

U^S-PC

Université Sorbonne
Paris Cité

université
PARIS
DIDEROT
PARIS 7

Thèse de doctorat de l'Université Sorbonne Paris Cité
Préparée à l'Université Paris Diderot
ECOLE DOCTORALE 386 — SCIENCES MATHÉMATIQUES DE PARIS CENTRE
INSTITUT DE RECHERCHE EN INFORMATIQUE FONDAMENTALE (I.R.I.F.)
ÉQUIPE PREUVES PROGRAMMES SYSTÈMES

NON-IDEMPOTENT TYPING OPERATORS, BEYOND THE λ -CALCULUS

par Pierre VIAL

Thèse de doctorat d'informatique

dirigée par

Delia KESNER
Damiano MAZZA

Directrice de thèse
Co-directeur de thèse

Thèse soutenue publiquement le 7 décembre 2017 devant le jury constitué de

Directrice de thèse	KESNER Delia	Professeur	Paris Diderot
Rapporteur	KLOP Jan Willem	Professeur émérite	Vrije Universiteit
Examineur	DAL LAGO Ugo	Maître de conférence	Università di Bologna
Chargé de Recherche	MAZZA Damiano	Co-directeur de thèse	Paris 13
Examinatrice	VAN RAAMSDONK Femke	Maître de conférence	Vrije Universiteit
Président du Jury	REGNIER Laurent	Professeur	Université Aix-Marseille
Rapporteur	REHOF Jakob	Professeur	TU Dortmund

Opérateurs de typage non-idempotents, au delà du λ -calcul

Contributions: L'objet de cette thèse est l'extension des méthodes de la théorie des types intersections non-idempotents, introduite par Gardner et de Carvalho, à des cadres dépassant le λ -calcul *stricto sensu*.

- Nous proposons d'abord une caractérisation de la normalisation de tête et de la normalisation forte du $\lambda\mu$ -calcul (déduction naturelle classique) en introduisant des types unions non-idempotents. Comme dans le cas intuitionniste, la non-idempotence nous permet d'extraire du typage des informations quantitatives ainsi que des preuves de terminaison beaucoup plus élémentaires que dans le cas idempotent. Ces résultats nous conduisent à définir une variante à petits pas du $\lambda\mu$ -calcul, dans lequel la normalisation forte est aussi caractérisée avec des méthodes quantitatives.
- Dans un deuxième temps, nous étendons la caractérisation de la normalisation faible dans le λ -calcul pur à un λ -calcul infiniement lié aux arbres de Böhm et dû à Klop et al. Ceci donne une réponse positive à une question connue comme le problème de Klop. À cette fin, il est nécessaire d'introduire conjointement un système (système **S**) de types infinis utilisant une intersection que nous qualifions de séquentielle, et un critère de validité servant à se débarrasser des preuves dégénérées auxquelles les grammaires coinductives de types donnent naissance. Ceci nous permet aussi de donner une solution au problème 20 de TLCA (caractérisation par les types des permutations héréditaires). Il est à noter que ces deux problèmes n'ont pas de solution dans le cas fini (Tatsuta, 2007).
- Enfin, nous étudions le pouvoir expressif des grammaires coinductives de types, en dehors de tout critère de validité. Nous devons encore recourir au système **S** et nous montrons que tout terme est typable de façon non triviale avec des types infinis et que l'on peut extraire de ces typages des informations sémantiques comme l'ordre (arité) de n'importe quel λ -terme. Ceci nous amène à introduire une méthode permettant de typer des termes totalement non-productifs, dits termes muets, inspirée de la logique du premier ordre. Ce résultat prouve que, dans l'extension coinductive du modèle relationnel, tout terme a une interprétation non vide. En utilisant une méthode similaire, nous montrons aussi que le système **S** collapse surjectivement sur l'ensemble des points de ce modèle.

Mots-clés: lambda calcul, type, non-idempotent, type intersection, typage infiniement, coinduction, intersection séquentielle, type rigide, lambda-mu calcul, logique classique, Curry-Howard, collapse, réduction productive, réduction non-productive

Non-idempotent typing operators, beyond the λ -calculus

Abstract In this dissertation, we extend the methods of non-idempotent intersection type theory, pioneered by Gardner and de Carvalho, to some calculi beyond the λ -calculus.

- We first present a characterization of head and strong normalization in the $\lambda\mu$ calculus (classical natural deduction) by introducing non-idempotent union types. As in the intuitionistic case, non-idempotency allows us to extract quantitative information from the typing derivations and we obtain proofs of termination that are far more elementary than those in the idempotent case. These results leads us to define a small-step variant of the $\lambda\mu$ calculus, in which strong normalization is also characterized by means of quantitative methods.
- In the second part of the dissertation, we extend the characterization of weak normalization in the pure λ -calculus to an infinitary λ -calculus narrowly related to Böhm trees, which was introduced by Klop et al. This gives a positive answer to a question known as Klop’s problem. In that purpose, it is necessary to simultaneously introduce a system (system **S**) featuring infinite types and resorting to an intersection operator that we call sequential, and a validity criterion in order to discard unsound proofs that coinductive grammars give rise to. This also allows us to give a solution to TLCA problem # 20 (type-theoretic characterization of hereditary permutations). It is to be noted that those two problem do not have a solution in the finite case (Tatsuta, 2007).
- Finally, we study the expressive power of coinductive type grammars, without any validity criterion. We must once more resort to system **S** and we show that every term is typable in a non-trivial way with infinite types and that one can extract semantical information from those typings e.g. the order (arity) of any λ -term. This leads us to introduce a method that allows typing totally unproductive terms (the so-called mute terms), which is inspired from first order logic. This result establishes that, in the coinductive extension of the relational model, every term has a non-empty interpretation. Using a similar method, we also prove that system **S** surjectively collapses on the set of points of this model.

Key words: lambda calculus, type, non-idempotent, intersection type, infinitary typing, coinduction, sequential intersection, rigid type, lambda-mu calculus, classical-logic, Curry-Howard, collapse, productive reduction, non-productive reduction



MARCELLE NULLANS
10 février 1931 - 21 août 2017
IN MEMORIAM

Cette thèse est dédiée à ma grand-mère, Marcelle Nullans,
en mémoire de sa générosité, de sa bienveillance et de tout ce qu'elle m'a
appris.

Remerciements

Mes remerciements vont d'abord à Delia Kesner, qui a accepté de me diriger en thèse environ un quart d'heure après m'avoir rencontré et a bien voulu composer un sujet à quatre mains en un temps record avant l'heure limite de remise des projets! Je suis aussi redevable à Delia de m'avoir posé la question qui devait être reconnue *a posteriori* comme le problème de Klop: c'est de là que presque tout est parti. Pendant ces trois années de thèse, Delia m'a prodigué ses conseils et a fait preuve d'une infinie patience pour lire et relire encore un grand nombre de versions de mes articles et de mon manuscrit, dont certaines étaient peut-être, avouons-le, un peu thomas-bernhardiennes.

Merci à Damiano Mazza de m'avoir d'abord encadré en stage "d'initiation à la recherche" en M2 et pour son enthousiasme communicatif, ses mille idées qui surgissent à chaque instant. Merci aussi pour la confiance et la liberté offerte pendant ces trois ans et demi, ses nombreux encouragements et pour m'avoir en premier orienté vers le λ -calcul infini !

Jan Willem Klop et Jakob Rehof m'ont fait l'honneur de bien vouloir être les rapporteurs de ma thèse. Je les remercie pour tout le temps et tous les efforts consacrés à lire l'intégralité du manuscrit, et pour les remarques détaillées qu'ils ont bien voulu me faire sur celui-ci. Je n'ose pas penser que cela a été une sinécure. Merci à Jan Willem Klop, pour avoir, avec d'autres, inventé le λ -calcul infinitaire et à Jakob Rehof pour les très agréables moments de discussion que nous avons eus à Budapest, Reykjavik ou Oxford. Que Ugo dal Lago, Femke van Raamsdonk et Laurent Regnier soient aussi remerciés d'avoir bien voulu faire partie de mon jury de thèse.

Je remercie Alexis Saurin, aussi bien l'ami, le responsable des thésards aux conseils avisés que le chercheur avec qui j'ai eu de nombreuses discussions éclairantes, notamment à travers sa connaissance et sa compréhension encyclopédiques des langages de programmation, qui vont bien au delà du λ -calcul.

Ma gratitude va aussi à Giulio Manzonetto, qui a parfois pris la casquette de co-co-directeur de thèse et s'est intéressé à ce qui constitue la dernière partie de ma thèse. Giulio a notamment relu plusieurs articles qui constituent la fin de ce manuscrit et j'ai profité extensivement de ses conseils en rédaction (et en typographie) scientifique. Je remercie aussi tous les chercheurs¹ avec qui j'ai eu des discussions scientifiques éclairantes: je pense notamment à Daniel de Carvalho, Antonio Bucciarelli, Thomas Ehrhard, Hugo Herbelin, Paul-André Melliès, Roman Péchoux, Matthieu Sozeau, Christine Tasson et Lionel Vaux.

Merci à Odile Ainardi et à Marie Fontanillas pour leur patience quasi-bouddhique et pour toute l'efficacité qu'elles ont déployée pendant mes missions et déplacements en

¹Selon la tradition, les précaires seront remerciés dans un deuxième temps.

France et à l'étranger, pour moi, comme pour tout le monde.

Le LMFI a été riche en rencontres. Je remercie Amina² Doumane, dont l'amitié a été très précieuse depuis que nous nous connaissons. Merci pareillement à Ludovic Patey, l'homme qui a le CNRS plusieurs fois chaque année, avec qui je ne compte plus les discussions de toutes sortes, autour d'un feu de camp ou ou d'un bassin rempli de poissons rouges. Je remercie Sonia Marin pour toutes ces heures passées sur les devoirs à la maison du LMFI et sur la géométrie filipendulaire (qui n'a rien à envier à la géométrie grothendieckienne).

Je remercie Charles Grellois de m'avoir fait profiter de sa culture scientifique, de ses slides et de ses conseils avisés, pour traverser les différentes ordalies que le thésard doit connaître avant de pouvoir soutenir. Merci à Cyrille Chenavier pour ses explications historiques sur la rivalité entre Attila et Charles Quint ou Hugues Capet, et pour ne toujours faire ressortir que la meilleure partie de chacun (et le faire savoir). Je ne dirai pas de mal d'Étienne Miquey, dernier compagnon³ de ligne gauche, car nous sommes appelés à nous revoir assez régulièrement dans un futur proche. Il faut quand même reconnaître que ses explications sur la logique classique et les types dépendants n'étaient pas inintéressantes. Merci à Maxime Lucas pour sa loquacité, et à Antoine Allieux pour sa résistance à l'alcool. Merci à Léonard Guetta pour sa magnanimité et son couscous à venir⁴. Merci à tous les autres thésards et stagiaires de l'IRIF, d'aujourd'hui ou d'hier: Clément Jacq (qui était de facto co-directeur de PPS), Thibaut Girka, Pierre-Marie Pedrot, Yann Hamdaoui, Gabriel Radanne, Théo Zimmermann, Tommaso Petrucciani, Théo Winterhalter, Nicolas Jeannerod, Victor Lanvin, Leo Stefanescu, Zeinab Galal, Jules Chouquet, Chaitanya Leena Subramaniam, Rémi Nollet, Cédric Ho Tanh, Matthieu Bouttier, Hadrien Batmalle, Raphaëlle Crubillé, Cyprien Mangin, Thibault Godin, Pierre Cagne et Adrien Husson. Merci aux *doctorantes honoris causa*, Clémentine, Daniel et Maud. Je n'oublie pas les thésards de Paris 13, Luc Pellissier, Alice Pavaux et Thomas Rubiano, avec qui je suis allé notamment à OPLSS, et plus encore, à Hot Cougar Springs. Toute ma gratitude politico-culturelle à Luc, qui m'a fait connaître Thomas Römer, les Brigandes et Liza Monet.

J'ai rencontré des professeurs exceptionnels à l'École normale supérieure, comme François Loeser, Jean-François Le Gall, Marc Rosso, quoique mon assiduité ne m'ait pas fait profiter de leurs enseignements autant qu'ils le méritaient. Marc Rosso a accepté de me recevoir au printemps 2012 alors que je m'interrogeais sur une réinscription à l'Université et m'a orienté vers le LMFI. Arnaud Durand a ensuite bien voulu accepter ma candidature et j'ai pu commencer les cours le lendemain de notre entretien le mois de septembre suivant. À tous les deux, merci.

Au cours de ma scolarité, j'ai eu la chance d'être élève dans la classe de Mme Alini, en CP et CE1 à l'école des Cités-Cécile, ainsi que celle de Mlle Welter et M. Dupuy, au collège Charles Guérin de Lunéville. Je leur exprime ma gratitude, pour m'avoir montré ce qu'était un cours vivant et à l'écoute⁵ des élèves. Merci aussi à Philippe Château, mon professeur en MP* au lycée Henri Poincaré. De l'autre côté du miroir, il me faut

²Qu'elle n'oublie pas cependant les pièces de tissu simples et de bon goût qu'elle a promises à une bonne partie des membres de l'IRIF du bas.

³Quoiqu'il ait, contrairement à l'auteur, astucieusement évité les frais de réinscription en "4ème année".

⁴Le couscous est à venir, pas la magnanimité, qui n'est plus à prouver

⁵Pour l'improbable lecteur qui aurait eu la chance de fréquenter l'IUFM et autres ESPE, cette expression vieillotte peut plus ou moins se traduire par "pédagog-i-e différencié-e".

remercier mes anciens élèves, sur qui j'ai expérimenté toutes les méthodes de travail qui me faisaient défaut. Sans eux, je n'aurais sans doute pas pu mettre le pied à l'étrier.

Enfin, un grand merci à Nicolas Tabareau de m'accueillir à Nantes pour une année de post-doc. C'est une immense opportunité de découvrir Coq et la théorie homotopique des types avec lui.

Reprendre les études n'a pas été de soi. Sans les nombreux encouragements de Camille Tauveron, Jean-Baptiste Bilger, Julien Rabachou, Hélène Valance, Sandra Collet et Stéphane Pouyaud, je n'aurais sans doute jamais dépassé mes hésitations et retrouvé le chemin des écoliers. Qu'ils soient tous remerciés pour m'avoir accompagné un moment ou un autre. Pour la même raison, je remercie aussi Michel Plouznikoff. Merci à Shirine, Marie-Cécile, David et Rossella pour leur amitié.

Mes années de thèse auraient été traversées bien différemment si Mahek ne m'avait pas apporté l'illumination quelques heures avant un départ en avion: je lui en serai toujours reconnaissant. Je suis particulièrement redevable à la famille Rollet, à Fanny et à Christine en particulier, de m'avoir chaleureusement accueilli au printemps 2015. Je remercie aussi Joseph Beaume pour son amitié, ainsi que pour les nombreux livres qu'il m'a envoyés. Ma reconnaissance aussi à Rachel Daniel, pour sa compréhension profonde des expériences intérieures. Merci à Mariana pour les dernières semaines de septembre, alors que j'apportais les touches finales à ce manuscrit. Merci à Mathilde Régent et Aude Leblond, pour leur voisinesque et verre-siffleuse présence pendant ces trois années, matérialisée par d'innombrables tisanes, bières, vodkas et autres Yogi Teas[©].

Je remercie mes soeurs, Mélanie et Camille, pour tout ce que j'ai partagé avec elles, à Paris, à Lunéville ou en Bretagne. Enfin, je remercie mes parents pour le soutien qu'ils m'ont apporté tout au long de ma reprise d'études.

Contents

Contents	9
1 Introduction	15
1.1 Functions and Termination	17
1.2 Processing Computation in Functional Programming	21
1.3 Intersection (or not) Types in the λ -Calculus	26
1.4 Infinitary Computation, Coinduction and Productivity	34
1.5 Main Contributions (Technical Summary)	38
1.6 How to read this thesis	41
I PRELIMINARIES	43
2 Lambda Calculus	47
2.1 Pure Lambda Calculus	48
2.1.1 Tracks and Labelled Trees	48
2.1.2 Lambda Terms and Alpha Equivalence	51
2.1.3 Beta Reduction, Redexes and Normal Forms	53
2.1.4 Notable Lambda Terms	54
2.1.5 Residuals and Quasi-Residuals	55
2.1.6 Reduction Sequences and Residuation	58
2.1.7 Contexts	58
2.2 Normalizations and Reduction Strategies	59
2.2.1 Head Redexes, Head Normal Forms, Head Reduction	60
2.2.2 Weak Normalization and Leftmost Reduction	61
2.2.3 Strong Normalization	63
2.3 Tinkering with Normalization	63
2.3.1 Stable Positions and Sets of Normal Forms	64
2.3.2 Mute Terms and Order of a Lambda Term	65
2.3.3 Toward Infinitary Normalization	66
2.3.4 Partial Normal Forms	68
2.3.5 Böhm Reduction Strategies	70
2.4 A Lambda-Calculus with Explicit Substitutions	72
3 Intersection Type Systems	75
3.1 From the λ -Calculus to Intersection Types Theory	76

3.1.1	The Curry-Howard Correspondence	76
3.1.2	Lambda-Calculus and Type Theory	76
3.1.3	A Simple Type System	78
3.1.4	Types and Termination	79
3.2	Principles and Examples of Intersection Types	80
3.2.1	Towards Strictness and Relevance	80
3.2.2	Intersection Operator, Sets and Multisets	83
3.2.3	System \mathcal{D}_0 (Idempotent Intersection)	85
3.2.4	System \mathcal{R}_0 (Non-Idempotent Intersection)	86
3.3	Discussing Subject Reduction and Subject Expansion	87
3.3.1	Uses and Behaviors of Intersection Type Systems	88
3.3.2	Subject Reduction and Subject Expansion	89
3.3.3	Context Preservation	91
3.3.4	Failure of Subject Expansion with Relevant Idempotent Intersection	93
3.3.5	Weakening and Irrelevant Intersection Types Systems	94
3.4	Study of Head Normalization in System \mathcal{R}_0	97
3.4.1	Typed and Untyped Parts of a Term	97
3.4.2	Typing (Head) Normal Forms	98
3.4.3	Weighted Subject Reduction	101
3.4.4	Characterization of Head Normalization and Completeness of Head Reduction	101
3.4.5	Order Discrimination	102
4	A Few Complements on Intersection Types	105
4.1	A Bit of This and a Bit of That	105
4.1.1	An Alternative Presentation of Relevant Derivations	105
4.1.2	Confluence (and Non-Confluence) of Type Systems	106
4.1.3	Type Systems and their Features (Summary)	109
4.2	Intersection Types for the Lambda Calculus with Explicit Substitutions	109
4.3	Tait's Realizability Argument	111
4.3.1	The Failure of an Induction	111
4.3.2	Interpretation	113
5	Characterizing Weak and Strong Normalization	117
5.1	Characterizing Weak Normalization	117
5.1.1	Positive and Negative Occurrence of a Type	117
5.1.2	Unforgetfulness	118
5.1.3	Unforgetfulness and Typing Rules	119
5.1.4	Weak Normalization	121
5.2	Characterizing Strong Normalization	122
5.2.1	Erasable Subterms	122
5.2.2	Subject Reduction and Expansion in \mathcal{S}	124
5.2.3	Strong Normalization as an Inductive Predicate	125
5.2.4	Characterizing Strong Normalization	127
5.2.5	Obtaining an Upper Bound for Normalizing Sequence	128
II	RESOURCES FOR CLASSICAL NATURAL DEDUCTION	131
	Presentation	133

6	The Lambda-Mu Calculus	135
6.1	Classical Logic in Natural Deduction	135
6.1.1	Getting Classical Logic	135
6.1.2	Focused Classical Natural Deduction	136
6.2	The Lambda-Mu Calculus	137
6.2.1	Lambda-Mu Terms	138
6.2.2	Simply Typed Lambda-Mu Calculus	139
6.2.3	Operational Semantics	139
6.2.4	Subject Reduction for Simply Typed Lambda-Mu	140
6.2.5	Normalization in Lambda-Mu Calculus	141
7	Non-Idempotent Intersection and Union Types for Lambda-Mu	143
7.1	Auxiliary Judgments and Choice Operators	144
7.2	Quantitative Type Systems for the λ_μ -Calculus	147
7.2.1	Types	147
7.2.2	System $\mathcal{H}_{\lambda_\mu}$	148
7.2.3	Design of System $\mathcal{H}_{\lambda_\mu}$	151
7.2.4	System $\mathcal{S}_{\lambda_\mu}$	153
7.3	Typing Properties	154
7.3.1	Forward Properties	154
7.3.2	Backward Properties	160
7.4	Strongly Normalizing λ_μ -Objects	164
7.5	Relevance (an Inquiry)	167
8	A Resource Aware Semantics for the Lambda-Mu-Calculus	169
8.1	The $\lambda_{\mu\mathbf{x}}$ -calculus	169
8.1.1	Syntax	170
8.1.2	Operational Semantics	171
8.1.3	Typing System	171
8.2	Typing Properties	173
8.2.1	Forward Properties	173
8.2.2	Backward Properties	178
8.3	Strongly Normalizing $\lambda_{\mu\mathbf{x}}$ -Objects	182
8.4	Conclusion	184
III INFINITARY NORMALIZATION AND SEQUENTIAL INTERSECTION		185
	Presentation	187
9	The Infinitary Lambda-Calculus	189
9.1	Böhm Trees	189
9.2	Induction and Coinduction	191
9.2.1	Infinite Labelled Trees	191
9.2.2	Smallest and Biggest Invariant Subsets	192
9.2.3	Inductive <i>vs.</i> Coinductive Grammars	192
9.3	The Infinitary Lambda Calculi	194
9.3.1	The Full Infinitary Calculus	194
9.3.2	The Infinitary Calculus of Böhm Trees	198
9.3.3	Böhm Trees Revisited	201

10 Klop’s Problem	203
10.1 How to Answer Positively to Klop’s Problem	206
10.1.1 The Finitary Type System \mathcal{R}_0 and Unforgetfulness	206
10.1.2 Finitarily Typing the Infinite Terms	207
10.1.3 Infinitary Subject Expansion by Means of Truncation	209
10.2 Intersection by means of Sequences	211
10.2.1 Towards System \mathbf{S}	212
10.2.2 Rigid Types	213
10.2.3 Rigid Derivations	214
10.3 Statics and Dynamics	216
10.3.1 Bipositions and Bisupport	216
10.3.2 Quantitativity and Coinduction	217
10.3.3 One Step Subject Reduction and Expansion	217
10.3.4 Safe Truncations of Typing Derivations	220
10.3.5 A Proof of the Subject Reduction Property	221
10.4 Approximable Derivations and Unforgetfulness	222
10.4.1 The Lattice of Approximation	222
10.4.2 Approximability	223
10.4.3 Unforgetfulness	224
10.4.4 The infinitary Subject Reduction Property	226
10.5 Typing Normal Forms and Subject Expansion	227
10.5.1 Support Candidates	228
10.5.2 Natural Extensions	229
10.5.3 Approximability	230
10.5.4 The Infinitary Subject Expansion Property	231
10.6 Conclusion	232

IV UNDERSTANDING UNPRODUCTIVE REDUCTION THROUGH LOGICAL METHODS **233**

Presentation 235

11 An Informal Presentation of Threads **239**

11.1 Threads, Syntactic Polarity and Consumption	239
11.1.1 Ascendance, Polar Inversion and Threads	239
11.1.2 Syntactic Polarity and Consumption	241
11.1.3 Referents of Threads, Applicative Depth and Brotherhood	244
11.2 Collapsing Redex Towers	246
11.2.1 Negative Left-Consumption and Redex Towers	246
11.2.2 Collapsing a Redex Tower Sequence	248
11.3 Formalizing Ascendance and Polar Inversion in System \mathbf{S}	250
11.3.1 Applications and Tracking in System \mathbf{S}	250
11.3.2 Abstractions and Tracking in System \mathbf{S}	251

12 Complete Unsoundness: a Linearization of the λ -Calculus **253**

12.1 Coinductive Type Systems	256
12.1.1 A Coinductive Simple Type System	256
12.1.2 Complete Unsoundness and Relevance	256
12.1.3 Typing some Notable Terms in System \mathcal{R}	257

12.1.4	Type System \mathbf{S} (Sequential Intersection)	258
12.2	Bisupport Candidates	260
12.2.1	A Toy Example: Support Candidates for Types	260
12.2.2	Toward the Characterization of Bisupport Candidates	261
12.2.3	Tracking a Type in a Derivation	262
12.2.4	Type Formation, Type Destruction	265
12.2.5	Threads and Minimal Bisupport Candidate	266
12.3	Nihilating Chains	269
12.3.1	Polarity and Threads	269
12.3.2	Interactions in Normal Chains	271
12.3.3	Complete Unsoundness (almost) at Hand	274
12.4	Normalizing Nihilating Chains	275
12.4.1	Quasi-Residuals	276
12.4.2	The Collapsing Strategy	280
12.4.3	Redex Towers	280
12.5	Applications	282
13	The Surjectivity of the Collapse of Sequential Intersection Types	287
13.1	From Representing Types in System \mathbf{S} to Representing Derivations	290
13.1.1	Multiset Types as Collapses of Sequential Types	290
13.1.2	The Representation Theorem and Hybrid Derivations	293
13.1.3	System \mathcal{R} and the Hybrid Construction	295
13.2	Subject Reduction	296
13.2.1	Encoding Reduction Choices with Interfaces	296
13.2.2	Residuation and Encoding Reduction Choices	298
13.3	Representation Theorem and Isomorphisms of Derivations	299
13.3.1	Isomorphisms of Operable Derivations	300
13.3.2	Resetting an Operable Derivation	301
13.3.3	Relabelling a derivation	302
13.4	Edge Threads	304
13.4.1	Threads and Consumption of Mutable Edges	305
13.4.2	Edge Threads and Syntactic Polarity	307
13.4.3	Brother Chains and Representation	307
13.4.4	Towards the Final Stages of the Proof	309
13.4.5	Top Ascendants, Syntactic Polarity and Referents	310
13.5	Residuation of Threads and the Collapsing Strategy	313
13.5.1	Edges and Residuation	314
13.5.2	The Collapsing Strategy for Operable Derivations	315
13.5.3	Conclusion of the Proof	315
13.6	Conclusion	317
	Conclusion	319
	Bibliography	323
A	Complements to Klop's Problem	333
A.1	Expanding the Π'_n and Π'	333
A.2	Equinecessity, Reduction and Approximability	336
A.2.1	Equinecessary bipositions	336
A.2.2	Approximability is stable under (anti)reduction	337

A.2.3	Equinecessity and Bipositions of Null Applicative Depth	337
A.3	Lattices of (finite or not) approximations	338
A.3.1	Meets and Joins of Derivations Families	342
A.3.2	Reach of a derivation	343
A.3.3	Proof of the subject expansion property	344
A.4	Approximability of the quantitative NF-derivations	344
A.4.1	Degree of a position inside a type in a derivation	344
A.4.2	Truncation of degree n	345
A.4.3	A Complete Sequence of Derivation Approximations	347
A.5	Isomorphisms between \mathbf{S} -Derivations	347
A.6	Approximability cannot be defined by means of Multisets	349
A.6.1	Quantitativity in System \mathcal{R}	349
A.6.2	Representatives and Dynamics	349
A.7	A Positive Answer to TLCA Problem 20	351
B	Residuation, Threads and Isomorphisms in System \mathbf{S}_{op}	355
B.1	Subject Reduction	355
B.1.1	Residual Derivation (Hybrid)	356
B.1.2	Residual Types and Contexts (Hybrid Derivations)	357
B.1.3	Residual Interface	359
B.1.4	Proof of the Representation Lemma	361
B.2	Isomorphisms and Relabelling of Derivations	362
B.2.1	Isomorphisms of Operable Derivations	362
B.2.2	Resetting an Operable Derivation	363
B.3	Edge Threads, Brotherhood and Consumption	364
B.4	Residuation for Mutable Edges and Threads	365
B.4.1	Edges and Residuation	365
B.4.2	Residuals of Edges Threads	366

Chapter 1

Introduction

Mise en bouche

This thesis is about the relations between type theory and mathematical logic. We specialize into the **non-idempotent intersection type theory** applied to the λ -calculus. Type theory provides syntactic *certificates* that some programs behave well *e.g.*, *terminate* and the λ -calculus can be seen as the mold on which *functional programming* is designed. Thus, forgetting temporarily about the words “intersection” and “non-idempotent”, we are interested in finding guarantees that some programs of a language called the λ -calculus terminate.

The matter of finding such certificates of good behavior is indeed fundamental when working with expressive programming languages and it is a crucial step for having programs that meet their *specification*: the specification of a program is everything what it is supposed to do *e.g.*, you do not want the embedded system of your aircraft to open the doors while it is flying¹ or to bug because of an unnoticed coding mistake)

The syntactic nature of the certificates – meaning that they rely on the source code – provided by type theory is an invaluable asset: this avoids all coding mistakes, contrary to semi-random benchmarking methods that only *sample* the instances of a program. As it turns out, type theory also ensures that *streams* – *i.e.* programs running without limit of time – keep on *producing* computations instead *e.g.*, of looping in a cycle of states.

But what is a type? Roughly speaking, typing is a very simple idea: it just consists in describing what kind of data (integer? floating number? string of characters? array of integers? function?) is stored in the memory (in the *variables*) used by the program. We may then assign types to programs *e.g.*, the function `sumLength` that takes *two* strings `s1` and `s2` and outputs the sum of their respective lengths has the type $(\mathbf{String} \times \mathbf{String}) \rightarrow \mathbf{int}$. On the left-hand side (the *source*), we find the types of the inputs² and on its right-hand side (the *target*), the type of the output. From the practical perspective, typing provides a form of safety for the program developer: one cannot inadvertently switch the name of an integer variable with that of a string variable without having an error, whereas some untyped languages may let the mistake pass unbeknownst to the programmer.

It is not this kind of type-safety we are interested in in this thesis, but that of the types as a guarantee of termination. But how does such a guarantee hold? Here is where

¹Except if you are Tom Cruise.

²Separated by the operator \times , roughly corresponding to a cartesian product.

mathematical logic comes into play. William A. Howard, extending some observations by Haskell Curry, noticed that type systems really look like some *inference* systems coming from mathematics, not only because *assignment rules* of types system and *inference rules* of logic are oddly similar (*static* correspondence), but also because the *execution-steps* of programs are in almost every point homologous to the so-called *cut-elimination steps* (*dynamical* correspondence). As we will see, the cut-elimination procedure was introduced by Gerhard Gentzen [44] to give a (partial) proof of coherence of some logic underlying arithmetic: from a high-level perspective, this procedure consists in transforming a proof of an assertion A subdivided in several lemmas into a self-contained one-block proof of the same statement A .

An interesting aspect of Gentzen’s contribution, later extended to intuitionistic³ *Natural Deduction* by Prawitz [94], is that he proved that this cut-elimination procedure is terminating. As a consequence of the correspondence developed by Curry and Howard (the so called **Curry-Howard correspondence**), the fact that the typed programs (w.r.t. some type systems) are terminating became a straightforward consequence of some proofs of cut-elimination. This was the first of many fruitful back-and-forth exchanges between programming languages and logic – each one shedding light on the other –, leading to powerful type theories (as the Martin-Löf type theory) and proof assistants like Coq or Agda, used both in the industry and in fundamental mathematics.

This introduction chapter is dedicated to making the keywords of this foreword explicit as “function”, “types”, “ λ -calculus”, “intersection”, “non-idempotent” as well as some important phenomena and difficulties occurring in functional programming, before presenting our contributions in an informal way, then in a more concise and technical manner from p. 38 on. We also present some historical elements regarding the birth of computation theory (before the invention of the first computers!) and the undecidability of the so-called halting problem, which is the limitation result that intersection types are constantly at odds with. More precisely, we follow the plan below:

- Section 1.1:
 - We first present some very basic features of functional programming and typing and also very basic examples of terminating/non-terminating programs.
 - We present the circumstances leading to the creation of computer science (the *Entscheidungsproblem*, Gödel incompleteness theorems, Turing machines and the undecidable of the halting problem).
- Section 1.2: we say a few words on the λ -calculus as a paradigm of functional programming and present some fundamental notions (duplication, creation, reduction paths, reduction strategies) related to functional computation.
- Section 1.3: we explain how intersection types are situated in the general picture of type theory (notably in regard to higher-order typing), we present some of their uses and sketch the mechanisms of non-idempotent intersection.
- Section 1.4: we discuss infinitary semantics of the λ -calculus and how unsoundness arises from infinitary type systems.

To the reader who is familiar with the concepts above, we suggest to pass directly to Sec. 1.5 and Sec. 1.6, respectively presenting a technical description of all the contributions of this thesis and a road-map with the dependencies between chapters.

³We say a few words about intuitionism p. 33.

1.1 Functions and Termination

Functional Programming Functional programming is a high-level paradigm of language, in which a program is thought as a function, that takes parameters as inputs, performs a computation on these parameters, and then outputs a return value, if the computation terminates. The main aspects that differentiate functional programming from other paradigms (*e.g.*, imperative/assembler programming) are the following:

- Functions are taken as first-class objects. Some programs can be applied to functions (and not only to data). Such programs are said to be of **higher-order**. For instance, the function `map` can take *e.g.*, a function `f` from \mathbb{N} to \mathbb{N} as input and then outputs the lifted *function* from the set of the *arrays* of integers to itself, that applies `f` to each element of the array: thus, `map(f)` is the function that takes an array `[n1 n2 ... nk]` and outputs `[f(n1) f(n2) ... f(nk)]`.
- **Absence of side-effects**: side-effects occur when the state *e.g.*, of a variable is modified because some function/routine has been executed. For instance, arrays are sensitive to side-effects in most languages, particularly when imperative features are involved. The modification of a variable by a function call is often a convenient feature, but because of side-effects, a function fed with the *same* arguments can output different results, depending on the place of the call in the execution. This makes programs with side-effects difficult to verify, especially those with lengthy source codes. In contrast to that, functional programming advocates *referentially transparent* or *pure* functions *i.e.* functions that do not cause side-effects. Most usual functional languages actually have imperative features and are thus not impervious to side-effects, except for instance Haskell, which is an example of purely⁴ functional language.
- Functions are suitable for typing, which helps avoid bugs and coding errors. They also help developers to organize their programs, to make them more readable, to subdivide them into modules and thus, to make them easier to maintain.

The Problem of Termination One very simple observation is that some programs do not terminate. This problem is pervasive in most expressive programming language (and is independent from the functional features of the language). For instance, in imperative style, the program below never ends:

```
x = 1
while (x ≠ 0) {
  x = x + 1 }
print("It's quite late, don't you think?")
```

That is: the initial value of `x` is 1. One keeps on incrementing `x` so long that the value of `x` is not 0, so that `x` is equal to 1, then to 2, then to 3, etc... Of course, `x` will never be equal to 0, so that the `while` loop will keep on running and the message will never be printed.

⁴Haskell features explicit *monadic* constructions to process side-effects.

Some other programs also obviously terminate (although one is never sure that no mistake has been made) *e.g.*, the one below, computing the sum $1+2+\dots+100$:

```
sum = 0
for(n = 0; n < 100; n++){
    sum = sum + n}
return sum
```

Since this is computers and programs we are talking about, one may wonder whether there is an *automatic* way to check whether a given program P terminates when applied to a given input x . This is the core of the **Halting Problem** which, more precisely, consists in finding (for a given programming or computing language) an algorithmic procedure that, given a source code of a program P and an input x of P , determines whether P terminates on x or not, *if such a procedure exists*. As it turns out, the halting problem does not have a solution. We present in the next section how this negative result came to knowledge.

Hilbert's Program stops, some computations do not

To understand the importance of termination problems in programming, one must go back to the early 30s. Hilbert's program of formalizing completely mathematics, presented in 1901, was brought to a sudden stop following the publishing of Gödel's incompleteness theorems. According to the first incompleteness theorem, *effective* axiomatization of arithmetic cannot be both *consistent* and *complete*. The three italicized words demand precision:

1. By *effective* axiomatization, one means that it can be checked whether a possible proof Π in this axiomatization is correct or not.
2. By *consistent*, one means that one cannot prove both a proposition and its negation in the axiomatization. Here, consistent is a synonym of coherent.
3. By *complete*, one means that, for any given closed formula⁵ F of the arithmetic, there is, in the axiomatization, a proof of F or there is a proof of $\neg F$, the negation of F .

If other words, by this theorem, no effective and consistent theory \mathcal{T} axiomatizing arithmetic can ensure that every arithmetic statement is provable or disprovable. Thus, there are arithmetic theorems that are true, but the fact that they are true cannot be established by syntactical means (*i.e.* by human-designed means!). The second incompleteness theorem states that the consistency of an effective theory \mathcal{T} cannot be proved by means of a proof of \mathcal{T} *i.e.* an effective theory *cannot* prove its own consistency (unless it is contradictory, in which case the proofs of \mathcal{T} do not mean anything).

⁵In short, a formula of first order arithmetic is a well-formed expression (*e.g.*, not “ $+ 5 \times 8 =$ ”) using the operators “+”, “-”, “ \times ”, the connective \wedge (“and”), \vee (“or”) and \neg (“not”), the quantifiers “ \forall ” (“for all”) and \exists (“there exists”), the binary relation “=” and possibly involving variables. This is enough to express most concepts involving integers (*e.g.*, comparison \leq , euclidean division, etc). A formula is *closed* when every variable is quantified.

The Decision Problem Hilbert had complemented his consistency program with the so-called *Entscheidungsproblem* (literally, the Decision Problem), which consisted in finding an algorithmic procedure that, given a first order formula F and an (effective) set of axioms, would output **True** if F is a syntactic consequence of the axioms and **False** if not, *in the case such a procedure existed*. Gödel's theorems were a shock for many mathematicians, philosophers and logicians, and moreover, they were a strong indication that an algorithm of the *Entscheidungsproblem* did not exist as well, which had not been hitherto suspected. However, Gödel's results did not straightforwardly give this negative answer, because their proof did not address the topic of computation, which was essential to understand what algorithmic procedures are and how they behave. Thus, the notion of computation, that had actually been overlooked by mathematicians and by logicians since the introduction of mathematics, came into light and caused intense reflection on its nature. Several alternative paradigms were proposed to provide a formal and comprehensive definition of computation. In his proof of the incompleteness theorems, Gödel had considered some obviously computable functions that are nowadays known as the *primitive recursive functions*. Integrating some remarks from Herbrand, he then defined the set of (partial) **recursive functions**, despite the fact he did not believe them to capture all possible computations (see [99], chapter 17). Church, who had introduced the λ -calculus in 1928, was convinced that a function was effectively computable iff it could be encoded by a λ -term, but many researchers, including Gödel, were skeptical. Finally, Turing defined his celebrated abstract machine [103] model, ever since known as the **Turing machines**. Turing explicitly conceived his machines by emulating (*i.e.* imitating in an abstract way) the human mind, seen as a device having a finite number of possible states and a reading/writing head interacting with an infinite tape, that is empty at the beginning of the execution (except for finitely many symbols). Very roughly, this captured the idea that (1) a human mind (or a cluster thereof) can handle only a finite number of data (*i.e.* what is already written on the tape) and this, in finitely many ways (captured by a finite transition function) (2) a human being writes/erases one letter after the other. Last, the assumption that the tape is infinite gives rise to the possibility to conduct a computation (or a reasoning) without limitation in space or time (just, the computation or the reasoning must stop at some point), which is what the notions of decidability and computability are about.

The very design of Turing machines made them a very convincing comprehensive formalization of computation. It soon turned out [24, 25, 66, 104] that they had an *equivalent expressive power* to those of the λ -calculus and of the Herbrand-Gödel recursive functions up to some encoding *i.e.* the complete behavior of each one of these three models of computation can be implemented in the two others. This led to the **Church-Turing thesis**:

A function is effectively computable iff it can be implemented in a Turing Machine/by means of a recursive function/a λ -term.

This is a *thesis*, and not a *theorem*, because there is still the very thin possibility that one day, one may find effective computing devices/languages that compute more than Turing machines. What is sure for now is that (1) every such devices/languages that has been made or conceived hitherto has been *proved* to fit within the scope of the Church-Turing thesis (2) for now, nothing suggests that this thesis could become obsolete.

With a formal definition of computation, it was a small step to adapt Gödel's techniques and to prove that the *Entscheidungsproblem* did not have a solution. Turing proved this negative result along with the introduction of his machines in 1936 [103] (Church [25] had a proof of the same fact using λ -calculus dating back from 1934, when it was not yet surely established that the λ -calculus encompassed the whole notion of computation). For instance, with Turing machines, the technique of Gödel consisting in arithmetizing⁶ the set of arithmetic *statements* is replaced by the implementation of a *Universal Turing Machine* *i.e.* a Turing machine that emulates any other Turing machine (modulo some encodings).

The Church-Turing thesis gave rise to the notion of **Turing-complete languages**: a programming language \mathcal{L} or a calculus is Turing-complete when its expressive power is equivalent to that of Turing-machines/the λ -calculus *i.e.* one can encode and emulate the Turing-machines in the language \mathcal{L} . The Church-Turing thesis reformulates into: every implementable calculus is *at best* Turing-complete.

Termination Along with the negative answer to the *Entscheidungsproblem* came other limitation results. Let us just talk about two of them:

- The **halting problem** is undecidable: there is no algorithmic procedure that can decide, given *any* program P and input x , whether P terminates on x .
- **Extensional equality** is undecidable: there is no algorithmic procedure that can decide, given *any* programs P and Q , whether P and Q are extensionally equal *i.e.* whether, for all input x , P terminates on x iff Q does and in that case, they output the same value.

Turing [103] and Church [25] respectively proved that no Turing machine and no λ -term could determine whether a Turing machine and a λ -term terminates. This, along with the Church-Turing thesis, means that there are no mechanical or humanly implementable ways to determine whether *any* given program terminates. It is actually possible to check when considering only programs of some poorly expressive language, but only in a strictly more powerful language (a language cannot decide the termination of its own programs, by a diagonal argument). In the case of the Turing-complete languages, this means that termination⁷ is impossible to check (in all generality) since there are no effective and more powerful languages than them, by the Church-Turing thesis.

⁶That is, encoding the *formulas* and *proofs* of first order arithmetic with *natural numbers* and its *inference rules* with primitive recursive *functions*. He then used the fact that arithmetic statements and their possible proofs could be represented by integers to express predicates on the set of arithmetic formulas by means of arithmetic formulas (!), such as “This formula has a proof in first order arithmetic”. This allowed him to enunciated, in an elaborated variation of the liar's paradox (“I am lying”) an auto-referential proposition G more or less saying “There is a proof of my negation”: thus, neither G nor its negation can be provable. This is *a priori* not a contradiction: this is just that truth cannot be captured by provability.

⁷Note that when a program is running, without supplementary information, one cannot know whether it just needs some more time to produce its output or it is locked in a loop that will never end.

Theory of Computation

- There is a universal and robust definition of computation, encompassing everything that is *effectively* computable by a machine or a human being (CT thesis).
- There *cannot* exist a general method verifying whether any given program terminates or not (*i.e.* there are programs whose non-termination cannot be determined).

1.2 Processing Computation in Functional Programming

The λ -Calculus as a Model of Functional Programming Language The λ -calculus can be seen as the skeleton on which functional programming languages are built and incidentally, the kernel of Caml was initially nothing more than an implementation of the *pure* (*i.e.* untyped) λ -calculus. From the mathematical point of view, the λ -calculus is a language in which *everything is a function*. As we will see, from this notion (of function), one can reconstruct many basic notions (*e.g.*, integers, lists, etc). One can draw a parallel between this approach and the Zermelo-Fraenkel set theory (ZF) in which every object (numbers, lists, functions) is built from the notion of set. For instance, the program/ λ -term \mathbf{app}_3 representing the natural number 3, is the higher-order function taking two arguments \mathbf{f} and \mathbf{x} (intuitively, \mathbf{f} is a function and \mathbf{x} is an argument of \mathbf{f}) and applying \mathbf{f} three times to \mathbf{x} *i.e.* $\mathbf{app}_3(\mathbf{f}, \mathbf{x})$ outputs $\mathbf{f}(\mathbf{f}(\mathbf{f}(\mathbf{x})))$.

Let us now informally explain a few aspects of computation in functional programming languages, including the notion of reduction paths, reduction strategies and final state/normal form. We will use a hybrid syntax (not exactly that of the λ -calculus) in which functional programs are literal expressions that can be rewritten into other. The arrow \rightarrow_β , called⁸ **β -reduction** in the λ -calculus, represents one execution-step *e.g.*, $2 + 3 + 5 \rightarrow_\beta 5 + 5 \rightarrow_\beta 10$ or $\mathbf{f}(4) \rightarrow_\beta 4 \times 4 \rightarrow_\beta 16$ if \mathbf{f} is the function that takes an integer as input and outputs its square. We use the words “computation”, “expression” and “program” as synonyms.

One immediate observation is that some (*e.g.*, arithmetical) expressions correspond to **final states** of a computation/program whereas others are not (we say that they are **reducible**): for instance, $2 + 4 \times 5$ is not the final state of a computation nor $2 + 20$ whereas 22 is. Intuitively, the final state of an integer is its decimal notation. In the λ -calculus, a final state is called a **normal form** and termination is called **normalization**. The β -reduction of the λ -calculus can be used to encode almost all the operations of a given structure (*e.g.*, addition, multiplication for integers, concatenation for lists etc).

The λ -calculus

- An universal (but rudimentary) model of functional programming.
- Is Turing-complete.
- Every computation rule is subsumed by β -reduction.
- Allows studying both typed and untyped functions.

⁸Not to be confused with \mapsto , which represents functional mapping and \rightarrow , that represents functional types *e.g.*, the square function \mathbf{f} is defined by $\mathbf{n} \mapsto \mathbf{n} \times \mathbf{n}$ and is of type $\mathbf{Nat} \rightarrow \mathbf{Nat}$ here, meaning that both its single input and its output are natural numbers.

Computing a Functional Expression A step of computation (β -reduction) may consist in just reducing an operation (and **destroying** an operand) *e.g.*, $2 + 3 \times 5 \rightarrow_{\beta} 2 + 15 \rightarrow_{\beta} 17$, but actually, many computation steps make the current expression more complex. For instance, let \mathbf{f} be the function from \mathbf{Nat} to \mathbf{Nat} , that takes a natural number \mathbf{n} and outputs $(2 + 3) \times \mathbf{n} \times \mathbf{n} \times \mathbf{n}$. We write $f(\mathbf{n}) \rightarrow_{\beta} (2 + 3) \times \mathbf{n} \times \mathbf{n} \times \mathbf{n}$. Now, how do we evaluate $\mathbf{f}(2 \times 3 + 1)$ step-by-step? An efficient way is this one:

$$\begin{aligned} \mathbf{f}(2 \times 3 + 1) &\rightarrow_{\beta} \mathbf{f}(6 + 1) \rightarrow_{\beta} \mathbf{f}(7) \rightarrow_{\beta} (2 + 3) \times 7 \times 7 \times 7 \\ &\rightarrow_{\beta} 5 \times 7 \times 7 \times 7 \rightarrow_{\beta} 35 \times 7 \times 7 \rightarrow_{\beta} 245 \times 7 \rightarrow_{\beta} 1715 \end{aligned}$$

Thus, in 7 elementary computation steps, we obtain the final result. But we may be less shrewd and proceed like this:

$$\begin{aligned} \mathbf{f}(2 \times 3 + 1) &\rightarrow_{\beta} (2 + 3) \times (2 \times 3 + 1) \times (2 \times 3 + 1) \times (2 \times 3 + 1) \\ &\rightarrow_{\beta} 2 \times (2 \times 3 + 1) \times (2 \times 3 + 1) \times (2 \times 3 + 1) \\ &\quad + 3 \times (2 \times 3 + 1) \times (2 \times 3 + 1) \times (2 \times 3 + 1) \\ &\rightarrow_{\beta} \dots \end{aligned}$$

and we go on like this: instead of reducing the parenthesized expressions, we expand the outer products. We also reach 1715 (the final state) at the end, but in several dozens of steps instead of just 7 and by manipulating considerably bigger expressions.

This example epitomizes:

- The notion of **reduction paths**: since a computation may contain several reducible sub-expressions, some computations can be processed in different ways.
- The possible **duplication** of the argument: $\mathbf{f}(\mathbf{n})$ unfolds to $(2 + 3) \times \mathbf{n} \times \mathbf{n} \times \mathbf{n}$ with 3 occurrences of \mathbf{n} . Note that in the second reduction path, we duplicate 3 times the reducible expression $2 \times 3 + 1$ and this is one of the reasons why the first computation is better. In general, an execution-step of a program can duplicate several routines of this program. Since executing each copy of these routines can also entail duplications of subroutines and so on, this sometimes entails an explosion of computational complexity.

Some other phenomenons occur, that will be of great importance in this thesis:

- **Erasure**: One may think from the previous example that it is more efficient to first compute the argument of a function in order to avoid duplication of sub-computations. It is sometimes false *e.g.*, when some argument does not impact the result. For instance, if \mathbf{f} is the function that takes an integer \mathbf{n} as inputs and outputs $\mathbf{n} \times (3 - 3) \times \mathbf{n}$, then, the best way to compute $\mathbf{f}(2 \times (4 - 1) - 2 \times 4 + 7)$ is to reduce $\mathbf{f}(\mathbf{n})$ first *i.e.*

- given \mathbf{n} , $\mathbf{f}(\mathbf{n}) \rightarrow_{\beta} \mathbf{n} \times (3 - 3) \times \mathbf{n} \rightarrow_{\beta} \mathbf{n} \times 0 \times \mathbf{n} \rightarrow_{\beta} 0$
- $\mathbf{f}(2 \times (4 - 1) - 2 \times 4 + 7) \rightarrow_{\beta} 0$

In this case, since the value of $\mathbf{f}(\mathbf{n})$ does not depend on \mathbf{n} , it would be absolutely useless to compute first the sub-expression $(2 \times (4 - 1) - 2 \times 4 + 7)$.

- **Created computations:** Because of higher-order functions, an execution-step may create (not only duplicate) new computable expression. For instance, let \mathbf{app}_2 be the higher-order function taking its first argument and apply it twice to its second one *e.g.*, if the 1st argument is a function $\mathbf{f} : \mathbf{Nat} \rightarrow \mathbf{Nat}$ and the 2nd one is a natural number \mathbf{n} , then \mathbf{app}_2 applies \mathbf{f} twice to \mathbf{n} . *i.e.* $\mathbf{app}_2(\mathbf{f}, \mathbf{n})$ outputs⁹ $\mathbf{f}(\mathbf{f}(\mathbf{n}))$. Then \mathbf{f} is not applied to anything in the expression $\mathbf{app}_2(\mathbf{f}, \mathbf{n})$, whereas, in the outputted expression $\mathbf{f}(\mathbf{f}(\mathbf{n}))$, \mathbf{f} occurs twice with an argument (\mathbf{n} and $\mathbf{f}(\mathbf{n})$).
- **An example of unsound computational behavior:** created computations can be a source of non-termination in the untyped case. For instance, one can define the **auto-application** $\mathbf{autoapp}$, which is a higher-order function that takes a function \mathbf{f} and applies it to itself *i.e.* $\mathbf{autoapp}(\mathbf{f})$ outputs $\mathbf{f}(\mathbf{f})$ for *any* function \mathbf{f} . If we apply $\mathbf{autoapp}$ to itself, note that $\mathbf{autoapp}(\mathbf{autoapp})$ outputs (after one execution-step) $\mathbf{autoapp}(\mathbf{autoapp})$, which outputs $\mathbf{autoapp}(\mathbf{autoapp})$, which... This computation will never stop but it is licit and meaningful in some *untyped* programming language *e.g.*, the pure (untyped) λ -calculus. In the latter case, auto-application is just encoded by the term $\Delta := \lambda x.x x$ and $\mathbf{autoapp}(\mathbf{autoapp})$ by the term $\Omega = \Delta \Delta$.
- **Order of a program** The *order* of a program is the number of its top-level inputs. Let us notice that the top function of a program can change during its execution. For instance, taking the example $\mathbf{app}_2(\mathbf{f}, \mathbf{n})$ on p. 23, the main function is \mathbf{app}_2 , but after one execution-step, it is \mathbf{f} (in the reduced expression $\mathbf{f}(\mathbf{f}(\mathbf{n}))$). This kind of phenomenon can mixed up with complex branching instructions. For instance, let \mathbf{branch}_2 be the higher-order function that uses the output of its 2nd parameter to feed the input of its first parameter. In particular, the two parameters of \mathbf{branch}_2 must be functions. Let \mathbf{sum} be the function that computes the sum of two integers and \mathbf{mult}_3 the function that computes the product of three integers. Thus, \mathbf{sum} and \mathbf{mult}_3 can be respectively be written by $x, y \mapsto x + y$ and $x, y, z \mapsto x \times y \times z$. Then $\mathbf{branch}_2(\mathbf{mult}_3, \mathbf{sum})$ outputs the *function* with four parameters that can be written $x, y, z, t \mapsto (x + y) \times z \times t$ (up to some unfoldings). Because of higher-order computations, the order n of a program t is not easy to obtain when t is not in a sufficiently reduced state (*i.e.* n is difficult to capture statically), and although, in the typed case, types usually give some information on the order n of t , determining its value is undecidable in the *untyped* λ -calculus.
- **Small-step calculi:** A way to have a more fine-grained control on duplication is given by small-step calculi and to avoid for instance the following situation. We assume that:
 - \mathbf{F} is a higher-order function and $\mathbf{F}(\mathbf{x})$ reduces into an expression where the variable \mathbf{x} occurs 45 times.
 - \mathbf{expr} is also a complex higher-order expression.

Thus, $\mathbf{F}(\mathbf{expr}) \rightarrow_{\beta} \mathbf{Expr}$, where the argument \mathbf{expr} of \mathbf{F} is duplicated 45 times in \mathbf{Expr} . But assume moreover that one of the occurrence of \mathbf{expr} in \mathbf{Expr} interacts

⁹In this example, \mathbf{app}_2 can be assigned the type $((\mathbf{Nat} \rightarrow \mathbf{Nat}) \times \mathbf{Nat}) \rightarrow \mathbf{Nat}$. But more generally, \mathbf{app}_2 can be assigned *any* type of the form $((A \rightarrow A) \times A) \rightarrow A$, see p. 27.

via higher-order computations with other parts of Expr and erases 40 copies of expr . Then, it was useless to replace the corresponding 40 occurrences of the variable x of F with expr since they are to be erased. Small-step computation provides an alternative method by allowing the substitution of x by expr to be processed occurrence by occurrence. So the computation occurs like this: $F(\text{expr})$ reduces in an expression Expr' where, for the moment, only one occurrence of x has been replaced by expr : the one that allows us to erase 40 occurrences of x in Expr . In some execution steps, we obtain a much simpler expression Expr'' in which x occurs only $45-40-1=4$ times. We then replace these 4 remaining occurrences of x by the expression expr . This example epitomizes the use of small-step *operational semantics*.

Executions paths

- An execution-step (materialized by \rightarrow_β) can cause erasure duplication and creation of some computations.
- A same expression can be computed in different ways (\rightsquigarrow reduction paths) with different efficiencies, to reach a *final state*.
- Some computations do not terminate (in the untyped case).
- The order of a functional expression is its number of top-level inputs.

Final states and how to reach them An important observation is that some computations terminate when they are handled in some ways whereas they do not in other cases. For instance, assume that expr is an arithmetic expression whose computation does not terminate (think of an imperative-style program with a bad “while” loop or of an expression using `autoapp(autoapp)` on p. 23). Let proj_2 be the function taking two integers as inputs and outputting the second one *i.e.* $\text{proj}_2(m, n) \rightarrow_\beta n$. Thus, proj_2 does use its first argument. We then consider $\text{proj}_2(\text{expr}, 2 + 3)$: if we try to execute expr while it is not in final state, the computation will never stop, by hypothesis. But if we start by unfolding \mathbf{f} , we obtain a final state in two steps:

$$\text{proj}_2(\text{expr}, 2 + 3) \rightarrow_\beta 2 + 3 \rightarrow_\beta 5$$

This is one case where starting evaluating the arguments makes the computation non-terminating!

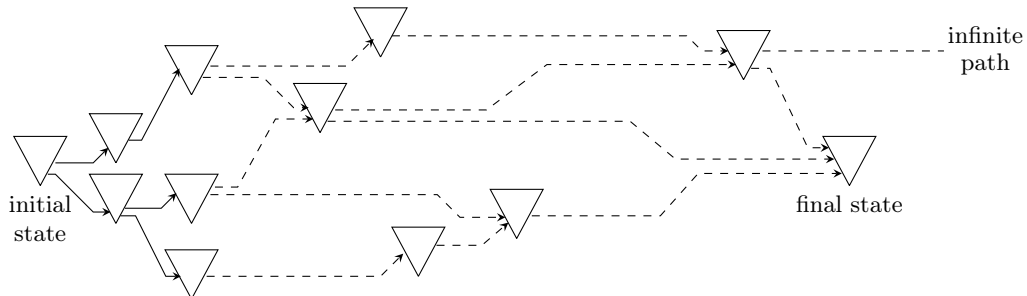


Figure 1.1: Execution Paths (Weak Normalization Case)

In the λ -calculus, a β -normal form is a program such that every computation is in its final state (the program does not contain reducible expressions). One then distinguishes between *weakly* normalizing and *strongly* normalizing λ -terms. A program t is

weakly normalizing when there is at least one reduction path from t to a final state (a β -normal form) whereas it is *strongly normalizing* when there are no infinite reduction paths starting at t . For instance, the program $\text{proj}_2(\text{expr}, 2 + 1)$ above is weakly, but not strongly, normalizing, whereas the expression $\mathbf{f}(2 \times 3 + 1)$ from p. 22 corresponds to a strongly normalizing program (it does not even contain “for” or a “while” loop). In Fig. 1.1, we have represented different execution/reduction paths, starting at an initial state/expression of a program. All the paths represented in the figure lead to a final state, except for one. Thus, the program can be executed in some ways that will make it terminate, but some other will not: this roughly corresponds to a case of weak (not strong) normalization in the λ -calculus.

Operational completeness and reduction strategies To get into the subject of reduction strategies, let us give a very high-level representation of a computable *functional* expression expr as a set of computations with nestings. A simple example of nestings is $\mathbf{f}(2 \times 3 + 1)$ on p. 22: the application of \mathbf{f} to $2 \times 3 + 1$ is the most shallow process (with $\mathbf{f} : \mathbf{n} \mapsto (2 + 3) \times \mathbf{n} \times \mathbf{n} \times \mathbf{n}$). The sums $2 + 3$ and $2 \times 3 + 1$ are nested in $\mathbf{f}(2 \times 3 + 1)$ in the expression of \mathbf{f} and in its argument respectively. The product 2×3 is nested in the sum $2 \times 3 + 1$. In Fig 1.2, the computations of a given expr are represented by the nodes and the nestings by the edges. Intuitively, these computations are more or less deeply nested in the expression. We consider the computations that do not contain sub-computations¹⁰ as *inputs* of the computation (although it is somewhat¹¹ of an oversimplification) but there is exactly *one* computation that is the closest to the output. Every sub-computation of expr has a main process and this main process is either in a final state (we label it with “T”, standing for terminated) or it is not completed – *i.e.* it can still be reduced – and we label it with “?”.

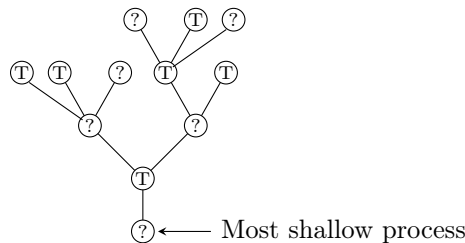


Figure 1.2: Dependencies between Computations in a Function

A **reduction strategy** consists in reducing (executing) only sub-expressions of some particular form or in some particular places *e.g.*, in λ -calculus, the *head reduction strategy* informally corresponds to keeping on reducing the most *shallow* process of the expression (the one that is the closest to the output) while it is not in terminal state (the strategy will go on forever if this is not possible).

Still in the λ -calculus, a *head normal form* is a program whose most shallow process is in terminal state whereas a β -normal form (see above) is a program whose processes are all in final state. One says that a program t is *head normalizing* if there exist a reduction path (an execution path) from t to a head normal form. Obviously, if the head reduction

¹⁰Computations containing only one operand.

¹¹In general, Fig. 1.2 is far too rudimentary to suitably represent computation dependencies in a program. Compare with the figures of Chapter 2.

strategy terminates on a program, then it is head normalizing. An equivalence actually holds: a program is head normalizing iff the head reduction strategy terminates on it. We then say that the head reduction **strategy** is **complete** for head normalization.

The converse implication of the above equivalence is difficult to prove: the head reduction consists in reducing only the most shallow node *i.e.* the node of interest, but as we saw on p. 22, each execution-step can displace, duplicate, erase or create new computations. It is sometimes clearly more efficient to reduce first the argument of a computation instead of executing this computation (recall $f(2 \times 3 + 1)$ on p. 22) and the head reduction strategy is often quite sub-optimal to reach a head normal form, but still, it ensures that a head normal form will be reached (if one is reachable from the initial state of the program).

Usually, the developer or the user of a program must not be bothered and asked to specify at each execution step which expression should be computed next, and languages feature default *deterministic* reduction strategies. It is then very useful to identify reduction strategies for given definitions of termination, so that one ensures that computation stops iff a final state is reached. It turns out that intersection type theory provides semantic proofs of the operational completeness of a strategy, instead of syntactical ones, and this is one of their main interests.



Reduction Strategies

- For a given program, some execution paths can be infinite whereas some others reach a final state in a finite number of steps.
- A reduction strategy sets priorities of reduction/execution.
- Some reduction strategies are complete w.r.t. some notions of termination.

1.3 Intersection (or not) Types in the λ -Calculus

Type systems, typing derivations The typing of a program roughly consists in a proof, called a **typing derivation**, following the structure of the instructions of the program. These proofs contain statements called typing **judgments** and the typing rules of the system describe the *licit* transitions from one judgment to another. As hinted before, the most important constructor of type theory is the arrow/implication “ \rightarrow ”: the judgment $f : A \rightarrow B$ means that f is a typed function that takes an input of type A and outputs an object of type B .

For instance, if `twice` is a function computing the double of a natural number (*i.e.* the type of `twice` is $\text{Nat} \rightarrow \text{Nat}$) and `len` computes the length of a string (*i.e.* the type of `len` is $\text{String} \rightarrow \text{Nat}$), then the program f defined taking a string s as input and outputting `twice(len(s))` should of course be typed with $\text{String} \rightarrow \text{int}$. From the syntactical point of view, this typing is checked like this: under the assumption that s is an object of type `String`, we prove that `twice(len(s))` is a well-defined object of type `Nat` (this is denoted by the judgment $s : \text{String} \vdash \text{twice}(\text{len}(s)) : \text{Nat}$: on the left-hand side of \vdash , one finds the typing assumptions). This implies that the *function* f , defined by $s \mapsto \text{twice}(\text{len}(s))$, is of type $\text{String} \rightarrow \text{Nat}$. Formally, this corresponds to this typing derivation:

$$\frac{\frac{\frac{}{\vdash \text{twice} : \text{Nat} \rightarrow \text{Nat}}{\vdash \text{twice} : \text{Nat} \rightarrow \text{Nat}} \quad \frac{\frac{}{\vdash \text{len} : \text{String} \rightarrow \text{Nat}}{\vdash \text{len} : \text{String} \rightarrow \text{Nat}} \quad \frac{}{\text{s} : \text{String} \vdash \text{s} : \text{String}}{\text{s} : \text{String} \vdash \text{s} : \text{String}}}{\text{s} : \text{String} \vdash \text{len}(\text{s}) : \text{Nat}}}{\text{s} : \text{String} \vdash \text{twice}(\text{len}(\text{s})) : \text{Nat}}}{\vdash \text{s} \mapsto \text{twice}(\text{len}(\text{s})) : \text{String} \rightarrow \text{Nat}}$$

In a simple or a higher-order type system, a derivation typing a program featuring 83 functions-calls in its source code will be a proof featuring more or less 83 typing rules/judgments and following the general structure of this source code.

When a type system ensures termination for typed programs, a lot of undecidable problems become decidable *e.g.*, one can usually determine the order of a given typed program by executing it, or decide whether some typed expressions expr_1 and expr_2 represent states of a same program or not by reducing them.

Types and Termination Revisited As recalled on p. 16, Curry proved in the 50s that typed λ -terms are terminating (they are strongly normalizing), using cut-elimination techniques, and Howard observed that this was just a particular consequence of an isomorphism between the simply typed λ -calculus and intuitionistic natural deduction. Shortly after, Landin [71,72] used the λ -calculus to construct the language Algol. Curry's type system does not have much expressive power¹² and **Higher-order types systems**, deriving from higher-order deduction systems *via* the Curry-Howard correspondence, were introduced in the 60s to extend the computational power of the programming languages while ensuring termination of the typed program (see [21], Sec. 8.3 for some historical aspects of higher-order types).

In short, higher-order type systems¹³ allow manipulating types as first-class objects (one may for instance quantify over types). This approach is embodied by Jean-Yves Girard' system \mathcal{F} [45, 46], Per Martin-Löf type theory [78, 79] and the calculus of constructions (CoC), developed by Thierry Coquand, Gérard Huet and Christine Paulin-Mohring [29–31], and gave birth to a generation of powerful proof assistants like COQ or Agda.

So, why do we need higher order typing? It stems from the observation that, intuitively, some program can be assigned several different types. For instance, the function app_2 from p. 23, satisfying $\text{app}_2(\mathbf{f}, \mathbf{x}) \rightarrow_{\beta} \mathbf{f}(\mathbf{f}(\mathbf{x}))$, can be typed with $((\text{Nat} \rightarrow \text{Nat}) \times \text{Nat}) \rightarrow \text{Nat}$, since it can take a function $\mathbf{f} : \text{Nat} \rightarrow \text{Nat}$ as first argument, a natural number $\mathbf{n} : \text{Nat}$ as second argument, and outputs $\mathbf{f}(\mathbf{f}(\mathbf{n}))$ which is also of type Nat . In contrast, app_2 cannot be typed with $(\text{String} \rightarrow \text{Nat} \times \text{Nat}) \rightarrow \text{Nat}$ because one may not apply a function \mathbf{f} of type $\text{String} \rightarrow \text{Nat}$ (outputting a number from a string) to a natural number \mathbf{n} ! But note that app_2 can be fed, for any type A , with a function \mathbf{f} of type $A \rightarrow A$ as first input and an argument \mathbf{x} of type A as second input, so that $\text{app}_2(\mathbf{f}, \mathbf{x})$ outputs $\mathbf{f}(\mathbf{f}(\mathbf{x}))$, which is of type A . In other words, \mathbf{f} can be typed with $((A \rightarrow A) \times A) \rightarrow A$ for all types A . In a higher-order type theory, one then assigns to app_2 the type $\forall A.((A \rightarrow A) \times A) \rightarrow A$, meaning that the type of app_2 may be instantiated so that app_2 can be used w.r.t. any "base type" A (*e.g.*, $A = \text{Nat}$ as in

¹²Schwichtenberg [97] proved in 1976 that a function from Nat to Nat can be encoded with a simply typed λ -term iff it is an extended polynomial *i.e.* a polynomial using some conditional operator

¹³this is not to be confused with the fact that the λ -calculus is of higher-order, as a *programming* language (*i.e.* allows passing functions as parameters of other functions)

$\text{app}_2(\mathbf{f}, \mathbf{n})$ above, or $A = \text{String}$ or $A = \text{Nat} \rightarrow \text{Nat}$). One then says that the type of app_2 is **polymorphic**. Let us give an example of use of this polymorphism:

- Let \mathbf{G} be the function that takes a function $\mathbf{f} : \text{Nat} \rightarrow \text{Nat}$ and outputs the function $\text{Nat} \rightarrow \text{Nat}$ defined by $\mathbf{n} \mapsto f(3 \times \mathbf{n}) + 1$. Thus, \mathbf{G} is a higher-order function of type $(\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \rightarrow \text{Nat})$. For instance, if \mathbf{f} is $\mathbf{n} \mapsto \mathbf{n}^2$, then $\mathbf{G}(\mathbf{f})$ is the function $\mathbf{n} \mapsto 9 \times \mathbf{n}^2 + 1$.
- Then, if \mathbf{f} is a function $\text{Nat} \rightarrow \text{Nat}$, $\text{app}_2(\mathbf{G}, \mathbf{f})$ outputs $\mathbf{G}(\mathbf{G}(\mathbf{f}))$ *i.e.* (up to some execution-steps), the function $\mathbf{n} \mapsto \mathbf{f}(9 \times \mathbf{n}) + 2$ which is also of type $\text{Nat} \rightarrow \text{Nat}$. In this case, app_2 was instantiated with the type $A = \text{Nat} \rightarrow \text{Nat}$ and not $A = \text{Nat}$.

Thus, $\text{app}_2(\text{app}_2(\mathbf{G}, \mathbf{f}), \mathbf{n})$ outputs (up to some execution-steps) the natural number $\mathbf{f}(9 \times \mathbf{f}(9 \times \mathbf{n}) + 18) + 2$. In this expression, app_2 occurs twice, but with different types: the type $((\text{Nat} \rightarrow \text{Nat}) \times \text{Nat}) \rightarrow \text{Nat}$ (case $A = \text{Nat}$) in $\text{app}_2(\dots, \mathbf{n})$ and the type $((\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \rightarrow \text{Nat})) \times (\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \rightarrow \text{Nat})$ (case $A = \text{Nat} \rightarrow \text{Nat}$) in $\text{app}_2(\mathbf{G}, \mathbf{f})$. Because of that, in the case of simple typing, the expression $\text{app}_2(\text{app}_2(\mathbf{G}, \mathbf{f}), \mathbf{n})$ would not be typable (one should have chosen the particular type of app_2).

Subject Reduction and Invariance of Typing by Execution An important property, usually verified by simple and higher-order type systems, as the one of the Calculus of Constructions for instance, is **subject reduction**, which means that a type assignment that is valid for the initial state of a program P remains valid for all the states of P during its execution.

For instance, if \mathbf{f} is the function $\mathbf{n} \mapsto 3 \times \mathbf{n} + 1$, the program \mathbf{g} from Nat to Nat , that takes a natural number \mathbf{n} as input and outputs $\mathbf{f}(\mathbf{n} + 2)$, can be executed in this manner:

$$\mathbf{g}(\mathbf{n}) \rightarrow_{\beta} \mathbf{f}(\mathbf{n} + 2) \rightarrow_{\beta} 3 \times (\mathbf{n} + 2) + 1 \rightarrow_{\beta} \dots \rightarrow_{\beta} 3 \times \mathbf{n} + 7$$

The program \mathbf{g} passes from its initial state $\mathbf{n} \mapsto \mathbf{f}(\mathbf{n} + 2)$ to its final state $\mathbf{n} \mapsto 3 \times \mathbf{n} + 7$ after some execution-steps. Then subject reduction simply ensures that the final expression $\mathbf{n} \mapsto 3 \times \mathbf{n} + 7$ is also a function from Nat to Nat , as the initial one is. This example is somewhat non-plussing because of its straightforwardness, but subject reduction is perhaps the only requirement demanded to every type system. Note that typing is specified by a set of *syntactical*¹⁴ rules, that can be applied to a given expression of a program. Usually, typing features an arrow constructor and $f : A \rightarrow B$ means that f is a function of type $A \rightarrow B$ *i.e.* a function that takes an output of type A and outputs an object of type B . But beyond that, any constructor (*e.g.*, product, coproduct, intersection, union, higher-order, dependent types, equality types, etc) and any typing rule can be specified without particular restriction, except that if subject reduction¹⁵ is not satisfied, the type system will not be meaningful since typing is not *invariant* under execution as it should be. When a type system satisfies subject reduction, it is said to be **(dynamically) sound**.

¹⁴ In contrast to the *realizability* approach pioneered by Kleene [67], which is not syntactical but more semantical.

¹⁵ *i.e.* if an expression expr of a given function is typable but not another that is obtained from expr after some execution-steps.



Type systems

- Typing relies on syntactic rules.
- The arrow \rightarrow represents functionality from one type to another.
- The typing of a program is statically checked for an initial state of this prog.
- Subject reduction holds when typing is invariant under execution.

Enriching Types with Intersection The λ -calculus is an amazingly rich playground to define typed programming languages, since its untyped version is Turing-complete whereas, when it is endowed with simple or higher-order type systems, it provides safe languages featuring only strongly normalizing programs. One of the purpose of type theory in the λ -calculus is to provide more and more descriptors to interpret pure λ -terms as effective programs with a computational content. An alternative extension of Curry’s original type system, differing from the higher-order approach, was proposed by Coppo and Dezani [27, 28, 77], who introduced the so-called **intersection type systems**.

Intersection types also allow polymorphism, but in a finite form *i.e.* there is no universal quantification on type as $\mathbf{app}_2 : \forall A. ((A \rightarrow A) \times A) \rightarrow A$, but one can specify a finite list of types to a same function *e.g.*, , if \mathbf{NN} shortens $\mathbf{Nat} \rightarrow \mathbf{Nat}$, in the example on p. 23, one can specify the type of \mathbf{app}_2 in $\mathbf{app}_2(\mathbf{app}_2(\mathbf{G}, \mathbf{f}), \mathbf{n})$ by $((\mathbf{Nat} \rightarrow \mathbf{Nat}) \rightarrow \mathbf{Nat}) \wedge (((\mathbf{NN} \rightarrow \mathbf{NN}) \wedge \mathbf{NN}) \rightarrow \mathbf{NN})$, which means that \mathbf{app}_2 can be both used w.r.t. type \mathbf{Nat} and type $\mathbf{Nat} \rightarrow \mathbf{Nat}$ (but no more, with this assignment). Since only a finite number of types can be specified, one could think that intersection types have a limited expressivity compared to that of higher-order types, but it is not true in some respects: with intersection types, type assignment is completely unconstrained *e.g.*, in higher-order, \mathbf{app}_2 is always instantiated with types of the same form $((A \rightarrow A) \times A) \times A$, but with intersection types, one can assign to a same program types of this form and of the form \mathbf{String} or $\mathbf{Nat} \rightarrow \mathbf{String}$ (the syntax of the assigned types is not necessarily the same). It turns out that intersection types usually type more pure λ -terms than the original calculus of constructions does (see Fig. 1.3).

- From the semantical point of view, intersection type systems provide a *characterization* of termination (and not only a *guarantee* of termination, as in the simple and higher-order cases).
- They are more modular regarding what is meant by termination *e.g.*, head, strong or weak normalization (in the λ -calculus) can be captured.
- From the **dynamical** point of view, they are usually **complete**: they often (not always) satisfy a **subject expansion property**, meaning that, if an expression \mathbf{expr} representing a state of a program gives \mathbf{expr}' after some execution-steps, and a typing is valid for \mathbf{expr}' , then this typing is also valid for the initial state represented by \mathbf{expr} . This is some sort of converse implication to that of subject reduction and it really relies on the “unconstrained” nature of intersection types alluded to above (this is addressed in detail in Sec. 3.3.2).

Subject expansion (along with subject reduction) implies that intersection type systems can be used to build *denotational models* of the λ -calculus. Moreover, it is not satisfied by simple and higher-order type systems. That is why these systems do not *characterize* terminating programs (whatever the sense given to “termination” is) whereas, in an intersection type system, we have equivalences of the form:

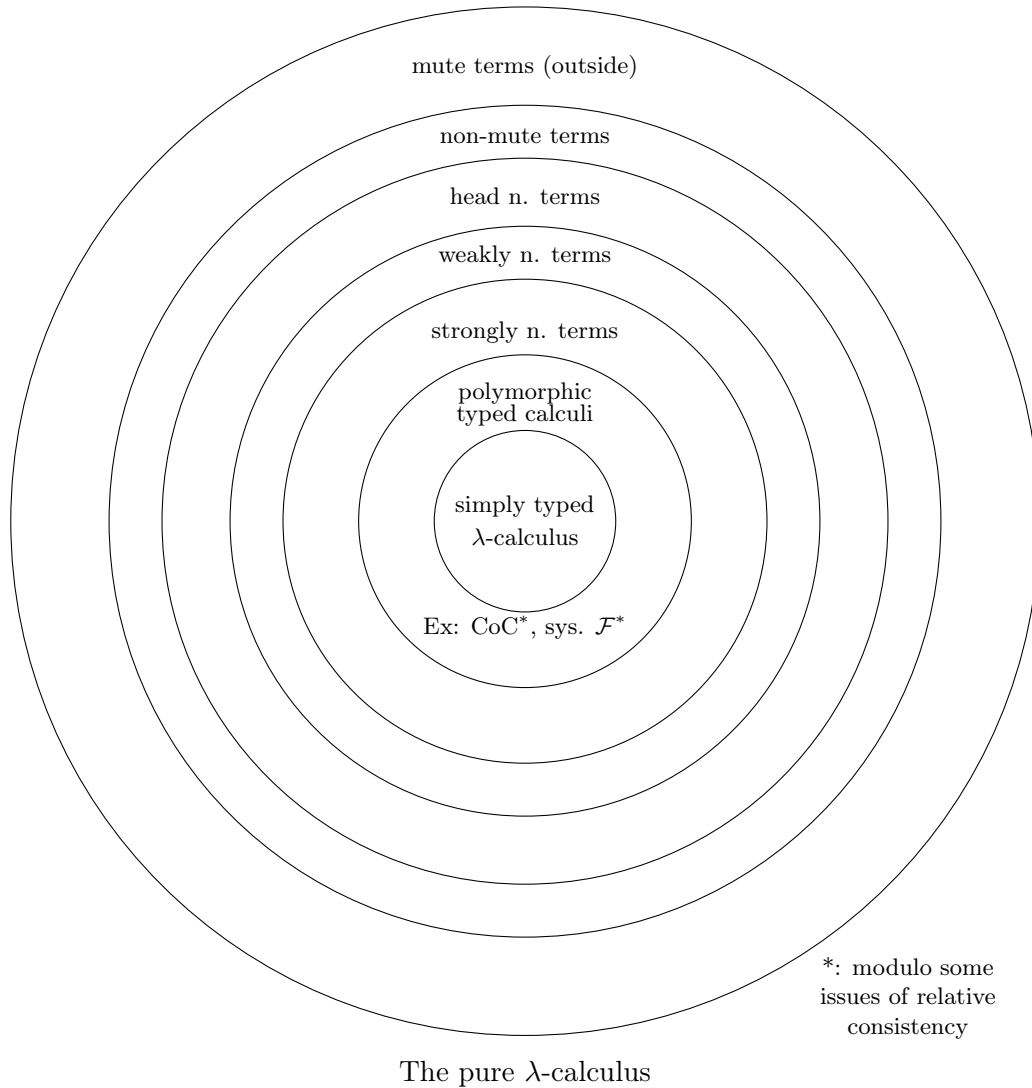


Figure 1.3: The Expressive Power of Types (λ -Calculus)

A program t is typable iff t terminates (\exists at least one red. path from t to a final state)

The two predicates “ t is typable” and “ t terminates” are usually undecidable, but in practice, this kind of equivalence is often proved by actually proving an equivalence between 3 predicates:

A program t is typable iff t terminates (there is at least one red. path. . .)
iff some reduction strategy terminates on t

Thus, as a by-product of these characterizations, intersection types provide a proof that some reduction strategy is *complete* for some given notion of termination *i.e.* intersection types prove operational properties that are independent from the notion of type. This aspect is very important in this thesis. We discuss it again in Sec. 3.3.1 and it is embodied by Theorems 7.1 and 10.4, in which we both characterize a form of termination and prove the completeness of a given reduction strategy for this form of termination.

If a function f is typed with $(A_1 \wedge A_2 \wedge A_3) \rightarrow B$, then it can be applied to an argument x that has been typed with A_1 , A_2 and A_3 *at the same time*. In simple or higher-order type systems, the argument of a function is typed exactly once. In an intersection type systems, it can be typed 3, 4, 5... or even 0 times. Thus, in this setting, since a sub-expression of a typed program t can be typed as many times as needed/desired (starting from 0), if t contains 83 instructions, the size of a derivation typing t can vary from 1 to far more than 83 judgments.

Types as Semantical Descriptors Types provide a guarantee of termination, but also many other semantical information, that are more than just “this function takes two integers as inputs and outputs an array of strings”. For instance:

- Some higher-order type systems (for instance, the one of Coq) feature *equality types* *i.e.* types that are able to assert that *e.g.*, two integer expressions have the same value. Equality types bring a lot to proof-checkers and can be used to give a certificate that two programs have the same observational behavior (*i.e.* that they meet the same specification).
- Moreover, typability brings more news than mere termination. For instance, a function $\mathbb{N} \rightarrow \mathbb{N}$ may be implemented/typed in system \mathcal{F} iff it is provably total¹⁶ in PA_2 , the Second Order Peano Arithmetic [45] (see Chapter 15 of [49]).

On the other hand, when a language both features untyped and typed programs (as the λ -calculus does), intersection type systems, besides their ability to characterize termination, can help to separate programs *i.e.* if given two expressions expr_1 and expr_2 , there is a type A than can be assigned to expr_1 but not to expr_2 , then expr_1 and expr_2 cannot represent two different states of a same program. This works because typing is invariant under execution (backward and forward) with intersection types, by subject reduction and expansion.

For instance, usually, the types of a program give indications on the value of the order of a typed program t because a function must be typed with an arrow type. However, by lack of subject expansion in simple and higher-order type systems, those usually only provide an upper bound¹⁷ to the order. On the other hand, intersection type systems often satisfy subject expansion and provide the precise value of the order of any typed term. One say that such systems are **order-discriminating**, since, if two programs do not have the same order, then, there is a type that can be assigned to one but not to the other, and the order of terms can be determined statically, without execution-step.

A last remark on higher-order and most intersection types is that they provide *qualitative* information about programs (termination) but no *quantitative* information (*e.g.*, in how many execution-steps a final state can be reached from a given typed program). The next section addresses quantitative issues.

Non-Idempotent Intersection Types Subject reduction is a guarantee that every typing of a state of program represented by an expression expr is also valid for any state of the same program expr' after some execution-steps. For instance, if there is a typing derivation Π concluding with $\Gamma \vdash \text{expr} : B$ and $\text{expr} \rightarrow_{\beta} \dots \rightarrow_{\beta} \text{expr}'$, then there is

¹⁶We say that f is *provably terminal* in PA_2 if f is terminal (*i.e.* for all $n \in \mathbb{N}$, $f(n)$ is defined) and that there is a proof in PA_2 that f is actually terminal

¹⁷Type system can give a precise account of the order of a typable program when it has been *sufficiently reduced*. Typing can go backward (w.r.t. execution) only when subject expansion is satisfied.

also a typing derivation Π' concluding with $\Gamma \vdash \mathbf{expr}' : B$. The derivation Π' is roughly obtained from Π by undergoing the same processes (duplication, erasure, etc) as \mathbf{expr} undergoes to give \mathbf{expr}' , since typing derivations follows the structure of the program (we refer to Fig. 3.1 for a more formal example).

Thus, it is possible that a derivation typing the initial state of a program is simpler than the corresponding derivations, obtained by subject reduction, in later stages of the execution. This is the reason why proving that typing ensures termination¹⁸ in simple and higher-order type systems and in most intersection type systems is technically difficult (see Sec. 4.3).

Gardner [43] and de Carvalho [22] independently proposed an intersection type system in which the termination of typed programs was straightforward to establish. This was due to the disallowing the duplication inside a typing derivation when it is executed. This approach takes its inspiration from Girard's Linear Logic [47] in which structural rules (weakening, contraction, co-contraction *i.e.* duplication) are handled by means of explicit modalities. Intersection types *à la* Gardner/de Carvalho are said to be **non-idempotent** because A and $A \wedge A$ are not the same, since duplication is not licit.

Let us illustrate the difference between the non-idempotent intersection approach on one hand, and higher-order or idempotent intersection types on the other. Assume that:

- $\mathbf{f} : A \rightarrow B$ and $\mathbf{x} : A$ (so that $\mathbf{f}(\mathbf{x}) : B$) *i.e.* the derivation typing $\mathbf{f}(\mathbf{x})$ concludes with a rule of this form (we omit the possible typing assumptions):

$$\frac{\mathbf{f} : A \rightarrow B \quad \Pi_{\mathbf{x}} \triangleright \mathbf{x} : A}{\mathbf{f}(\mathbf{x}) : B}$$

where $\Pi_{\mathbf{x}}$ is the derivation certifying that \mathbf{x} is typable with A .

- After one execution-step, \mathbf{f} duplicates its argument \mathbf{x} 28 times *i.e.* from $\mathbf{f}(\mathbf{x})$, we obtain an expression \mathbf{expr} in which \mathbf{x} occurs 28 times.

In the higher-order or idempotent intersection types settings, during reduction, the certificate that \mathbf{x} can be assigned the type A will be duplicated 28 times *i.e.* the derivation typing \mathbf{expr} features 28 copies of $\Pi_{\mathbf{x}}$. In the non-idempotent setting, duplication is disallowed and in the same situation, since the argument \mathbf{x} is copied 28 times, then the initial derivation typing $\mathbf{f}(\mathbf{x})$ must at least feature 28 certificates that \mathbf{x} is typable with A and the type of \mathbf{f} must specify that \mathbf{f} uses its argument 28 times *i.e.* the derivation typing $\mathbf{f}(\mathbf{x})$ must be of the form:

$$\frac{\mathbf{f} : (A \wedge \dots \wedge A) \rightarrow B \quad \Pi_{\mathbf{x}} \triangleright \mathbf{x} : A \dots \dots \Pi_{\mathbf{x}} \triangleright \mathbf{x} : A}{\mathbf{f}(\mathbf{x}) : B}$$

Thus, when $\mathbf{f}(\mathbf{x})$ is reduced to \mathbf{expr} , the 28 certificates are dispatched with the 28 copies of \mathbf{x} : the argument \mathbf{x} is duplicated, but not the 28 typing certificates stating that \mathbf{x} is typable with A . Of course, this will work only if \mathbf{x} cannot be duplicated anymore during execution: if \mathbf{x} is duplicated later in some execution path, then the initial typing of $\mathbf{f}(x)$ must take that into account (we refer to Fig. 3.2).

Intuitively, one sees that it is far more difficult to produce a typing certificate in a non-idempotent type system than in an idempotent one, because in the former case, the

¹⁸Not all programs terminate in the untyped case, because notably of created computation. So why should typed programs be safe from non-termination?

typing must reflect the exact use of sub-processes of the typed programs, whereas in the latter, this aspect is simply handled by the duplication/erasure etc of typing certificates. For this reason, the derivations in a non-idempotent intersection system can be huge, even when the source code of the typed program is small.

So, if terms are so complicated to type with non-idempotent intersection types, what do we gain with this approach? Straightforward proofs of termination: the absence of duplication entails that the size of non-idempotent derivations (*i.e.* the number of judgments that they contain) decreases along with execution. Thus, as expected, a typed program is terminating, since there are no infinite decreasing sequence of natural numbers. Moreover, it is not more difficult to type final states in the non-idempotent case than in the idempotent one. Thus, the characterizations provided by intersection type (see p. 30) are a lot easier to prove¹⁹ in the non-idempotent case, and in particular, the fact that a reduction strategy is complete for a given definition of termination.

As a by-product, non-idempotent types give new semantic information about programs: the size of a non-idempotent derivation typing a program t gives an upper bound to the number of steps needed by a reduction strategy to reach a final state from t . See for instance Theorems 7.1, 7.2 and 8.1.



Intersection type systems

- Characterize various forms of termination, usually satisfy subject expansion.
- Elegantly prove the completeness of many strategies.
- Non-idempotent intersection:
 - duplication disallowed, typing derivations decrease along with execution
 - programs are difficult to type, but the type systems are easy to study.
 - brings quantitative semantic information on execution strategies.

Classical Logic and Computation The isomorphism noticed by Curry and Howard was between the λ -calculus on one side and *intuitionistic* natural deduction on the other. Indeed, intuitionistic logic was designed by Brouwer and Heyting to be *constructive*, which was confirmed by Gentzen cut-elimination technique: for instance, from a proof, in intuitionistic logic of an existential proposition $\exists x \mathcal{P}(x)$, one can extract/compute an object x_0 (called an *existential witness* of \mathcal{P}) satisfying $\mathcal{P}(x_0)$ whereas it is impossible in classical logic. The non-constructivism of classical logic comes from the *law of excluded middle* (*i.e.* either a proposition or its negation is true).

On the other hand, some programming languages feature **control operators** *i.e.* operators allowing the programmer to manipulate the *control flow*, also called the *continuation*, and to pass it as the argument of a function. Such a control operator is **call-cc** (“call-with-current-continuation”) in the language Scheme, that allows backtracking *e.g.*, going back to some previous point of the execution if an undesired value is outputted and then running the program on another execution path. Griffin [53] typed **call-cc** with Peirce’s law, that is well-known to extend intuitionistic logic into classical logic as the law of excluded middle does (see the presentation of Part II, p. 133 and Sec. 6.1.1 for more details). This gave way to defining more and more computational interpretations of classical logic, which was hitherto thought impossible.

From a computation perspective, in an intuitionistic calculus (*e.g.*, the typed λ -calculus), a final state of a program typed **Nat** is an expression of the form 2, 5, 35, 101, by opposition to 3-1, 3+2, 7×5 and $1 + 200 \div 2$ (*cf.* p. 21). In a classical setting, one

¹⁹which is independent from the difficult of typing a particular term.

will also have final states inhabiting the type `Nat` meaning *e.g.*, “My value is 3 or 5” or “My value is 2 or 10 or 42” (by a theorem known as Herbrand’s witnesses). Through this extension of Curry-Howard correspondence to classical logic, one may identify the possibility to manipulate continuations as first-class objects in programs with that of taking proofs or counter-proofs as *e.g.*, existential witnesses in logic.

Part II is dedicated to a computational interpretation of classical natural deduction, called the λ_μ -calculus, which extends the λ -calculus, and also exists both in a typed and in an untyped version. The notions of head, weak and strong normalization described on p. 22 and 25 also generalize. We extend the tools of non-idempotent intersection type theory to the λ_μ -calculus. This leads us to consider non-idempotent union types, which are so to say the *duals* of the non-idempotent intersection types in classical logic. This allows us to characterize head and strong normalization and to use types to provide upper bounds to the length of some execution paths in the λ_μ -calculus.

We also define a *small-step operational semantics* for the λ_μ -calculus, that we call the $\lambda_{\mu x}$ -calculus. As we saw on p. 23, this allows to have a more precise control on reduction paths and resource-consumption. The small-step paradigm needs to be carefully adapted in order to apply to the classical setting. We then endow the $\lambda_{\mu x}$ -calculus with a non-idempotent type system extending that we have defined for the λ_μ , and we finally obtain a characterization of strong normalization for this small-step operation semantics.

Contribution 1 (Chapters 7 and 8)

- We endow the λ_μ -calculus, a computational interpretation of classical natural deduction, with non-idempotent intersection and *union* types.
- This allows us to characterize head and strong normalization in the λ_μ -calculus, and in one “small-step” version of the λ_μ .
- The typings bring quantitative information on the length of some exec. paths.

1.4 Infinitary Computation, Coinduction and Productivity

Böhm trees as an infinitary semantics for programs An important part of this thesis is dedicated to infinitary computations and typing. Infinitary computation is not necessarily meaningless (as the program with the “while” loop on p. 17 that will never print a message): for instance, a program that prints all the prime numbers one after the other never stops, but regularly produces something. Such a non-terminating program is meaningful. Some contributions of this thesis consist in assigning infinitary types to obtain semantical information from programs that do not terminate and, in some case, to characterize infinitary behaviors that are **productive**.

First, there are a lot of ways to represent the (possibly) infinitary behaviors of programs. The most canonical is perhaps the **Böhm trees**: the Böhm tree of a program t corresponds to all the sub-processes in terminal state that one can make out of t , with the additional convention that any sub-program of t whose head process cannot be stabilized (*i.e.* remains in the state ?) is replaced by the unique node \perp , representing meaningless/unproductive computation.

Böhm trees can be infinite because of duplication and creation *i.e.* a program can (asymptotically) output infinitely many terminal processes. This is the case of the program outputting all the prime numbers one after another: each prime number can be seen as a terminated sub-process. One may then formalize the notion of “program that does not terminate but is nevertheless productive” with the **hereditary head**

normalizing (HHN) λ -terms *i.e.* the programs that may have an infinite Böhm tree BT, but such that BT does not contain the symbol \perp . Thus, hereditary head normalizing λ -terms correspond to programs that may not terminate but that do not contain any²⁰ meaningless/unproductive sub-process and they are suitable for representing streams (see p. 15). Such programs run indefinitely, but they will keep on producing more and more terminated sub-computations (from a HHN term t , one can obtain a tree that has only “T” symbols under *any fixed depth*).

In this thesis, we answer to **Klop’s problem**, which is finding a type system that characterizes hereditary head normalizing terms.

Infinitary proofs and semantic unsoundness Tatsuta [101] proved that Klop’s problem does not have a solution with a *finite* type system, so our approach is to find an *infinite* type system characterizing the hereditary head normalizing programs.

A first observation is that considering infinitary formulas or proof systems gives birth to semantically unsound objects *i.e.* objects that cannot have a reasonable meaning. Note first that $A \rightarrow A$ (where A is a given formula) is intuitively true (a proposition implies itself). We can use this observation to prove A (for any formula A) if one allows proofs of infinite size:

$$\frac{\frac{\frac{\frac{\frac{\frac{\dots}{\vdash A}}{\vdash A \rightarrow A}}{\vdash A \rightarrow A}}{\vdash A \rightarrow A}}{\vdash A \rightarrow A}}{\vdash A \rightarrow A}}{\vdash A \rightarrow A}}{\vdash A}$$

This infinite proof corresponds to the following infinite question and answer game: “— Why A is true? — Because $A \rightarrow A$ and A are true. — But why is this latter occurrence of A true? — Because $A \rightarrow A$ and A are true. — But why is this latter occurrence of A true? — Because $A \rightarrow A$ and A are true. But...”

Interestingly, the mere fact of allowing infinite formulas (while considering proofs containing finitely many steps) also entails unsoundness. This may be illustrated as follows: let A be *any* formula. We then define the infinite formula $R_A := (((\dots) \rightarrow A) \rightarrow A) \rightarrow A$ as the implication whose target is A and whose source is an implication, whose target is A and whose source is an implication whose target is A and... Thus, R_A is an implication whose target is A and whose source is R_A itself *i.e.* $R_A = R_A \rightarrow A$. Since a formula proves itself, the following proof using R_A is syntactically correct:

$$\frac{\frac{\frac{\overline{R_A \vdash R_A \text{ i.e. } R_A \rightarrow A}}{R_A \vdash A}}{\vdash R_A \rightarrow A \text{ i.e. } R_A}}{\vdash A} \quad \frac{\frac{\overline{R_A \vdash R_A}}{R_A \vdash A}}{\vdash R_A}}{\vdash A}$$

As announced, just with the infinite formula R_A , we can prove any given formula A . Interestingly enough, the above unsound derivation corresponds to a derivation typing the λ -term Ω , which is the auto-application applied to itself (see `autoapp(autoapp)` p. 23), that keeps on looping without producing anything.

²⁰or at least, not any meaningless sub-process that cannot be erased in the sens of p. 22.

Retrieving soundness The examples above show that, naively infinitary type systems cannot work and provide certificates of productivity. However, infinite type systems seem more prone to describe precisely programs that have an infinitary behavior (here, the HHN terms) than finite type systems. Thankfully, there is a way to check semantic soundness: namely, by means of a criterion that we call **approximability**. This allows us to define a class of *sound* and meaningful infinitary proofs.

Intuitively, an infinite proof/typing derivation is approximable when it is obtained by superposing infinitely many *finite* proofs, growing over and over, as in Fig. 1.4: the outer triangle represents an infinite proof/typing derivation Π and the inner polygons represent finite proofs that “fit” in Π . The idea is that we know that finite proof systems are semantically sound, so that an approximable derivation may be infinite, but it is asymptotically obtained from finite/sound proofs.

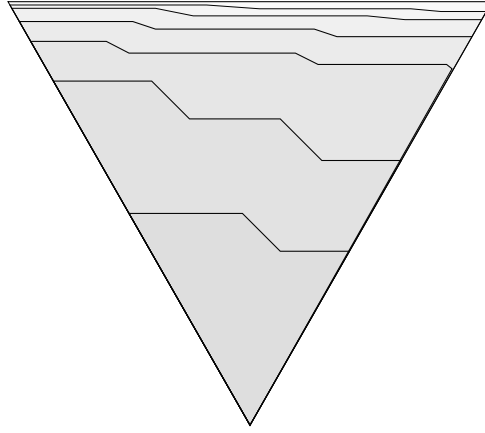


Figure 1.4: An Approximable Derivation as an Infinite Superposition

Just to give a more precise idea of what is meant by superposition, the derivation Π_2 below can be superposed upon Π_1 (we omit the typing assumption and ignore the possible premises of the top judgments):

$$\Pi_1 = \frac{\mathbf{f} : (A \wedge B) \rightarrow A \rightarrow C \quad \mathbf{x} : A \quad \mathbf{x} : B}{\mathbf{f}(\mathbf{x}) : A \rightarrow C}$$


$$\Pi_2 = \frac{\mathbf{f} : (A \wedge B \wedge (B \rightarrow C)) \rightarrow (A \wedge C) \rightarrow D \quad \mathbf{x} : A \quad \mathbf{x} : B \quad \mathbf{x} : B \rightarrow D}{\mathbf{f}(\mathbf{x}) : (A \wedge C) \rightarrow C}$$

Indeed, Π_1 is obtained from Π_2 by removing the symbols colored in red:

$$\Pi_2 = \frac{\mathbf{f} : (A \wedge B \wedge (B \rightarrow C)) \rightarrow (A \wedge C) \rightarrow D \quad \mathbf{x} : A \quad \mathbf{x} : B \quad \mathbf{x} : B \rightarrow D}{\mathbf{f}(\mathbf{x}) : (A \wedge C) \rightarrow C}$$

Infinite superposition strongly hints at the presence of complete lattices and complete partial orders, which is formalized in Chapter 10.


With this concept of approximation, we retrieve the main tools of finite intersection type theory (in an infinitary setting): subject reduction, subject expansion, typing of normal forms (terminal states). Actually, hereditary head normalization can be related to an infinitary reduction strategy and we prove that it is complete for a form of infinitary weak normalization (see p. 25):

 **Contribution 2 (Chapter 10)**

- Answer to **Klop’s problem**: characterization of the hereditary head normalizing λ -terms by means of an infinite intersection type system. For that:
 - Definition of a validity criterion (approximability) to discard the semantically unsound infinite proofs.
 - Introduction of a new intersection type system, called sys. **S** (sequential inter.).

Unsound semantics We are also interested in the typing of a given program when one consider infinite types *without* approximability. It is very easy to type *every* program by using techniques inspired from the unsound proofs of Sec. 1.4, involving a certain infinite type R satisfying $R = R \rightarrow R$. But this type R does not bring any information. We actually prove that we can type any program t with infinite terms in a non-trivial way *i.e.* with types that bring information on t . In particular, we show that infinite types enable to statically capture the order (p. 23) of any λ -term (whereas the known systems only capture the order of some sub-classes of λ -terms).

An interesting aspect of intersection types is that the description of the set of typable programs deeply relies on the typing of “sub-processes in terminal state” (p. 25). But working with infinite types forces us to consider very unstable programs, the so-called **mute terms**, that do not contain any process that can be stabilized. This is the most challenging aspect of the techniques that we develop in this thesis.

 **Contribution 3 (Chapter 12)**

- Proving that *every* λ -term is typable in a *non-trivial* way, by means of infinite intersection types.
- The order of *any* λ -terms can be extracted from these non-trivial typings.
- Typing without productivity/stability.

The problem of tracking and sequential intersection Throughout this thesis, we discuss a somewhat aesthetic feature of some type systems, namely **syntax-direction**, that becomes involved with fundamental issues when infinitary typing is considered. Roughly speaking, a type system is syntax-directed when the typing rules follow narrowly the structure of the programs. This is of course desirable, should it be only to simplify the characterization proofs of the type systems. Most of the first intersection type systems were not syntax-directed for numerous reasons. One is that syntax-direction is ensured by disallowing some very natural rules as “ t is of type $A \wedge B$ implies that t is of type A ”.

In the non-idempotent case, a practical consequence of syntax-direction is that it strongly suggests defining intersection types as *multisets* (one write respectively $[A, B]$, $[A, B, A]$ instead of $A \wedge B$ and $A \wedge B \wedge A$, with $[A, B] \neq [A, B, A] = [A, A, B]$). Intersection is then handled by means of the multiset sum *e.g.*, one writes $[A, B, A] = [A, B] + [A]$. The use of multisets is also desirable, in that it identifies the types that can be assigned to a term t to the interpretation of t in the relation model of Bucciarelli, Ehrhard and Manzonetto [17]

However, we prove that approximability (contribution 2, p. 36) cannot be formulated with multiset intersection. This leads us to introduce system **S**, in which an intersection type is a family of types indexed by integers, which is called **sequence type**. Concretely, we annotate types in intersections with pairwise distinct integers values called **tracks**

e.g., $(2 \cdot A, 3 \cdot B, 7 \cdot A)$ is a sequence type that contains two occurrences of A and one of B . Sequence types come along with a disjoint union operator \uplus *e.g.*, $(2 \cdot A, 3 \cdot B) \uplus (7 \cdot A) = (2 \cdot A, 3 \cdot B, 7 \cdot A)$ whereas $(7 \cdot A, 3 \cdot B) \uplus (7 \cdot A)$ is not defined since the track 7 occurs twice. A crucial aspect of sequential intersection is that it allows **tracking**: in $(2 \cdot A, 3 \cdot B, 7 \cdot A) = (2 \cdot A, 3 \cdot B) \uplus (7 \cdot A)$, we have the “pointers” 2 and 7 to distinguish the two occurrences of A and we can assert that the occurrence of A annotated with 2 in the left-hand side of the equality comes from the sequence type $(2 \cdot A, 3 \cdot B)$ on the right-hand side (and not from $(7 \cdot A)$). In contrast, with multisets $[A, B] + [A] = [A, A, B]$, but, in this equality, we have no way to relate one occurrence of A in $[A, A, B]$ to $[A, B]$ rather than $[A]$ and *vice versa*. For this reason, we say that sequential intersection is **rigid** whereas multiset intersection is not.

Interestingly, without approximability, finding the non-trivial infinite typings of a given term cannot be achieved while working with multisets but only with sequential intersection. Sequential intersection is far more basic than multiset-intersection and system \mathbf{S} is not endowed with any permutation rule that would enable us to rewrite sequences in different orders. Thus, one may wonder if multiset constructions are more rich and have a wider scope than sequential constructions, especially in the infinite case. We prove that it is actually not the case:



Contribution 4 (Chapter 13)

Despite the fact that sequential intersection is more constraining than multiset intersection, it does not cause any loss of expressivity, both from a static and a dynamical perspective.

1.5 Main Contributions (Technical Summary)

To summarize, in this thesis, we distinguish four main groups of contributions:

1. In the λ_μ -calculus (a computational interpretation of **classical natural deduction**):
 - Chapters 7, Theorems 7.1 and 7.2: two type-theoretical characterizations of head normalization and strong normalization respectively, by introducing two non-idempotent intersection and union type systems (systems $\mathcal{H}_{\lambda_\mu}$ and $\mathcal{S}_{\lambda_\mu}$), naturally providing upper bounds on the length of interesting normalizing reduction sequences.
 - Chapter 8, Theorem 8.1: a small-step operational semantics, denoted $\lambda_{\mu\tau}$, extending that of the λ_μ -calculus, along with an extension $\mathcal{S}_{\lambda_{\mu\tau}}$ of system $\mathcal{S}_{\lambda_\mu}$ characterizing strong normalization in $\lambda_{\mu\tau}$.
2. Chapter 10, Theorem 10.4: a positive answer to **Klop’s Problem**, which is finding whether the set of the so-called **hereditary head normalizing terms** can be characterized in a type system. A term t is hereditary head normalizing when its Böhm tree²¹ does not contain the symbol \perp . Tatsuta [101] proved that this was not possible with *inductive* (meaning finite) type systems.

²¹Böhm trees provide a semantics for the λ -calculus by means of possibly infinite trees and the symbol \perp represents a form of meaningless computation. Thus, a term is hereditary head normalizing when it has a possibly infinitary semantics that does not contain any meaningless sub-process.

- We answer positively to Klop’s Problem by introducing a *coinductive* (meaning infinitary) type system, namely system **S**, that we endow with a *validity* condition to discard some meaningless proofs.
 - The main features of system **S** is that it is **relevant**²² and intersection has a non-idempotent flavor and is represented by families of types indexed by integers (instead of lists, sets or multisets of types as in the usual intersection type systems). Thus, **intersection** is said to be **sequential** in system **S**. It is necessary to consider sequential intersection because the validity criterion discarding unsound derivation cannot be defined when intersection is represented by multisets.
 - Along with this contribution, system **S** also provides a positive answer to **TLCA Problem #20** in the coinductive case, whereas Tatsuta [102] also proved that the answer is negative in the inductive case. This consists in characterizing the set of **hereditary permutations**²³ with types. By lack of time, this contribution could only be sketched in this document (see Appendix A.7 and Theorem A.1).
3. In Chapter 12, we prove that the main intersection type systems resorting to a *coinductive* grammar are able to type every λ -term. This is very easy for irrelevant type systems but very difficult for the relevant ones.
- Theorem 12.1: we prove that every term is typable in the main *relevant* coinductive intersection type systems. Interestingly, the proof of this fact cannot be formulated *e.g.*, with multiset intersection. For this reason, we work with system **S**.
 - The derivable judgments of system \mathcal{R} , featuring coinductive multiset intersection, are the points of a relational model \mathcal{M} of the λ -calculus. As a consequence of the typability of every term in \mathcal{R} , no term has an empty interpretation in \mathcal{M} .
 - Theorem 12.2: we prove that some semantical information on λ -terms can be extracted from \mathcal{R} : for every term t , there is non-trivial coinductive typing capturing the *order* of t *i.e.*, roughly speaking, the number of inputs of the term t .
4. In Chapter 13, we prove that system **S** has full expressive power over system \mathcal{R} :
- Theorem 13.1: we prove that every \mathcal{R} -derivation (built by means of multiset constructions) is the **collapse** of a **S**-derivation (built by means of sequential construction). The collapse just consists in transforming sequences (in the above sense) in multisets and we prove that this collapse is surjective.
 - As a consequence, the points of the model \mathcal{M} can be studied through system **S** without loss of generality.
 - Theorem 13.2: subject reduction is non-deterministic when intersection is based on multisets, which gives rise to **reduction choices**. We prove that every sequence of reduction choices in system \mathcal{R} can be encoded in system **S**.

²²A type system is relevant when every typing assumption in a given derivation must be effectively used in this derivation *i.e.* derivations only introduce what they actually need (no weakening). Thus, relevance captures a form of resource-awareness.

²³which are a class of Böhm trees without \perp

Minor Contributions As a minor contribution, we explain how representing non-idempotent intersection types with multisets in the finite case yields a type system that is syntax directed but non-deterministic w.r.t. subject reduction (Sec. 4.1.2). This fact was perhaps already known and it has both advantages and disadvantages in the finite case, but we show that in the infinite case, the non-determinism of reduction makes it impossible to formulate the validity criterion that is used to solve Klop’s Problem.

We also suggest a visual presentation for relevant derivations (Sec. 4.1.1) that is used throughout this thesis and that makes the understanding of some technical arguments easier.

Technical Contributions For each group of contributions, we isolate an emblematic technical difficulty that needed to be solved, and we give pointers to the sections where they are explained in more details.

- Normalization in the λ_μ -calculus: the λ -calculus is a computational interpretation of intuitionistic natural deduction. In the case of the non-idempotent typing, reduction, which corresponds to a cut-elimination step in *intuitionistic* natural deduction, decreases the number of rules of the typing derivations. This provides a simple argument for the termination of the reduction. In the λ_μ -calculus, a μ -reduction step, which corresponds to some cut-elimination step in *classical* natural deduction, does not necessarily decrease the number of rules of the derivation even in the non-idempotent setting. We must then find a more elaborated measure for derivations to ensure a decrease under reduction. The design of this measure and of the associated non-idempotent typing systems is explained in Sec. 7.2.3.
- Klop’s Problem: the most technical point is to perform infinitary subject expansion (infinitary subject reduction is relatively easy) along a reduction sequence of infinite length *i.e.* we must prove that, if t reduces to t' after an infinite number of steps, then any typing of t' is also valid for t . The method to achieve this is described in Sec. 10.1.3.
- For the two last group of contributions: the considered type systems also type terms that do not stabilize *e.g.*, terms t such that do not output a “stable block” which could be easily typable (in the finite case, the typing of the normalizing terms usually relies on the typing of “stable blocks” *e.g.*, head normal forms or abstractions). These systems are not productive and we must find a method to handle typing. We then develop a corpus of methods inspired by first order logic and relying on the identification of some elements of a given derivation. Chapter 11 is dedicated to those methods and the associated notions, which are presented there from a high-level perspective.

On the Preliminaries The core of the preliminaries is Chapter 3 in which we tried at our best to dissect the designs of (some) intersection type systems, as well as their features (idempotence, syntax direction, relevance. . .). This exploration is complemented by Chapters 4 and 5. Here is the list of the preliminaries in this thesis:

- Chapter 2: the λ -calculus.
- Chapter 3: presentation of intersection type systems and a type-theoretic characterization of head normalization in the non-idempotent case.

- Chapter 4: an overview of some intersection type systems, including an example of non-confluence of subject reduction.
- Chapter 5: type-theoretic characterizations of weak and strong normalization.
- Chapter 6: classical logic, the λ_μ -calculus (simply typed and untyped version).
- Chapter 9: two infinitary λ -calculi.

1.6 How to read this thesis

We present now an overview of the structure of this thesis: we distinguish the chapters that present the state of the art from those that present a contributions and we discuss the relations between them. In short, we explain how this thesis can be read and we indicate the dependencies of various strength that may exist between the chapters.

Preliminaries *vs.* Contributions

The body of this thesis contains 4 parts and 12 chapters:

- Five chapters presenting a contribution (Chapters 7, 8, 10, 12 and 13).
- Six preliminary chapters, presenting the state of the art on various domains or questions, including:
 - Four general preliminaries chapters (Part I, from Chapter 2 to Chapter 5).
 - Two specific preliminary chapters: Chapter 6 and Chapter 9, respectively presenting the λ_μ -calculus and the infinitary λ -calculus.
- An in-between chapter (Chapter 11) giving a semi-informal presentation of some concepts and methods developed in the course of the thesis, that are extensively used in Chapters 12 and 13.
- Each non-preliminary part (*i.e.* Parts II, III and IV) begins with a short general presentation.

The content of the four parts is the following:

- Part I only contains background on the λ -calculus and intersection type systems, in particular, on non-idempotent intersection operators and the characterization of head, weak and strong normalization.
- Part II is dedicated to the λ_μ -calculus: one background chapter on the λ_μ -calculus and its simply typed version, one contribution chapter presenting two non-idempotent intersection and *union* type systems respectively characterizing head and strong normalization, then one contribution chapter introducing a small-step operational semantics for the λ_μ -calculus along with a type system characterizing strong normalization.
- Part III is dedicated to a characterization of weak normalization in an infinitary λ -calculus and it contains two chapters: one background chapter on the infinitary λ -calculus, and one contribution chapter, presenting the aforementioned type-theoretical characterization. This positively answers to a question known as Klop's Problem.

- Part IV is dedicated to two problems involving type systems that do not ensure any kind of productivity and the development of a method allowing us to study typing in this case. It contains a preliminary chapter (Chapter 11) giving an informal account of the technique, followed by two contribution chapters.

Dependencies

We give a short account of the dependencies between the chapters of this thesis:

- The only inescapable dependency is that of Chapter 8 on Chapter 7: the type system (system $\mathcal{S}_{\lambda_{\mu x}}$) presented in the former is an extension of the system $\mathcal{S}_{\lambda_{\mu}}$, which is presented in the latter. In particular, some elements of the design of $\mathcal{S}_{\lambda_{\mu x}}$ are not discussed since they are already featured in system $\mathcal{S}_{\lambda_{\mu}}$ and the proofs of Chapter 8 only address the new cases compared to $\mathcal{S}_{\lambda_{\mu}}$.
- Except for Chapter 8, each contribution chapter is (almost) self-contained.
- Moreover, in each contribution chapter, we give a summary of the background techniques to be involved with *pointers* to the preliminaries. Thus, it is not necessary to read the preliminaries to understand the contribution chapters. More specifically:
 - Part II: Chapters 7 and 8 can be read with a basic knowledge of the (simply typed or not) λ_{μ} -calculus and of non-idempotent intersection type systems. It is also helpful to know of the type-theoretic characterization of strong normalization in the λ -calculus.
 - Part III: Chapters 10 can be read with a basic knowledge of non-idempotent intersection type systems. It is also helpful to know of the type-theoretic characterization of weak normalization in the λ -calculus or of the infinitary λ -calculus.
 - Part IV: Chapters 12 and 13 are independent of one another and can be read with a basic knowledge of non-idempotent intersection type systems and a good understanding of residuation in the case of the λ -calculus. Although they are also self-contained, they are the most technical chapters in this thesis and reading first Chapter 11, which informally presents a corpus of useful concepts and methods, is strongly advised.

This thesis is globally self-contained, except for the properties of confluence of the calculi that are addressed in this dissertation (including the λ -calculus): we state these properties without proof.

Part I

Preliminaries

As no better man advances to take this matter in hand, I hereupon offer my own poor endeavors. I promise nothing complete; because any human thing supposed to be complete, must for that very reason infallibly be faulty.

Herman Melville, *Moby Dick*

Chapter 2

Lambda Calculus

The Lambda Calculus was introduced by Alonzo Church in 1928¹, in an attempt at founding mathematics on the notion of function, rather than that of sets. Then, after having the short-lived hope that it could escape Gödel's Incompleteness Theorems, Church used λ -calculus to formalize the notion of *computable function* and to give the first proof of Undecidability of the *Entscheidungsproblem* (Decision's Problem).

In the same period, other propositions were made to formalize the concept of computability, as the Herbrand-Gödel recursive functions (*cf.* [99], 17.2.4.) and Turing Machines (*cf.* [99], 1.4.). It was soon proved that those systems had the exact same expressive power as λ -calculus: every function that is implementable in one system may also be implemented in the two others, modulo some suitable and matterless encodings. This observation led to **Church-Turing's thesis**: every programming language or physical/mechanical computing device has at most the expressive power of recursive functions/Turing Machines/the Lambda Calculus (see p. 19). In other words, there is no actual way to compute a function that could not be already implemented and computed in λ -calculus or with Turing Machines/recursive functions.

The proof of Undecidability of the *Entscheidungsproblem* resorting to Turing Machines was considered as more natural and simple than the original one. After that, λ -calculus received only marginal attention until the suggestions, for instance of Peter Landin [71,72], to use λ -terms to study, understand and develop the newly introduced *functional* languages. Lambda calculus (and its variants) has been henceforth regarded as a paradigm for programming.

Outline In the first section, we present some basic aspects of λ -calculus. We stress out the importance of the heuristics of λ -term as trees. For instance, Figure 2.6 is a simple graphical representation of β -reduction, but it also illustrates the underlying mechanisms behind *subject reduction* and *subject expansion* in type systems (see the figures of Sec. 3.3, p. 3.3, Fig 10.5, p. 10.5 and 13.2, p. 297). The same figure allows us to support the well-known notion of residual (Sec. 2.1.5 and Figure 2.8), that we will also have to extend to typing derivations in the later parts of this thesis (*e.g.*, Sec. 10.3.5, 12.4.1 for system **S**, 13.2.2 and 13.5.1 for system **S_{op}**). We present some important λ -terms in Sec. 2.1.4.

¹We are greatly indebted to Hindley and Cardone's **History of Lambda-calculus and Combinatory Logic** [21] for the elements of historical context in this thesis. A downloadable free version can easily be found on the internet

Section 2.2 is dedicated to normalization. We present the principal variants (head, weak and strong normalization) and their relations with some particular reduction strategies.

In the last section, we discuss the ways in which the notion of normalization may be modulated. We recall the definition of *mute* terms (Sec. 2.3.2) and explain why those terms are considered as the most *non-normalizing* term. In Sec. 2.3.3, we give a few hints at *infinitary normalization* and infinite terms, including *Böhm trees*, just before describing the Böhm reduction strategies (Sec. 2.3.4). We end this chapter with a “small-step” version of the λ -calculus (Sec. 2.4).

2.1 Pure Lambda Calculus

In the present section 2.1, we give a short presentation of the λ -calculus. We present the formalism that we will use for labelled trees throughout this thesis (Sec. 2.1.1) and the key notions of:

- **Support** of a λ -term, that allows us to point inside λ -terms (Sec. 2.1.2).
- **β -reduction**, that corresponds to an execution step when λ -terms are seen as programs (Sec. 2.1.3).
- **Residuation**, that describe how positions in a λ -term are *traced* (*i.e.* affected or moved) by β -reduction (Sec. 2.1.6, 2.1.5).

We also observe phenomenons as **duplication** (Sec. 2.1.3) or the existence of **fixpoint combinators** (Sec. 2.1.4). All this will be illustrated with trees and figures.

2.1.1 Tracks and Labelled Trees

Many objects to be considered in this thesis may be regarded as **labelled trees**.

We present here the formalism that we will use for labelled trees and define some tools and notations associated with this notion. At the end of this section, we will actually distinguish two kinds of labelled trees: the *rigid* ones and the *non-rigid* ones. A **rigid tree** is just a labelled tree in the most common sense *i.e.* a tree whose nodes and edges have both labels with the additional hypothesis that brother edges are decorated with pairwise distinct labels *e.g.*, λ -terms are rigid trees (Sec. 2.1.2 to come next). A labelled tree is non-rigid when its nodes have labels, but its edges do not. As we will see in later chapters (Sec. 4.1.2), the typing derivations of many intersection type systems are intrinsically *non-rigid* trees.

Rigidity will play a central role to define and motivate the type system **S** in the later chapters (Chapters 10, 12 and 13, see *e.g.*, Sec. 10.3.4), but λ -terms constitute an elementary example of rigid labelled trees (see the figures of the following section).

If \mathcal{A} is a set, then \mathcal{A}^* denotes the set of **finite words** on the *alphabet* \mathcal{A} . In this document, \mathcal{A} will usually be chosen as a set of natural numbers. Mainly, $\mathcal{A} = \mathbb{N}$ or $\mathcal{A} = \{0, 1, 2\}^*$ will hold.

The *empty word* is denoted by ε and if $w_1, w_2 \in \mathcal{A}^*$, then $w_1 \cdot w_2$ is the *concatenation* of w_1 and w_2 . *e.g.*, with $\mathcal{A} = \mathbb{N}$, $\mathcal{A}^* = \mathbb{N}^*$ is the set of the words whose letters are natural numbers *e.g.*, $w_1 = 2 \cdot 1 \cdot 3 \cdot 7$, $w_2 = 2 \cdot 10 \cdot 3 \cdot 7$ and $w_3 = 2 \cdot 1$ are elements of \mathbb{N}^* . Sometimes, we may just write $w_1 w_2$ instead of $w_1 \cdot w_2$ (usually, there will be no ambiguity as with $2 \cdot 1 \cdot 0$, $2 \cdot 10$ and 210 *e.g.*, 10 will always denote $1 \cdot 0$).

The *prefix order* \leq is defined on \mathcal{A}^* by $w_1 \leq w_2$ if there exists w_3 such that $w_2 = w_1 \cdot w_3$. Note that, for all $w \in \mathcal{A}^*$, $\varepsilon \cdot w = w$, $w \cdot \varepsilon = w$ and $\varepsilon \leq w$. For instance, coming back to the previous w_1 , w_2 , w_3 , we have $w_3 \leq w_1$ since $w_1 = w_3 \cdot 3 \cdot 7$, but *not* $w_3 \leq w_2$.

If $w \in \mathcal{A}^*$, then $|w|$ is the *length* of w *i.e.* the number of letters that w contains *e.g.*, $|w_1| = 4$, $|w_2| = 4$ and $|w_3| = 2$. If $a \in \mathcal{A}$ and $n \in \mathbb{N}$, then a^n denotes the word $\underbrace{a \cdot \dots \cdot a}_n$ *e.g.*, $2^3 \cdot 0^2 := 2 \cdot 2 \cdot 2 \cdot 0 \cdot 0$.
n occ. of a

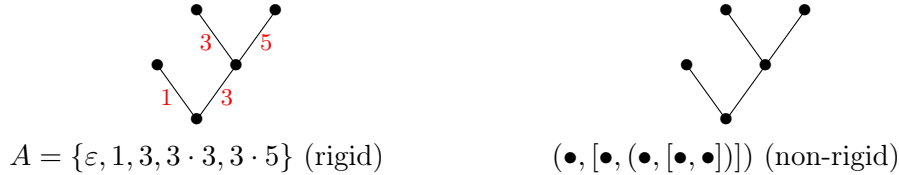


Figure 2.1: Unlabelled Trees (Rigid and non-Rigid)

A **(rigid) tree** A is a *non-empty* subset of \mathbb{N}^* that is closed under the prefix order *i.e.* for all $A \subseteq \mathbb{N}^*$, A is a tree iff $A \neq \emptyset$ and for all $w_1, w_2 \in \mathbb{N}^*$ such that $w_1 \leq w_2$, then $w_2 \in A$ implies $w_1 \in A$. A rigid tree can be thought as a set of (unlabelled) **positions**. Thus, the nodes of a rigid tree A are not labelled, but its edges *implicitly* are *i.e.* if $w \cdot k \in A$ with $k \in \mathbb{N}$, then the number k is taken as the label of the *edge* between positions w and $w \cdot k$ as in the left-hand side of Fig. 2.1. We then call any number that labels an edge of a rigid tree a **track**. Tracks will be a fundamental feature of system **S** in Parts III and IV.

Let A be a rigid tree and $w \in A$. Then the **subtree** of A rooted at w is $A|_w := \{w' \in \mathbb{N}^* \mid w \cdot w' \in A\}$ (A' is also a tree).

Rigid Labelled Trees Let Σ be a set. A **(rigid) labelled tree** on the **signature** Σ is a function $T : \mathbb{N}^* \rightarrow \Sigma$ such that the domain $\text{dom}(T)$ of T is a rigid tree. The domain of a labelled tree T is then called its **support** and denoted $\text{supp}(T)$. The **size** $|T|$ of a tree T is the cardinal of its support *i.e.* $|T| := \#\text{supp}(T)$.

In that case, for all $w \in \text{dom}(T)$, $T|_w$ denotes the *subtree* of T rooted at position w *i.e.* $T|_w$ is the labelled tree T' defined by $\text{dom}(T') = \text{dom}(T)|_w = \{w' \in \mathbb{N}^* \mid w \cdot w' \in \text{dom}(T)\}$ and $T'(w') = T(w \cdot w')$ for all $w' \in \text{dom}(T')$.

A word $w \in \text{dom}(T)$ is called a *position* of T and $T(w)$ is the label of the *node* of T located at position w . If $T(w) = \sigma \in \Sigma$, we also say that σ **occurs** or has an **occurrence** at position w in T . Positions ensure that we can point to any precise node or part of those trees.

Example 2.1. Let T be the rigid labelled tree on the signature $\Sigma = \{a, b, c\}$ such that $\text{supp}(T) = \{\varepsilon, 1, 1 \cdot 1, 1 \cdot 2, 2, 2 \cdot 2, 2 \cdot 3, 2 \cdot 5, 3, 3 \cdot 1, 3 \cdot 2\}$ and $T(\varepsilon) = T(1) = T(2) = T(3) = T(2 \cdot 2) = T(2 \cdot 3) = a$, $T(2 \cdot 5) = T(1 \cdot 2) = T(3 \cdot 2) = b$ and $T(1 \cdot 1) = T(3 \cdot 1) = c$. Then the two drawings of Fig. 2.2 both represent T .

Non-Rigid Trees Sometimes, it is unnecessary, if not problematic, to distinguish *e.g.*, two immediate subtrees of a given tree when those subtrees are equal. For instance, as it will be addressed later (Sec. 3.2.1, Remark 3.8 and 4.2), if two equal argument

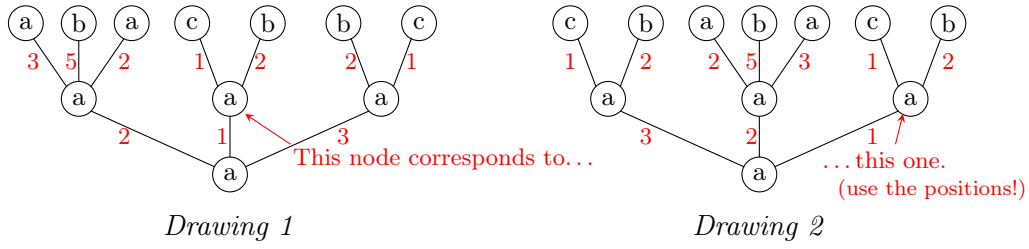


Figure 2.2: Rigid Labelled Trees

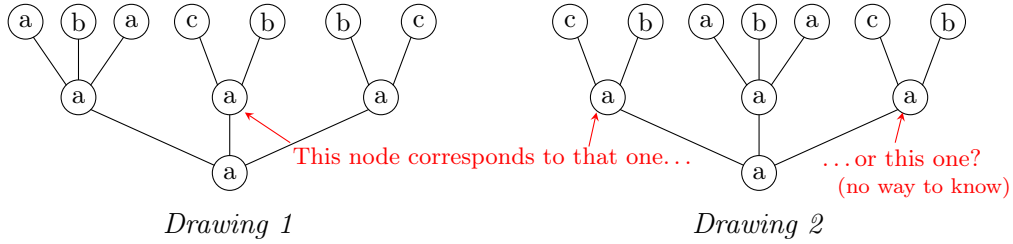


Figure 2.3: Non-Rigid Labelled Trees

subderivations of a typing derivation of Gardner/de Carvalho’s system \mathcal{R}_0 could be distinguished, the possibility of reduction choices would be lost.

For now, we may just give a few intuitions on how rigidity can be “disabled”, thus giving rise to the notion of **non-rigid tree**. The idea is to remove the edge annotations that exist *e.g.*, in Fig. 2.2. For that, we must consider trees that are not anymore subsets of \mathbb{N}^* and functions from \mathbb{N}^* to some signature Σ : the set of non-rigid (unlabelled) trees can be defined by the inductive grammar

$$\mathcal{T} := \bullet \mid (\bullet, [\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_n]) \quad (n \geq 1)$$

The notation $(\bullet, [\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_n])$ denotes² the tree whose immediate subtrees are $\mathcal{T}_1, \dots, \mathcal{T}_n$ and \bullet is the “root tree”. In Fig. 2.1, the right-hand side drawing represents a non-rigid unlabelled tree.

The set of the **non-rigid labelled trees** on a signature Σ is defined by the inductive grammar:

$$\mathcal{T} := \sigma \in \Sigma \mid (\sigma, [\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_n]) \quad (n \geq 1)$$

For instance, the non-rigid unlabelled tree $(a, [(a, [a, a, b]), (a, [b, c]), (a, [b, c])])$ corresponds to the specification “The root of \mathcal{T} is labelled with a . The root has 3 children, all labelled with a . One of the children has three children (which are leaves of \mathcal{T}) such that two are labelled with b and one with a . The other two children of the root have both two children (which are leaves of \mathcal{T}), one that is labelled with a and the other with b ”. The two drawings of Fig 2.3 represent \mathcal{T} .

Among the three children of the root, the node that has three children can be distinguished from the two others (on the left in Drawing 1, in the middle in Drawing 2), that only have two children. But those two cannot be distinguished one from another: indeed, they root identical subtrees. Of course, on a particular drawing, *e.g.*, the left

² $[\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_n]$ denotes a multiset (Sec. 3.2.2)

one, we may say “this one on the middle” or “that one on the right”, but it depends on the drawing and *not* on the (too vague) specification of \mathcal{T} . This is also illustrated by the fact that we cannot unambiguously identify every node of one drawing to a node of the other (see the commentary inside the figure). Concretely, a non-rigid tree does not have support and a node in a non-rigid tree does not have a position.

In contrast with Fig. 2.1 representing a rigid decoration of \mathcal{T} , we can associate each node of the drawing on the right-hand side to a node of the drawing on the left-hand side by just comparing their position. On this figure, we identify the nodes (labelled with a) that have the common position 1.

Remark 2.1. Note that non-rigid trees can be seen as *collapses* of rigid trees *i.e.* a rigid tree can be thought as an equivalence class of rigid trees that are equal when the tracks (edges labels) are forgotten *e.g.*, the non-rigid tree represented twice in Fig. 2.3 is the collapse of the rigid tree represented twice in Fig 2.2. This will be made formal and thoroughly studied in the last chapter of this thesis (Chapter 13).

2.1.2 Lambda Terms and Alpha Equivalence

Let \mathcal{V} be a countable set of term variables (metavariables x, y, z). To each $x \in \mathcal{V}$, we associate another the string λx . The set Λ of λ -terms is defined by the following *inductive*³ grammar:

$$t, u ::= x \in \mathcal{V} \mid (\lambda x.t) \mid (tu)$$

Notation 2.1.

- The term (tu) , called the *application* of t to u , is often written tu and we consider the application operator as left-associative *i.e.* $t_1 t_2 \dots t_q$ stands for $(\dots ((t_1 t_2) t_3) \dots t_q)$.
- A term of the form $(\lambda x.t)$ is an *abstraction*. We often write just $\lambda x.t$ instead of $(\lambda x.t)$. More generally, if a term contains several successive abstractions, we omit the λ except on the first one *e.g.*, we write $\lambda xyz.t$ instead of $(\lambda x.(\lambda y.(\lambda z.t)))$.

Lambda terms can be seen as rigid labelled trees following this pattern:

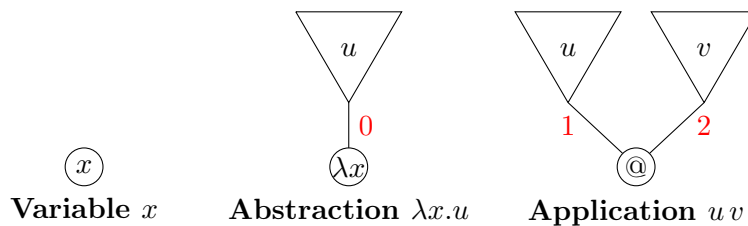


Figure 2.4: Lambda Terms as Labelled Trees

Nodes are labelled by $x, \lambda x$ (x ranging over \mathcal{V} , a countable set of term variables) or $@$. As described in the previous section, the natural numbers that label edges are called **tracks**: more precisely, track 0 is dedicated to abstractions, track 1 to application left-hand sides, track 2 to application their right-hand sides, also called the **argument** of

³The mechanisms of **induction** and **coinduction** are briefly explained in Sec. 9.2.3

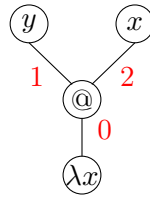


Figure 2.5: Parsing Tree of $\lambda x.yx$

the application. A position of a term t is some $b \in \{0, 1, 2\}^*$ *i.e.* a word in the alphabet $\{0, 1, 2\}^*$. The set of positions of a term t , known as its **support**, is written $\text{supp}(t)$.

For instance, the parsing tree of $t := \lambda x.yx$ is given by Fig. 2.5: Thus, we have $\text{supp}(t) = \{\varepsilon, 0, 0 \cdot 1, 0 \cdot 2\}$, $t(\varepsilon) = \lambda x$, $t(0 \cdot 1) = y$, $t(0 \cdot 2) = x$, $t(0) = @$ and $t|_0 = yx$.

Abstraction λx *binds* the variable x in $\lambda x.t$. We say then that variable x is **bound** in $\lambda x.t$. An occurrence of a variable that is not bound is said to be **free**. More precisely, a bound occurrence of a variable $x \in \mathcal{V}$ in a given term t is a position $b \in \text{supp}(t)$ such that $t(b) = x$ and there exists $b_* < b$ with $t(b_*) = \lambda x$. If b is a bound occurrence of the variable x in t , the *binding position* of this occurrence is the maximal prefix $b_* \leq b$ such that $t(b_*) = \lambda x$. We then say that b is bound by position b_* in t and we write $b_* = \lambda^t(b)$. For instance, let $t = \lambda x.x$, so that $\text{supp}(t) = \{\varepsilon, 0, 0^2\}$, $t(\varepsilon) = \lambda x$, $t(0) = \lambda x$ and $t(0^2) = x$. Then $\lambda^t(0^2) = 0$ *i.e.* the occurrence of x is bound at position 0 and *not* at the root of t .

For all term t , the set of free variables of t is denoted $\text{fv}(t)$ *i.e.* for all $x \in \mathcal{V}$, $x \in \text{fv}(t)$ iff x has at least one free occurrence in t . For instance, $\text{fv}(xy) = \{x, y\}$, $\text{fv}(\lambda x.x) = \emptyset$, $\text{fv}(\lambda x.yx) = \{y\}$ and $\text{fv}(x(\lambda xz.yxz)) = \{x, y\}$. A term t is **closed** when no variable occurs free in t *i.e.* when $\text{fv}(t) = \emptyset$.

We sometimes denote the number of free occurrences of a variable x in a term t by $|t|_x$ *e.g.*, $|(\lambda x.x)xx|_x = 2$.

As it will turn out with β -reduction (next section), bound variables are used to denote the entry of λ -terms-seen-as-functions (whereas the free variables of a term may be seen as *constants*). Usually, the variable denoting the entry of a function does not matter *e.g.*, in mathematics, if f is the square function, then we may indifferently write $f : x \mapsto x^2$ or $f : z \mapsto z^2$. Likewise, the choice of symbols (*e.g.*, x *vs.* z) for bound variables does not really matter for λ -terms, provided some common sense precautions are taken.

This yields the notion of **α -equivalence** (that will be more formally defined for the infinitary λ -calculus in Sec. 9.3.1): two terms t and t' are α -equivalent iff (1) the terms t and t' have the same support *i.e.* $\text{supp}(t) = \text{supp}(t')$ (2) free variables occur at the same positions in t and t' (3) for all $b \in \{0, 1, 2\}^*$ such that b is a bound occurrence of a variable x (resp. a variable x') in t (resp. in t'), then b is bound by the same position in t and in t' (*i.e.* $\lambda^t(b) = \lambda^{t'}(b)$).

When two terms t and t' are α -equivalent, we say that one can be obtained by α -*renaming* the other. A term satisfies the **Barendregt convention** if (1) its set of free variables and its set of bound variables are disjoint (2) for all $x \in \mathcal{V}$, λx occurs at most *once* in t .

When two terms t and t' are α -equivalent, we actually consider them as *equal* and we just write $t = t'$. This can be shown to be consistent with the definition of β -reduction

to come below. Modulo α -renaming, every term may satisfy Barendregt convention *e.g.*, $\lambda x.x$ may be rewritten into $\lambda xy.y$.

For instance, $\lambda z.z = \lambda x.x \neq \lambda z.x \neq \lambda x.z$ and $\lambda x.xx = \lambda y.yy \neq \lambda x.xy, \lambda z.yz$. We have $\lambda xx.x = \lambda yx.x = \lambda xy.y \neq \lambda xy.x$ and also $\lambda x.xz = \lambda y.yz$ but $\lambda z.xz \neq \lambda z.yz$. The terms $\lambda x.xy, \lambda y.xy, \lambda y.yx$ are pairwise distinct.

We write $t[u/x]$ for the **capture-free substitution** of x by u inside t , meaning that $t[u/x]$ is the term obtained from t by replacing (in t) every *free* occurrence of x by u and by α -renaming the bound variables of t so that no abstraction of t binds a variable occurring free in u .

For instance, $(\lambda y.xyx)[\lambda x.x/x] = \lambda y.(\lambda x.x)y(\lambda x.x), (\lambda x.x)[zy/x] = \lambda x.x$. Notice that $(\lambda z.xzx)[xz/x]$ is equal to $\lambda y.xzy(xz)$ but *not* to $\lambda z.xzz(xz)$ (z does not occur free in the latter term).

Definition 2.1. Let t be a term. The **size** $|t|$ of the λ -term t is defined by $|t| = \#\text{supp}(t)$.

Thus, the size of t is the size (number of nodes) of its parsing tree. Equivalently, we can define $|t|$ by induction: $|x| = 1, |\lambda x.t| = |t| + 1$ and $|tu| = |t| + |u| + 1$.

2.1.3 Beta Reduction, Redexes and Normal Forms

We first treat the case of *root reduction*. A *reducible expression* (for short, a **redex**) is a term of the form $(\lambda x.t)u$. The (*root*-)reduct of $(\lambda x.t)u$ is defined as $t[u/x]$. For instance, $(\lambda x.xyx)(zy) \rightarrow_{\beta} zyy(zy)$ (where \rightarrow_{β} denotes the reduction, whose full definition is below).

The representation of λ -terms by trees is of great help to understand reduction. We consider the reduction from the redex $(\lambda x.r)s$ to the reduct $r[s/x]$. Then the free occurrences of x in the subterm r correspond to some *leaves* labelled with x . Then, to obtain the tree of the reduct $r[s/x]$, we destroy those leaves and replace each of them (inside r) by the tree of s . We also destroy the application and the abstraction of the redex. In Figure 2.6, we consider the situation in which r holds exactly 3 (free) occurrences of x .

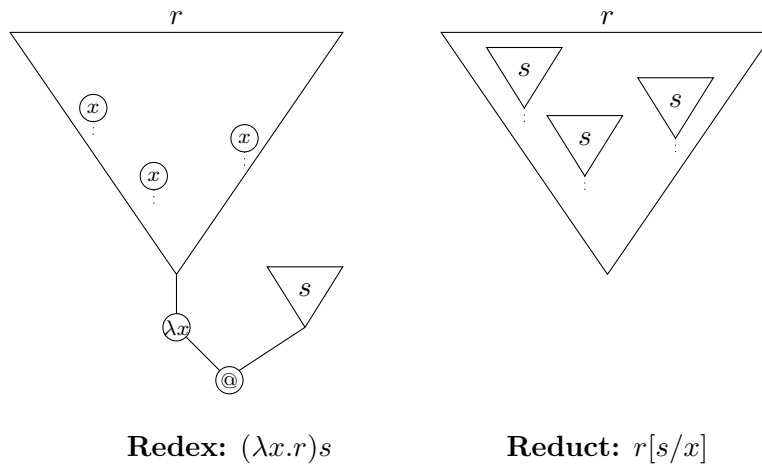


Figure 2.6: Reduction from the Tree Perspective

Figure 2.6 is very important in many aspects: in type theory, it turns out that the subject reduction and subject expansion properties may be seen as the *adaptation* of this figure to *typing derivations* (instead of terms). It provides a very good intuition about why those properties hold (see Figures 3.1, 3.2, 3.3, 10.5 and 13.2). Moreover, it gives a graphical illustration of the possible duplication of the argument during reduction: in the figure, argument s is duplicated 3 times.

A redex may be reduced *inside* a term at some given position. For that, we define **pointed β -reduction** $t \xrightarrow{b}_\beta u$ by induction on $b \in \{0, 1, 2\}^*$ as follows:

$$\frac{}{(\lambda x.t)u \xrightarrow{\varepsilon}_\beta t[u/x]} \qquad \frac{t \xrightarrow{b}_\beta t'}{\lambda x.t \xrightarrow{0 \cdot b}_\beta \lambda x.t'}$$

$$\frac{t \xrightarrow{b}_\beta t'}{t u \xrightarrow{1 \cdot b}_\beta t' u} \qquad \frac{u \xrightarrow{b}_\beta u'}{t u \xrightarrow{2 \cdot b}_\beta t u'}$$

We notice that, for each $b \in \{0, 1, 2\}^*$, relation \xrightarrow{b}_β is functional. The **full β -reduction** \rightarrow_β is defined by $t \rightarrow_\beta t'$ iff there exists $b \in \text{supp}(t)$ such that $t \xrightarrow{b}_\beta t'$ *i.e.* \rightarrow_β is the union of the pointed reductions:

$$\rightarrow_\beta = \bigcup_{b \in \{0, 1, 2\}^*} \xrightarrow{b}_\beta$$

The full β -reduction is not deterministic *e.g.*, with $t = (\lambda z.z((\lambda x.xx)z))u$, we have $t \rightarrow_\beta u((\lambda x.xx)u)$ (at ε) and $t \rightarrow_\beta (\lambda z.z(zz))u$ (at $1 \cdot 0 \cdot 2$).

Let t be a term. If there exists a term u such that $t \rightarrow_\beta u$, then we say that t is **reducible**: t is reducible iff t contains a redex as a subterm (see Figure 2.7). In that case, u is called a *reduct* of t . Thus, a redex is a term that is reducible at its *root*. More generally, a **reduct** of a term t is a term u that may be obtained from t by several reduction steps *e.g.*, if $I := \lambda x.x$ and $\Delta = \lambda x.xx$, then $\Delta I \rightarrow II \rightarrow I$, so that I is a (rank 2) reduct of ΔI . We write \rightarrow_β^* for the reflexive transitive closure of \rightarrow_β and $t \rightarrow^n t'$ to mean that t' is obtained from t in n reduction steps.

Conversely, if $t \rightarrow^* t'$, we say that t is an **expansion** of t' . Notice that for a given term t' and $b \in \{0, 1, 2\}^*$, there may be several terms t such that $t \xrightarrow{b}_\beta t'$ *e.g.*, with $t' = xx$, $t_1 = (\lambda x.x)xx$, $t_2 = (\lambda y.xy)x$, we both have $t_1 \xrightarrow{\varepsilon}_\beta t'$ and $t_2 \xrightarrow{\varepsilon}_\beta t'$.

We denote \equiv_β the reflexive symmetric transitive closure of \rightarrow . Thus, \equiv_β is an equivalence relation on Λ . When $t \equiv_\beta t'$, we say that t and t' are **β -equivalent**.

If t is not reducible *i.e.* t does not contain any redex, we say that t is a **Normal Form (NF)** *e.g.*, $I = \lambda x.x$, $\Delta = \lambda x.xx$ and $x I \Delta$ are normal forms.

From now on, we will usually write \rightarrow instead of \rightarrow_β *e.g.*, we write $(\lambda x.x)y \rightarrow y$ instead of $(\lambda x.x)y \rightarrow_\beta y$, with the important exception of Part IV.

2.1.4 Notable Lambda Terms

The following terms will be used throughout this thesis as examples or objects of study. They will be redefined when they are needed, but we briefly explain here why they may be of future interest and look at some aspects of their *dynamic* behaviours. By

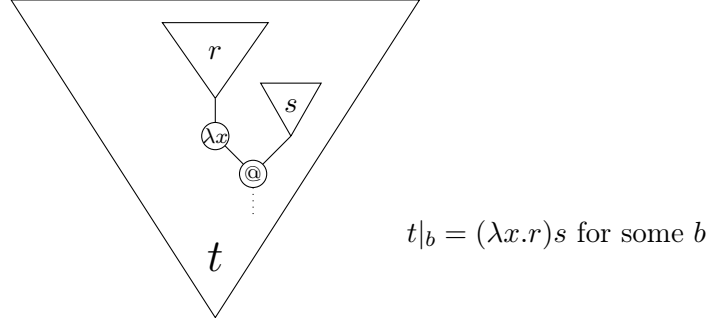


Figure 2.7: Tree of a Reducible Term

dynamics, we mean whatever concerns the effects of β -reduction on λ -terms.

$$\begin{array}{lll}
 I = \lambda x.x & K_y = \lambda x.y & K = \lambda yx.y \\
 \Delta = \lambda x.xx & \Omega = \Delta\Delta & \Delta_f = \lambda x.f(xx) \\
 Y_f = \Delta_f\Delta_f & Y = \lambda f.Y_f & Y_\lambda = (\lambda xy.xx)(\lambda xy.xx) \\
 \omega_3 = \lambda x.xxx & \Omega_3 = \omega_3\omega_3 &
 \end{array}$$

The term I is the *identity*: indeed, for all $t \in \Lambda$, $I t \rightarrow t$. The term K_y represents the constant function equal to y : for all $t \in \Lambda$, $K_y t \rightarrow y$. We then say that the argument is **erased** during this reduction. More generally, for all $t, u \in \Lambda$, $K t u \rightarrow (\lambda x.t)u \rightarrow t$ (since $x \notin \text{fv}(t)$): thus, $K t$ represents the constant function equal to t .

The term Δ is the *auto-application*: for all $t \in \Lambda$, $\Delta t \rightarrow t t$ (t applied to itself). In particular, $\Omega = \Delta \Delta$ satisfies $\Omega \rightarrow \Delta \Delta = \Omega$: reduction loops for Ω . This term is to be called later (see Sec. 2.3.2) a **mute** term.

The term Y is called **Curry fixpoint combinator** for the following reason: for all $t \in \Lambda$, $Y t \rightarrow (\lambda x.t(xx))(\lambda x.t(xx)) \rightarrow t(\lambda x.t(xx))(\lambda x.t(xx))$. Thus, we have $Y t \equiv_\beta t(Y t)$ *i.e.* $Y t$ is a fixpoint for t w.r.t. β -equivalence. We notice that $Y I \rightarrow \Omega$ *i.e.* Ω is an *instance* of Curry fixpoint combinator.

We have $Y f \rightarrow Y_f$. Moreover, $Y_f \xrightarrow{\varepsilon} f(Y_f) \xrightarrow{2} f(f(Y_f)) \xrightarrow{2^2} f(f(f(Y_f))) \xrightarrow{2^3} \dots \xrightarrow{2^{n-1}} f^n(Y_f)$ for all $n \in \mathbb{N}$, where $f^n(Y_f)$ stands for $\underbrace{f(f(\dots(f(Y_f))\dots))}_{n \text{ occ. of } f}$.

We have $Y K \rightarrow Y_\lambda$. Moreover, $Y_\lambda \xrightarrow{\varepsilon} \lambda x.Y_\lambda \xrightarrow{0} \lambda x.\lambda x.Y_\lambda \xrightarrow{0^2} \lambda x.\lambda x.\lambda x.Y_\lambda \xrightarrow{0^3} \dots \xrightarrow{0^{n-1}} \lambda x^n.Y_\lambda$, where $\lambda x^n.Y_\lambda = \underbrace{\lambda x.\lambda x \dots \lambda x.\lambda x}_{n \text{ occ. of } \lambda x}.Y_f$. Thus, for all finite sequence u_1, \dots, u_n of terms, $Y_\lambda u_1 \dots u_n \rightarrow^* Y_\lambda$.

We have $\Omega_3 \xrightarrow{\varepsilon} \Omega_3 \omega_3 \xrightarrow{1} \Omega_3 \omega_3 \omega_3 \xrightarrow{1^2} \dots \xrightarrow{1^{n-1}} \Omega_3 \underbrace{\omega_3 \dots \omega_3}_{n \text{ occ. of } \omega_3}$.

2.1.5 Residuals and Quasi-Residuals

As suggested by Figure 2.6, if t' is a (one step) reduct of t , the tree of t' (*i.e.* the support and the labels of the nodes) is simply obtained from that of t : intuitively, each position of t' may be seen as a **residual** of a position of t .

So, assume that $t|_b = (\lambda x.r)s$ and $t \xrightarrow{b} t'$ (so that $t'|_b = r[s/x]$). Thus, $t(b) = @$ and $t(b \cdot 1) = \lambda x$. The subterm r occurs at position $b \cdot 1 \cdot 0$ (that we abusively write $b \cdot 10$) and

the subterm s at position $b \cdot 2$. We write $\mathfrak{o}_\beta^t(b)$ for the occurrences of the variable x to be substituted during reduction: those occurrences are the positions that are bound by λx at position $b \cdot 1$ i.e. $\mathfrak{o}_\beta^t(b) = \{b_0 \in \text{supp}(t) \mid \lambda^t(b_0) = b \cdot 1\}$ (see Sec. 2.1.2 for notation λ^t).

Before giving a complete definition, let us start by considering a few examples that will be supported by Figure 2.8 (extending Figures 2.6 and 2.7). We still assume that there are 3 occurrences of x to be substituted during reduction. Those occurrences are in r (position $b \cdot 10$) so that their positions may be written $b \cdot 10 \cdot b_i$ ($i = 1, 2, 3$) respectively.

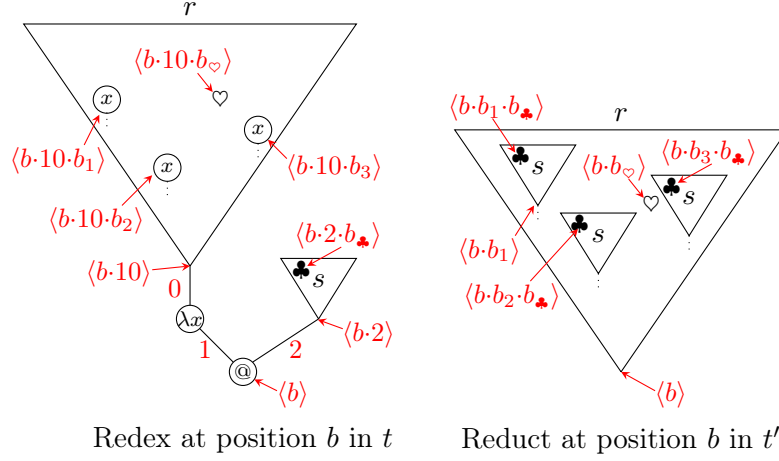


Figure 2.8: Residuals of Positions

We indicate positions of a node with angle brackets *e.g.*, in t , the leftmost node labelled with x is at position $b \cdot 10 \cdot b_1$. We define now, for all $b_* \in \text{supp}(t)$ the set $\text{Res}_b(b_*) \subseteq \text{supp}(t')$ of residuals of b_* after reduction at position b . The fact that we define $\text{Res}_b(b_*)$ as a subset (and not an element) of $\text{supp}(t')$ can be related to the possible duplication of the argument s during reduction.

- During reduction, the application and abstraction (of respective positions b and $b \cdot 1$) are “destroyed”, so that we set $\text{Res}_b(b) = \text{Res}_b(b \cdot 1) = \emptyset$. Likewise, the nodes labelled with x that are replaced by the tree of s are destroyed, so we set $\text{Res}_b(b \cdot 10 \cdot b_i) = \emptyset$ ($i = 1, 2, 3$).
- The symbol \heartsuit represents a position nested in r (that is not in $\mathfrak{o}_\beta^t(b)$), so that its position is of the form $b \cdot 10 \cdot b_\heartsuit$. Since the application and the abstraction of the redex are destroyed, its position after reduction will be $b \cdot b_\heartsuit$. We set then $\text{Res}_b(b \cdot 10 \cdot b_\heartsuit) = \{b \cdot b_\heartsuit\}$.
- The argument s , at position $b \cdot 2$ is duplicated 3 times during reduction. The duplications will take the former positions of x i.e. $b \cdot b_1$, $b \cdot b_2$ and $b \cdot b_3$ (since the infix 01 is destroyed during reduction). Thus, we set $\text{Res}_b(b \cdot 2) = \{b \cdot b_i \mid i \in \{1, 2, 3\}\}$. More generally, the symbol \clubsuit represents a position nested in the argument s , so that its position is of the form $b \cdot 2 \cdot b_\clubsuit$. It will be duplicated 3 times and its residuals will be at positions $b \cdot b_1 \cdot b_\clubsuit$, $b \cdot b_2 \cdot b_\clubsuit$, $b \cdot b_3 \cdot b_\clubsuit$. We set $\text{Res}_b(b \cdot 2 \cdot b_\clubsuit) = \text{Res}_b(b \cdot 2) \cdot b_\clubsuit = \{b \cdot b_i \cdot b_\clubsuit \mid i \in \{1, 2, 3\}\}$
- If $b_* \not\preceq b$, then b_* is a position outside the redex to be fired and is not affected by reduction, so that we set $\text{Res}_b(b_*) = \{b_*\}$.

In order to state the complete definition, for $t \in \Lambda$ and $b \in \text{supp}(t)$ such that $t|_b$ is a redex $(\lambda x.r)s$, we set $\text{pf}_\beta^t(b) = \{b_* \in \{0, 1, 2\}^* \mid b \cdot 10 \cdot b_* \in \mathfrak{o}_\beta^t(b)\}$: it is the set of postfixes of the $b_0 \in \mathfrak{o}_\beta^t(t)$ w.r.t. position $b \cdot 10$ (root of r) and we have $\text{pf}_\beta^t(b) \subset \text{supp}(r)$.

Definition 2.2. Let t be a term and $b \in \text{supp}(t)$ such that $t|_b$ is a redex. For all $b_* \in \text{supp}(t)$, we define $\text{Res}_b^t(b_*)$ the set of residuals of b in t below:

- If $b_* \not\preceq b$, then $\text{Res}_b^t(b_*) = \{b_*\}$.
- $\text{Res}_b^t(b) = \text{Res}_b^t(b \cdot 1) = \emptyset$
- If $b_* \in \mathfrak{o}_\beta^t(b)$, $\text{Res}_b^t(b_*) = \emptyset$.
- If $b_* = b \cdot 10 \cdot b_\heartsuit$ for some $b_\heartsuit \in \{0, 1, 2\}^*$ and $b_* \notin \mathfrak{o}_\beta^t(b)$, then $\text{Res}_b^t(b_*) = \{b \cdot b_\heartsuit\}$ (paradigm \heartsuit).
- If $b_* = b \cdot 2 \cdot b_\clubsuit$ for some $b_\clubsuit \in \{0, 1, 2\}^*$, then $\text{Res}_b^t(b_*) = b \cdot \text{pf}_\beta^t(b) \cdot b_\clubsuit$ (paradigm \clubsuit).

Residuation preserves labelling and two distinct positions cannot have a common residual (“pseudo-injectivity” of residuation):

Lemma 2.1. Let t, t' be two terms and $b \in \text{supp}(t)$ such that $t \xrightarrow{b} t'$. Then:

- For all $b_* \in \text{supp}(t)$, for all $b'_* \in \text{Res}_b^t(b_*)$, we have $t'(b'_*) = t(b_*)$.
- If $b_1 \neq b_2$, then $\text{Res}_b^t(b_1) \cap \text{Res}_b^t(b_2) = \emptyset$.

The notion of residual can be extended, yielding that of **quasi-residual**. Assume again that $t|_b = (\lambda x.r)s$ and $t \xrightarrow{b} t'$. The nodes labelled with x are destroyed in t' , but they are replaced by copies of s . In that respect, they are still present in the term t' . Likewise, the subterm $(\lambda x.r)s$ at position b is destroyed, but it is replaced by its computation $r[s/x]$, still at position b . We set then:

- For all $b_* \in \text{supp}(t)$ such that $\text{Res}_b^t(b_*) \neq \emptyset$, $\text{QRes}_b^t(b_*) = \text{Res}_b^t(b_*)$.
- For all $b_* \in \text{pf}_\beta^t(b)$, $\text{QRes}_b^t(b \cdot 10 \cdot b_*) = \{b \cdot b_*\}$ (paradigm \heartsuit).
- $\text{QRes}_b^t(b) = \{b\}$.

We still have $\text{QRes}_b^t(b \cdot 1) = \emptyset$. The above lemma is false for quasi-residuals *e.g.*, if $t = (\lambda x.x)y \xrightarrow{\varepsilon} y = t'$, then $\varepsilon \in \text{QRes}_\varepsilon^t(\varepsilon)$, but $t(\varepsilon) = @ \neq y = t'(\varepsilon)$.

For instance, if $t = y((\lambda x.y x x)z z) \xrightarrow{2} y(y(z z)z z) = t'$, we check that:

- Out of the redex: $\text{Res}_2^t(1) = \{1\}$ (label y)
- In the left-hand side of the redex (paradigm \heartsuit): $\text{Res}_b^t(2 \cdot 1 \cdot 0) = \{2\}$ (label $@$), $\text{Res}_2^t(2 \cdot 1 \cdot 0 \cdot 1^2) = \{2 \cdot 1^2\}$ (label y).
- In the argument of the redex (paradigm \clubsuit) $\text{Res}_b^t(2^2) = \{2^2, 2 \cdot 1 \cdot 2\}$ (label $@$), $\text{Res}_b^t(2^2 \cdot 1) = \{2^2 \cdot 1^2, 2 \cdot 1 \cdot 2 \cdot 1\}$ (label z)

2.1.6 Reduction Sequences and Residuation

Let us generalize the notion of residuals along a reduction sequence: for that, we briefly the following notation: for all term t and $b \in \text{supp}(t)$ such that $t|_b$ is a redex, we write $\text{Red}_b(t)$ for the *unique* term t' such that $t \xrightarrow{b} t'$.

Formally, a **reduction sequence** w.r.t. a term t is a sequence $\mathbf{rs} = (b_i)_{i < n}$ such that $n \in \mathbb{N}$ or $n = \infty$ and, for all $i < n$, $t_i|_{b_i}$ is a redex, where $(t_i)_{i < n}$ is the sequence of terms defined by $t_0 = t$ and $t_{i+1} = \text{Red}_{b_i}(t_i)$ for all $i < n - 1$. In that case, n is the *length* of the reduction sequence \mathbf{rs} and the sequence $t = t_0 \xrightarrow{b_0} t_1 \xrightarrow{b_1} t_2 \xrightarrow{b_2} \dots$ is called a *reduction path*. When n is *finite*, we write $\text{Red}_{\mathbf{rs}}(t)$ for the aforementioned term t_n .

With the same notations, when the length n is *finite*, we can inductively define the set of *residuals* of a position b in a term t after the reduction sequence \mathbf{rs} :

- If $n = 0$, then we set $\text{Res}_{\mathbf{rs}}^t(b_0) = \{b_0\}$ for all $b_0 \in \text{supp}(t)$.
- For all $b_0 \in \text{supp}(t)$ and b redex position of t_n , we set $\text{Res}_{\mathbf{rs}.b}^t(b_0) = \cup_{b_* \in \text{Res}_{\mathbf{rs}}^t(b_0)} \text{Res}_b^{t_n}(b_*)$.

We can proceed likewise to extend quasi-residuation to *finite* reduction sequences.

2.1.7 Contexts

Let \square be a new constant symbol, intuitively representing a box inside which a λ -term can be placed. The set of **contexts** (metavariable \mathbf{C}) is inductively defined by the following grammar:

$$\mathbf{C} = \square \mid (\lambda x.\mathbf{C}) \mid (\mathbf{C}u) \mid (t\mathbf{C})$$

Thus, a context is like a λ -term along with the additional variable \square except that \square cannot be bound and must occur exactly once in a context. We use the same simplifications in notations as for λ -terms (see 2.1.2).

Let \mathbf{C} be a context and t a term: we define the term $\mathbf{C}[t]$ (t in the context \mathbf{C}) by induction: (1) $\mathbf{C}[t] = t$ if $\mathbf{C} = \square$ (2) $(\lambda x.\mathbf{C})[t] = \lambda x.\mathbf{C}[t]$ (3) $(\mathbf{C}u)[t] = (\mathbf{C}[t]u)$ (4) $(u\mathbf{C})[t] = u\mathbf{C}[t]$. Intuitively, $\mathbf{C}[t]$ is \mathbf{C} in which the box \square was replaced by t . The double brackets $[\cdot]$ indicate that capture is possible.

We may define α -equivalence for contexts as we did for λ -calculus in Sec. 2.1.2. Notice that no α -conversion may be performed in \mathbf{C} when we compute $\mathbf{C}[t]$, since we may want to allow capture in this term.

However, the **capture free** substitution $\mathbf{C}[[t]]$ is defined like $\mathbf{C}[t]$ except we must α -convert \mathbf{C} so that, for all variable $x \in \mathcal{V}$ occurring free in t , λx must not occur in \mathbf{C} .

For instance, if $\mathbf{C} = \lambda x.\square x$, then $\mathbf{C}[x] = \lambda x.xx$ but $\mathbf{C}[[x]] = \lambda y.xy$.

We now present an in-between notation. Let $\mathcal{S} \subseteq \Lambda$ be a set of terms. We write $\mathbf{C}^{\mathcal{S}}$ for a term context \mathbf{C} which does not capture the free variables of terms in \mathcal{S} *i.e.* there are no abstraction symbols in the context that bind the symbols of a term in \mathcal{S} (if λx is in \mathbf{C} , then $x \notin \text{fv}(t)$ for all $t \in \mathcal{S}$). For instance $\mathbf{C} = \lambda y.\square$ can be specified as TT^x while $\mathbf{C} = \lambda x.\square$ cannot. We may omit \mathcal{S} when it is clear from the context.

Remark 2.2 (Notation choices and capture). As many authors in λ -calculus, we chose the notation $[u/x]$ to denote a *capture-free* substitution (e.g., $t[u/x]$ is the capture-free substitution of x by u in t). But, using notations coming from the rewriting community, when \mathbf{C} is a context, the notation $\mathbf{C}[u]$ (also using simple brackets) allows capture whereas $\mathbf{C}[[u]]$ does not. This is a bit unfortunate, but note that the capture-free substitution $t[u/x]$ and the “capture-allowing” substitution $\mathbf{C}[u]$ may be told apart not only thanks to the metavariables (t vs. \mathbf{C}) but because of the $'/x'$ in the former notation.

2.2 Normalizations and Reduction Strategies

For λ -terms, a β -reduction step may be seen as an *execution* step. From that perspective, a term is said to be **normalizing** if it terminates. Normalization is a *dynamic* property (in the sense of Sec. 2.1.4): in order to know whether a term is normalizing or not, we have to reduce it over and over, till we reach – or not – a final state.

Thus, finding out whether a term normalizes is an instance of the *Halting Problem*. Since pure λ -calculus is Turing-complete, this is semi-decidable for pure λ -terms.

Internally to the λ -calculus, the notion of normalization immediately raises three questions.

1. What is the final state of a λ -term? The most natural idea coming to mind is that the notion of final state corresponds to that of normal form (Sec. 2.1.3) *i.e.* a term that do not hold any redexes and thus cannot be executed anymore.

However, the choice of the set of normal forms as the set of final states can be relaxed, yielding different sets of normal forms *e.g.*, the set of Head Normal Forms (HNF) (that will be properly defined in Sec. 2.2.1). When there is an ambiguity, we call the terms that do not hold any redex the **β -Normal Forms**.

A set of normal forms is usually defined a set of λ -terms that do not hold some kind of redex (*e.g.*, head redexes for HNF), and that is closed under β -reduction. This will be made more precise in Sec. 2.3.1.

2. How may we execute terms *i.e.* reduce them? Indeed, β -reduction is not **deterministic** *i.e.* for a given term t , there may be several t' such that $t \rightarrow t'$. Thus, as soon as a term or one of its reducts contains more than one redex, it may be executed in different ways.

Usually, for a given set \mathcal{N} of normal forms, a term t is defined as \mathcal{N} -*normalizing* if there is at least one⁴ reduction path from t to an element of \mathcal{N} *e.g.*, a term is Head Normalizing (HN) if it can be reduced to a Head Normal Form (HNF).

For most choices of \mathcal{N} , there is actually a (deterministic) **reduction strategy** that yields a \mathcal{N} -NF from *any* \mathcal{N} -normalizing (*e.g.*, a term t is HN iff the so called *head reduction strategy* terminates for t).

As we will see in Chapter 3, type theory can simplify the exhibition of a reduction strategy in relation with \mathcal{N} -normalization for many choices of \mathcal{N} .

3. The last question is: can a λ -term reach distinct final states (depending on reduction paths)? For instance, some HN terms may be reduced to several HNF. But if

⁴This is not true for strong normalization, see Sec. 2.2.3.

t reduces to t_1 and t_2 , two distinct HNF, there always is a third HNF t_3 such that t_1 and t_2 may both be reduced to t_3 , intuitively meaning that the HNF reachable by reduction from a given term t may be pairwise equated modulo additional series of reductions: reduction cannot *separate* two reducts of a same term.

More generally, for all term t , if t reduces to t_1 and t_2 , then t_1 and t_2 have a common reduct t_3 . This property of β -reduction is known as the **confluence** or the **Church-Rosser property** of λ -calculus and it is usually proved using⁵ *parallel reduction*. Thus, since we assume that a set \mathcal{N} of normal forms must be closed under β -reduction, the notion of \mathcal{N} -NF of a given \mathcal{N} -normalizing term is (practically) unambiguous (modulo extra reduction steps). Moreover, still by confluence⁶, the set of \mathcal{N} -normalizing terms is stable under reduction.

2.2.1 Head Redexes, Head Normal Forms, Head Reduction

We define here *head normal forms*, the associated notion of normalization *i.e.* *head normalization*, and a *deterministic* reduction strategy *i.e.* *head reduction*, that turns out to be *complete* for head normalization.

Let t be a term. Intuitively, to identify the **head** of t , we start at the root, visit as many abstractions as possible, then as many application left-hand sides as possible. This means that we stop at a node labelled with x or with λx . In the first case, t is a head normal form and in the second one, we have reached the abstraction of a redex to be called the head redex of t .

Formally, let $p \geq 0$ be the maximal integer such that $0^p \in \text{supp}(t)$ (thus, t is of the form $\lambda x_1 \dots \lambda x_p.t_0$ for some $t_0 \in \Lambda$ that is not an abstraction). Then, let $q' \geq 0$ be the maximal integer such that $0^p \cdot 1^{q'} \in \text{supp}(t)$. We have two possible cases (Figure 2.9):

- $t(0^p \cdot 1^{q'}) = x$ for some $x \in \mathcal{V}$. Then, we set $q = q'$ and t is of the form $\lambda x_1 \dots \lambda x_p.x t_1 \dots t_q$ for some terms t_1, \dots, t_q (and with possibly $x = x_i$ for some $i \in \{0, \dots, p\}$).

In that case, we say that t is a **Head Normal Form (HNF)**, x is the **head variable** of t and t_1, \dots, t_q are the **head arguments** of this HNF.

- $t(0^p \cdot 1^{q'}) = \lambda x$ (for some $x \in \mathcal{V}$). Then, by maximality of p , we have $q' \geq 1$ and we set $q = q' - 1$ so that $t(0^p \cdot 1^q) = @$. Thus, $t|_{0^p \cdot 1^q}$ is a redex $(\lambda x.r)s$ and t is then of the form $\lambda x_1 \dots \lambda x_p.(\lambda x.r)s t_1 \dots t_q$.

In that case, we say that t is a **head reducible term** and that $(\lambda x.r)s$ is the **head redex** of t .

We call a HNF that is not an abstraction (*i.e.* a term of the form $x t_1 \dots t_q$) a **Zero Head Normal Form (ZHNF)** (why “zero”? See Definition 2.8) *e.g.*, $x(\lambda x.x)\Omega$ is a ZHNF whereas $\lambda x.x y y$ is not. HNF and ZHNF can be defined by the alternative inductive definition:

$$\frac{}{x \text{ is a ZHNF}} \quad \frac{t \text{ is a ZHNF}}{t u \text{ is a ZHNF}} \quad \frac{t \text{ is a ZHNF}}{t \text{ is a HNF}} \quad \frac{t \text{ is a HNF}}{\lambda x.t \text{ is a HNF}}$$

⁵This method is apparently due to Tait *cf.* [21], 7.2.

⁶Another very important consequence of confluence is that λ -calculus is **consistent**: the β -equivalence does not equate all the term *e.g.*, by confluence, two distinct normal forms cannot be β -equivalent.

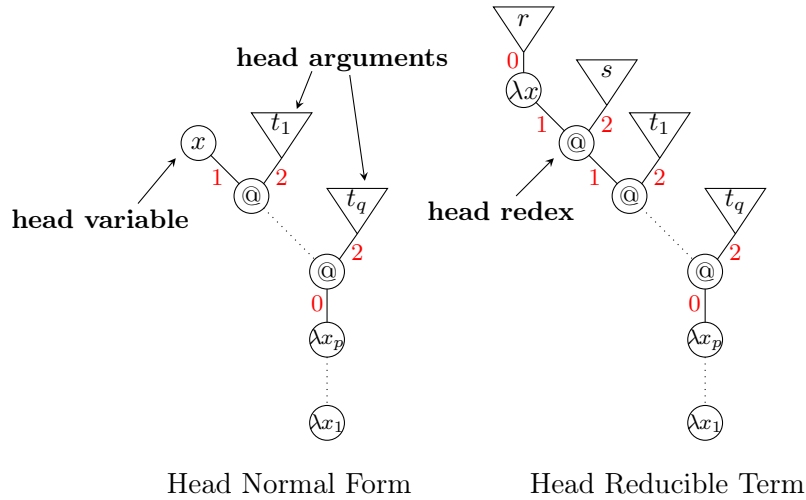


Figure 2.9: Head Forms

Definition 2.3. Let $t \in \Lambda$. Then t is **Head Normalizing (HN)** if there is a reduction sequence from t to a head normal form.

The **Head Reduction** is the restriction of β -reduction to the reduction of head redexes only. We write $t \rightarrow_h t'$ to mean that t head-reduces to t' . Since a term has at most one head redex, head reduction is deterministic. Given a term t , the **Head Reduction Strategy** consists in starting with t and performing head-reduction steps as long as it is possible. The head reduction strategy may be non-terminating and produce a reduction sequence of infinite length *e.g.*, $\Omega \rightarrow_h \Omega \rightarrow_h \Omega \rightarrow_h \dots$

Equivalently, if a term that is not an abstraction is called **neutral**, head reduction can be inductively defined by:

$$\frac{}{(\lambda x.r)s \rightarrow_h r[s/x]} \quad \frac{t \rightarrow_h t' \text{ and } t \text{ is neutral}}{tu \rightarrow_h t'u} \quad \frac{t \rightarrow_h t'}{\lambda x.t \rightarrow_h \lambda x.t'}$$

An important property of head reduction is its *completeness* for head normalization:

Proposition 2.1. For all term t , t is head normalizing iff the head reduction strategy terminates for t .

Proof. This proposition is a direct corollary of the Standardization Theorem (Sec. 11.4 of [8]), which was first proved by using syntactic methods involving residuals, developments and algorithmic transformations of reduction sequences.

Remarkably, a far simpler proof of the present proposition is given by Gardner and de Carvalho's type system \mathcal{B}_0 , although the statement of the present proposition does not concern type theory. We sketch this proof in Sec. 3.4. \square

2.2.2 Weak Normalization and Leftmost Reduction

Let t be a term and $b \in \text{supp}(t)$. Informally, when b is closed to the root (*i.e.* $|b|$ is small), b is regarded as an *outer* position of t , and when b is distant to the root (*i.e.* $|b|$ is high), b is regarded as an *inner* position. Thus, the more a position is nested in the

term, the more it is inner.

The following lemma states that normal forms are inductive assemblages of head normal forms: inductively, a term t is normal if it is a HNF and all its head arguments are themselves NF.

Lemma 2.2. The set of normal forms can be defined by the following inductive grammar:

$$t, u = \lambda x_1 \dots \lambda x_p. x t_1 \dots t_q \quad (p, q \geq 0)$$

Proof. This lemma is proved by double inclusion, each inclusion being proved inductively. \square

Remark 2.3. We could give a more elementary inductive grammar for normal forms. Let us call a **Zero Normal Form (ZNF)** a normal form that is not an abstraction. Then, the sets of NF and ZNF are defined by induction as follows:

$$\frac{}{x \text{ is a ZNF}} \quad \frac{t \text{ is a ZNF} \quad u \text{ is a NF}}{t u \text{ is a ZNF}} \quad \frac{t \text{ is a ZNF}}{t \text{ is a NF}} \quad \frac{t \text{ is a NF}}{\lambda x. t \text{ is a NF}}$$

Definition 2.4. Let $t \in \Lambda$. Then t is **Weakly Normalizing (WN)** if there is a reduction sequence from t to a normal form.

For instance, the term $(\lambda x. y)\Omega$, that reduces to y , is WN. Likewise, the term $I^n = \underbrace{I \dots I}_{n \text{ occ. of } I}$ is WN since $I^n \rightarrow^{n-1} I$, which is a normal form.

The terms Ω , Y , Y_f , Y_λ and Ω_3 from Sec. 2.1.4 are not WN, whereas the term $\lambda x. y \Omega$ is an example of a head normal form which is not WN, since the redex Ω cannot be eliminated by reduction.

The **Leftmost (Outermost) Reduction Strategy**, starting from a given term t , consists in keeping reducing the leftmost-outermost redex, as long as it is possible. This redex, when it exists, is defined as the one whose position is minimal for the *lexicographical* order. From the tree perspective, this redex is the leftmost and outermost one.

If a term is head reducible, then its head redex is its leftmost one. Thus, the leftmost reduction strategy extends the head reduction strategy. It is deterministic as well and we write $t \rightarrow_\ell t'$ to mean that t' is the (one step) left reduct of t .

Equivalently, leftmost outermost reduction can be inductively defined by:

$$\frac{}{(\lambda x. r)s \rightarrow_\ell r[s/x]} \quad \frac{t \rightarrow_\ell t'}{t u \rightarrow_\ell t' u} \quad \frac{t \text{ is a ZNF} \quad u \rightarrow_\ell u'}{t u \rightarrow_\ell t u'} \quad \frac{t \rightarrow_\ell t'}{\lambda x. t \rightarrow_\ell \lambda x. t'}$$

From this definition, we see that, when a λ -terms is considered as a string of characters, the leftmost-outermost redex is indeed the leftmost one *i.e.* the redex whose abstraction is leftmost.

Leftmost reduction is complete for weak normalization:

Proposition 2.2. For all term t , t is weakly normalizing iff the leftmost reduction strategy terminates for t .

Proof. This proposition also is a direct corollary of the Standardization Theorem (11.4 in [8]). \square

Remark 2.4. As for Proposition 2.1, a simple proof of this proposition can be given using type theory (Sec. 3.4.4).

The notion of NF can be inductively defined (Lemma 2.2), but the above proposition allows us to reformulate the definition of weak normalization inductively via the leftmost reduction strategy:

Corollary 2.1. A term t is weakly normalizing iff t is head normalizing and all the head arguments of its HNF are themselves WN.

When induction is replaced by coinduction, weak normalization becomes hereditary head normalization, that is studied in Chapter 10 of Part III.

2.2.3 Strong Normalization

Definition 2.5. Let $t \in \Lambda$. Then t is **Strongly Normalizing (SN)** if there is no infinite reduction path starting from t .

Thus, a term t is SN if all the reduction paths starting from t terminate. In particular, if t is SN, then t is WN. But the converse implication is not true *e.g.*, $t := (\lambda x.y)\Omega$ is weakly but not strongly normalizing, since $t \xrightarrow{\varepsilon} y$ but the reduction path $t \xrightarrow{2} t \xrightarrow{2} t \xrightarrow{2} \dots$ is infinite. In the last example, the fact that the (non-terminating) argument Ω of the redex is *erased* (see Sec. 2.1.4) plays a pivotal role, that will have to be addressed in order to understand how type theory may characterize strong normalization (Sec. 5.2).

Actually, those two terms t and y also show that the set of the strongly normalizing terms is not stable under *expansion*, meaning that there are some terms t and t' such that $t \rightarrow t'$, t' is SN but t is not. In contrast to that, by Definitions 2.3 and 2.4, the sets of head and weakly normalizing terms are stable under expansion. The set of SN terms does not correspond to the general definition of \mathcal{N} -normalization given in Sec. 2.2, that yields sets of normalizing terms which are stable under expansion.

Remark 2.5. A reduction strategy \mathcal{S} satisfying “For all term t , the strategy \mathcal{S} terminates on t iff t is SN, and in that case, outputs the normal form of t ” (*i.e.* a reduction strategy that is *complete* for strong normalization) is said to be **perpetual**. Equivalently, \mathcal{S} is perpetual if (1) $t \rightarrow_{\mathcal{S}} t'$ implies that t' is not SN when t is not SN and (2) if t is SN and reducible, there exists t' such that $t \rightarrow_{\mathcal{S}} t'$. Even though strong normalization is quite different from head or weak normalization, one may prove that such strategies exist (in many variants) *e.g.*, [112].

König’s Lemma states that every finitely branching (every node has a finite number of children) and well-founded (there is not infinite branches) tree is finite. This entails that, given a strongly normalizing term t , the reduction sequences starting at t are bounded in length, so that we may set:

Notation 2.2. Let t be a strongly normalizing term. Then, the maximal length of a reduction sequence starting at t is denoted $\eta(t)$.

2.3 Tinkering with Normalization

In this section, we explore different aspects of normalization and of reduction strategy. To do that, we first exhibit the notion of *stable positions* of a term (Sec. 2.3.1) *i.e.*

positions that cannot be affected by reduction in a sense to be defined. In Sec. 2.3.2, we define the *mute terms*, that are the terms in which no position may be “stabilized”. For this reason, those terms are seen as very undefined. In 2.3.3, the notion of stable positions leads us to notice that some λ -terms do not weakly normalize, but produce asymptotically an “infinite normal form”. This is a first glimpse at infinitary normalization. Some parts of a term may be already in normal form, which yields the notion of *partial normal form* in Sec. 2.3.4. This allows us to outline some defects of the leftmost outermost strategy, leading to define the **Böhm** reduction strategies and have a first look at Böhm trees.

2.3.1 Stable Positions and Sets of Normal Forms

In this section, we consider again the terms defined in Sec. 2.1.4.

As discussed at the beginning of Sec. 2.2, normal forms are regarded as *final states* of λ -terms (indicating the end of an execution) but there are many candidates for the set \mathcal{N} of normal forms.

Some candidates are even bigger than the set of head normal forms, such as the set of **Weak Head Normal Forms (WHNF)**: a term is a WHNF if it is an abstraction or a HNF. For instance, $t = \lambda x.\Omega$ is a WHNF whereas it is not HN, and the zero head normal forms (*e.g.*, $x\Omega$) are the WHNF that are not abstractions.

Notice that the set of WHNF is indeed stable under reduction: if $t = \lambda x.t_0$ for some t_0 , then any of the reducts of t will be of the form $t' = \lambda x.t'_0$ where t'_0 is a reduct of t_0 .

So, can we give some more intuitions about what may be regarded as a *final state*? Heuristically, a term may be regarded as final if some of its parts are stabilized. Let us give a formal definition of stability:

Definition 2.6.

- Let t be a term and $b \in \mathbf{supp}(t)$. The position b is **stable** in t if for all residuals b' of b in a reduct t' of t , b' is not nested in a redex of t' .
- Let t be a term. Then t is **root-stable** if ε is stable in t .
- The set of stable positions in t is denoted $\mathbf{stab}(t)$.

Thus, a position in a term is stable when it cannot be affected by reduction. More formally, b_0 is stable in t iff, for all reduction sequence \mathbf{rs} starting from t and $b'_0 \in \mathbf{Res}_{\mathbf{rs}}^r(b_0)$, there is no $b' \leq b'_0$ such that $t'|_{b'}$ is a redex, where $t' = \mathbf{Red}_{\mathbf{rs}}^t(t)$ (notation of Sec. 2.1.6).

For instance, if $t = \lambda x_1 \dots x_p. x t_1 \dots t_q$ is a HNF, then the positions $\varepsilon, 0, 0^2, \dots, 0^p, 0^p \cdot 1, \dots, 0^p \cdot 1^2, \dots, 0^p \cdot 1^q$, labelled with $\lambda x_1, \dots, \lambda x_p, @$ and x , are stable (see Figure 2.9).

If t is an abstraction *i.e.* $t = \lambda x.t_0$, then ε is stable *i.e.* the root of t is stable.

Notice that, although Ω (that reduces to itself) is “graphically” invariant under reduction, no position of Ω is stable since Ω is a redex.

The term $t = (\lambda x.x)(\lambda x.x)y$ is a term that is not root-stable and not a redex: we have $t \xrightarrow{1} (\lambda x.x)y \xrightarrow{\varepsilon} y$, so that ε is not stable in t .

Remark 2.6. If b is stable in t , then, for all reduction sequence \mathbf{rs} starting from t , we have $\mathbf{Res}_{\mathbf{rs}}^t(b) = \{b\}$, but this is not a sufficient condition. Indeed, let $t = (\lambda x.yx)z \xrightarrow{\varepsilon} yz$. We have $\mathbf{Res}_b^t(2) = \{2\}$ (label z) and this is the only (non-empty) reduction path starting from t , but 2 is not stable in t since t is a redex.

The following observation, obtained by induction on the length of a reduction sequence, is useful:

Lemma 2.3.

- If b is stable in t , then any prefix of b also is.
- If t is not root-stable, no position $b \in \text{supp}(t)$ is stable in t .
- A term t is root-stable iff t may not be reduced to a redex.

In conclusion, a set of normal forms can usually be defined as a set of terms in which some positions are stable. Formally, for any $B \subseteq \{0, 1, 2\}^*$, let \mathcal{N}_B be the set of terms such that, for all $b \in B \cap \text{supp}(t)$, b is stable in t . Then, \mathcal{N}_B is stable under reduction (by Definition 2.6) and for $B = \{0, 1\}^*$, \mathcal{N}_B is the set of HNF, for $B = \{0, 1, 2\}^*$, \mathcal{N}_B is the set of β -NF and for $B = \{1\}^*$, \mathcal{N}_B is the set of WHNF (see below).

Choice of B	The normal forms are...
$\{0, 1, 2\}^*$... the β -normal forms
$\{0, 1\}^*$... the head normal forms
$\{1\}^*$... the weak head normal forms

2.3.2 Mute Terms and Order of a Lambda Term

The *mute* terms, introduced by Berarducci [11], correspond to the terms that cannot be defined as normal forms, for any reasonable choice of \mathcal{N} . They have been referred as the “most undefined lambda terms” [15]. Let us understand why.

Definition 2.7. Let $t \in \Lambda$. Then t is **mute** if every reduct of t may be reduced to a redex.

Formally, t is mute iff, for all $t' \in \Lambda$ such that $t \rightarrow^* t'$, there exists $(\lambda x.r)s$ such that $t' \rightarrow^* (\lambda x.r)s$. Thus, mute terms can be regarded as *persisting redexes*.

For instance, Ω is a mute term since its unique reduct – Ω itself – is a redex. Buciarrelli, Carraro, Favro, Salibra [15] defined the class of n -regular mute terms, generalizing the construction of Ω (the reducts of those terms are redexes each n head reductions steps *e.g.*, Ω and $(\lambda x.(\lambda y.yx))\Delta$ are 1-regular mute terms and AAA (with $A = \lambda xy.x(\lambda zt.tzx)y$) is a 2-regular mute term whereas BB (with $B = \lambda x.x(\lambda y.xy)$) is a mute term that is not regular (those examples were found in [15]).

In contrast, the term Y_λ from Sec. 2.1.4, satisfying $Y_\lambda \xrightarrow{\varepsilon} \lambda x.Y_\lambda$ is not mute since $\lambda x.Y_\lambda$ is an abstraction and cannot be reduced to a redex (all its reducts are of the form $\lambda x^n.Y_\lambda$ with $n \geq 1$). The reducts of Ω_3 are of the form $\Omega_3 \omega_3^n$ with $n \geq 1$. No one is a redex, so Ω_3 is not mute.

By Lemma 2.3, a term is mute when no one of its reduct is root-stable and muteness can be reformulated as follows:

Lemma 2.4. A term is mute iff no reduct of t has a stable position.

Let us say that a position $b \in \{0, 1, 2\}^*$ is **stabilizable** or **may be stabilized** in t if there is $t \rightarrow^* t'$ such that $b \in \text{supp}(t')$ and b is stable in t' . Then, by the above Lemma, a term is mute iff no position can be stabilized in t . If we follow the discussion ending the previous section, this justifies why mute terms are improper to be regarded as normal forms.

Order of a Lambda-Term and Stabilization An interesting *semantical* aspect of λ -terms is their orders: the notion of *order* of a λ -term will be studied in later parts of this document (see Chapter 12 and Theorem 12.2). It is also related to *stable positions*.

Let t be a term. If t is an abstraction *i.e.* $t = \lambda x.t_0$ for some t_0 , then any of the reduct of t will be of the form $t' = \lambda x.t'_0$ where t'_0 is a reduct of t_0 *i.e.* the root of an abstraction⁷ is a stable position. This suggests the notion of *order* of a λ -term t , defined as the supremal number of abstractions that prefix a reduct of t :

Definition 2.8.

- Let t be a λ -term. The **order** of t is

$$\sup\{n \in \mathbb{N} \mid \exists x_1, \dots, x_n, u \text{ s.t. } t \rightarrow_{\beta}^* \lambda x_1 \dots \lambda x_n. u\}$$

- A term of order 0 is called a **zero term**

By confluence, if the order of t is n , then the order of any reduct of t is also n .

For instance, a term that reduces to a zero head normal form (Sec. 3.7) is indeed a zero term. The order of a head normalizing term is the order of its head normal form.

Let us have a look at the orders of some terms of Sec. 2.1.4:

- The mute term $\Omega = \Delta \Delta$, is a zero term (its unique reduct is Ω which is not an abstraction). More generally, if a term t is of order $n \geq 1$, then the position 0^n can be stabilized in a reduct of t , so that a mute term is of order 0.
- The term $\lambda x.\Omega$ is of order 1 (its unique reduct is $\lambda x.\Omega$). Since Y_f head normalize to $f(Y_f)$ (zero head normal form), then Y_f is a zero term, so that $Y = \lambda f.Y_f$ (the Curry Fixpoint Combinator) is of order 1.
- The reduct of term Ω_3 (which is not HN) are of the form $\Omega_3 \omega_3^k$. No one is an abstraction, so that Ω_3 is a zero term, that is neither a mute nor a head normalizing term.
- Since $Y_\lambda \rightarrow \lambda x.Y_\lambda$ and $Y_\lambda \rightarrow^n \lambda x^n.Y_\lambda$, the order of Y_λ is infinite.

2.3.3 Toward Infinitary Normalization

The study of some terms of Sec. 2.1.4 suggests other variants of normalization.

We recall that Y_f , Y_λ and Ω_3 satisfy respectively $Y_f \xrightarrow{\varepsilon} f(Y_f)$, $Y_\lambda \xrightarrow{\varepsilon} \lambda x.Y_\lambda$ and $\Omega_3 \xrightarrow{\varepsilon} \Omega_3 \omega_3$. Let Δ'_f and Y'_f be defined by $\Delta'_f = \lambda x.x x f$ and $Y'_f = \Delta'_f \Delta'_f$, so that $Y(\lambda x.x f) \rightarrow^3 Y'_f$ and $Y'_f \xrightarrow{\varepsilon} Y'_f f$.

Neither Y_f nor Y_λ , Ω_3 and Y'_f are weakly normalizing. Indeed, $Y_f \rightarrow_\ell^n f^n(Y_f)$, $Y_\lambda \rightarrow^n \lambda x^n.Y_\lambda$, $\Omega_3 \rightarrow_\ell^n \Omega_3 \omega_3^n$ and $Y'_f \rightarrow_\ell^n Y'_f f^n$.

Let us first consider Y_f . Although the left reduction starting from Y_f does not terminate, we notice that, along the left reduction path, more and more positions are stabilized. In Figure 2.10, all the apparent nodes (*i.e.* the nodes outside u_1 , u_2 and u_3) are stabilized.

⁷Thus, abstractions are good candidates to be considered as finale states *i.e.* a normal forms. For instance, in the so-called **call-by-value** evaluation strategy, they are precisely completed computations *i.e.* normal forms.

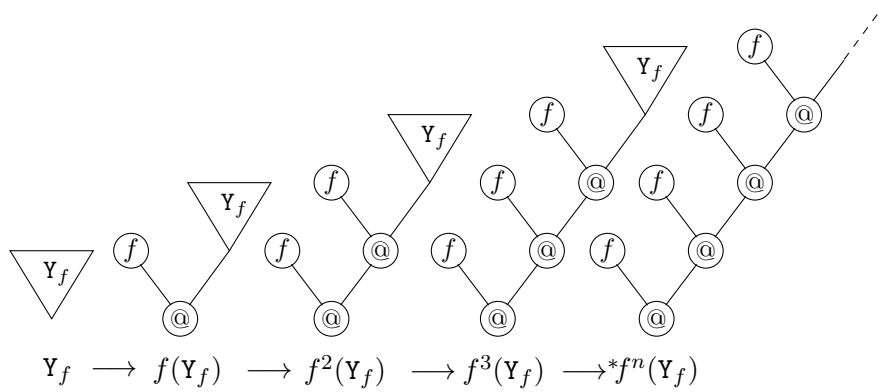


Figure 2.10: Reducing Y_f

Actually, any position of the form 2^i or $2^i \cdot 1$ with $i \leq n$ is stable in $f^{n+1}(Y_f)$ (in particular, we have $(f^k(Y_f))(2^n) = @$ and $(f^k(Y_f))(2^n \cdot 1) = f$ for all $k \geq n + 1$).

Thus, any position is stabilized in $f^n(Y_f)$, the rank n reduct of Y_f , above a certain reduction rank. So Y_f is not normalizing, but intuitively, after an *infinite* number of reduction steps, the redex Y_f disappears and $f^n(Y_f)$ “converges” toward an *infinite* term, that we call f^ω , whose tree could be the obtained by repeating infinitely many times the pattern of the tree on the right-hand side of Figure 2.10. The term f^ω can be written $f(f(\dots))$ and we notice that $f^\omega = f(f^\omega)$ (whereas $Y_f \equiv_\beta f(Y_f)$). Intuitively, f^ω does not contain any redex (compare with Figure 2.7) and may be thus considered as the *infinite* normal form of the term Y_f .

More generally, when we keep reducing Y_λ , Ω_3 and Y'_f , we notice that the lower parts of their trees also stabilize. Asymptotically, every position is stabilized and we obtain the infinite terms $\text{NF}_\infty(Y_\lambda) = \lambda x.\lambda x.\lambda x \dots$, $\text{NF}_\infty(\Omega_3) := ((\dots)\omega_3)\omega_3$ and $\text{NF}_\infty(Y'_f) = ((\dots)f)f$ as their respective normal forms. Their trees can also be seen as infinite repetitions of a same pattern (Figure 2.11) and they satisfy the following fixpoint equalities $\text{NF}_\infty(Y_\lambda) = \lambda x.\text{NF}_\infty(Y_\lambda)$, $\text{NF}_\infty(\Omega_3) = \text{NF}_\infty(\Omega_3)\omega_3$ and $\text{NF}_\infty(Y'_f) = \text{NF}_\infty(Y'_f)f$, whereas $Y_\lambda \equiv_\beta \lambda x.Y_\lambda$, $\Omega_3 \equiv_\beta \Omega_3 \omega_3$ and $Y'_f \equiv_\beta Y'_f f$.

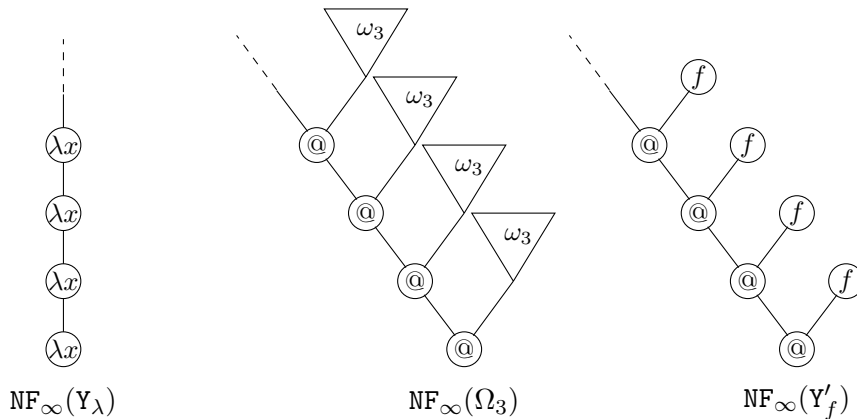


Figure 2.11: Infinite Normal Forms

All this suggests that there is a notion of normalization for which $Y := \lambda f.Y_f$, Y_λ , Ω_3 and Y'_f are normalizing terms w.r.t. some infinite term calculus. Historically, Böhm trees (Chapter 10 of [8]) were the first infinitary normal forms to be introduced, followed by Lévy-Longo [1, 74, 75] and Berarducci [11] trees. Then Kennaway, Klop, Sleep and de Vries [57] built a general framework, featuring infinitary reduction sequence and subsuming these three cited kind of trees. This framework contains seven variant of infinitary λ -calculus and two of these calculi are presented in Chapter 9.

2.3.4 Partial Normal Forms

In this section, we develop the notion of **partial normal forms** *i.e.* terms whose outer parts are already in normal form (the tree of such a term begins with a nesting of head normal forms) but whose inner parts are not. Thus, a partial normal form is a term in-between a head normal form and a β -normal form (a term without redex), that may be met in the course of a normalizing reduction sequence (or at least, a reduction sequence *trying* to normalize a term).

Normal forms are inductive assemblages of head normal forms (Sec. 2.2.2). When t is a normal form, then for all $x \in \mathcal{V}$ and $b \in \text{supp}(t)$ such that $t(b) = x$, we call x the **local head variable** of t at position b . If $t(b) = x$, there is a maximal $q \geq 0$ such that $b = b_0 \cdot 1^q$ for some $b_0 \in \text{supp}(t)$. In that case, $t|_{b_0} = x t_1 \dots t_q$ for some t_1, \dots, t_q , that we call the **arguments** of x . Thus, each variable of a normal form t corresponds to an inductive “call” of Lemma 2.2.

We recall that a zero head normal form (Sec. 2.2.1) is a term of the form $x t_1 \dots t_q$. Let t be a term. We define the set $\text{stab}_B(t)$ of **Böhm stable positions** of t as the subpart of t (*i.e.* $\text{stab}_B(t) \subseteq \text{supp}(t)$) that is “already” in normal form, upwards from the root of the term *i.e.* inductively, we set:

- If t is not an HNF, $\text{stab}_B(t) = \emptyset$.
- If $t = x$, then $\text{stab}_B(t) = \{\varepsilon\}$.
- If t is a ZHNF, then $\text{stab}_B(t u) = \{\varepsilon\} \cup 1 \cdot \text{stab}_B(t) \cup 2 \cdot \text{stab}_B(u)$.
- If $t = \lambda x.t_0$, then $\text{stab}_B(t) = \{\varepsilon\} \cup 0 \cdot \text{stab}_B(t_0)$.

The set $\text{stab}_B(t)$ is closed under the prefix order, if b is a leaf of $\text{stab}_B(t)$, then b is a leaf of $\text{supp}(t)$ (*i.e.* $t(b) = x$ for some $x \in \mathcal{V}$) and if t is a normal form, then $\text{stab}_B(t) = \text{supp}(t)$. Moreover, for all $b \in \text{stab}_B(t)$, $t|_b$ is a head normal form. Equivalently, a position b of t is Böhm -stable iff b , as well as all its prefixes, are the positions of head normal subterms of t . Thus, the Böhm stable positions of t corresponds to the positions of the head normal forms in t that are hereditarily nested in head normal forms.

Lemma 2.5. Let t be a λ -term and $b \in \text{supp}(t)$. Then b is Böhm -stable in t iff, for all $b_0 \leq b$, $t|_{b_0}$ is a head normal form.

For that reason, we say, for all Böhm stable position b points to a **Hereditarily Nested Head Normal Form (HNHNF)**.

Let u_1, u_2, u_3 be 3 terms that are *not* head normal forms. We set:

$$t = \lambda x_1 x_2. y (x (x u_1)) (\lambda x.x) u_2 (\lambda x.x_2 y u_3)$$

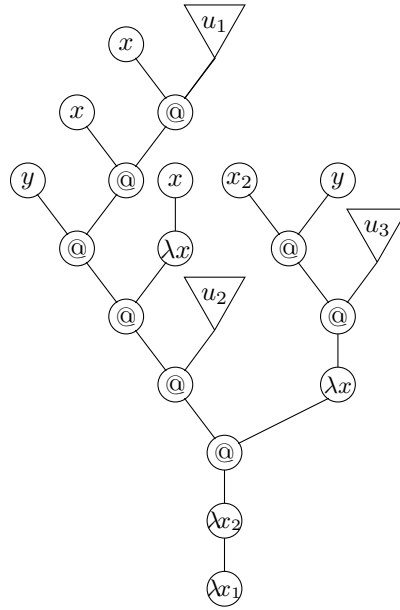


Figure 2.12: Minimal Redexes

Then $\mathbf{stab}_{\mathcal{B}}(t)$ corresponds to the set of the *apparent* nodes of Figure 2.12 (the nodes outside u_1 , u_2 and u_3).

Remark 2.7. The notion of *Böhm* stable position is more restrictive than that of stable position (Definition 2.6). For instance, ε is stable in $\lambda x.\Omega$ (see Sec. 2.3.1), but it is not Böhm stable, since $\lambda x.\Omega$ is not a head normal form, so that $\mathbf{stab}_{\mathcal{B}}(\lambda x.\Omega) = \emptyset$.

Intuitively, the minimal redexes of a term t would correspond to the leftmost outermost redexes of the subterms obtained if we removed the Böhm stable parts of t :

Definition 2.9. Let $t \in \Lambda$ and $b \in \mathbf{supp}(t)$

- There is a **maximal head reducible subterm** of t at position b if $t|_b$ is head reducible (*i.e.* $t|_b$ is not HNF) and for all $b_0 < b$, $t|_{b_0}$ is a head normal form.
- There is a **minimal redex** at position b in t if b is the position of head redex of a maximal head reducible subterm of t .

Equivalently, b is the position of a maximal head reducible subterm of t if $t|_b$ is head reducible and for all $b_0 < b$, $b_0 \in \mathbf{stab}_{\mathcal{B}}(t)$. When t is not a head normal form, its only maximal head reducible subterm is t itself.

Let us define the order $\leq_{\mathcal{B}}$ on $\{0, 1, 2\}^*$ (where \mathcal{B} stands for Böhm) by $b \leq_{\mathcal{B}} b'$ if

- $b \leq b'$ or ...
- ... $\mathbf{ad}(b) < \mathbf{ad}(b')$ and there is a $b_0 \leq b$ and $q \in \mathbb{N}$ such that $b = b_0 \cdot 1^q$ and $b_0 \leq b'$.

Let us remember that 1 points to the left hand-side of an application and assume that $b, b' \in \mathbf{supp}(t)$ correspond to two redexes of t and that $b \leq_{\mathcal{B}} b'$.

- If $b \leq_{\mathcal{B}} b'$ matches the first case of the definition of $\leq_{\mathcal{B}}$, then $t|_{b'}$ is nested in the redex $t|_b$.

- If $b \leq_{\mathcal{B}} b'$ matches the second case, then $t|_{b_0} = t|_b u_1 \dots u_q$ and b' is nested in one of the u_k . Thus, $t|_b$ is the head redex of $t|_{b_0}$ whereas $t|_{b'}$ is an *inner* redex of $t|_{b_0}$.

Thus, we observe that, in a reducible term t , a redex at position b is minimal in the sense of Definition 2.9 iff b is minimal among the positions of the redexes of t , which justifies the vocable.

A minimal redex is not nested in any other redex: thus, a minimal redex is an *outermost* redex, but the converse is not true *e.g.*, in $t := (\lambda x.r)s(\lambda y.u)v$, $(\lambda y.u)v$ is an outermost redex, but is not a minimal redex since $(\lambda y.u)v$ is not a maximal head reducible subterm of t (that is head reducible).

We call the **Minimal Reduction** the restriction of β -reduction to the reduction of outermost redexes only and we write $t \rightarrow_{\mathbf{m}} t'$ when t' is obtained by firing a minimal redex of t . Minimal reduction is not deterministic since a term can contain several minimal redexes *e.g.*, in Fig. 2.12, if u_1 , u_2 and u_3 are redexes, then t has 3 minimal redexes: u_1 , u_2 and u_3 themselves. The **Minimal Reduction Strategy** consists in starting with t and performing minimal reduction steps as long as it is possible

Since the leftmost outermost redex – when it exists – is the redex whose position is minimal for the lexicographical order (Sec. 2.2.2), there is a minimal redex at position b in t if (1) $t|_b$ is a redex (2) for all $b_0, b_* \notin \text{stab}_{\mathcal{B}}(t)$ such that $b_0 \leq b, b_*$ and $t|_{b_*}$ is also a redex, then $b \leq_{\ell} b_*$ (lexicographical order).

If t is not a normal form, then the leftmost outermost redex of t is a minimal one. With the example of Figure 2.12, the leftmost outermost redex of t is the head redex of u_1 , but t has exactly two more minimal redexes: the respective head redexes of u_2 and u_3 .

2.3.5 Böhm Reduction Strategies

In this section, we notice a defect of the leftmost outermost reduction strategy, thanks to the notions of maximal head reducible subterm (Sec. 2.3.4) and applicative depth (see below). This leads us to consider the Böhm reduction strategies, that try to normalize terms from the bottom to the top more “equitably” than the leftmost outermost one does. This will give a glimpse of finite or infinite Böhm trees.

The **applicative depth** $\text{ad}(b)$ of a subterm u at position b in a term t (with $b \in \text{supp}(t)$) is the number of nestings of u inside application arguments. Formally, we set $\text{ad}(b) = \#\{0 < i \leq \ell \mid b_i = 2\}$ where $\ell = |b|$ and $b = b_1 \cdot b_1 \cdot \dots \cdot b_\ell$ with $b_i \in \{0, 1, 2\}$. Inductively, $\text{ad}(\varepsilon) = 0$, $\text{ad}(b \cdot 0) = \text{ad}(b \cdot 1) = \text{ad}(b)$ and $\text{ad}(b \cdot 2) = \text{ad}(b) + 1$. For instance, $\text{ad}(0 \cdot 1) = 0$, $\text{ad}(2 \cdot 1 \cdot 0 \cdot 2) = 2$. Thus, $\text{ad}(b)$ is the number of nestings of b inside application *arguments*.

Notice that, if b is the position of the head (variable or redex) of t , then $\text{ad}(b) = 0$ (see Sec. 2.2.1). If $t = \lambda x_1 \dots x_p. x t_1 \dots t_q$, then the head arguments t_1, \dots, t_q are at applicative depth 1 (see Figure 2.9). In Figure 2.12, subterm u_1 occurs at applicative depth 3, so the leftmost-outermost redex of t also occurs at applicative depth 3. The maximal head reducible subterms u_2 and u_3 occur respectively at applicative depth 1 and 2.

Now, assume that u_2 and u_3 are both weakly normalizing and that u_1 is *not*. Since the head redex of u_1 is the leftmost outermost redex of t , then the leftmost outermost reduction strategy will keep on reducing the leftmost innermost redex of u_1 (and its head redexes) and will never perform reduction inside u_2 and u_3 , which is a pity, since these

two terms are normalizing. This motivates to define the *Böhm reduction strategy*, that we present under two variants.

Böhm Reduction Strategy Let t be a term.

- First variant: we keep on reducing a redex of minimal applicative depth (which one does not matter).
- Second variant: as long as the current term is not a normal form
 1. We list the current maximal head reducible subterms (whose redexes are at position b_1, \dots, b_n).
 2. We fire those redexes (the order in which these reductions are performed does not matter)
 3. We got back to step 1.

As the leftmost reduction strategy is, these two strategies are restriction of the minimal reduction strategy (Sec. 2.3.5). These two variants are usually not deterministic and if t is not weakly normalizing, they will not terminate. We shall prove (Sec. 5.1.4) that they are both complete for weak normalization, as the leftmost outermost reduction strategy is, but only they have a nice infinitary behaviour (see Part III, Remark 9.3 in particular).

With the first variant, we keep on head reducing t as long as we do not get a head normal form $\lambda x_1 \dots x_p. x t_1 \dots t_q$. Then we keep on head-reducing the head arguments t_1, \dots, t_q as long as they are not all in head normal form (applicative depth 1). Then, we keep on head-reducing their head arguments (applicative depth 2) as long as they are not all in head normal form. And so on. The strategy stops if we meet a normal form at some point.

For instance, if we consider the term t of Figure 2.12 and assume that t is weakly normalizing, then this strategy will not start by reducing inside u_1 (as would the leftmost outermost one) but by keeping on head-reducing u_2 (whose applicative depth is minimal) till we get a head normal form u'_2 . Then, we keep on reducing u_3 and the head arguments of u'_2 till they are put in head normal forms. Then, there are no more redexes at applicative depth 2 and we may start head-reducing u_3 and other remaining redexes of applicative depth 3.

Assume this time that u_1 and u_3 are weakly normalizing, but that u_2 is not head normalizing. The same criticism that we made above towards leftmost reduction holds for the first variant of Böhm reduction strategy: here, since we must start with u_2 and that it is not head normalizing, we will never reduce u_1 and u_3 , whereas they are normalizing and may reach a final state.

The second strategy avoids all those drawbacks: since we never perform more than one reduction step in a maximal head reducible subterm before skipping to another, we are never captive of a non-weakly normalizing subterm of t and all the parts of t that can be normalized shall be. Thus, this does not matter whether u_1 , u_2 or u_3 are normalizing or not, the second variant will normalize the ones that are.

Remark 2.8. Assume that a (non erasable) subterm of t is not weakly normalizing. Then the 2nd variant of the Böhm reduction strategy (BRS2) will not terminate, but all

the positions that may be Böhm stabilized will be *i.e.* if $t \rightarrow^* t'$ and $b \in \mathbf{stab}_B(t')$, then $b \in \mathbf{stab}_B(t_n)$ for all $n \geq N$ for N great enough, where $t = t_0, t_1, t_2, \dots$ is an instance of BRS2. In other words, even if this strategy does not terminate on t , it will progressively output all the Hereditarily Nested Head Normal Forms (HNHNF) in t (Lemma 2.5). There are actually two subcases:

- After a certain rank, the set of Böhm stable positions of (the reduct of) t does not change anymore. Thus, only a finite number of HNHNF can be outputted and there is an integer n such that the applicative depth of the Böhm stable positions is bound by n . This will correspond to a *finite Böhm tree* (Sec. 9.1).
- The set of Böhm stable positions keep on growing (this is the case with $t = Y_f$). Then, their applicative depth is not bound. Asymptotically, we obtain an infinite number of HNHNF and this will correspond to an *infinite Böhm tree*. For instance, when $t = Y_f$, we have $Y_f \rightarrow^n f^n(Y_f)$ with $\mathbf{stab}_B(f^n(Y_f)) = \{2^i, 2^i \cdot 1 \mid i \leq n\}$. Asymptotically, when n tends towards infinity, the set of Böhm stable positions is $\{2^i, 2^i \cdot 1 \mid i \in \mathbb{N}\}$. This is the support of the infinitary term f^ω (see Sec. 2.3.3), which is the Böhm tree of Y_f .

2.4 A Lambda-Calculus with Explicit Substitutions

We shortly present in this section a λ -calculus with **explicit substitution** $\Lambda_{\mathbf{ex}}$ due to Accattoli and Kesner [3]. $\Lambda_{\mathbf{ex}}$ corresponds to a “small-step” version of Λ : in the latter, when a redex $(\lambda x.r)s$ is reduced in one step into the term $r[s/x]$ and to do that, we replace *all* the (free) occurrences of x in r by a copy of s *e.g.*, if there are n free occurrences of x in r , then those n free occurrences of x are all replaced *simultaneously* by s during reduction (“big-step”). In $\Lambda_{\mathbf{ex}}$, a reduction step will be allowed to replace only *one* occurrence of x by the argument s : the reduction $\rightarrow_{\beta_{\mathbf{x}}}$ of $\Lambda_{\mathbf{ex}}$ is then said to be **linear**. One of the contribution of this thesis is to extend $\Lambda_{\mathbf{ex}}$ to a small-step version of the λ_μ -calculus (Chapter 8).

An Example of Explicit Substitution Reduction Sequence For instance, x has two free occurrences in $r := xyx$ and $(\lambda x.r)s = (\lambda x.xy x)s$ reduces into $r[s/x] = sy s$. In $\Lambda_{\mathbf{ex}}$, this reduction will be emulated by

$$(\lambda x.xy x) \rightarrow_{\beta_{\mathbf{x}}} (xyx)\langle x \setminus s \rangle \rightarrow_{\beta_{\mathbf{x}}} (s y x)\langle x \setminus s \rangle \rightarrow_{\beta_{\mathbf{x}}} s y s$$

or by

$$(\lambda x.xy x) \rightarrow_{\beta_{\mathbf{x}}} (xyx)\langle x \setminus s \rangle \rightarrow_{\beta_{\mathbf{x}}} (xy s)\langle x \setminus s \rangle \rightarrow_{\beta_{\mathbf{x}}} s y s$$

The notation $\langle x \setminus s \rangle$ is a new construction of the calculus $\Lambda_{\mathbf{ex}}$ and denotes an explicit substitution.

The operational semantics of $\Lambda_{\mathbf{ex}}$ may be understood as this: firing a redex $(\lambda x.r)s$ triggers an explicit substitution and we have $(\lambda x.r)s \rightarrow_{\beta_{\mathbf{x}}} r\langle x \setminus s \rangle$. A term $r\langle x \setminus s \rangle$ may be reduced in $r'\langle x \setminus s \rangle$ where r' is obtained from r by replacing exactly one free occurrence of x by s (this free occurrence is non-deterministically chosen) *except* when:

- The term r contains exactly one free occurrence of x : in that case, the explicit substitution is completed by the reduction step and we have $r\langle x \setminus s \rangle \rightarrow_{\beta_{\mathbf{x}}} r'$, where r' is obtained from r by replacing the *unique* free occurrence of x by s . This occurs in $(xy s)\langle x \setminus s \rangle \rightarrow_{\beta_{\mathbf{x}}} s y s$

- The term r does not have a free occurrence of x : in that case, the explicit substitution is eliminated by the reduction step and we have $r\langle x \setminus s \rangle \rightarrow_{\beta_x} r$. This occurs in $(\lambda z.y z)\langle x \setminus s \rangle \rightarrow_{\beta_x} \lambda z.y z$.

Naturally, the construction $\langle x \setminus s \rangle$ binds x .

Operational Semantics of Λ_{ex} Now, let us give a formal definition of Λ_{ex} and state the main properties that it enjoys.

The terms t, u of Λ_{ex} are defined inductively by:

$$t, u = x \in \mathcal{V} \mid (\lambda x.t) \mid (tu) \mid t\langle x \setminus u \rangle$$

Notice that $\langle x \setminus u \rangle$ is a new operator of the language (it is in the grammar of Λ_{ex}) and *not* a meta-operator as $[u/x]$ is for the λ -calculus. In Λ_{ex} , $[u/x]$ is also a meta-operator with the expected definition and behaviour.

The contexts \mathbf{C} of Λ_{ex} are defined as expected with the additional constructor $\langle x \setminus u \rangle$. Among them, we distinguish the **list contexts** \mathbf{L} , which are defined inductively by:

$$\mathbf{L} ::= \square \mid \mathbf{L}\langle x \setminus u \rangle$$

The reduction rules of the λ_{ex} -calculus aim to give a resource aware semantics to the λ -calculus, based on the *substitution at a distance* paradigm [2,3]. Indeed, the reduction relation λ_{ex} of the calculus is given by the context closure of the following rewriting rules.

$$\begin{array}{ll} \mathbf{L}[\lambda x.t]u & \rightarrow_{\mathbf{B}} \mathbf{L}[t\langle x \setminus u \rangle] \\ \mathbf{C}^x[[x]\langle x \setminus u \rangle] & \rightarrow_{\mathbf{c}} \mathbf{C}[[u]\langle x \setminus u \rangle] \text{ if } |\mathbf{C}[[x]]|_x > 1 \\ \mathbf{C}^x[[x]\langle x \setminus u \rangle] & \rightarrow_{\mathbf{d}} \mathbf{C}[u] \text{ if } |\mathbf{C}[[x]]|_x = 1 \\ t\langle x \setminus u \rangle & \rightarrow_{\mathbf{w}} t \text{ if } x \notin \mathbf{fv}(t) \end{array}$$

where we remember \mathbf{C}^x means that the context \mathbf{C} does not capture x (Sec. 2.1.7) and that $|t|_x$ denotes the number of free occurrences of x in t (Sec. 2.1.2).

Notice that, as expected from the above example, the occurrences of x are (arbitrarily) substituted one after another *i.e.* substitution is *linearly* processed. When there is just one occurrence of x left, the small reduction step \mathbf{d} performs the last substitution and removes the explicit substitution $\langle x \setminus u \rangle$, thus completing the operation.

More generally, not only the syntax of the λ_{ex} -calculus can be seen as a refinement of the λ -calculus, but also its operational semantics. Formally:

Lemma 2.6. If $t \in \Lambda$, then $t \rightarrow_{\beta} t'$ implies $t \rightarrow_{\beta_x}^+ t'$.

This lemma was illustrated by the above example. Conversely, we can project λ_{ex} -reduction sequences into λ -reduction sequences. Indeed, consider the projection function $\mathbf{P}(_)$ computing all the explicit substitution defined inductively by $\mathbf{P}(x) = x$, $\mathbf{P}(\lambda x.t) = \lambda x.\mathbf{P}(t)$, $\mathbf{P}(tu) = \mathbf{P}(t)\mathbf{P}(u)$ and $\mathbf{P}(t\langle x \setminus u \rangle) = \mathbf{P}(t)[\mathbf{P}(u)/x]$. Then:

Lemma 2.7. If $t \in \Lambda_{\text{ex}}$, then $t \rightarrow_{\beta_x} t'$ implies $\mathbf{P}(t) \rightarrow_{\beta}^* \mathbf{P}(t')$

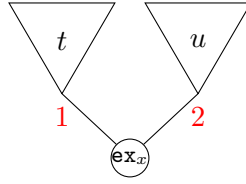


Figure 2.13: $t\langle x\langle u\rangle$ as a Labelled Tree

Positions and Head Normal Forms We may also consider $\lambda_{\mathbf{ex}}$ -terms as labelled trees by extending the signature $\Sigma = \mathcal{V} \cup \{\lambda x \mid x \in \mathcal{V}\} \cup \{\@\}$ to $\Sigma_{\mathbf{ex}} = \Sigma \cup \{\mathbf{ex}_x \mid x \in \mathcal{V}\}$, where \mathbf{ex}_x is a binary node representing an explicit substitution of x .

We may then define a notion of positions (words on $\{0, 1, 2\}$) and support for $\lambda_{\mathbf{ex}}$ -terms accordingly. The *Head Normal Forms* of $\Lambda_{\mathbf{ex}}$ are those of Λ . More precisely, a term $t \in \Lambda_{\mathbf{ex}}$ may be of three different forms: $t = \lambda x_1 \dots x_p. x t_1 \dots t_q$, $t = (\lambda x. r) s t_1 \dots t_q$ or $t = r\langle x\langle s\rangle t_1 \dots t_q$ ($p, q \geq 0$). Indeed, let p be maximal such that $0^p \in \text{supp}(t)$, then let q be maximal such that $0^p \cdot 1^q \in \text{supp}(t)$ and $t(0^p \cdot 1^q) \neq \@$. Then $t(0^p \cdot 1^q)$ will be x , λx or \mathbf{ex}_x for some $x \in \mathcal{V}$.

Head reduction and the head reduction strategy are naturally extended to $\Lambda_{\mathbf{ex}}$, as well as leftmost outermost reduction. We may also define the notions of head, weak or strong normalization for $\Lambda_{\mathbf{ex}}$ in the obvious way.

Chapter 3

Intersection Type Systems

Well-typed programs cannot “go wrong”.

Milner, 1978 [83]

We present in this chapter a few aspects of Intersection Type Theory. We start first with a discussion about Simple Type Theory in Lambda Calculus and the Curry-Howard Isomorphism. We then explain some limitations of simple types and how intersection types came into play to overcome those limitations. Indeed, we identify here three uses of **Intersection Type Systems (ITS)**.

- Providing a **static** (see Sec. 3.1.1 and *e.g.*, Proposition 3.7) characterization of different (variants of) normalization *i.e.* termination.
- Proving that a reduction strategy is **complete** for a given notion of normalization (*e.g.*, Propositions 3.8 and 5.2). This kind of result is *external* to Type Theory but ITS provide alternative proofs that can arguably be seen as the simplest ones in the non-idempotent case.
- Providing **denotations** to λ -terms *i.e.* for us, **invariants of execution**. This aspect is discussed in Chapter 12.

As it turns out, the main expected properties of a well-behaved intersection type system are known as **Subject Reduction** and **Subject Expansion**, stating the stability of typing under (anti)reduction. We discuss them throughout Sec. 3.3.

Outline

In Sec. 3.1, we present the simply typed λ -calculus and quickly recall the Curry-Howard Isomorphism and some of its applications. Some limitations of simple types suggest considering intersection type systems, which are the subject of Sec. 3.2. We discuss the possible idempotency of the intersection operator and two intersection type systems are presented: system \mathcal{D}_0 (idempotent) and system \mathcal{R}_0 (non-idempotent). We explain how two very important *dynamic* properties of intersection type systems, namely *subject reduction* and *subject expansion* are used to provide type-theoretic characterization of normalization, as well as proofs of completeness of some reduction strategies. In Sec. 3.3, we explain the mechanisms of subject reduction and subject expansion and we show in which case they can fail. Finally, in Sec. 3.4, we illustrate the general principles presented in the first sections of this chapter by proving that head normalization is characterized by typability in system \mathcal{R}_0 . This characterization comes along with the following semantical

result (that is external to type theory): a λ -term is head normalizing (there is a reduction path from t to a head normal form) iff the head reduction strategy terminates on t .

3.1 From the λ -Calculus to Intersection Types Theory

In Sec. 3.1, after presenting some basic ideas of the Curry-Howard Correspondence (Sec. 3.1.1), we give a short overview of the motivations and uses of the intersection type discipline (Sec. 3.1.2). We then present Curry’s simple type system (Sec. 3.1.3) and we discuss in more detail the relation between typability and normalization (Sec. 3.1.4). From the impossibility of simple types to characterize normalization originates the birth of intersection types.

3.1.1 The Curry-Howard Correspondence

The notion of **type** was introduced by Whitehead and Russell in [115], among other attempts (*e.g.*, Zermelo’s Separation Axiom in Set Theory) at avoiding Russell’s Paradox and providing a non-contradictory foundation to mathematics. Types were then adapted in Combinatory Logic by Curry ([21], 4.2.) who first noticed, in this framework, the nowadays well-known correspondence of “Propositions-as-Types”.

Church himself introduced types in λ -calculus in [26]. Those types furthered the intuition of λ -terms as functions. For instance, a term t typed with $A \rightarrow B$ represents a function from A to B . When fed with a term u of type A , t yields a term of type B *i.e.* the term $t u$ will be of type B when u is of type A . Typing is **static** (by opposition to *dynamic*, Sec. 2.1.4): no reduction is needed either to find a typing derivation of a term or, given a typing tree, to check whether it is correct derivation or not.

In 1969 ([21], 8.1.4 and [55]), Howard noticed that the correspondence observed by Curry could be extended to Lambda Calculus. This yielded the famous **Curry-Howard Correspondence**: there is an equivalence between some notions in Simply Typed λ -Calculus and some others in Natural Deduction.

Natural Deduction	Simply Typed Lambda Calculus
Formula	Type
Proof	Simply Typed Term
Cut-Elimination Step	Reduction-Step

A practical consequence of this correspondence (that can be extended to many other notions) is that logical methods and concepts can be exported to (functional) programming languages and *vice versa*. The first published application of this correspondence was due to Curry himself [33], who used cut-elimination techniques from logic to prove that if a term was typable, then it was weakly normalizing (Definition 2.4).

3.1.2 Lambda-Calculus and Type Theory

As recalled in Chapter 2, an important dynamical property of λ -terms is **normalization**: for instance, a term is *Head Normalizing (HN)* if it can be reduced to a *Head Normal Form (HNF)* (*i.e.* a term of the form $\lambda x_1 \dots x_p. x t_1 \dots t_q$) and a term is *Weakly Normalizing (WN)* when it can be reduced to a *β -Normal Form (NF)* (*i.e.* a term without redex). A term t is *Strongly Normalizing (SN)* if there is no infinite reduction path starting at t .

We saw in Sec. 3.1.1 that a simple type system for the λ -calculus was introduced by Church not long after its creation and notions of type assignment systems for λ -calculus. One of the developments of the historical simple type systems were the *polymorphic* (or *higher-order*) type systems.

However, despite their expressive power, polymorphic types have some limitations. For example, it is not possible to assign a type to a term of the form $\lambda x.xx$ (see Sec. 3.1.4), which can be understood as a meaningful program specified by a terminating term ($\lambda x.xx$ represents the *auto-application*, see Sec. 2.1.4). First, tu is not typable when t and u have the same type (the equality $A = A \rightarrow B$ is impossible in Curry_0). Thus, the normal forms xx or $\Delta = \lambda x.xx$ cannot¹ be typed.

Intersection types, pioneered by Coppo and Dezani [27, 28], introduce a new constructor \wedge for types, allowing to assign a type of the form $((\sigma \rightarrow \sigma) \cap \sigma) \rightarrow \sigma$ to the term $\lambda x.xx$ because the left occurrence of x and the right one can be assigned different types.

The main feature of an **Intersection Type System (ITS)** is the following: each time that an occurrence of a variable x is met, it may be assigned a new type. For instance, xx becomes easily typable (if x is assigned the types A and $A \rightarrow B$, then we can type xx with B). Then Δ can be assigned a type of the form $((A \rightarrow B) \wedge A) \rightarrow B$. Thus, in a sense, intersection type systems also are polymorphic, but they feature a sort of “unconstrained” polymorphism with no higher-order rules.

The intuition behind a term t of type $A_1 \wedge A_2$ is that t has both types A_1 and A_2 . The symbol \wedge is to be understood as a mathematical intersection, so in principle, intersection type theory was developed to ensure *idempotent* ($A \wedge A = A$), *commutative* ($A \wedge B = B \wedge A$), and *associative* ($(A \wedge B) \wedge C = A \wedge (B \wedge C)$) laws.

Intersection types have been used as a *behavioural* tool to reason about several operational and semantical properties of programming languages: whereas in a higher-order type system, typability ensures strong normalization (if the term t is typable, then it is SN), an intersection type system will usually provide a *characterization* of normalization, with equivalences of the form “ t is normalizing iff it is typable”.

For example, a λ -term/program t is strongly normalizing/terminating if and only if t can be assigned a type in an appropriate intersection type assignment system. Similarly, intersection types are able to describe and analyze models of λ -calculus [9], characterize *solvability* [77], *head normalization* [77], *linear-head normalization* [60], and *weak-normalization* [68, 77] among other properties.

These frameworks turn out to be a powerful tool to reason about **qualitative** properties of programs, but not for **quantitative** ones. Indeed, for example, there is a type system that assigns a type to a term t if and only if t is head normalizing (qualitative information), but the type system gives no information about the number of reduction steps that are needed to obtain a HNF from t (quantitative information).

Here is where **non-idempotent** types come into play, by making a clear distinction between $\sigma \wedge \sigma$ and σ : with non-idempotency, using the resource σ twice or once is not the same. This change of point of view can be related to the essential spirit of Girard’s Linear Logic [47], which removes the contraction and weakening structural rules in order to provide an explicit control of the use of logical resources, *i.e.* to give a full account of the number of times that a given proposition is used to derive a conclusion.

¹The details of this argument can be found in Sec. 3.1.3 in the case of Curry’s system, but it also works for any simple type system of higher order.

The use of non-idempotent types was pioneered by Gardner [43] and Kfoury [63]. Relational models of λ -calculi based on non-idempotent types have been investigated in [22, 41] (a version of Gardner/de Carvalho type system is given in Sec. 3.2.4). D. de Carvalho [22] established in his PhD thesis a relation between the size of a typing derivation in a non-idempotent intersection type system for the lambda-calculus and the head/weak-normalization execution time of head/weak-normalizing lambda-terms, respectively (this is recalled in Remarks 3.15 and 5.6). Non-idempotency has been used, in particular by Bernadet and Lengrand, to reason about the longest reduction sequence of strongly normalizing terms in both the lambda-calculus [13, 14, 36] and in different lambda-calculi with explicit substitutions [14, 60]. Non-idempotent types also appear in linearization of the lambda-calculus [63] and type inference [64, 85]. Bucciarelli, Kesner and Ronchi della Rocca proved that inhabitation² is decidable with non-idempotent intersection types [18] whereas it is not in the idempotent case (Urzyczyn [105]), and Dudenhefner and Rehof characterized the complexity of inhabitation at bounded dimension, both in the idempotent and the non-idempotent case [39, 40]. Finally, non-idempotent types provided different characterizations of solvability [88] and have been used in verification of higher-order programs [52, 86].

3.1.3 A Simple Type System

In this section, we define a simple type system³ \mathbf{Curry}_0 , which corresponds to the type system originally introduced for λ -calculus by Church and then studied by Turing and Curry.

Let \mathcal{O} be a countable set of *base types* or *type variables* (metavariable o). We consider the set of *simple types* that is the set of words generated by the following inductive grammar:

$$A, B ::= o \in \mathcal{O} \mid A \rightarrow B$$

A **context** (metavariables Γ, Δ) is a partial function from the set of variables \mathcal{V} to the set of simple types. If for all $x \in \text{dom}(\Gamma) \cap \text{dom}(\Delta)$, $\Gamma(x) = \Delta(x)$, we write $\Gamma :: \Delta$ for the context of domain $\text{dom}(\Gamma) \cup \text{dom}(\Delta)$ extending Γ and Δ . If $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset$, we may write $\Gamma; \Delta$ instead of $\Gamma :: \Delta$. The context $x : A$ is the context Γ such that $\text{dom}(\Gamma) = \{x\}$ and $\Gamma(x) = A$.

The set of typing derivations in system \mathbf{Curry}_0 is defined *inductively* by the following rules:

$$\frac{}{\Gamma; x : A \vdash x : A} \text{ax} \quad \frac{\Gamma; x : A \vdash t : B}{\Gamma \vdash \lambda x. t : A \rightarrow B} \text{abs} \quad \frac{\Gamma \vdash t : A \rightarrow B \quad \Delta \vdash u : A}{\Gamma :: \Delta \vdash t u : B} \text{app}$$

Notice that the **app**-rule can only be applied when $\Gamma :: \Delta$ is defined *i.e.* when Γ and Δ agree on the common parts of their domain.

² Given a type A , is there a term having type A ? From the Curry-Howard perspective, this problem corresponds to deciding the provability of a proposition A .

³The subscript 0 indicates *finite* type systems in this document, by opposition to their (coinductive) infinite extensions, that will be considered in the later parts of this thesis.

Example 3.1. We give \mathbf{Curry}_0 -derivations typing the terms $I = \lambda x.x$, $K_x = \lambda y.x$ and $x I$:

$$\frac{\overline{x : A; y : B \vdash x : A}^{\mathbf{ax}}}{y : B \vdash \lambda x.x : A \rightarrow A}^{\mathbf{abs}} \quad \frac{\overline{x : A; y : B \vdash x : A}^{\mathbf{ax}}}{x : A \vdash \lambda y.x : B \rightarrow A}^{\mathbf{abs}}$$

$$\frac{\overline{x : (A \rightarrow A) \rightarrow B \vdash x : (A \rightarrow A) \rightarrow B}^{\mathbf{ax}} \quad \frac{\overline{x : A; y : B \vdash x : A}^{\mathbf{ax}}}{y : B \vdash \lambda x.x : A \rightarrow A}^{\mathbf{abs}}}{x : (A \rightarrow A) \rightarrow B; y : B \vdash x (\lambda x.x) : B}^{\mathbf{app}}$$

Remark 3.1. The rules can also be given *additively*:

$$\frac{\overline{\Gamma; x : A \vdash x : A}^{\mathbf{ax}}}{\Gamma \vdash \lambda x.t : A \rightarrow B}^{\mathbf{abs}} \quad \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash t u : B}^{\mathbf{app}}$$

This yields another type system \mathbf{Curry}'_0 . Every \mathbf{Curry}'_0 -derivation is a \mathbf{Curry}_0 -derivation, and it is very easy to see that any \mathbf{Curry}_0 -derivation can be rewritten as a \mathbf{Curry}'_0 -derivation, provided we enrich the contexts given in the axiom rules so that they are all equal (and we take care to avoid the capture of free variables).

This type system enjoys subject reduction *i.e.* typing is stable under reduction:

Proposition 3.1. System \mathbf{Curry}_0 enjoys subject reduction. Namely, assume $t \rightarrow t'$. If $\Gamma \vdash t : B$ is derivable in system \mathbf{Curry}_0 , then $\Gamma \vdash t' : B$ also is.

As we will see, subject reduction is often crucial to prove properties of termination/normalization, as Theorem 3.1.

3.1.4 Types and Termination

Lost in the notes of Alan Turing ([21], 5.1.) was found the first proof, dating back to the early 40s, of normalization of the Typed Terms. Another proof of this result was later found by Curry [33] using cut-elimination. Indeed, for the first type systems introduced in λ -calculus:

Theorem 3.1. If a term is typable in system \mathbf{Curry}_0 , then it is strongly normalizing.

Let us remind that normalization is a *dynamic* property (beginning of Sec. 2.2). One of the interesting aspects of this result is that typing is *static* (in the sense of Sec. 3.1.1).

Unfortunately, system \mathbf{Curry}_0 does not allow typing some meaningful terms like $\Delta = \lambda x.x x$ (which represents auto-application, Sec. 2.1.4). More generally, the converse implication of the above theorem is not true in a regular simple type system: normalization does not imply typability. For instance, we cannot even type the normal form $x x$ because x should be both typed with $A \rightarrow B$ (left occurrence) and A (right occurrence) for some types A and B . However, in a simple type system, a variable may be only assigned *one* type and in \mathbf{Curry}_0 , the equality $A \rightarrow B = A$ is impossible (the type $A \rightarrow B$ contains strictly more symbols than A does).

As it turns out in the next section, Intersection Type Systems (ITS) were designed to overcome this limitation.

3.2 Principles and Examples of Intersection Types

From Sec. 3.1.4, we know that with a simple type system, typability implies normalization: if a term is typable, then it is normalizing. We saw that the converse implication does not hold anymore, notably because in such a type system, a variable can only be assigned one type. Intersection Type Systems (ITS) were introduced by Coppo and Dezani in 78 [28] to relax this condition and obtain a type-theoretic *characterization* of normalization: in ITS, typability is equivalent to normalization (for all term t , t is typable iff t is normalizing) and not only a *guarantee* of normalization as it is in STS.

Remark 3.2.

- The price to pay to characterize normalization is the loss of decidability, since λ -calculus is Turing-complete for head reduction (Chapter 2 of [68]): in an ITS, typability is a semi-decidable predicate whereas in the simple type system \mathbf{Curry}_0 , it is decidable (*i.e.* there is a terminating algorithm that decides whether a term t is simply typable or not).
- Generally speaking, *Higher Order* simple type systems are not decidable *e.g.*, system \mathcal{F} is not decidable since a function $\mathbb{N} \rightarrow \mathbb{N}$ may be implemented in system \mathcal{F} iff it is provably total in \mathbf{PA}_2 (recall p. 31). But the predicate “ f is provably total in \mathbf{PA}_2 ” is not decidable by Gödel Incompleteness Theorem. It is semi-decidable though, since the set of proofs of \mathbf{PA}_2 is obviously recursively enumerable.

Concretely, in an ITS, a variable may be assigned a new type (or several new ones) each time that it is the subject of an axiom rule. Thus, the argument used in the previous section does not hold and a term like xx is easily typable. This is not enough though to explain why ITS are able to characterize normalization.

The two ITS that are going to be presented first are *relevant*, meaning that weakening (see Sec. 3.3.5) is forbidden, whereas the first ITS to be introduced were rather *irrelevant*. However, it is easier to understand subject reduction and subject expansion in relevant ITS and, especially, why they hold or do not (Figures 3.1 and 3.2). We will introduce irrelevant ITS later, in Sec. 3.3.5. See [107] for an extensive survey of idempotent ITS.

3.2.1 Towards Strictness and Relevance

Let us give Krivine’s presentation of the intersection type system $\mathcal{D}\Omega$, corresponding to one of the original systems of Coppo and Dezani (chapter 3 of [68]).

The set of types of system $\mathcal{D}\Omega$ is defined inductively by:

$$A, B := \perp \mid o \in \mathcal{O} \mid A \rightarrow B \mid A \wedge B$$

We write $A_1 \wedge A_2 \wedge \dots \wedge A_n$ for $(\dots(A_1 \wedge A_2) \wedge \dots) \wedge A_n$. The derivations of system $\mathcal{D}\Omega$ are defined inductively by:

$$\begin{array}{c} \frac{}{\Gamma; x : A \vdash x : A} \text{ax} \qquad \frac{\Gamma; x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \rightarrow B} \text{abs} \\ \\ \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash x : A}{\Gamma \vdash tu : A} \text{app} \qquad \frac{}{\Gamma \vdash t : \perp} \perp \\ \\ \frac{\Gamma \vdash t : A \quad \Gamma \vdash t : B}{\Gamma \vdash t : A \wedge B} \wedge \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash t : A} \text{proj}_L \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash t : B} \text{proj}_R \end{array}$$

Intuitively, \perp is an **empty intersection type** which explains why it types any term. It is necessary to ensure subject expansion (stability of typing under expansion) in the case on an erasing step $(\lambda x.r)s \rightarrow r$ (with $x \notin \text{fv}(r)$).

From those rules, we may easily derive the following admissible *structural* rules:

$$\frac{\Gamma; x : A \wedge A \vdash t : B}{\Gamma; x : A \vdash t : B} \text{contr} \quad \frac{\Gamma; x : A \vdash t : B}{\Gamma; x : A \wedge A \vdash t : B} \text{duplic}$$

$$\frac{\Gamma; x : A \vdash t : B}{\Gamma; x : A \wedge C \vdash t : B} \text{weak}$$

$$\frac{\Gamma \vdash t : (A \wedge B) \wedge C}{\Gamma \vdash t : A \wedge (B \wedge C)} \text{asso} \quad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash t : B \wedge A} \text{comm}$$

$$\frac{\Gamma; x : A_1 \wedge \dots \wedge A_n \vdash t : B \quad \pi \text{ is a permutation of } \{1, \dots, n\}}{\Gamma; x : A_{\pi(1)} \wedge \dots \wedge A_{\pi(n)} \vdash t : B} \text{perm}$$

In particular, the typing rules of $\mathcal{D}\Omega$ imply that, as an *operator*, \wedge enjoys some well-known features up to equivalence: it is associative (**asso**), commutative (**comm**) and also *idempotent* ($A \wedge A$ and A are equivalent by rules \wedge and **proj_L**). Moreover, \perp behaves like the neutral element of \wedge , since $\Gamma \vdash t : A$ and $\Gamma \vdash t : A \wedge \perp$ are interderivable (by rules \perp and \wedge).

Let us observe an important difference between the typing rules of $\mathcal{D}\Omega$:

- The premises and the conclusions of the rules \wedge and **proj_{L/R}** have the same subject. We say that they are not **subject directed**.
- The premises and the conclusions of the rules **ax**, **abs** and **app** have a different subject: actually, each of these rules introduces a constructor of λ -calculus (x , λx or $@$) to the subjects of their premises. We say that they are **subject directed**.

Strictness and Relevance Historically, another intersection type system [77], to be called here \mathcal{D}_0 , was introduced just after [28]. System \mathcal{D}_0 is a restriction of $\mathcal{D}\Omega$. We give now a high-level presentation⁴ of the features of system \mathcal{D}_0 before hinting at their consequences on its dynamical behavior compared to that of $\mathcal{D}\Omega$:

- System \mathcal{D}_0 only allows **strict intersection types** *i.e.* types in which intersection is only allowed in the source (the left-hand sides) of arrows. Thus, strict intersection types are defined by the inductive grammar:

$$A_k, B := o \in \mathcal{O} \mid (A_1 \wedge A_2 \wedge \dots \wedge A_n) \rightarrow B \quad (n \geq 0)$$

The empty intersection type \perp is implicitly present in this grammar: when $n = 0$, the arrow type above denotes $\perp \rightarrow B$.

- In particular, the rules \wedge and **proj_{L/R}** are put aside. Rules **ax** and **abs** are also restricted compared to system $\mathcal{D}\Omega$.

⁴A formal presentation system \mathcal{D}_0 will be given in Sec. 3.2.3 after the discussion of Sec. 3.2.2 that suggests to represent the intersection types of \mathcal{D}_0 by *sets* of types.

- System \mathcal{D}_0 is **relevant** [35, 106]. For now, let us just say that relevance only validates implication/arrow types of the form $(A_1 \wedge A_2 \wedge \dots \wedge A_n) \rightarrow B$ for which the premises $A_1 \dots A_n$ are actually needed to prove B . Thus, relevance ensures some **resource-awareness**. For instance, $(A \wedge B) \rightarrow A$ is an irrelevant implication since we only need the premise A (and not B) to prove the target B . In particular, the rule **weak** above is not admissible in system \mathcal{D}_0 .
- More generally, relevance demands that if the variable x has been assigned some strict types A in the context, then the *subterm* x occurs in the derivation with the type A *i.e.* every assigned type must be used in the derivations of system \mathcal{D}_0 . In particular, if a derivation Π of system \mathcal{D}_0 concludes with $\Gamma \vdash t : B$ and $x \notin \text{fv}(t)$, then $x \notin \text{dom}(\Gamma)$. But normal forms like $\lambda x.y$ should still be typable: the **abs**-rule is then modified so that the empty type appears when a non-occurring/untyped variable is abstracted:

$$\frac{\Gamma \vdash t : B \quad x \notin \text{dom}(\Gamma)}{\Gamma \vdash \lambda x.t : \perp \rightarrow A} \perp$$

Intuitively, $\perp \rightarrow A$ means that A is universally valid *i.e.* A is provable in an empty context/does not need any hypothesis to hold.

- Contexts assign *intersection types* to variables. Since the subterms of a typed term can have different variables and thus, by relevance, be typed in different contexts, a point-wise extension of \wedge on contexts is used to *collect* typing information *i.e.* the **app**-rule becomes:

$$\frac{\Gamma \vdash t : (A_1 \wedge \dots \wedge A_n) \rightarrow B \quad \Delta_1 \vdash u : A_1 \quad \dots \quad \Delta_n \vdash u : A_n}{\Gamma \wedge \Delta_1 \wedge \dots \wedge \Delta_n \vdash t u : B}$$

Strictness and relevance actually greatly simplify (the proof) of subject reduction in \mathcal{D}_0 (compared to $\mathcal{D}\Omega$), but this will be made clear only with Fig. 3.1, p. 90.

Towards Syntax Direction It is easy to see that in system $\mathcal{D}\Omega$, the associativity and the commutativity of \wedge , as well as the **perm**-rule, are proved from the typing rules by using the rules \wedge and **proj**_{R/L}, that are discarded in \mathcal{D}_0 . But the equivalence between $(A \wedge B) \wedge C$ and $A \wedge (B \wedge C)$ (“associativity” of \wedge) and that between $A \wedge B$ and $B \wedge A$ are still desirable⁵ features. To ensure them in relevant typing, several choices are possible:

- Gardner [43] takes the **perm**-rule above as a typing rule.
- Coppo et al. [77] use subtyping relations to identify “equivalent” types.

Those two choices have the same drawback: they burden derivations with rules that do not follow the syntax of λ -calculus (in the sense suggested above) *i.e.* rules whose premises and conclusions have the same subject. In the next section, we explain how using sets or multisets to represent intersection types makes it possible to avoid these problems and obtain type systems whose rules only follow the construction of λ -calculus (what we informally call **subject direction**) and are thus more readable.

⁵ *Idempotency* is a different matter, that will be discussed throughout Sec. 3.3: actually, idempotency makes subject expansion fail (Sec. 3.3.4).

Remark 3.3 (Strictness and Subject Direction). Let us understand why subject direction demands to consider *strict types*. Informally, judgments $y : A \rightarrow B \rightarrow C$, $x : A \wedge B \vdash y x x$ and $z : D \rightarrow (A \rightarrow B)$, $z' : D \vdash z z : A \rightarrow B$ are easily derivable, the latter one featuring the non-strict type $D \rightarrow (A \wedge B)$. Thus, $(\lambda x.r)s$ is typable with C , with $r = y x x$ and $s = z z'$. In order to have subject reduction (*i.e.* a typing of $r[s/x]$ with C), the first occurrence of x in r should be replaced with $s : A$ and the second one with $s : B$. But this demands the use of the non-subject directed rules $\text{proj}_{L/R}$ to obtain $s : A$ and $s : B$ from $s : A \wedge B$.

Remark 3.4 (Syntax Direction). Informally, a type system is syntax directed when not only it is **subject directed**, but also the subject and the type enable to infer the last rule of a derivation *i.e.* if the only thing we know of a derivation Π is its conclusion $\Gamma \vdash t : B$, we may infer the name of the last rule of Π from t and B . This is more than mere subject direction *e.g.*, a glimpse at the rules of system \mathcal{S} (Sec. 5.2.1) shows that is subject direct but that it is not syntax directed since it features *two* rules typing the application, although we later present a formalism that allows those two rules to be subsumed in one and recover syntax direction (Sec. 7.1).

3.2.2 Intersection Operator, Sets and Multisets

We explain in this section how some structural rules identifying equivalent types (*e.g.*, $(A \wedge B) \rightarrow C$ and $(B \wedge A) \rightarrow C$) can be implicitly implemented by representing intersection types with sets or multisets.

As discussed in the previous section, we mostly consider in this thesis *relevant* intersection type systems (every type that has been assigned must be effectively used) that feature *strict* intersection types (intersection is allowed in the source of arrows only). Strictness and relevance allow designing subject directed type systems, meaning that here that each typing rule introduce a constructor (x , λx or $@$) of the λ -calculus. A last step to achieve subject direction is to fully identify types that are equal modulo structural rules that should otherwise be treated implicitly.

Still according to Sec. 3.2.1, intersection can be seen as an operator \wedge that “collects” the types assigned to a same variable. This operator will always be assumed to be *associative* and *commutative*. We will consider both the cases where it is *idempotent* and where it is not.

Sets, Multisets and Free Operators Intuitively, a **multiset** (a set with multiplicity) is like a set, except that the number of occurrences of each element matters. Thus, if the notation $[a_i]_{i \in I}$ denotes the multiset whose elements are the a_i , we shall have $[a, a, b] = [a, b, a] = [b, a, a]$ but $[a, a, b]$, $[a, b]$, $[b, b, a]$, $[a, b, a, b]$ are pairwise distinct (assuming $a \neq b$). With regular sets, we also have $\{a, a, b\} = \{a, b, a\} = \{b, a, a\}$ but in contrast, $\{a, a, b\} = \{a, b\} = \{b, b, a\} = \{a, b, a, b\}$ holds. We write $[a]_n$ to denote the multiset containing a with multiplicity n .

The **cardinal** of a multiset $[a_i]_{i \in I}$, denoted $\#[a_i]_{i \in I}$, is the cardinal of I *e.g.*, $\#[a, b, a] = 3$. A multiset is **finite** if its cardinal is finite. The cardinal of a set $\{a_i\}_{i \in I}$ is also denoted $\#\{a_i\}_{i \in I}$. Notice however that we only have $\#\{a_i\}_{i \in I} \leq \#I$ *e.g.*, $\#\{a, b, a\}$ is equal to 2 (if $a \neq b$) because $\{a, b, a\} = \{a, b\}$. It is equal to 1 if $a = b$.

We use the meta-annotation \neq in $\{a_i\}_{i \in I}^{\neq}$ to denote the set $\{a_i\}_{i \in I}$ when the a_i are pairwise distinct. Thus, $\#\{a_i\}_{i \in I}^{\neq} = \#I$ when this notation is licit.

The **multiset sum** (also called *multiset union*, denoted by $+$, is defined by: $[a_i]_{i \in I} + [b_j]_{j \in J} = [c_k]_{k \in K}$, where $K = (\{1\} \times I) \cup (\{2\} \times J)$, $c_{1,i} = a_i$ for all $i \in I$ and $c_{2,j} = b_j$ for all $j \in J$. We may check that this definition is sound and that $+$ is an associative and commutative operator on multisets, that has a neutral element, which is the empty multiset (denoted $[]$).

Remark 3.5. Let X be a set.

- A subset Y of X can be identified to a function f from X to Bool (for all $x \in X$, $x \in Y$ iff $f(x) = \text{True}$) and conversely. Likewise, when X is finite, a finite multiset \mathcal{M} of X can be identified to a function m (standing for “multiplicity”) from X to \mathbb{N} (and conversely): for all $x \in X$, the natural number $m(x)$ is multiplicity (number of occurrences) of x in \mathcal{M} *e.g.*, if $M = [a, a, b]$ (with $a \neq b$) and m represents \mathcal{M} , then $m(a) = 2$ and $m(b) = 1$. If X is infinite, we must assume that, for all $x \in X$ except a finite number, $m(x) = 0$.
- With this formalism, given two multisets \mathcal{M}_1 and \mathcal{M}_2 represented by m_1 and m_2 , the multiset sum $\mathcal{M}_1 + \mathcal{M}_2$ may be just defined as the multiset associated to the function $m_1 + m_2$. Associativity and commutativity are then straightforward.

Multiset order: let $[a_i]_{i \in I}$ and $[a'_i]_{i \in I'}$ be two multisets. We write $[a_i]_{i \in I} \leq [a'_i]_{i \in I'}$ if there is a multiset $[b_j]_{j \in J}$ such that $[a'_i]_{i \in I'} = [a_i]_{i \in I} + [b_j]_{j \in J}$. We may also say that \leq is a relation of *multiset inclusion*.

Representation of Operators Notice that:

- If an operator \wedge is associative, commutative and idempotent (for all a , $a \wedge a = a$), then $\{a_i\}_{i \in I} = \{a'_i\}_{i \in I'}$ implies that $\wedge_{i \in I} a_i = \wedge_{i \in I'} a'_i$.
- If an operator \wedge is associative, commutative (but not necessarily idempotent), then $[a_i]_{i \in I} = [a'_i]_{i \in I'}$ implies $\wedge_{i \in I} a_i = \wedge_{i \in I'} a'_i$, which is weaker.

Moreover, if we consider *free* operators, the converse implication hold:

- If \wedge is the free associative, commutative and idempotent operator on a set \mathcal{A} , then $\wedge_{i \in I} a_i = \wedge_{i \in I'} a'_i$ iff $\{a_i\}_{i \in I} = \{a'_i\}_{i \in I'}$.
- If \wedge is the free associative and commutative operator on \mathcal{A} , then $\wedge_{i \in I} a_i = \wedge_{i \in I'} a'_i$ iff $[a_i]_{i \in I} = [a'_i]_{i \in I'}$.

Intersection Types as Sets or Multisets? A last remark before concluding this chapter regards the choice of considering intersection types as sets or multisets. We define in Sec. 3.2.3 and 3.2.3 two type systems:

- System \mathcal{D}_0 , in which intersection is assumed to be a free associative and idempotent commutative operator. In \mathcal{D}_0 , an intersection of types will be thus naturally represented by a set of types in \mathcal{D}_0 *i.e.* instead of writing $\wedge_{i \in I} A_i$, we just write $\{A_i\}_{i \in I}$
- System \mathcal{R}_0 , in which intersection is assumed to be a free associative and commutative operator *i.e.* instead of writing $\wedge_{i \in I} A_i$, we just write $\{A_i\}_{i \in I}$.

3.2.3 System \mathcal{D}_0 (Idempotent Intersection)

System \mathcal{D}_0 is relevant and idempotent. Thus, as seen at the end of Sec. 3.2.2, it is natural to define the set $\mathbf{Typ}_{\mathcal{D}_0}$ of types of system \mathcal{D}_0 by the following inductive grammar:

$$A, B ::= o \mid \{A_i\}_{i \in I} \rightarrow B$$

We call $X = \{A_i\}_{i \in I}$ a **set type**. The set types represent intersection in system \mathcal{D}_0 and the intersection operator \wedge is the set-theoretic union: $\wedge_{i \in I} X_i = \cup_{i \in I} X_i$ (*i.e.* $\wedge_{i \in I} \{A_j\}_{j \in J(i)} := \cup_{j \in J} \{A_j\}_{j \in J(i)}$). The empty set type⁶ is denoted $\{\}$.

A \mathcal{D}_0 -**context** (metavariables Γ, Δ) is a *total* function from \mathcal{V} to the set of set types. The **domain** of Γ is given by $\{x \in \mathcal{V} \mid \Gamma(x) \neq \{\}\}$. The intersection of contexts $\cup_{i \in I} \Gamma_i$ is defined point-wise, as well as the inclusion $\Gamma \subseteq \Gamma'$. We may write $\Gamma; \Delta$ instead of $\Gamma \cup \Delta$ when $\mathbf{dom}(\Gamma) \cap \mathbf{dom}(\Delta) = \emptyset$. Given a set type $\{A_i\}_{i \in I}$, we write $x : \{A_i\}_{i \in I}$ for the context Γ s.t. $\Gamma(x) = \{A_i\}_{i \in I}$ and $\Gamma(y) = \{\}$ for all $y \neq x$. In particular, $x : \{B\}$ and $x : \{B\}; y : \{\}$ denote the same context. A \mathcal{D}_0 -**judgment** is a triple $\Gamma \vdash t : A$ where Γ is a \mathcal{D}_0 -context, t a term and A a type.

The set of typing derivations of system \mathcal{D}_0 , named $\mathbf{Deriv}_{\mathcal{D}_0}$, is defined inductively by the following rules:

$$\frac{}{x : \{A\} \vdash x : A} \mathbf{ax} \qquad \frac{\Gamma; x : \{A_i\}_{i \in I} \vdash t : B}{\Gamma \vdash \lambda x.t : \{A_i\}_{i \in I} \rightarrow B} \mathbf{abs}$$

$$\frac{\Gamma \vdash t : \{A_i\}_{i \in I}^{\neq} \rightarrow B \quad (\Delta_i \vdash u : A_i)_{i \in I}}{\Gamma \cup (\cup_{i \in I} \Delta_i) \vdash t u : B} \mathbf{app}_{\neq}$$

Remark 3.6. In the \mathbf{app}_{\neq} -rule, the condition $\{A_i\}_{i \in I}^{\neq}$ means that $i \neq i'$ implies $A_i \neq A_{i'}$ (for all $i, i' \in I$). Thus, the argument u is not typed redundantly *i.e.* for all type A , there is at most *one* premise typing u with A . We explain why this is necessary in Sec. 3.3.4. This condition will be relaxed in system $\mathcal{D}_{0,w}$ (Sec. 3.3.5).

Example 3.2. Notice that, in the \mathbf{abs} -rule, if x is not in the domain of the context, then $\lambda x.t$ is typed with $\{\} \rightarrow B$, as in the example on the right (we recall that $\lambda x.x$ is denoted by I and $\lambda y.x$ by K_x):

$$\frac{}{x : \{B\} \vdash x : B} \mathbf{ax} \qquad \frac{}{x : \{B\} \vdash x : B} \mathbf{ax}$$

$$\frac{}{\vdash \lambda x.x : \{B\} \rightarrow B} \mathbf{abs} \qquad \frac{}{x : \{B\} \vdash \lambda y.x : \{\} \rightarrow B} \mathbf{abs}$$

Contrary to system \mathbf{Curry}_0 , $\Delta = \lambda x.x x$ is typable in system \mathcal{D}_0 :

$$\frac{\frac{}{x : \{\{A\} \rightarrow A\} \vdash x : \{A\} \rightarrow A} \mathbf{ax} \quad \frac{}{x : \{A\} \vdash x : A} \mathbf{ax}}{x : \{\{A\} \rightarrow A, A\} \vdash x x : A} \mathbf{app}_{\neq}}{\vdash \lambda x.x x : \{\{A\} \rightarrow A, A\} \rightarrow A}$$

Remark 3.7 (Untyped Argument). In the \mathbf{app}_{\neq} -rule, when I is empty *i.e.* t is typed with $\{\} \rightarrow B$, then there is no judgment typing u in the premise and $t u$ may be typed with B for *any* term u . See also Sec. 3.4.1.

⁶Of course, $\{\} = \emptyset$, but $\{\}$ will only be used for the empty set as a set type.

We write $\Pi \triangleright_{\mathcal{D}_0} \Gamma \vdash t : B$ to mean that derivation Π concludes with the judgment $\Gamma \vdash t : B$ in system \mathcal{D}_0 and $\triangleright_{\mathcal{D}_0} \Gamma \vdash t : B$ to mean that $\Gamma \vdash t : B$ is derivable. When there is no ambiguity with any other type systems (*i.e.* most of the time), we write \triangleright instead of $\triangleright_{\mathcal{D}_0}$.

3.2.4 System \mathcal{R}_0 (Non-Idempotent Intersection)

System \mathcal{R}_0 was introduced by Philippa Gardner [43] and rediscovered by Daniel de Carvalho in his PhD Thesis [22]. It is relevant and features a non-idempotent intersection free operator, so the end of Sec. 3.2.2 suggests to define the set $\mathbf{Typ}_{\mathcal{R}_0}$ of types of system \mathcal{R}_0 inductively by:

$$\sigma, \tau ::= o \in \mathcal{O} \mid [\sigma_i]_{i \in I} \rightarrow \tau$$

We call $\mathcal{I} := [\sigma_i]_{i \in I}$ a **multiset type**. The multiset types represent intersection in system \mathcal{R}_0 and the intersection operator \wedge is the multiset-theoretic sum: $\wedge_{i \in I} \mathcal{I}_i = +_{i \in I} \mathcal{I}_i$ (*i.e.* $\wedge_{i \in I} [\sigma_j^i]_{j \in J(i)} := +_{i \in I} [\sigma_j^i]_{j \in J(i)}$).

A **\mathcal{R}_0 -context** (metavariables Γ, Δ) is a *total* function from \mathcal{V} to the set of multiset types. The **domain** of Γ is given by $\{x \mid \Gamma(x) \neq []\}$. The intersection of contexts $+_{i \in I} \Gamma_i$ is defined point-wise, as well as inclusion $\Gamma \leq \Gamma'$ (see Sec. 3.2.2). We may write $\Gamma; \Delta$ instead of $\Gamma + \Delta$ when $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset$. Given a multiset type $[\sigma_i]_{i \in I}$, we write $x : [\sigma_i]_{i \in I}$ for the context Γ s.t. $\Gamma(x) = [\sigma_i]_{i \in I}$ and $\Gamma(y) = []$ for all $y \neq x$. A **\mathcal{R}_0 -judgment** is a triple $\Gamma \vdash t : \sigma$ where Γ is a context, t a term and σ a type.

The set of typing derivations of system \mathcal{R}_0 , named $\mathbf{Deriv}_{\mathcal{R}_0}$, is defined inductively by the following rules:

$$\frac{}{x : [\tau] \vdash x : \tau} \mathbf{ax} \qquad \frac{\Gamma; x : [\sigma_i]_{i \in I} \vdash t : \tau}{\Gamma \vdash \lambda x. t : [\sigma_i]_{i \in I} \rightarrow \tau} \mathbf{abs}$$

$$\frac{\Gamma \vdash t : [\sigma_i]_{i \in I} \rightarrow \tau \quad (\Delta_i \vdash u : \sigma_i)_{i \in I}}{\Gamma +_{i \in I} \Delta_i \vdash t u : \tau} \mathbf{app}$$

One of the fundamental intuitions behind non-idempotent intersection is that, in this setting, a type is a **resource** that cannot be duplicated or merged/contracted and is possibly consumed under reduction. This is embodied by the linear nature of subject reduction in system \mathcal{R}_0 (see Sec. 3.3.2), which induce a decrease of measure (Proposition 3.6). Termination/normalization of typed terms often becomes straightforward (*e.g.*, Proposition 3.9).

Remark 3.8.

- A derivation tree in system \mathcal{R}_0 may be seen as *non-rigid* in the sense of Sec. 2.1.1: in an **app**-rule, there is no way to distinguish two equal argument⁷ subderivations (*i.e.* two subderivations typing the argument) and if we wanted to do it, this would change the dynamics of the system (Remark 4.2, p. 107).
- As alluded to in Sec. 3.2.1, Gardner's original presentation of system \mathcal{R}_0 does not resort to multiset intersection, but to an explicit permutation rule. Thus, Gardner's presentation gives rigid derivations trees, but it is not syntax directed. System **S** (Sec. 10.2) will be both rigid and syntax directed (in particular, there is not permutation rule in system **S**).

⁷Note however that the subderivation typing t , the left-hand side of the application, can always be distinguished from the arguments subderivations typing u

- Another argument in the favor of the representation of non-idempotent intersection with multisets (instead of using the rule **perm**) is that the \mathcal{R}_0 -judgments that can be assigned to a term t precisely corresponds to the points of the interpretation of t in the **relational model** [17]. In other words, the presentation \mathcal{G} using the rule **perm** makes some distinctions between judgments/derivations that are not relevant from the semantical point of view.

Example 3.3. As in Example 3.2, we type $I = \lambda x.x$ and K_x . Implicitly, in the **abs**-rule, if x is not in the domain context, then $\lambda x.t$ is typed with $[] \rightarrow \tau$, as in the example on the right:

$$\frac{\overline{x : [\tau] \vdash x : \tau}^{\mathbf{ax}}}{\vdash \lambda x.x : [\tau] \rightarrow \tau}^{\mathbf{abs}} \quad \frac{\overline{x : [\tau] \vdash x : \tau}^{\mathbf{ax}}}{x : [\tau] \vdash \lambda y.x : [] \rightarrow \tau}^{\mathbf{abs}}$$

Moreover, Δ is also typable in system \mathcal{R}_0 :

$$\frac{\frac{\overline{x : [[\sigma] \rightarrow \sigma] \vdash x : [\sigma] \rightarrow \sigma}^{\mathbf{ax}} \quad \overline{x : [\sigma] \vdash x : \sigma}^{\mathbf{ax}}}{x : [[\sigma] \rightarrow \sigma, \sigma] \vdash x x : \sigma}^{\mathbf{app}}}{\vdash \lambda x.x x : [[\sigma] \rightarrow \sigma, \sigma] \rightarrow \sigma}}$$

Remark 3.9 (Untyped Argument). In the **app**-rule, when I is empty *i.e.* t is typed with $[] \rightarrow \tau$, then there is no judgment typing u in the premise and $t u$ may be typed with τ for *any* term u . See also Sec. 3.4.1.

We write $\Pi \triangleright_{\mathcal{R}_0} \Gamma \vdash t : \tau$ to mean that derivation Π concludes with the judgment $\Gamma \vdash t : \tau$ and $\triangleright_{\mathcal{R}_0} \Gamma \vdash t : \tau$ to mean that $\Gamma \vdash t : \tau$ is derivable.

3.3 Discussing Subject Reduction and Subject Expansion

In this section, we present the dynamical behavior to be expected from an intersection type system (namely, subject reduction and subject expansion) and explain how this dynamical behavior (1) can be ensured (2) helps to prove semantical properties of typing (*i.e.* characterization of normalization).

In Sec. 3.3.1, we explain why subject reduction and subject expansion are crucial for an ITS to characterize normalization because subject reduction ensures termination (*i.e.* a term that is typable is normalizing) most of the times and with subject expansion, it is enough to type the normal forms to ensure that every normalizing term is typable.

We then see how subject reduction and subject expansion rely on a delicate equilibrium:

- In Sec. 3.3.2, we try to understand how subject reduction is processed in systems \mathcal{D}_0 and \mathcal{R}_0 : there may be some duplications (of argument derivations) in \mathcal{D}_0 but not in \mathcal{R}_0 . The comparison between Fig. 3.1, 3.1 on one hand and Fig. 2.6 is discussed and shows how β -reduction gives guidelines to design or understand the behavior of the intersection type systems. We then explain why subject expansion should fail in simple type systems (there, with **Curry**₀).
- In Sec. 3.3.3, we explain why having an equivalence of the form ‘When $t \rightarrow t'$, t is typable iff t' typable’ is not enough to be expected from an ITS, but that we

should also demand some kind of **context** and **type preservation** (meaning that t and t' should be typable with the same types and in the same contexts).

- In Sec. 3.3.4, we show that subject expansion does not hold in system \mathcal{D}_0 by lack of context preservation.
- Subject expansion can be retrieved by modifying system \mathcal{D}_0 into system $\mathcal{D}_{0,w}$ that allows a weakening rule (this is done in Sec. 3.3.5).

Thus, the lesson that we can learn from this Section 3.3 is that idempotency brings a lot of complications and that non-idempotent ITS are more natural to engineer than idempotent ones (despite being more recent). This is one of the reason why this thesis is dedicated to non-idempotent intersection types.

3.3.1 Uses and Behaviors of Intersection Type Systems

Usually, an intersection type system \mathcal{I} provides both a characterization of (some notion of) normalization and a proof that a given reduction strategy is complete for this normalization notion. For instance, if we are considering head normalization, we will simultaneously prove that typability characterizes head normalization (*i.e.* t is head normalizing iff t is typable in system \mathcal{I}) and that the head reduction strategy is **complete** for head normalization (*i.e.* t is head normalizing iff the head reduction strategy terminates for t).

Let us outline why, by making the following assumptions on \mathcal{I} :

1. Subject Reduction: assume $t \rightarrow t'$. If $\triangleright_{\mathcal{I}} \Gamma \vdash t : B$, then $\triangleright_{\mathcal{I}} \Gamma \vdash t' : B$.
2. Subject Expansion: assume $t \rightarrow t'$. If $\triangleright_{\mathcal{I}} \Gamma \vdash t' : B$, then $\triangleright_{\mathcal{I}} \Gamma \vdash t : B$.
3. Typing of head normal forms: if t is a HNF then t is typable in \mathcal{I} .

The structure of the proof of the claims above by using these 3 assumptions is the following:

- The typing of head normal forms and subject expansion entail that every head normalizing term is typable in system \mathcal{I}
- Subject reduction and an extra argument depending on the type system (see Remark 3.10 below) usually⁸ help us prove a termination property:

Proposition (Termination). If t is typable in system \mathcal{I} , then the head reduction strategy terminates for t .

This proves the circular implications $\langle \text{head normalizing} \Rightarrow \mathcal{I}\text{-typable} \Rightarrow \text{the head reduction strategy terminates} \Rightarrow \text{head normalizing} \rangle$ (the last one being obvious), which is enough to conclude.

⁸Subject reduction by itself is not enough to ensure a Termination Property: by declaring fixpoint equation as $o = [o]_2 \rightarrow o$ for some type variable o , it is easy to type Ω or other mute terms, while preserving subject reduction. We will observe in Sec. 10.1.3 that usually, in type systems featuring a co-inductive type grammar (they usually enjoy subject reduction), typability does not entail normalization (some mute terms are typable).

Thus, an intersection type system must be designed so that subject reduction and subject expansion hold. In the sections to come, we discuss these properties in systems \mathbf{Curry}_0 , \mathcal{D}_0 and \mathcal{R}_0 (the three of them satisfy subject reduction, but only \mathcal{R}_0 satisfies subject expansion).

Remark 3.10. The technicity of the proof of the Termination Property above varies a lot from a type system to another:

- For *idempotent* intersection type systems (or higher-order simple type systems), one resorts to Tait's Realizability Argument [100] (presented in Sec. 4.3).
- For *non-idempotent* intersection type systems, there is usually a straightforward arithmetical argument (see Sec. 3.4.3). This is one of their most interesting features, that we extensively use in this Thesis.
- For the simple type system \mathbf{Curry}_0 , there is also a Gentzen/Prawitz style proof, similar to the one use for Cut Elimination in Sequent Calculus (see for instance Chapter 4 in [49]).
- Moreover, some type systems satisfy subject reduction, but typability does not ensure normalization (see *e.g.*, the typing of Ω in Appendix A.1) *i.e.* subject reduction is not enough to ensure a Termination Property.

3.3.2 Subject Reduction and Subject Expansion

Systems \mathcal{D}_0 and \mathcal{R}_0 both enjoy subject reduction, as in system \mathbf{Curry}_0 (see Proposition 3.1) and \mathcal{R}_0 also enjoys subject expansion. We explain why in this section and then why simple type systems do not enjoy subject expansion (system \mathcal{D}_0 does not enjoy subject expansion either, but this is addressed in Sec. 3.3.4).

Proposition 3.2 (Subject Reduction for \mathcal{D}_0). Assume $t \rightarrow t'$. If $\Gamma \vdash t : B$ is derivable in system \mathcal{D}_0 , then $\Gamma \vdash t' : B$ also is.

Proposition 3.3 (Subject Reduction and Subject Expansion for \mathcal{R}_0). Assume $t \rightarrow t'$:

- *Subject Reduction:* If $\Gamma \vdash t : \tau$ is derivable in system \mathcal{R}_0 , then $\Gamma \vdash t' : \tau$ also is.
- *Subject Expansion:* If $\Gamma \vdash t' : \tau$ is derivable in system \mathcal{R}_0 , then $\Gamma \vdash t : \tau$ also is.

Proof sketch.

- In both systems, the root case $t \xrightarrow{\varepsilon} t'$ relies on a substitution lemma (Lemmas 3.1, 3.2 to come). Then, an induction (performed in the proof of Proposition 3.4) yields the general case.
- Subject expansion (in system \mathcal{R}_0) relies on a reverse substitution lemma for the root case and an induction on the position of the reduction.
- For now, we focus on graphical intuitions explaining why these propositions hold and how they work, from a global perspective.

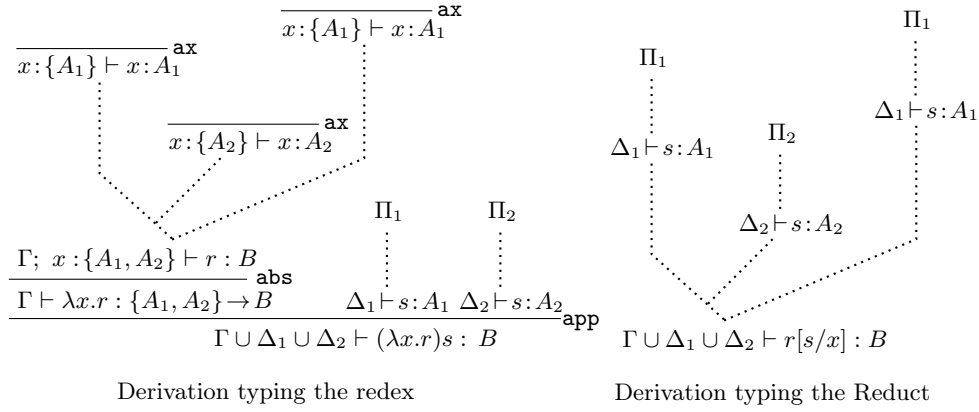


Figure 3.1: Subject Reduction in System \mathcal{D}_0

□

Subject reduction usually holds in simple type systems (*e.g.*, Curry_0) but subject expansion does not. Subject expansion does not hold in system \mathcal{D}_0 , but it can be naturally retrieved. This is addressed in Sec. 3.3.4.

As an example, we assume that there is a derivation Π typing $t = (\lambda x.r)s$ with type B in system \mathcal{D}_0 and type τ in system \mathcal{R}_0 . From that, we produce a derivation Π' typing $t' = r[s/x]$ with B (resp. τ). Figures 3.2 and 3.1 are the type-theoretic versions of Figure 2.6 representing β -reduction.

Instead of replacing occurrences of the variable x by copies of the argument s , we replace *axiom rules* typing x with *argument derivations* typing s . From the typing perspective, the maneuver is licit, because each typed occurrence of x in the derivation is replaced by an occurrence of s that has the exact same type (and the fact that we operate a *capture-free* substitution ensures that the contexts do not interact wrongly).

In system \mathcal{D}_0 , the argument derivations typing s may be duplicated, but they may not in system \mathcal{R}_0 , for reasons to be discussed below. We assume moreover that there are 3 axiom rules typing x in the subderivation typing r and moreover, that 2 of these axiom leaves involve the same type (A_1 for \mathcal{D}_0 , σ_1 for \mathcal{R}_0), whereas the third one is distinct (resp. A_2 and σ_2).

- In \mathcal{D}_0 , there is only one argument derivation Π_1 concluding with $s : A_1$: this argument derivation is *duplicated* during derivation and there is two copies of Π_1 in the derivation typing the reduct.
- In \mathcal{R}_0 , by typing constraints, there are two derivations Π_1^a and Π_1^b concluding with $s : \sigma_1$. Each one replaces an axiom rule concluding with $x : \sigma_1$. No duplication is performed. System \mathcal{R}_0 is **linear**.

A similar figure could be done for Curry_0 : in a Curry_0 -derivation Π , every occurrence of x (that is free in r) will be typed with the same type A . Each one will be replaced by a copy of the argument derivation typing s with A .

Subject expansion is the inverse process: from a derivation Π' typing t' with τ in system \mathcal{R}_0 (see remark below), we produce a derivation Π typing t with τ . We replace each subderivation typing an occurrence of s with σ_i (*i.e.* an occurrence of s that has

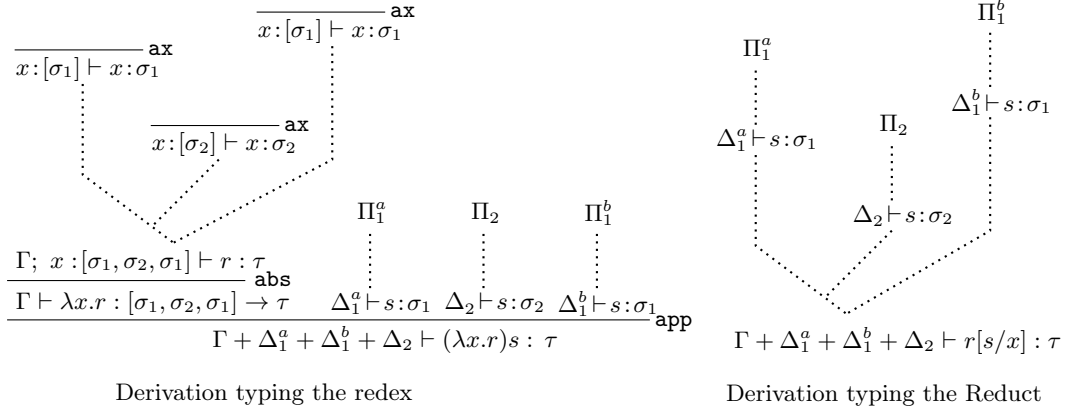


Figure 3.2: Subject Reduction and Expansion in System \mathcal{R}_0

replaced an occurrence of x during β -reduction) by an axiom rule concluding with $x : \sigma_i$. Those subderivations typing s are moved as argument derivations of the redex.

It is more subtle to explain why subject expansion fails in system \mathcal{D}_0 (see Sec. 3.3.4) but it can already be done for the simple type system Curry_0 .

Failure of Subject Expansion in Curry_0 : Let us informally explain why subject expansion does not hold in Curry_0 .

Assume that the copies of s (obtained by β -reduction) occur as different subterms of $r[s/x]$. Those subterm may be typed with two distinct types A_1 and A_2 as in the figures. So, in order to perform subject expansion and produce a derivation Π typing the redex $(\lambda x.r)s$, we should create two axiom rules concluding respectively with $x : A_1$ and $x : A_2$. This is impossible with simple type systems since a variable is assigned a *unique* type.

For instance, if $\Delta = \lambda x.x x$ and $I = \lambda x.x$, then $\Delta I \rightarrow I I \rightarrow I$. The term I is easily typable in Curry_0 with $A \rightarrow A$ (see Sec. 3.1.3). We can perform one step of subject expansion and type $I I$ with $A \rightarrow A$, where the right occurrence of I is necessarily typed with $A \rightarrow A$ and the left one with $(A \rightarrow A) \rightarrow (A \rightarrow A)$. Thus, in $I I$, the *subterm* I is typed with two distinct types. But we cannot perform the second step of subject expansion because the *variable* x (in the subderivation typing the subterm Δ) should be both typed with the distinct types $A \rightarrow A$ and $(A \rightarrow A) \rightarrow (A \rightarrow A)$.

3.3.3 Context Preservation

In this section, we explain why, from an operational point of view, it is crucial that subject reduction (resp. subject expansion) not only demands the stability of typability under reduction (resp. under expansion) *i.e.* an implication of the form “if $t \rightarrow t'$ and t is typable, then so is t' ” (resp. “if $t \rightarrow t'$ and t' is typable, then so is t ”, but also that the reduct (resp. the expanded term) should be typed with the same type in the same context.

Lemma 3.1 (Substitution (System \mathcal{D}_0)). If $\Pi_r \triangleright \Gamma; x : \{A_i\}_{i \in I} \vdash r : B$ and for all $i \in I$, $\Pi_i \triangleright \Delta_i \vdash s : \sigma_i$, then, there is a derivation Π' concluding with $\Gamma \cup (\cup_{i \in I} \Delta_i) \vdash r[s/x] : B$.

Lemma 3.2 (Substitution (System \mathcal{R}_0)). If $\Pi_r \triangleright \Gamma; x : [\sigma_i]_{i \in I} \vdash r : \tau$ and for all $i \in I$, $\Pi_i \triangleright \Delta_i \vdash s : \sigma_i$, then, there is a derivation Π' concluding with $\Gamma + (+_{i \in I} \Delta_i) \vdash r[s/x] : \tau$.

Proof. The two lemmas are proved by induction on Π_r . See *e.g.*, p. 154 for an example of a complete proof of a Substitution Lemma (Lemma 7.4). \square

Lemmas 3.1 and 3.2 immediately entail:

Lemma 3.3.

- System \mathcal{D}_0 : If $\Pi \triangleright \Gamma \vdash (\lambda x.r)s : B$, then there is a derivation Π' concluding with $\Gamma \vdash r[s/x] : B$.
- System \mathcal{R}_0 : If $\Pi \triangleright \Gamma \vdash (\lambda x.r)s : \tau$, then there is a derivation Π' concluding with $\Gamma \vdash r[s/x] : \tau$.

The above lemma is subject reduction (for \mathcal{D}_0 and \mathcal{R}_0) in the *root* case. We notice that, in both systems, there is **context preservation** in the root case: if t is typable with type B (resp. τ) in the context Γ , then t' is also typable with type B (resp. τ) in the same context Γ .

The proof of subject reduction by induction from the root case enlightens the importance of context preservation (Remark 3.11 below):

Proposition 3.4 (Subject Reduction). Assume $t \rightarrow t'$.

- System \mathcal{D}_0 : If $\Gamma \vdash t : B$ is derivable in system \mathcal{D}_0 , so is $\Gamma \vdash t' : B$.
- System \mathcal{R}_0 : If $\Gamma \vdash t : \tau$ is derivable in system \mathcal{R}_0 , so is $\Gamma \vdash t' : \tau$.

Proof. We prove the proposition for system \mathcal{D}_0 . The proof is similar for system \mathcal{R}_0 . Let us proceed by induction on the position b of the reduction $t \xrightarrow{b} t'$.

- If $b = \varepsilon$ (root case), this is Lemma 3.3.
- If $b = 1 \cdot b_1$. Then $t = t_1 t_2$ and $t' = t'_1 t_2$ for some $t_1, t'_1, t_2 \in \Lambda$ such that $t_1 \xrightarrow{b_1} t'_1$. Then Π is of the form:

$$\Pi = \frac{\Pi_1 \triangleright \Gamma_1 \vdash t_1 : \{A_i\}_{i \in I}^{\neq} \rightarrow B \quad (\Pi_{2,i} \triangleright \Gamma_{2,i} \vdash t_2 : A_i)_{i \in I}}{\Gamma \vdash t : \tau} \text{app}_{\neq}$$

for some $\Pi_1, \Gamma_1, (A_i)_{i \in I}, (\Pi_i)_{i \in I}, (\Gamma_{2,i})_{i \in I}$.

By Induction Hypothesis, there is a Π'_1 concluding with $\Gamma_1 \vdash t_1 : \{A_i\}_{i \in I}$. We then set:

$$\Pi' = \frac{\Pi'_1 \triangleright \Gamma_1 \vdash t'_1 : \{A_i\}_{i \in I}^{\neq} \rightarrow B \quad (\Pi_{2,i} \triangleright \Gamma_{2,i} \vdash t_2 : A_i)_{i \in I}}{\Gamma \vdash t : \tau} \text{app}_{\neq}$$

- If $b = 2 \cdot b_2$, we proceed likewise.
- If $b = 0 \cdot b_0$. Then $t = \lambda x.t_0$ and $t' = \lambda x.t'_0$ for some $t_0, t'_0 \in \Lambda$ such that $t_0 \xrightarrow{b_0} t'_0$. Then Π is of the form:

$$\Pi = \frac{\Pi_0 \triangleright \Gamma_0 \vdash t_0 : B_0}{\Gamma \vdash t : B} \text{abs}$$

for some Π_0 , Γ_0 and B_0 such that $B = \Gamma_0(x) \rightarrow B_0$ and $\Gamma = \Gamma_0 \setminus x$.

By Induction Hypothesis, there is a Π'_0 concluding with $\Gamma_0 \vdash t'_0 : B_0$. We then set:

$$\Pi = \frac{\Pi'_0 \triangleright \Gamma_0 \vdash t'_0 : B_0}{\Gamma \vdash t' : B} \text{abs}$$

□

Remark 3.11.

- The proof relies on the fact that not only t' is typable, but also that the type is preserved under reduction (if not, we could not handle the **app**-case: it is crucial that t'_1 is typed with the same arrow type $\{A_i\}_{i \in I} \rightarrow B$ as t_1 if we want to feed it with t_2 ! A similar argument holds when we reduce inside t_2).
- Since the type of an abstraction is computed from the context in the premise (recall that $B = \Gamma_0(x) \rightarrow B_0$ in the last case), it is also crucial that the contexts are also preserved under reduction for the induction to be correct.
- In other words, in a given type system, if subject reduction/expansion in the root case does not ensure type and context preservation, the general proof scheme of Sec. 3.3.1 will not work. However, some type systems in which subject reduction/expansion (with context preservation) only holds for *non-erasing* reduction steps are still handleable, but they rely on a more complex proof scheme. Such a system (system \mathcal{S}), characterizing strong normalization, is presented in Sec. 5.2.

3.3.4 Failure of Subject Expansion with Relevant Idempotent Intersection

In Sec. 3.3.2, we explained why subject expansion was not a natural feature of simple type systems (taking Curry_0 as an example). As we show here, the failure of subject expansion in system \mathcal{D}_0 is not so irrecoverable and can be explained w.r.t. context preservation. Before that, we explain why context preservation constrained us to prevent redundant typing of the argument in the **app**-rule.

Subject Reduction in \mathcal{D}_0 : First, let us explain why, in \mathcal{D}_0 , the rule **app** forbids the argument to be typed redundantly (see Remark 3.6). This is actually to ensure subject reduction in \mathcal{D}_0 .

Let \mathcal{D}_0^* be the variant of \mathcal{D}_0 in which an argument may be typed several times with the same type *i.e.* we remove the condition $(A_i \neq A_j)_{i,j \in I, i \neq j}$, indicated by $\{A_i\}_{i \in I}^{\neq}$ from the **app** _{\neq} -rule. Consider the left part of Figure 3.3 in which the typing of the argument is redundant (there are two distinct derivation Π^a and Π^b typing s with the same type A). Thus, this is a derivation of system \mathcal{D}_0^* but not of \mathcal{D}_0 . We set $\Gamma_+ = \Gamma \cup \Delta^a \cup \Delta^b$, so that we have $\Pi \triangleright \Gamma_+ \vdash (\lambda x.r)s : B$. For instance, let o and o' be two distinct type variables: we have $\triangleright z : \{\{o\} \rightarrow o, o\} \vdash zz : o$ and $\triangleright z : \{\{o'\} \rightarrow o, o'\} \vdash zz : o$, so that, if we type twice the argument zz , we find a \mathcal{D}_0^* -derivation Π concluding with $z : \{\{o\} \rightarrow o, o, \{o'\} \rightarrow o, o'\} \vdash (\lambda x.x)(zz) : o$ (here, $r = x$, $s = zz$).

In the subderivation typing r , there is only one axiom rule typing x . If we want to produce a derivation typing $r[s/x]$, there is only one place for Π^a or Π^b to replace the axiom rule typing x with A : we have to choose one subderivation and discard the other.

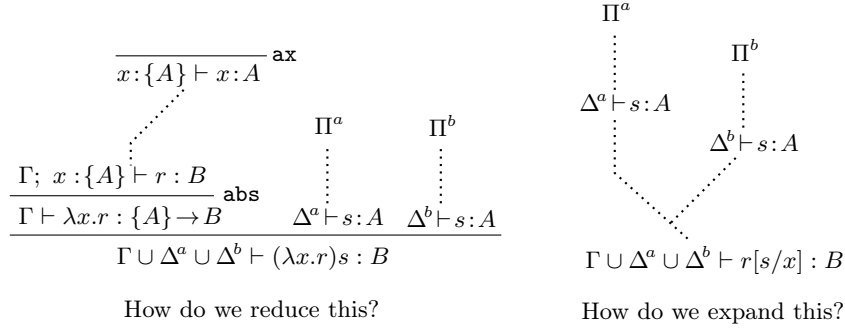


Figure 3.3: Problematic Subject Reduction and Subject Expansion

Then, we obtain a derivation Π' concluding with $\Gamma_- \vdash r[s/x] : B$, where $\Gamma_- = \Gamma \cup \Delta^a$ or $\Gamma_- = \Gamma \cup \Delta^b$. Since we can have $\Gamma \cup \Delta^a \neq \Gamma \cup \Delta^a \cup \Delta^b \neq \Gamma \cup \Delta^b$, it is possible that $\Gamma_+ \neq \Gamma_-$ and we lose context preservation (see Remark 3.11) in system \mathcal{D}_0^* . With the same example, we notice that $z : \{\{o\} \rightarrow o, o, \{o'\} \rightarrow o, o'\} \vdash z z : o$ is not derivable in \mathcal{D}_0^* .

Failure of Subject Expansion in \mathcal{D}_0 : Subject expansion does not “always” hold in \mathcal{D}_0 because of rule **app** is too restrictive in this system. Indeed, consider the derivation typing $r[s/x]$ in Figure 3.3. We set $\Gamma_+ = \Gamma \cup \Delta^a \cup \Delta^b$ so that Π' concludes with $\Gamma_+ \vdash r[s/x] : B$. For instance, let o and o' two distinct type variables: we have $\triangleright z : \{o \rightarrow o, o\} \vdash z z : o$ and $\triangleright z : \{o' \rightarrow o, o'\} \vdash z z : o$, so that there is a \mathcal{D}_0 -derivation Π' concluding with $y : \{\{o\} \rightarrow \{o\} \rightarrow o\}, z : \{\{o\} \rightarrow o, o, \{o'\} \rightarrow o, o'\} \vdash r[s/x] : o$ with $r = y x x$, $s = z z$ and $r[s/x] = y(z z)(z z)$.

Due to rule **app**_≠, we may type the argument s with A_1 only *once* in the derivation typing $(\lambda x.r)s$. This means that we have to choose Π^a or Π^b as the *unique* argument derivation of the redex and discard the other one. Then, we obtain a derivation Π concluding with $\Gamma_- \vdash (\lambda x.r)s : B$, where $\Gamma_- = \Gamma \cup \Delta^a$ or $\Gamma_- = \Gamma \cup \Delta^b$. Since $\Gamma_+ \neq \Gamma_-$ is possible, this compromise the correctness of the typing rules *outside* the redex (see Remark 3.11). With the same example, a case analysis shows that $z : \{\{o\} \rightarrow o, o, \{o'\} \rightarrow o, o'\} \vdash z z : o$ is not derivable in \mathcal{D}_0 , so that neither is $y : \{\{o\} \rightarrow \{o\} \rightarrow o\}, z : \{\{o\} \rightarrow o, o, \{o'\} \rightarrow o, o'\} \vdash (\lambda x.r)s : o$, where $(\lambda x.r)s = (\lambda x.y x x)(z z)$.

Thus, \mathcal{D}_0 is too restrictive. This problem can be bypassed by suitably allowing *weakening*: this yields system $\mathcal{D}_{0,w}$, that both enjoys subject reduction and subject expansion (Sec. 3.3.5).

3.3.5 Weakening and Irrelevant Intersection Types Systems

Weakening In Propositional Logic, the notation $A_1, \dots, A_n \vdash B$ has the intuitive meaning that, from the hypotheses A_1, A_2, \dots, A_{n-1} and A_n , we can prove the formula B *i.e.* B is a *syntactic* consequence of A_1, A_2, \dots, A_{n-1} and A_n . Of course, if we can prove formula B when we assume A_1, A_2, \dots, A_n , then *a fortiori* we can prove B when we assume A_1, A_2, \dots, A_n, A , where A can be *any* formula. This is known as **weakening** since, *a priori*, the more assumptions we make, the easier it is to prove a given formula B . Formally, the left weakening rule can be specified in Propositional Logic with the

following rule:

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}$$

where Γ is a sequence of formula A_1, \dots, A_n . Usually, weakening can be limited to axiom rules only, which yields axioms of the form:

$$\overline{\Gamma, A \vdash A}$$

This is enough to ensure that if $\triangleright \Gamma \vdash B$ is derivable, then $\triangleright \Gamma, AB$ is derivable for any formula A .

A type system is said to be **relevant** when it forbids weakening. For instance, \mathcal{D}_0 and \mathcal{R}_0 are relevant. Let us present a type system $\mathcal{D}_{0,w}$ featuring an idempotent intersection operator. System $\mathcal{D}_{0,w}$ amends the main defect of system \mathcal{D}_0 *i.e.* the failure of subject expansion (Sec. 3.3.4).

Irrelevant and Idempotent Intersection The types, contexts and judgments of $\mathcal{D}_{0,w}$ are those of system \mathcal{D}_0 . The set $\text{Deriv}_{\mathcal{D}_{0,w}}$ is defined by the following inductive rules:

$$\frac{i_0 \in I}{\Gamma; x : \{A_i\}_{i \in I} \vdash x : A_{i_0}} \text{ax}_w \quad \frac{\Gamma; x : \{A_i\}_{i \in I} \vdash t : B}{\Gamma \vdash \lambda x. t : \{A_i\}_{i \in I} \rightarrow B} \text{abs}$$

$$\frac{\Gamma \vdash t : \{A_i\}_{i \in I} \rightarrow B \quad (\Delta_i \vdash u : A_i)_{i \in I}}{\Gamma \cup (\cup_{i \in I'} \Delta_i) \vdash t u : B} \text{app}$$

Remark 3.12. Thus, contrary to system $\mathcal{D}_{0,w}$, the argument of an application may be typed redundantly: several argument derivation may be concluded with $u : A$ for some A .

Example 3.4. From Sec. 3.2.3, it can be noticed that, in system \mathcal{D}_0 , $I := \lambda x.x$ (resp. $K_x := \lambda y.x$) can only be typed with a type of the form $\{A\} \rightarrow A$ (resp. $\{ \} \rightarrow A$). This is due to the relevant rules of system \mathcal{D}_0 . System $\mathcal{D}_{0,w}$ is less restrictive:

$$\frac{}{x : \{A, B, C\} \vdash x : B} \text{ax}_w \quad \frac{}{x : \{B\}; y : \{B\} \vdash x : B} \text{ax}_w$$

$$\frac{}{\vdash \lambda x.x : \{A, B, C\} \rightarrow B} \text{abs} \quad \frac{}{x : \{B\} \vdash \lambda y.x : \{B\} \rightarrow B} \text{abs}$$

Additive Presentation Alternatively, we may consider the type system $\mathcal{D}_{0,w}^+$, that is defined inductively by the rules ax_w , abs and the rule app^+ below:

$$\frac{\Gamma \vdash t : \{A_i\}_{i \in I} \rightarrow B \quad (\Gamma \vdash u : A_i)_{i \in I}}{\Gamma \vdash t u : B} \text{app}^+$$

Notice that every $\mathcal{D}_{0,w}^+$ -derivation is a $\mathcal{D}_{0,w}$ -derivation. Conversely, from a $\mathcal{D}_{0,w}$ -derivation Π concluding with $\Gamma \vdash t : B$, we may build (by induction on Π) a $\mathcal{D}_{0,w}^+$ -derivation, that we write Π^+ , also concluding with $\Gamma \vdash t : B$. From that, systems $\mathcal{D}_{0,w}$ and $\mathcal{D}_{0,w}^+$ are equivalent.

We recall from Sec. 3.2.3 that $\Gamma \subseteq \Gamma'$ if, for all $x \in \mathcal{V}$, $\Gamma(x) \subseteq \Gamma'(x)$. The following Lemma is useful:

Lemma 3.4. If $\Gamma_- \vdash t : B$ is derivable in system $\mathcal{D}_{0,w}$ (resp. $\mathcal{D}_{0,w}^+$), then, for all context $\Gamma_+ \supseteq \Gamma_-$, the judgment $\Gamma_+ \vdash t : B$ is derivable in system $\mathcal{D}_{0,w}$ (resp. $\mathcal{D}_{0,w}^+$).

Proof. Assume that $\Pi_- \triangleright \Gamma_- \vdash t : B$. We replace, in Π_- one axiom rule concluding with say $\Gamma; x : \{A_i\}_{i \in I} \vdash x : A_{i_0}$ by that concluding with $(\Gamma; x : \{A_i\}_{i \in I}) \cup \Gamma_+ \vdash x : A_{i_0}$. Some α -renaming may be performed beforehand to prevent the capture of free variables of Γ_+ . \square

Dynamic Study of $\mathcal{D}_{0,w}^+$ Systems $\mathcal{D}_{0,w}$ and $\mathcal{D}_{0,w}^+$ enjoys both subject reduction and subject expansion.

Proposition 3.5 (Subject Reduction and Subject Expansion for $\mathcal{D}_{0,w}$). Assume that $t \rightarrow t'$:

- *Subject Reduction:* If $\Gamma \vdash t : B$ is derivable in system $\mathcal{D}_{0,w}$ or in system $\mathcal{D}_{0,w}^+$, then $\Gamma \vdash t' : B$ also is.
- *Subject Expansion:* If $\Gamma \vdash t' : B$ is derivable in system $\mathcal{D}_{0,w}$ or in system $\mathcal{D}_{0,w}^+$, then $\Gamma \vdash t : B$ also is.

Proof sketch. Let us have another look at the left derivation of Fig 3.3. This is a derivation that uses the rules **ax** and **app_w** (since the argument is typed redundantly). If we read the discussion that follows, we notice that, from a derivation Π concluding with $\Gamma_+ \vdash (\lambda x.r)s : B$ we are able to produce a derivation Π' concluding with $\Gamma_- \vdash r[s/x] : B$ where $\Gamma_- \subseteq \Gamma_+$. Since there is not *context preservation* (Sec. 3.3.3), we cannot ensure subject reduction in general case when using the rules **ax**, **abs** and **app_w** (see Remark 3.11).

Now, if we look at the right derivation of Fig 3.3 and the associated discussion, we notice that, in system \mathcal{D}_0 , from a derivation Π' concluding with $\Gamma_+ \vdash r[s/x] : B$, we are able to produce a derivation Π concluding with $\Gamma_- \vdash (\lambda x.r)s : B$ where $\Gamma_- \subseteq \Gamma_+$. Once again, since there is no context preservation, subject reduction cannot be guaranteed in the general case.

By contrast, in systems $\mathcal{D}_{0,w}$ and $\mathcal{D}_{0,w}^+$, those problems can be mended by using Lemma 3.4 in order to ensure context preservation in the root case, so that subject reduction holds. Subject expansion holds in \mathcal{D}_0 and $\mathcal{D}_{0,w}$ because the argument of an application may be typed redundantly, contrary to system \mathcal{D}_0 (see Sec. 3.3.4). \square

An Irrelevant Non-Idempotent Intersection Type System Derivations of system $\mathcal{R}_{0,w}$ are defined inductively by the following rules:

$$\frac{i_0 \in I}{\Gamma; x : [\sigma_i]_{i \in I} \vdash x : \sigma_{i_0}} \mathbf{ax}_w \qquad \frac{\Gamma; x : [\sigma_i]_{i \in I} \vdash t : \tau}{\Gamma \vdash \lambda x.t : [\sigma_i]_{i \in I} \rightarrow \tau} \mathbf{abs}$$

$$\frac{\Gamma \vdash t : [\sigma_i]_{i \in I} \rightarrow \tau \quad (\Delta_i \vdash u : \sigma_i)_{i \in I}}{\Gamma + (+_{i \in I} \Delta_i) \vdash t u : \tau} \mathbf{app}$$

Thus, we get system $\mathcal{R}_{0,w}$ from system \mathcal{R}_0 by replacing the relevant rule **ax** by the rule **ax_w**, that allows weakening.

3.4 Study of Head Normalization in System \mathcal{R}_0

In this section, we present a few applications of system \mathcal{R}_0 , mainly, the characterization of head normalizing terms (Sec. 3.4.4) and the fact that the head reduction strategy is complete for head normalization. We follow the general scheme that was outlined in Sec. 3.3.1. For that, the use of subject expansion and subject reduction is fundamental (Proposition 3.3). But this also demands (1) that we type head normal forms (Sec. 3.4.2), and (2) that we prove a Termination Property (as suggested in Sec. 3.3.1). Termination is actually very easy to prove because subject reduction is *weighted* in system \mathcal{R}_0 (Sec. 3.4.3): head reduction decreases the size of the derivations. Before that, we discuss the possible (and sometimes necessary) existence of *untyped* subterms of a typed term (Sec. 3.4.1). We also prepare for the characterizations of WN and SN by *inductively* typing every normal form.

The structure of the proof of characterization and completeness is the following:

- The typing of head normal forms and subject expansion entail that every head normalizing term is typable in system \mathcal{R}_0 .
- *Weighted* subject reduction and an additional observation (Remark 3.13) entail that, for every \mathcal{R}_0 -typable term, the head reduction strategy terminates (Termination Property).

This proves the circular implications $\langle \text{head normalizing} \Rightarrow \mathcal{R}_0\text{-typable} \Rightarrow \text{the head reduction strategy terminates} \Rightarrow \text{head normalizing} \rangle$ (the last one being obvious), which is enough to conclude.

3.4.1 Typed and Untyped Parts of a Term

As noticed in Remark 3.9 (and in Remark 3.7 for \mathcal{D}_0), the argument of an application may be untyped *e.g.*, for all term u , we may derive:

Example 3.5.

$$\frac{\frac{\frac{}{x : [\tau] \vdash x : \tau} \text{ax}}{x : [\tau] \vdash \lambda y.x : [] \rightarrow \tau} \text{abs}}{x : [\tau] \vdash (\lambda y.x)u : \tau} \text{app}}$$

Thus, the subterm u of $t = (\lambda y.x)u$ is indeed untyped. But t reduces to x and u does not occur anymore in the normal form of t (the subterm u has been erased or equivalently, the position 2 of t does not have a residual under root reduction).

On the contrary, in the derivation concluding with $\vdash \Delta : [[\sigma] \rightarrow \sigma, \sigma] \rightarrow \sigma$ given in Sec. 3.2.4, the two occurrences of x in Δ have been typed: in this derivation, every subterm of Δ is typed.

Intuitively, a typing derivation Π cannot tell us anything about the untyped parts of its subject. We may say that the untyped parts of a typed term are **invisible** for Π . Typability in system \mathcal{R}_0 is equivalent to head normalization (Proposition 3.7 to come). Indeed, by instantiating the derivation of Example 3.5 above with $u = \Omega$, we may type

$(\lambda x.y)\Omega$ with τ , but the non-normalizing subterm Ω is of course left untyped⁹ in the derivation. If it were, we could assert that Ω is head normalizing!

We further this discussion in the next section.

Remark 3.13. By the typing constraints, it is obvious that the head variable/head redex of a typed term is necessarily typed, since, to reach it, we only visit abstractions or application left-hand sides (see Figure 2.9).

The notion of *typed position* of a term t in a derivation Π typing it can be formally defined as follows:

Definition 3.1. Let Π a derivation typing a term t . We define the set $\hat{\Pi} \subset \text{supp}(t)$ of the **typed positions** in Π by the following induction on Π :

- If Π is an axiom rule ($t = x$), then $\hat{\Pi} = \{\varepsilon\} = \text{supp}(t)$.
- If Π ends with an **abs**-rule ($t = \lambda x.t_0$) and Π_0 is its immediate subderivation, then $\hat{\Pi} = \{\varepsilon\} \cup 0 \cdot \hat{\Pi}_0$.
- If Π ends with an **app**-rule ($t = t_1 t_2$), Π_1 is its left premise (concluding with say $t_1 : [\sigma_i]_{i \in I} \rightarrow \tau$) and the $(\Pi_i)_{i \in I}$ (concluding respectively with $t_2 : \sigma_i$) are its right premises, then $\hat{\Pi} = \{\varepsilon\} \cup 1 \cdot \hat{\Pi}_1 \cup \cup_{i \in I} \hat{\Pi}_i$.

In the **app**-case, if $I = \emptyset$, we only have $\hat{\Pi} = \{\varepsilon\} \cup 1 \cdot \hat{\Pi}_1$ and t_2 is not typed, as expected. This definition can be adapted for \mathcal{D}_0 or any type system. In system Curry_0 , every position of a typed term t is typed, for any given derivation.

3.4.2 Typing (Head) Normal Forms

By Sec. 3.3.1, we need to type head normal forms in \mathcal{R}_0 in order to characterize head normalization. Let us recall that a zero head normal form (Sec. 2.2.1) is a term of the form $x t_1 \dots t_q$.

Lemma 3.5. Si $t = x t_1 \dots t_q$ is a ZHNF, then t is typable in \mathcal{R}_0 (resp. in \mathcal{D}_0) with any type τ (resp. B).

Proof. This lemma could be proved by induction, using the inductive definition of ZHNF of Sec. 2.2.1. We chose here to rather give a “global” argument. The idea is to type x with $\underbrace{[] \rightarrow \dots \rightarrow []}_{q \text{ occ. of } []} \rightarrow \tau$, so that t_1, \dots, t_q are left untyped and $x t_1 \dots t_q$, as it is represented on the left part of Fig. 3.4 with $\tau = o$ (see Sec. 4.1.1 for more detail about this presentation).

More formally, let τ be a type. We write $[]^i \rightarrow \tau$ for $\underbrace{[] \rightarrow \dots \rightarrow []}_{i \text{ occ. of } []} \rightarrow \tau$. (inductively, $\tau_0 = \tau$ and $\tau_{i+1} = [] \rightarrow \tau$) We consider the derivation:

⁹We can also directly prove that Ω is not \mathcal{R}_0 -typable by reasoning *ad absurdum* on a derivation Π typing Ω

$$\frac{\frac{\overline{x : [[]^q \rightarrow \tau] \vdash x : [[]^q \rightarrow \tau}}{\text{ax}}}{x : [[]^q \rightarrow \tau] \vdash x t_1 : [[]^{q-1} \rightarrow \tau}}{\text{app}}$$

$$\frac{\begin{array}{c} \vdots \\ x : [[]^q \rightarrow \tau] \vdash x t_1 \dots t_{q-1} : [] \rightarrow \tau \end{array}}{x : [[]^q \rightarrow \tau] \vdash x t_1 \dots t_{q-1} t_q : \tau} \text{app}$$

Thus, $t = x t_1 \dots t_q$ is typable in \mathcal{R}_0 . We can proceed likewise in \mathcal{D}_0 . \square

Corollary 3.1. If t is a head normal form, then t is typable in \mathcal{R}_0 and in \mathcal{D}_0 .

Proof. Let $t = \lambda x_1 \dots x_p. x t_1 \dots t_q$ and $t_0 = x t_1 \dots t_q$. Then, by Lemma 3.5, there is a derivation Π_0 typing t_0 . When we complete Π_0 by means of *p ad hoc abs*-rules, we obtain a derivation Π typing t . \square

Invisible (Untyped) Arguments Notice that the head argument of $x t_1 \dots t_q$ (namely t_1, \dots, t_q) are *not* typed in the derivation used in the proof of Lemma 3.5 (they can even be not *typable i.e.* the HNF $x \Omega \Omega$ is typable, but the subterms $t_1 = t_2 = \Omega$ are not). With the same derivation, we can observe that the head normal form of a term, when it exists, is the minimal part of the term that we have to type: the deeper part of the term are *invisible* (see the discussion of the previous section) for the left derivation of Fig. 3.4.

According to this very discussion, if we want to characterize weak (or strong) normalization, it is important that we type every part of each normal form, what will be done now. For that, we use the fact that β -normal forms are “inductive assemblages of HNF” (Sec. 2.2.2).

Lemma 3.6. Let t be a normal form. Then there is a derivation Π in \mathcal{R}_0 and in \mathcal{D}_0 typing t such that $\hat{\Pi} = \mathbf{supp}(t)$.

Proof. We use Lemma 2.2 and reason by induction¹⁰ on the structure of t . Thus, let us write $t = \lambda x_1 \dots x_p. x t_1 \dots t_q$ with $p, q \geq 0$ and t_1, \dots, t_q normal forms.

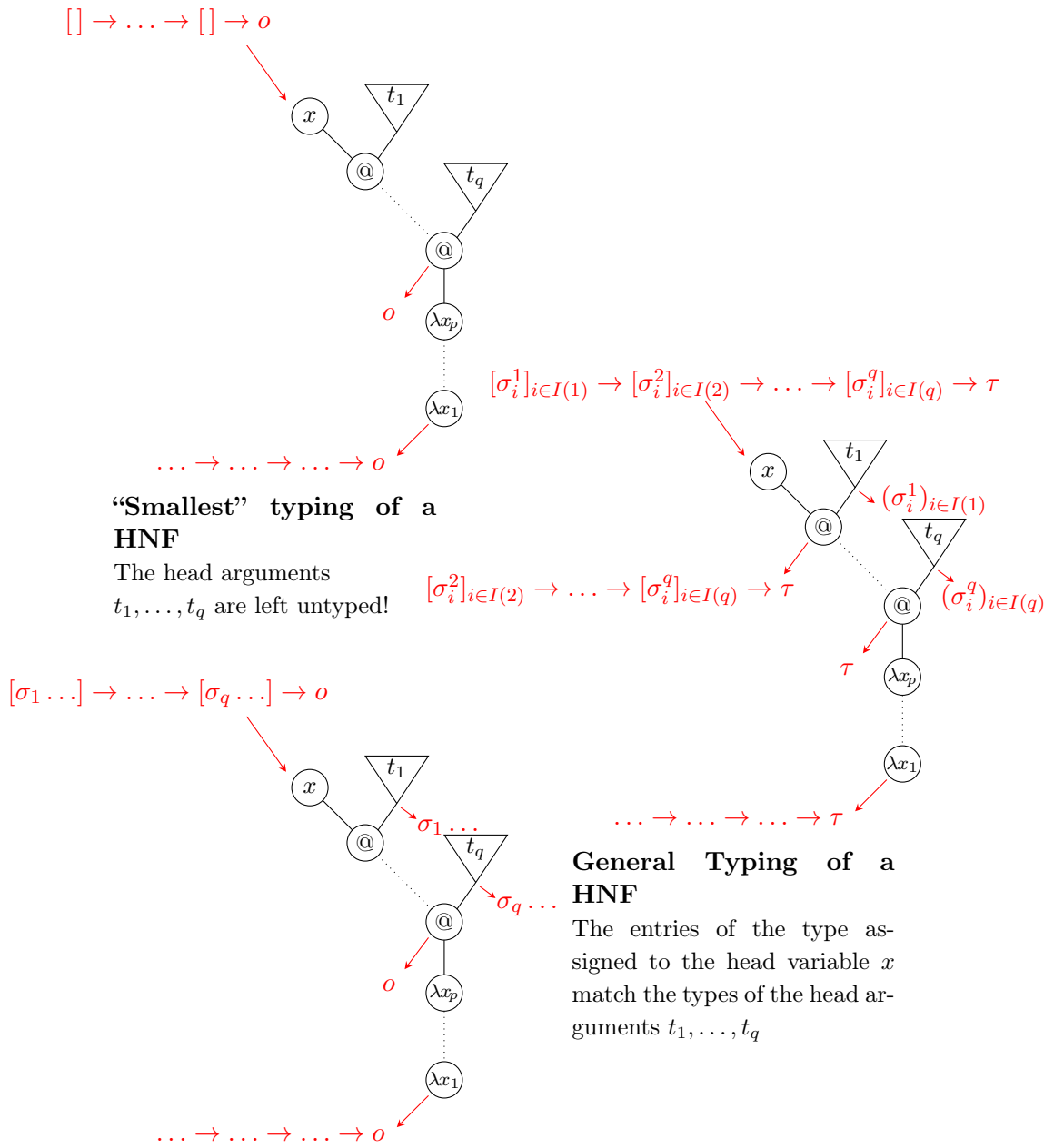
By Induction Hypothesis, for all $1 \leq i \leq q$, there is a derivation Π_i concluding with say $\Delta_i \vdash t_i : \sigma_i$ such that $\hat{\Pi}_i = \mathbf{supp}(t_i)$.

Let o be a *type variable*. The idea is to assign to x the type $[\sigma_1] \rightarrow \dots \rightarrow [\sigma_q] \rightarrow o$ so that x can be fed with the *stack* $t_1 : \sigma_1, \dots, t_q : \sigma_q$.

Formally, we set $\tau_q = o$, and, for $1 \leq k \leq q-1$, $\tau_k = [\sigma_{k+1}] \rightarrow \tau_{k+1}$ so that $\tau_k = [\sigma_{k+1}] \rightarrow \dots \rightarrow [\sigma_q] \rightarrow \tau$. We set $\Gamma_0 = x : \tau_0$ and for $0 \leq k \leq q-1$, $\Gamma_{k+1} = \Gamma_k + \Delta_k$. We consider the derivation Π_0 typing $x t_1 \dots t_q$ below:

$$\frac{\frac{\overline{\Gamma_0 \vdash x : \tau_0} \quad \Pi_1 \triangleright \Delta_1 \vdash t_1 : \sigma_1}{x : \Gamma_1 \vdash x t_1 : \tau_1}}{\vdots} \frac{x : \Gamma_{q-1} \vdash x t_1 \dots t_{q-1} : \tau_{q-1} \quad \Pi_q \triangleright \Delta_1 \vdash t_q : \sigma_q}{\Gamma_q \vdash x t_1 \dots t_{q-1} t_q : o}$$

¹⁰A more “small-step” induction could be given using Remark 2.3.



“Full” typing of a NF

- No empty type in the entries of the type assigned to x
 \sim the arguments are typed!
e.g., t_1 at least typed with σ_1 , t_2 at least typed with $\sigma_2 \dots$
- We proceed hereditarily to type the full Normal Form.

Figure 3.4: Typing (Head) Normal Forms

Then, to get a derivation Π typing t , we complete derivation Π_0 by means of *p ad hoc* abs-rules.

Since, for all $1 \leq i \leq q$, $\hat{\Pi}_i = \text{supp}(t_i)$, we have $\hat{\Pi} = \text{supp}(t)$ as desired. A similar argument works for \mathcal{D}_0 . \square

3.4.3 Weighted Subject Reduction

Definition 3.2. Let Π a derivation of system \mathcal{R}_0 . The **size** $\text{sz}(\Pi)$ of Π (also called its **weight**) is the number of rules of Π .

The size of a derivation can also be defined by using the same kind of induction as that of Definition 3.1. Equivalently, the size of Π is the number of nodes (*i.e.* judgments) that it contains. We can define $\text{sz}(\Pi)$ inductively: $\text{sz}(\Pi) = \sum_{i \in I} \text{sz}(\Pi_i) + 1$ where the Π_i are the depth one subderivations of Π ($I = \emptyset$ if Π types a variable and $\#I = 1$ if Π types an abstraction).

From Figure 3.2, we make the fundamental observation that if Π types a redex $(\lambda x.r)s$, then we may produce a derivation Π' typing $r[s/x]$ that contains few rules than Π does: indeed, the application and the abstraction of the redex have been destroyed (2 rules) as well as each axiom rule typing x (3 rules in the figure). Thus, $\text{sz}(\Pi') = \text{sz}(\Pi) - 5$. More generally, we observe that:

Proposition 3.6. Let t and t' be two terms such that $t \xrightarrow{b} t'$. If $\Pi \triangleright \Gamma \vdash t : \tau$ and b is a typed position in Π , then there is a derivation $\Pi' \triangleright \Gamma \vdash t' : \tau$ such that $\text{sz}(\Pi') < \text{sz}(\Pi)$.

Remark 3.14.

- The proposition is not true when a untyped redex is reduced: in that case, the rules of derivation Π are not affected (only the typed subterms may change) and $\text{sz}(\Pi') = \text{sz}(\Pi)$. For instance, in the derivation Π typing $t = (\lambda y.x)u$ (Example 3.5), assume that $u \xrightarrow{\varepsilon} u'$ (*i.e.* u is a redex and u' its reduct), so that $t \xrightarrow{2} (\lambda y.x)u'$. Then $\text{sz}(\Pi) = \text{sz}(\Pi') = 3$, where Π' is the same derivation with u' instead of u .
- By Remark 3.13, the head redex of a typed term, when it exists, is typed, so that, by Proposition 3.6, head reduction makes the size of a derivation strictly decrease.
- Subject reduction does not entail a decrease in size in system \mathcal{D}_0 due to the possible duplications of the argument derivations (*e.g.*, Π_1 in Figure 3.1). However, \mathcal{D}_0 -typability also entails normalization thanks to Tait's Realizability Argument, that is presented in Sec. 4.3.

3.4.4 Characterization of Head Normalization and Completeness of Head Reduction

We can now prove simultaneously the characterization of head normalization in system \mathcal{R}_0 and Proposition 2.1:

Proposition 3.7. [22, 23] Let t be a term. Then t is typable in system \mathcal{R}_0 iff t is head normalizing.

Proposition 3.8. [22, 23] A term t is head normalizing iff the head reduction strategy terminates on t .

We say then that head reduction strategy is **complete** for head normalization. Once again, the latter result is external to Type Theory and is usually proved *via* the Standardization Theorem (Sec. 11.4 of [8]).

Those propositions are a simple consequence of the two below:

Proposition 3.9. If t is typable in system \mathcal{R}_0 , then the head reduction strategy applied to t terminates.

Proof. Assume that $\Pi \triangleright \Gamma \vdash t : \tau$. We write t_n for the rank n head reduct of t (*i.e.* $t \rightarrow_{\mathbf{h}}^n t_n$) when it exists (*i.e.* a HNF is not reached within $n-1$ steps).

- By Remark 3.13, the head redex of a typed term, when it exists (*i.e.* when the term is not already a HNF), is typed.
- Thus, by Proposition 3.6 and Remark 3.13, there is a sequence of derivations $\Pi_0 \Pi_1, \Pi_2, \dots$ typing respectively t_0, t_1, t_2, \dots such that $\mathbf{sz}(\Pi_0) > \mathbf{sz}(\Pi_1) > \mathbf{sz}(\Pi_2) \dots$
- Since there is no strictly decreasing sequence of natural numbers that is of infinite length, the head reduction strategy must stop at some point *i.e.* a head normal form is reached.

□

Proposition 3.10. If t is head normalizing, then t is typable in system \mathcal{R}_0

Proof. Let t be a HN term. Thus, there are a head normal form t' and a reduction sequence \mathbf{rs} such that $t \xrightarrow{\mathbf{rs}} t'$.

- By Corollary 3.1, there is a derivation $\Pi' \triangleright \Gamma \vdash t' : \tau$ for some Γ, τ .
- By subject expansion (Proposition 3.3), there is a derivation Π concluding with $\Gamma \vdash t : \tau$.

□

Remark 3.15 (Quantitative Information given by a Derivation). If we look at the proof of Proposition 3.9, we notice that actually, the number of steps of head reduction needed to head-normalize t is bound by $\mathbf{sz}(\Pi)$. Thus, the size of \mathcal{R}_0 -derivation gives an upper bound for the length to complete the head reduction strategy (for a term that is head normalizing).

3.4.5 Order Discrimination

The *order* of a λ -term t (Definition 2.8, p. 66) is the supremal number of abstractions that prefixes a reduction of t . The same vocable exists for types. We give here a definition in system \mathcal{R}_0 that can be straightforwardly extended to any type system (with simple or strict intersection types):

Definition 3.3. The **order** of a \mathcal{R}_0 -type is the number of top-level arrows in t . Inductively, $\mathbf{order}(o) = 0$ $o \in \mathcal{O}$ is 0 and $\mathbf{order}([\sigma_i]_{i \in I} \rightarrow \tau) = \mathbf{order}(\tau) + 1$.

For instance, $\text{order}([\sigma_1, \sigma_2, \sigma_3] \rightarrow [] \rightarrow o) = 2$. A zero head normal form is a zero term and it can be easily typed with a type of order 0, by Lemma 3.5, p. 98 (see in particular the derivation on top of Fig. 3.4).

Observe now that, due to subject reduction, a typable term of order n can only be typed with a type of order $\geq n$. Indeed, if $\Pi \triangleright \Gamma \vdash t : \tau$ and $t \rightarrow^* \lambda x_1 \dots x_n. t'_0$, then, by subject reduction, there is $\Pi' \triangleright \Gamma \vdash \lambda x_1 \dots x_n. t'_0 : \tau$. But Π' must conclude with n **abs**-rule under a premise of the form $\Gamma; x_1 : [\sigma_i^1]_{i \in I(1)}; \dots; x_n : [\sigma_i^n]_{i \in I(n)} \vdash t'_0 : \tau_0$, so that $\tau = [\sigma_i^1]_{i \in I(1)} \rightarrow \dots \rightarrow [\sigma_i^n]_{i \in I(n)} \rightarrow \tau_0$ and $\text{order}(\tau) = n + \text{order}(\tau_0) \geq n$. But thanks to subject expansion, the typing of ZHNF and the Termination Property (Proposition 3.8), the case of equality is possible for any typable term t :

Proposition 3.11. Let t be a \mathcal{R}_0 -typable term whose order is n .

- If t is typable with τ , then the order of τ is at least n .
- Moreover, t is typable with a type whose order is equal to n .

Proof. • The first point was just discussed above.

- Since t is \mathcal{R}_0 -typable, then, by Proposition 3.8, t reduces to some $t' = \lambda x_1 \dots x_p. x t_1 \dots t_q$. Since the order of t' is p , we have $p = n$. By Lemma 3.5, $t'_0 = x t_1 \dots t_q$, there is a derivation Π'_0 and a context Γ such that $\Pi'_0 \triangleright \Gamma \vdash t'_0 : o$. Since the order of o is 0, after n **abs**-rules, we obtain from Π'_0 a derivation Π' concluding with $\Gamma \vdash t' : \tau$ for some context Γ and type τ whose order is n . By subject expansion, there is a derivation $\Pi \triangleright \Gamma \vdash t : \tau$. This concludes the proof. □

An interesting consequence of Proposition 3.11 is that system \mathcal{R}_0 is **order discriminating**: if two *typable* terms have distinct orders, then they do not inhabit the same types, and intuitively, system \mathcal{R}_0 is able to distinguish them. Since the set of \mathcal{R}_0 -types of a term t gives the denotation of t in the relational semantics [22], two terms with distinct orders whose semantics are not empty have distinct denotations in the model.

This is valid only for *typable* terms. One of the contribution of this thesis is to prove that the infinitary version of system \mathcal{R}_0 , called system \mathcal{R} , is also order discriminating but this time, for *all* the λ -terms (Theorem 12.2 in Chapter 12).

Chapter 4

A Few Complements on Intersection Types

In this chapter, we present some tools and observations to be sometimes used in this document. In Sec. 4.1, we discuss the representation of relevant derivations, the possible confluence of some type systems and summarize their main features. In Sec. 4.2, we present the extension of system \mathcal{R}_0 to the explicit λ -calculus Λ_{ex} (Sec. 2.4), which will be adapted in Chapter 8. Sec. 4.3, presenting Tait's Realizability Argument, is just here to appreciate the relative simplicity of non-idempotent i.t.s. w.r.t. some important aspects as termination.

4.1 A Bit of This and a Bit of That

In Sec. 4.1.1, we suggest a new presentation for the derivations in relevant intersection type systems, that makes use of the syntax-direction of the rules. We hope that this presentation is visual and is beamer-friendly. In Sec. 4.1.2, we give a counter-example to the confluence of Subject reduction in system \mathcal{R}_0 and in Sec. 4.1.3, we summarize in a table the main features of many type systems.

4.1.1 An Alternative Presentation of Relevant Derivations

When a derivation is relevant, the context of any axiom rule may be computed from the type in its right hand side (compare rule \mathbf{ax} with rule \mathbf{ax}_w). More generally, the context of each judgment may be computed from the \mathbf{ax} -rules that are located above, so that we only need to indicate the types assigned to variables in axioms. Consider for instance the derivation Π_{ex} typing Δ in system \mathcal{R}_0 :

$$\frac{\frac{\frac{\frac{x : [[o, o', o] \rightarrow o'] \vdash x : [o, o', o] \rightarrow o' \quad x : [o] \vdash x : o \quad x : [o'] \vdash x : o' \quad x : [o] \vdash x : o}{x : [[o, o', o] \rightarrow o', o, o', o] \vdash xx : o}}{\vdash \lambda x.xx : [[o, o', o] \rightarrow o', o, o', o] \rightarrow o'}}$$

Thanks to the relevance of \mathcal{R}_0 , we give with Figure 4.1 an alternative and more visual presentation of this typing derivation (we see the types assigned in axiom rules as *inputs* and the type of the term as the *output*).

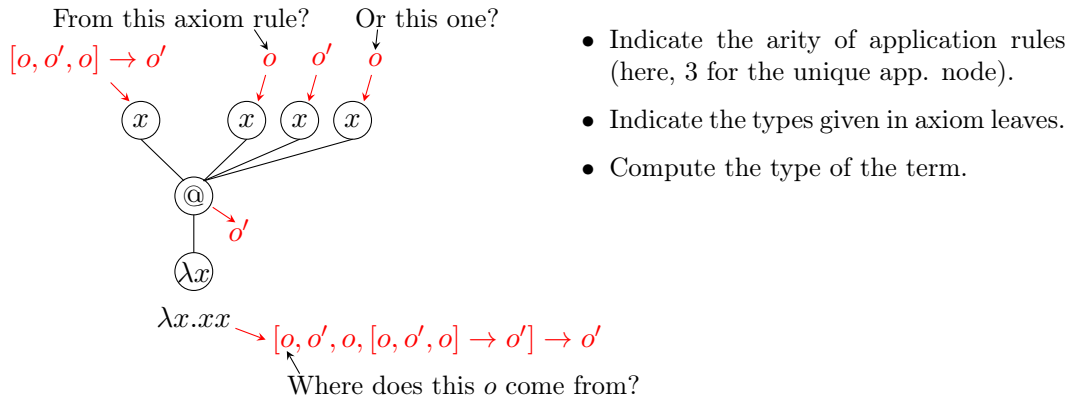


Figure 4.1: Alternative Presentation for Relevant Derivations

The same presentation can be adapted to the relevant ITS \mathcal{D}_0 . We may say that relevant derivations behave like λ -terms more than irrelevant ones do.

Remark 4.1.

- We may notice that derivations of \mathcal{R}_0 are not labelled trees in the sense of Sec. 2.1.1. Indeed, the edges of the \mathcal{R}_0 -derivations do not have labels (heuristically, there is no hard-wired way to point to one of the axiom rules assigning o to x in Figure 4.1, since no label distinguish them). Moreover, this prevents us from **tracking** a type inside a derivation. For instance, in Figure 4.1, the term Δ is typed with $[o, o', o, [o, o', o] \rightarrow o] \rightarrow o'$. The two occurrences of o in red can be traced back to the two axiom rules assigning o to x . However, there is no way to associate an occurrence in red to a particular axiom rule.
- This is to be related to the fact that “tracking” type symbol (*e.g.*, o or \rightarrow) is impossible when using multisets. More precisely, we have for instance $[\sigma, \tau] + [\sigma] = [\sigma, \sigma, \tau]$, but, in this equality, we have no way to relate one occurrence of σ in $[\sigma, \sigma, \tau]$ to $[\sigma, \tau]$ rather than $[\sigma]$ and *vice versa*.

4.1.2 Confluence (and Non-Confluence) of Type Systems

Let us explicit the notion of **reduction choices** in system \mathcal{R}_0 : when Π is a \mathcal{R}_0 -derivation concluding with $\Gamma \vdash t : \tau$ and $t \rightarrow_{\beta} t'$, then subject reduction (Proposition 3.3) ensures that there exists a derivation Π' concluding with $\Gamma \vdash t' : \tau$. But actually, there may be more than one such Π' and this will depend on the way the argument derivations (of the redex) are substituted during reduction (which is a matter of “choice”) as it is described in Sec. 3.3.2. For instance, it is possible in Fig. 3.2 p. 3.2 as soon as Π_1^a and Π_1^b are distinct. In this section, we build actual instances of reduction choices on a same derivation Π , and explain how this makes confluence fail for system \mathcal{R}_0 .

Let $t_0 = x(x x x) x x x$. For all \mathcal{R}_0 -type τ , there are two distinct derivations Π_1^τ and Π_2^τ both concluding with $\Gamma_0^\tau \vdash t_0 : \tau$, where $\Gamma_0^\tau = x : [[\] \rightarrow [\tau] \rightarrow \tau, [\tau] \rightarrow [\] \rightarrow \tau, \tau]$. Those two derivations are represented in Fig. 4.2 using the diagrams introduced in Sec. 4.1.1. By lack of space, some parts of the terms have been shrunk.

Now, we set $t = (\lambda y. y(y z))t_0$ and $t' = t_0(t_0 z)$, so that $t \rightarrow t'$. Let Π be the derivation represented in Fig. 4.3. The derivation Π would be standardly written (with

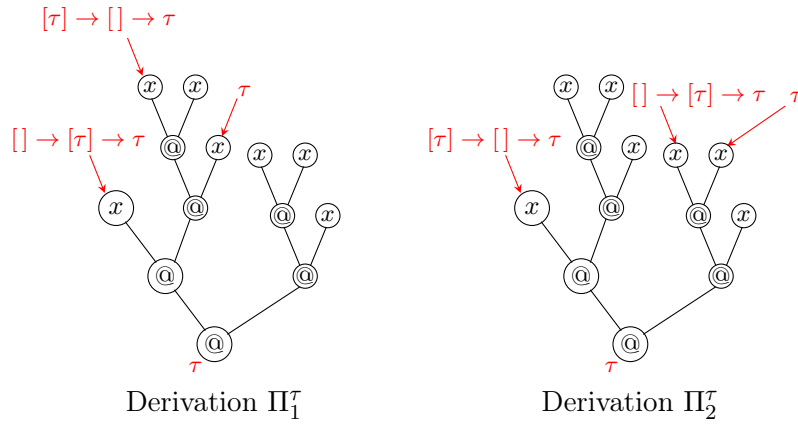


Figure 4.2: Reduction Choices in \mathcal{R}_0

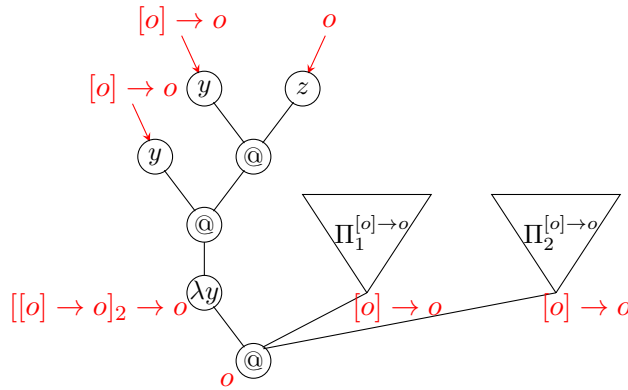


Figure 4.3: Derivation Π typing $(\lambda y. y (y z))t_0$ with o

$\tau = [o] \rightarrow o$ and $\Gamma = 2 \times \Gamma_0^\tau; z : [o]$:

$$\frac{\frac{\frac{}{y : [[o] \rightarrow o] \vdash y : [o] \rightarrow o} \quad \frac{}{z : [o] \vdash z : o}}{y : [[o] \rightarrow o]; z : [o] \vdash y z : o}}{y : [[o] \rightarrow o]_2; z : [o] \vdash y (y z) : o}}{z : [o] \vdash \lambda y. y (y z) : [[o] \rightarrow o]_2 \rightarrow o} \quad \frac{\Pi_1^\tau \triangleright \Gamma_0^\tau \vdash t_0 : \tau \quad \Pi_2^\tau \triangleright \Gamma_0^\tau \vdash t_0 : \tau}{\Gamma \vdash (\lambda y. y (y z))t_0 : o}}$$

Since, in Π , the argument of the redex is typed twice with the same type, we may replace the leftmost **ax**-rule typing y with $\Pi_1^{[o] \rightarrow o}$ or by $\Pi_2^{[o] \rightarrow o}$. Thus, from Π , we may produce two distinct derivation Π_1' and Π_2' that both type t' with o . We say then there are two **reduction choices**. We represent Π_1' and Π_2' in Fig. 4.4.

Below Π_1' and Π_2' , we have coloured in red the nodes of t' that correspond to typed positions. Thus, from the derivation Π , we have obtained two derivation Π_1' and Π_2' that type different parts of t' , the normal form of t . In particular, this shows that β -reduction is *not* confluent for derivation of system \mathcal{R}_0 : there is no Π'' such that Π_1' and Π_2' reduce to Π'' , since Π_1' and Π_2' are not reducible.

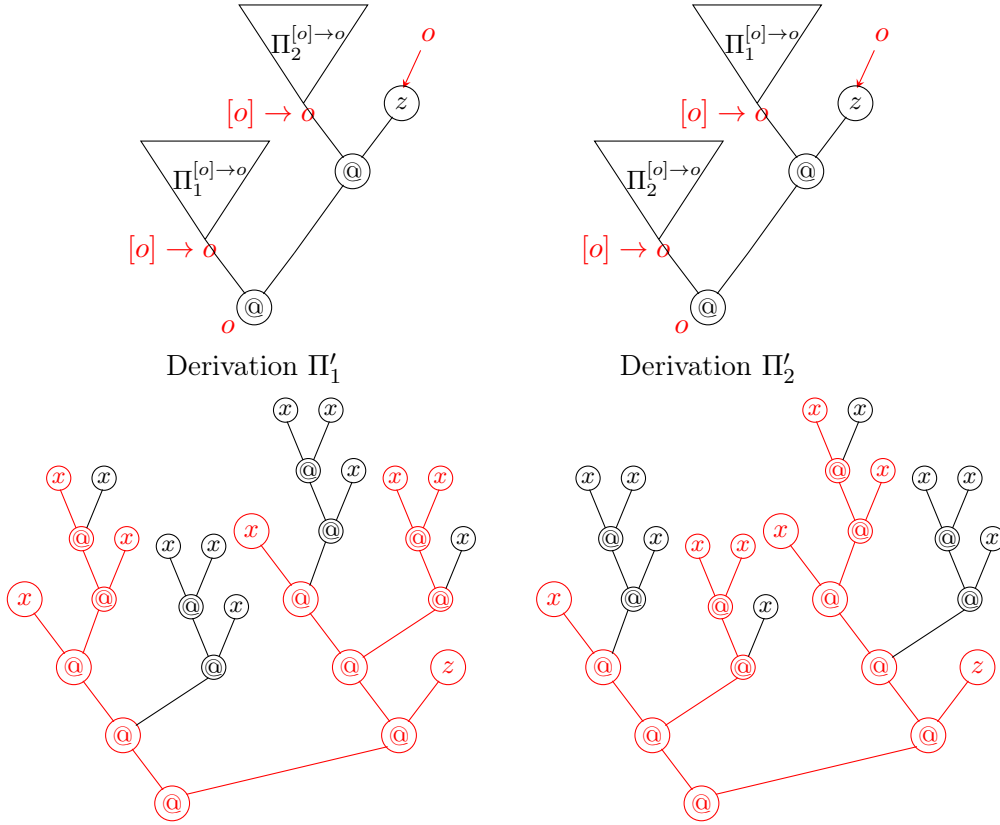


Figure 4.4: Reduction Choices

Remark 4.2.

- The non-determinism of reduction in system \mathcal{R}_0 is to be related with the fact that derivations are non-rigid labelled trees in the sense of Sec. 2.1.1 (see Remark 3.8): since the children of a same node (*i.e.* here, the argument premises of a same judgment typing an application) are not explicitly or implicitly distinguished (if they conclude equal subderivations), different reduction choices arise. For now, there is no reason to remove these reduction choices in \mathcal{R}_0 (this could be done by constraining the multiset types to never contain two occurrences of a same type, which may be seen as a bit “unnatural”).
- But, for reasons that will be shown later, this lack of rigidity will be problematic in later parts of this thesis (see Klop’s Problem and the presentation of Part IV, p. 235) and we will introduce system \mathcal{S} , which is deterministic and that can be seen as a *rigidification* of system \mathcal{R}_0 . In Chapter 13, we present two intermediary systems between system \mathcal{R}_0 and system \mathcal{S} , called system \mathcal{S}_h and \mathcal{S}_{op} , but the former is not totally rigid and whereas the latter is not syntax directed (see Remark 13.4, p. 299). The contribution of this chapter (Theorem 13.2) is to prove that, interestingly, there is no loss of expressive power with system \mathcal{S} , compared to systems \mathcal{R}_0 , \mathcal{S}_h and \mathcal{S}_{op} .

4.1.3 Type Systems and their Features (Summary)

We have seen hitherto some possible features of types systems: subject reduction (Sec. 3.3), relevance (Sec. 3.3.5) and Confluence (4.1.2). We give below a summary of features of the different type systems that we have already met. Recall from Remark 3.4, p. 83 that a type system is syntax directed the subject of a given derivation and its type give the name of the last rule of this derivation. Some short arguments:

- System $\mathcal{D}\Omega$ is Coppo and Dezani original intersection type system, as presented by Krivine (see Sec. 3.2.1).
- System \mathcal{G} is Gardner's original presentation of system \mathcal{R}_0 (Sec. 3.2.1 and Remark 3.8, p. 86), without multiset intersection but with a **perm**-rule (it is thus *almost* syntax-directed).
- System \mathcal{D}_0 is *implicitly* rigid in that, since it forbids redundant typing, two subderivations of a same \mathcal{D}_0 -derivations Π are distinct, so that subderivations can be unambiguously pointed at.
- The type systems actually enjoy *deterministic* subject reduction (for the reduction of a given redex) are actually confluent.
- System $\mathcal{D}_{0,w}$ is not confluent, since we must sometimes make choices during reduction in case of redundant typing, as it was suggested in the examples of Sec. 3.3.4. The same is true for $\mathcal{D}_{0,w}^+$. As seen in Sec. 4.2, the absence of confluence is roughly related to the fact that the typing derivations in the syntax directed i.t.s. are usually *non-rigid* trees, because in those systems, types are collapsed to avoid structural typing rules (Sec. 3.2.2).

We summarize the previous paragraph in the following table:

Type System	Syn. Dir.	Rigidity	S. Reduction	S. Expansion	Relevance	Confluence
Curry_0	+	+	+			+
$\mathcal{D}\Omega$		+	+	+		+
\mathcal{G}		+	+	+	+	+
\mathcal{D}_0	+	+	+		+	+
$\mathcal{D}_{0,w}$	+		+	+		
$\mathcal{D}_{0,w}^+$	+		+	+		
\mathcal{R}_0	+		+	+	+	
\mathbf{S}_0	+	+	+	+	+	+

We have not presented system \mathbf{S}_0 yet. System \mathbf{S}_0 is an intersection type system that features **sequences** as intersection types. The use of sequential intersection type to solve infinitary problems is one of the main contribution of this thesis (Parts III and IV).

4.2 Intersection Types for the Lambda Calculus with Explicit Substitutions

System \mathcal{R}_0 (Sec. 3.2.4) may be extended to the linear substitution calculus Λ_{ex} defined in Sec. 2.4 by adding the rule:

$$\frac{\Gamma; x : [\sigma_i]_{i \in I} \vdash t : \tau \quad (\Delta_i \vdash u : \sigma_i)_{i \in I}}{\Gamma + (+_{i \in I} \Delta_i) \vdash t(x \setminus u) : \tau} \text{exs}$$

This yields a type system for $\Lambda_{\mathbf{ex}}$ (due to Kesner and Ventura [60]) that we call $\mathcal{R}_{\mathbf{ex}}$. One of the contributions of this thesis (Chapter 8) is to extend the counterpart of this system characterizing strong normalization in $\Lambda_{\mathbf{ex}}$ so that it characterizes strong normalization in a new small-step version of the λ_{μ} -calculus.

The definition of size of a derivation is naturally extended by:

$$\mathbf{sz}\left(\frac{\Pi \triangleright \Gamma; x: [\sigma_i]_{i \in I} \vdash t: \tau \quad (\Pi_i \triangleright \Delta_i \vdash u: \sigma_i)_{i \in I} \mathbf{exs}}{\Gamma + (+_{i \in I} \Delta_i) \vdash t \langle x \setminus u \rangle : \tau} \mathbf{exs}\right) = \mathbf{sz}(\Pi) + (+_{i \in I} \mathbf{sz}(\Pi_i)) + 1$$

The notion of typed positions (*i.e.* Definition 3.1) extends to $\mathcal{R}_{\mathbf{ex}}$ in an obvious way and weighted subject reduction holds for $\mathcal{R}_{\mathbf{ex}}$:

Proposition 4.1. Assume $\Pi \triangleright_{\mathcal{R}_{\mathbf{ex}}} \Gamma \vdash t : \tau$ and $t \rightarrow t'$.

- There exists a $\mathcal{R}_{\mathbf{ex}}$ -derivation Π' such that $\Pi' \triangleright_{\mathcal{R}_{\mathbf{ex}}} \Gamma \vdash t' : \tau$.
- If the reduced redex is *typed* in Π , then there exists a $\mathcal{R}_{\mathbf{ex}}$ -derivation Π' such that $\Pi' \triangleright_{\mathcal{R}_{\mathbf{ex}}} \Gamma \vdash t' : \tau$ and $\mathbf{sz}(\Pi') < \mathbf{sz}(\Pi)$.

Let us informally understand why system $\mathcal{R}_{\mathbf{ex}}$ also enjoys weighted subject reduction. The fact that it enjoys subject reduction is no more complicated than for system \mathcal{R}_0 (see Sec. 3.3.2). How is the decrease of measure ensured?

- If $\Pi \triangleright \Gamma \vdash \mathbf{L}[(\lambda x.r)s] : \tau$ and $t = \mathbf{L}[(\lambda x.r)s] \rightarrow_{\beta_{\mathbf{x}}} \mathbf{L}[r \langle x \setminus s \rangle] = t'$: say that, for some $k \geq 0$, there are k subderivations of Π that type $(\lambda x.r)s$, the redex to be fired. Each such subderivation derivation is concluded by an **app**-rule whose right premise is an **abs**-rule. To obtain a derivation Π' typing t' , each one of those pairs of **app**-rule and **abs**-rule are replaced by a unique **exs**-rule. This yields a Π' such that $\mathbf{sz}(\Pi') = \mathbf{sz}(\Pi) - k$.
- If $\Pi \triangleright \Gamma \vdash \mathbf{C}[x] \langle x \setminus u \rangle$ and $t = \mathbf{C}[x] \langle x \setminus u \rangle \rightarrow_{\mathbf{c}} \mathbf{C}[u] \langle x \setminus u \rangle = t'$ with $|\mathbf{C}[x]|_x > 1$ and $\mathbf{C} = \mathbf{C}^x$: one occurrence of x of t (that between brackets) is replaced by u . This occurrence of x corresponds to k **ax**-rules in Π for some $k \geq 0$. To obtain a derivation Π' typing t' , we destroy those k **ax**-rules and replace them by moving k subderivations of Π typing u (with matching types). This yields a Π' typing t' such that $\mathbf{sz}(\Pi') = \mathbf{sz}(\Pi) - k$.
- If $\Pi \triangleright \Gamma \vdash \mathbf{C}[x] \langle x \setminus u \rangle$ and $t = \mathbf{C}[x] \langle x \setminus u \rangle \rightarrow_{\mathbf{d}} \mathbf{C}[u] = t'$ with $|\mathbf{C}[x]|_x = 1$ and $\mathbf{C} = \mathbf{C}^x$: this case handled as the the previous one, except that k **exs**-rules are destroyed, so that we get $\mathbf{sz}(\Pi') = \mathbf{sz}(\Pi) - 2 \times k$.
- If $\Pi \triangleright \Gamma \vdash t \langle x \setminus u \rangle : \tau$ and $t \langle x \setminus u \rangle \rightarrow_{\mathbf{w}} t$ with $x \notin \mathbf{fv}(t)$: to obtain a derivation Π' typing t , we destroy the **exs**-rule concluding Π , so that $\mathbf{sz}(\Pi') = \mathbf{sz}(\Pi) - 1$.

System $\mathcal{R}_{\mathbf{ex}}$ also enjoys subject expansion:

Proposition 4.2. Let $t \in \Lambda_{\mathbf{ex}}$. If $\Gamma \vdash t' : \tau$ is derivable in system $\mathcal{R}_{\mathbf{ex}}$, then $\Gamma \vdash t : \tau$ also is.

We easily adapt the contents of Sec. 3.4.4 to generalize Propositions 3.7 and 3.8:

Proposition 4.3. Let $t \in \Lambda_{\mathbf{ex}}$. Then t is typable in system $\mathcal{R}_{\mathbf{ex}}$ iff t is head normalizing.

Proposition 4.4. A λ_{ex} -term t is head normalizing iff the head reduction strategy applied to t terminates.

Remark 4.3 (Characterizing Strong normalization in Λ_{ex}). Kesner and Ventura [60] also defined a variant of \mathcal{R}_{ex} that characterizes strong normalization in Λ_{ex} . This type system is an extension of type system \mathcal{S} (to be presented in § 5.2) that characterizes strong normalization in λ -calculus and is a variant of system \mathcal{R}_0 .

4.3 Tait's Realizability Argument

Let us have a last look at the idempotent intersection type systems. System $\mathcal{D}_{0,w}$ and its additive variant $\mathcal{D}_{0,w}^+$ (Sec. 3.3.5) also characterize head normalization, as \mathcal{R}_0 does, and can also be used to prove that the head reduction strategy is complete for head reduction.

More precisely, systems $\mathcal{D}_{0,w}$ and $\mathcal{D}_{0,w}^+$ both satisfy subject reduction and subject expansion. It is also quite easy to type every head normal form in $\mathcal{D}_{0,w}$ and $\mathcal{D}_{0,w}^+$ (see Sec. 3.4.2). Thus, by subject expansion, every head normalizing term is easily typable in $\mathcal{D}_{0,w}$ and $\mathcal{D}_{0,w}^+$.

From a technical point of view, the big difference between $\mathcal{D}_{0,w}$ and $\mathcal{D}_{0,w}^+$ on one hand and \mathcal{R}_0 on the other (and more generally, between idempotent and non-idempotent ITS) is that the termination of the typable terms is far more convoluted to prove in the former than in the latter. To well appreciate this difference, we will present in this Section 4.3, Tait's famous Realizability Argument, introduced in [100] and later extended to higher-order type systems by Girard [45, 46, 49]. We will reason with $\mathcal{D}_{0,w}^+$ (it is slightly easier), but the proofs can be adapted to $\mathcal{D}_{0,w}$ at the cost of minor modifications.

- In Sec. 4.3.1, we see why head normalization is difficult to prove for the $\mathcal{D}_{0,w}^+$ -typed terms and why we are driven to try to reason modulo substitution, which turns out to become the Realizability Argument later.
- In Sec. 4.3.2, we explain the mechanism of Realizability and sketch a proof of the Termination Property for system $\mathcal{D}_{0,w}^+$.

4.3.1 The Failure of an Induction

Tait's Realizability Argument is often used when an induction fails because of the presence of redexes. Let us understand why, while trying to prove by induction the following statement, which corresponds to Proposition 3.9 in system $\mathcal{D}_{0,w}^+$ ("Termination Property"):

Claim 4.1. If t is typable in system $\mathcal{D}_{0,w}^+$, then t is head-normalizing and more precisely, the head reduction strategy terminates when it is applied in t .

Proof Attempt. Let us assume $\Pi \triangleright \Gamma \vdash t : B$. We try to reason by induction on the structure of Π . Only the last subcase will fail.

- When Π contains only one **ax**-rule: then $t = x$ for some $x \in \mathcal{V}$. Of course, t (that is x) is HN.

- When the last rule of Π is an **abs**-rule: then $t = \lambda x.t_0$ for some t_0 and Π is of the form:

$$\frac{\Pi_0 \triangleright \Gamma; x : \{A_i\}_{i \in I} \vdash t_0 : B_0}{\Gamma \vdash t : B}$$

with $B = \{A_i\}_{i \in I} \rightarrow B_0$. By induction hypothesis, t_0 is HN, so $t = \lambda x.t_0$ is obviously HN.

- When the last rule of Π is an **app**-rule: then $t = t_1 t_2$ and as it turns out, this case cannot be directly handled and makes the proof fail. Indeed, Π is of the form:

$$\frac{\Pi_1 \triangleright \Gamma \vdash t_1 : \{A_i\}_{i \in I} \rightarrow B \quad (\Pi_i \triangleright \Gamma \vdash t_2 : A_i)_{i \in I}}{\Gamma \vdash t : B}$$

By induction hypothesis, we may assert that both t_1 and t_2 are head normalizing. Let us consider two cases:

- The term t is of order 0. Thus, t_1 may be reduced to a ZHNF $x u_1 \dots u_q$, so that $t = t_1 t_2$ may be reduced to $x u_1 \dots u_q t_2$, which is also a ZHNF. So t is HN and the induction also works in that case.
- The order of t is ≥ 1 . Thus, t_1 may be reduced to an HNF that is an abstraction $\lambda x.t_0$. But even if $\lambda x.t_0$ is a HNF, the term $t' = (\lambda x.t_0)t_2$ is a redex and is not a HNF and right now, we cannot assert anything about t' (in particular, nothing ensures yet that t' is head normalizing). If we reduce t' , we obtain $t'' = t_0[t_2/x]$. Can we easily assert that t'' is HN? No, even if t_2 is a HNF: for instance, if $t_2 = \lambda y.t'_2$ is a HNF and $t_0 = x u_1 \dots u_q$ with $q \geq 1$, then we have $t'' = (\lambda y.t'_2) u_1[t_2/x/x] \dots u_q[t_2/x]$, which is a head-reducible term and there is no reason why a HNF could be reached from t'' . Can we still proceed by induction? There again, no: since in $\mathcal{D}_{0,w}^+$, the argument derivations of redexes may be duplicated during reduction, the size of the derivation t'' may be bigger than those typing t' and t . The term t'' itself may be bigger than t' and t .

Thus, as it was announced, trying to simply reason by induction on the structure of Π may not prove Claim 4.1.

□

If we give an attentive look at this proof attempt, we notice that it fails because, given two terms t and u , “ t and u are head normalizing” does not imply (without additional hypothesis) that $t u$ is head normalizing (thus, we could not assert that $(\lambda x.t_0)t_2$ was HN in the last case). For instance, Δ is a head normal form, but $\Omega = \Delta \Delta$ is not head normalizing.

But still, in the last subcase ($t = t_1 t_2$ and t_1 is of order ≥ 1), it may be felt that intuitively, in order to prove that t is head normalizing, we *have to* prove that $t'' = t_0[t_2/x]$ (*i.e.* the head normal form t_0 in which x was substituted with t_2) is head normalizing *i.e.* morally, we have to perform the reduction $(\lambda x.t_0)t_2 \rightarrow t_0[t_2/x]$ to show that t is HN.

We should also make use of the fact that the term t'' obtained by substitution is typable (by subject reduction) since the assumption “ t_0 and t_2 are HN” alone is not enough to prove that $t_0[t_2/x]$ is HN (*e.g.*, with $t_0 = x x$ and $t_2 = \Delta$).

The difficulty to get this is that the head normalization of t_0 and t_2 does not easily imply that of t'' and moreover, that t'' may be way bigger than t , as it was noticed

above. We may say that the idea behind Tait's Realizability Argument, which is of higher-order (see Remark 4.5) and that we are going to present now is to reason on typed terms *modulo substitution* (of typed terms) so that the head normalization of $t'' = t_0[t_2/x]$ is treated as the same level as that of t_0 . This may be seen with the Lemma of Interpretation (Lemma 4.4) below.

4.3.2 Interpretation

For all type B :

- Let $\mathbf{R}(B)$ be the set of the terms t such that the head reduction terminates and that are typable with type B in $\mathcal{D}_{0,w}^+$ (i.e. $t \in \mathbf{R}(B)$ iff the head reduction strategy starting at t terminates and there is a $\mathcal{D}_{0,w}^+$ -derivation Π and a context Γ such that $\Pi \triangleright \Gamma \vdash t : B$).
- Let $\mathbf{R}_0(B)$ be the set of zero head normal forms that are typable with type B in $\mathcal{D}_{0,w}^+$.

We notice again the importance of zero head normal forms, as in Sec. 3.4.2: their use is illustrated in the proof of the pivotal Lemma 4.2 below. The definition of \mathbf{R} and \mathbf{R}_0 naturally extends to set types and we set $\mathbf{R}(\{A_i\}_{i \in I}) = \bigcap_{i \in I} \mathbf{R}(A_i)$ and $\mathbf{R}_0(\{A_i\}_{i \in I}) = \bigcap_{i \in I} \mathbf{R}_0(A_i)$. We call $\mathbf{R}(B)$ the set of **reducibility candidates** of type B .

Definition 4.1. Let $X \subseteq \Lambda$. We say that X is **saturated** if, for all $x, r, s, t_1 \dots t_q$, $r[s/x]t_1 \dots t_q \in X$ implies that $(\lambda x.r)t_1 \dots t_q \in X$.

Thus, X is saturated if X is stable under "head expansion". Since $\mathcal{D}_{0,w}$ satisfies subject expansion, for all type B (resp. set type $\{A_i\}_{i \in I}$), $\mathbf{R}(B)$ (resp. $\mathbf{R}(\{A_i\}_{i \in I})$) is saturated.

Definition 4.2. Let $X, Y \subseteq \Lambda$. We set $X \rightarrow Y = \{t \in \Lambda \mid \forall u \in X, tu \in Y\}$.

Lemma 4.1.

- Let $X, X', Y, Y' \subseteq \Lambda$. If $X \supseteq X'$ and $Y' \subseteq Y$, then $X \rightarrow Y \subseteq X' \rightarrow Y$.
- Let $X, Y \subseteq \Lambda$. If Y is saturated, then $X \rightarrow Y$ is saturated.

Proof. Straightforward. □

Definition 4.3.

- An **interpretation** is a function ϕ from \mathcal{O} to $\mathbf{P}(\Lambda)$ such that, for all $o \in \mathcal{O}$ and $t \in \phi(o)$, the term t is $\mathcal{D}_{0,w}^+$ -typable with o .
- An interpretation ϕ is saturated if, for all $o \in \mathcal{O}$, $\phi(o)$ is saturated.
- An interpretation ϕ is **HN-suitable** if, for all $o \in \mathcal{O}$, $\phi(o)$ is saturated and $\mathbf{R}_0(o) \subseteq \phi(o) \subseteq \mathbf{R}(o)$.
- The **HN-interpretation** is the interpretation defined by $\phi(o) = \mathbf{R}(o)$ for all $o \in \mathcal{O}$.

The HN-interpretation is a HN-suitable interpretation, since $\mathbf{R}(o)$ is obviously saturated.

Let ϕ be an interpretation. We extend ϕ on $\mathbf{Typ}_{\mathcal{D}_0}$ and on the set of set types by mutual *induction*:

- We set $\phi(\{A_i\}_{i \in I}) = \bigcap_{i \in I} \phi(A_i)$.
- We set $\phi(\{A_i\}_{i \in I} \rightarrow B) = \phi(\{A_i\}_{i \in I}) \rightarrow \{A_i\}_{i \in I}$.

Lemma 4.2. For all set type $\{A_i\}_{i \in I}$ and type B , $\mathbf{R}_0(\{A_i\}_{i \in I}) \rightarrow \mathbf{R}(B) \subseteq \mathbf{R}(\{A_i\}_{i \in I} \rightarrow B)$ and $\mathbf{R}(\{A_i\}_{i \in I}) \rightarrow \mathbf{R}_0(B) \supseteq \mathbf{R}_0(\{A_i\}_{i \in I} \rightarrow B)$.

Proof.

- Let $t \in \mathbf{R}_0(\{A_i\}_{i \in I}) \rightarrow \mathbf{R}(B)$ and $x \in \mathcal{V}$. In particular, t is typable with $\{A_i\}_{i \in I} \rightarrow B$. Since $x \in \mathbf{R}_0(\{A_i\}_{i \in I})$, $tx \in \mathbf{R}(B)$. Thus, the head reduction strategy starting at tx terminates. This easily¹ implies that it also terminates when starting at t (note that x is a variable and not a general term). Thus, $t \in \mathbf{R}(\{A_i\}_{i \in I} \rightarrow B)$.
- Let $t \in \mathbf{R}_0(\{A_i\}_{i \in I} \rightarrow B)$. Thus, $t = x t_1 \dots t_q$ for some x, t_1, \dots, t_q . Let $u \in \mathbf{R}(\{A_i\}_{i \in I})$. Then, $t u$ is also a zero head normal form, so and $t u$ is typable with B , so $t u \in \mathbf{R}_0(B)$. Since this holds for all $u \in \mathbf{R}(\{A_i\}_{i \in I})$, $t \in \mathbf{R}_0(\{A_i\}_{i \in I} \rightarrow B)$.

□

Lemma 4.3. Let ϕ be an interpretation. For all type B :

- For all $t \in \Lambda$, if $t \in \phi(B)$, then t is \mathcal{D}_0 -typable with B .
- If ϕ is saturated, then $\phi(B)$ is saturated.
- If ϕ is HN-suitable, then $\mathbf{R}_0(B) \subseteq \phi(B) \subseteq \mathbf{R}(B)$.

The three statements are also true for set types.

Proof. By induction on B , using Lemma 4.1 for the second point and Lemma 4.2 for the third one. □

Assume that ϕ is *the* HN-interpretation (Definition 4.3). Thus, $\phi(B) = \mathbf{R}(B)$ when B is a type variable. Lemma 4.3 seems to only ensure that $\phi(B) \subseteq \mathbf{R}(B)$ when B is not a type variable. However, the Lemma of Interpretation below easily implies the converse inclusion: actually, we have $\phi(B) = \mathbf{R}(B)$ for all type B .

Lemma 4.4 (Interpretation). Let ϕ be a saturated interpretation.

Assume that $\triangleright_{\mathcal{D}_{0,w}^+} x_1 : \{A_{1,i}\}_{i \in I(1)}; \dots; x_p : \{A_{p,i}\}_{i \in I(p)} \vdash t : B$ and, for all $k \in \{1, \dots, p\}$, $u_k \in \phi(\{A_{k,i}\}_{i \in I(k)})$. Then $t[u_1/x_1, \dots, u_n/x_n] \in \phi(B)$.

Intuitively, this lemma means that typing is compatible with interpretations *i.e.* may help us to compute realizability candidates by substituting them to free variables: we use reducibility candidates of types $\{A_{k,i}\}_{i \in I(k)}$ as entries, to output a reducibility candidate of type B . This brings us close to reasoning *modulo substitution* as we suggested in Sec. 4.3.1.

¹If t is a zero term, it is obvious. If $t \rightarrow_h^* \lambda y. t_0$, then $tx \rightarrow_h^* (\lambda y. t_0)x \rightarrow_h t_0[x/y] \rightarrow_h^n t'$ for some $n \geq 0$ and HNF t' . A straightforward induction on n shows that $t' = t'_0[x/y]$ for t'_0 satisfying $t_0 \rightarrow_h^n t'_0$. Since t' is a HNF, t'_0 also is. Since $t \rightarrow_h^* \lambda y. t_0 \rightarrow_h^n \lambda y. t'_0$, t is HN.

Proof. By induction on the structure of t . This induction is interesting, because it makes a tight use of the design of HN-suitability (cf. Remark 4.4).

We denote by Π a $\mathcal{D}_{0,w}^+$ -derivation concluding $\Gamma \vdash t : B$, with $\Gamma = x_1 : \{A_{1,i}\}_{i \in I(1)}; \dots; x_p : \{A_{p,i}\}_{i \in I(p)}$. We write $t[\vec{u}/\vec{x}]$ for $t[u_1/x_1, \dots, u_n/x_n]$.

- Case $t = x$: obvious. Indeed, $x = x_k$ for some k and Π contains only one axiom rule. We have $B = A_{k,i_0}$ for some i_0 . Thus, $t[\vec{u}/\vec{x}] = u_k \in \phi(\{A_{k,i}\}_{i \in I(k)}) \subseteq \phi(A_{k,i_0}) = \phi(B)$.
- Case $t = \lambda x.t_0$: then $B = \{A_i\}_{i \in I} \rightarrow B_0$ for some $\{A_i\}_{i \in I}$ and B_0 , and Π is of the form:

$$\frac{\Gamma; x : \{A_i\}_{i \in I} \vdash t_0 : B_0}{\Gamma \vdash \lambda x.t_0 : \{A_i\}_{i \in I} \rightarrow B_0}$$

By induction hypothesis, for all $u \in \phi(\{A_i\}_{i \in I})$, $t_0[\vec{u}/\vec{x}, u/x] \in \phi(B_0)$. Since, by Lemma 4.3, $\phi(B_0)$ is saturated, we have $(\lambda x.t_0)[\vec{u}/\vec{x}]u \in \phi(B_0)$. Since this holds for all $u \in \phi(\{A_i\}_{i \in I})$, then, by definition of $X \rightarrow Y$, $\lambda x.t_0[\vec{u}/\vec{x}] \in \phi(\{A_i\}_{i \in I} \rightarrow B_0)$ i.e. $t[\vec{u}/\vec{x}] \in \phi(B)$.

- Case $t = t_1 t_2$: Π is of the form:

$$\frac{\Gamma \vdash t_1 : \{A_i\}_{i \in I} \rightarrow B \quad (\Gamma \vdash t_2 : A_i)_{i \in I}}{\Gamma \vdash t_1 t_2 : B}$$

By induction hypothesis, $t_1[\vec{u}/\vec{x}] \in \phi(\{A_i\}_{i \in I} \rightarrow B)$ and $t_2[\vec{u}/\vec{x}] \in \phi(A_i)$ for all $i \in I$, so that $t_2[\vec{u}/\vec{x}] \in \phi(\{A_i\}_{i \in I})$. Since, by definition, $\phi(\{A_i\}_{i \in I} \rightarrow B) = \phi(\{A_i\}_{i \in I}) \rightarrow \phi(B)$, we have $t_1[\vec{u}/\vec{x}] t_2[\vec{u}/\vec{x}] \in \phi(B)$ i.e. $t[\vec{u}/\vec{x}] \in \phi(B)$.

□

Remark 4.4.

- We note how saturation is used in the abstraction case.
- We see that, in the abstraction case, we need that $\phi(\{A_i\}_{i \in I} \rightarrow B) \supseteq \phi(\{A_i\}_{i \in I}) \rightarrow \phi(B)$ (to ensure that $\lambda x.t_0[\vec{u}/\vec{x}]$ is in the interpretation of $\{A_i\}_{i \in I} \rightarrow B$), and in the application case, that $\phi(\{A_i\}_{i \in I} \rightarrow B) \subseteq \phi(\{A_i\}_{i \in I}) \rightarrow \phi(B)$ (to ensure that $t_1[\vec{u}/\vec{x}] t_2[\vec{u}/\vec{x}]$ is in the interpretation of B). This strongly suggests that the definition of interpretation of a type B by *induction* on B (see Definition 4.3) is canonical.

Corollary 4.1.

- Let ϕ be a HN-suitable interpretation. For all term $t \in \Lambda$, if $\triangleright_{\mathcal{D}_{0,w}^+} \Gamma \vdash t : B$ then $t \in \phi(B)$.
- For all term $t \in \Lambda$, if $\triangleright_{\mathcal{D}_{0,w}^+} \Gamma \vdash t : B$, then t is head normalizing.
- The HN-interpretation ϕ is the only HN-suitable interpretation and actually, $\phi(B)$ is the set of λ -terms that are $\mathcal{D}_{0,w}^+$ -typable with B .

Proof. Let ϕ be a HN-suitable interpretation.

- The context Since ϕ is HN-suitable, by Lemma 4.3 for all $x \in \mathcal{V}$ and all set type $\{A_i\}_{i \in I}$, $x \in \phi(\{A_i\}_{i \in I})$. By applying Lemma 4.4 to $u_k = x_k$ for all $k \in \{1, \dots, p\}$, we obtain $t \in \phi(B)$.
- Since ϕ is HN-suitable, then $\phi(B) \subseteq \mathbf{R}(B)$, so the head reduction strategy terminates for t .
- The first point entails that $\Lambda(B)$, the set of the terms that are $\mathcal{D}_{0, \mathbf{w}}^+$ -typable with B , is a subset of $\phi(B)$. Since the converse inclusion holds by Lemma 4.3, $\phi(B) = \mathbf{R}(B) = \Lambda(B)$.

□

Remark 4.5 (Observations on Tait's Argument). • Tait's Realizability argument is of higher order: we must reason both on terms (*e.g.*, the induction on the structure of t used in the proof of the Lemma of Interpretation (Lemma 4.4)) and on types (*e.g.*, the inductions on the structure of B in Lemma 4.3).

- It is also used to prove normalization in higher order simple type systems, as recalled in Remark 3.10.

Chapter 5

Characterizing Weak and Strong Normalization

In this chapter, we explain how we can characterize weak and strong normalization with non-idempotent intersection types. For weak normalization, we will follow the general scheme presented in Sec. 3.3.1 that was implemented, for head normalization, in Sec. 3.4.4. In both cases (WN and SN), Termination will be easy to prove, as in Sec. 3.4.3 and will rely upon a simple arithmetic argument (there is no strictly decreasing sequence of natural numbers of infinite length) *via* weighted subject reduction (*e.g.*, Proposition 3.6). However, since strong normalization is not stable under subject expansion, the proof structure of Sec. 3.3.1 cannot be used and must be adapted. Both characterizations provide upper bounds on interesting reduction sequences, as in the case of head normalization.

5.1 Characterizing Weak Normalization

We recall (Definition 2.4) that a term t is weakly normalizing if there is a reduction path from t to a normal form. In this section, we explain how weak normalization is characterized by a class of \mathcal{R}_0 -derivations

5.1.1 Positive and Negative Occurrence of a Type

In this section, we present some technical points that will be needed in the discussions to follow.

As it will turn out, we need to describe the *occurrences* of $[]$ in a type τ : the empty type $[]$ occurs in τ if there is an arrow nested in τ whose source is empty *e.g.*, $[]$ occurs in $[o, [o, o'] \rightarrow [] \rightarrow o] \rightarrow o$ but not in $[o, [o, o'] \rightarrow [o'] \rightarrow o] \rightarrow o$.

Let us be more precise. The target of an arrow type is considered as *positive* whereas its source is considered as *negative*. This idea is extended inductively with respect to nesting (here, for the occurrences of $[]$ in τ):

Definition 5.1. The **positive** and **negative occurrences** of $[]$ are defined inductively as follows:

- $[]$ occurs positively in $[]$.
- $[]$ occurs positively (resp. negatively) in $[\sigma_i]_{i \in I}$ if there exists $i \in I$ s.t. $[]$ occurs positively (resp. negatively) in σ_i .

- $[]$ occurs positively (resp. negatively) in $[\sigma_i]_{i \in I} \rightarrow \tau$ if $[]$ occurs positively (resp. negatively) in τ or negatively (resp. positively) in $[\sigma_i]_{i \in I}$.

We say that there is a **top-level occurrence** of $[]$ in τ if τ is of the form $[\sigma_i^1]_{i \in I(1)} \rightarrow \dots [\sigma_i^q]_{i \in I(q)} \rightarrow \tau$ where some $I(k)$ is empty. A top-level occurrence is negative. Inductively, there is a top-level occurrence of $[]$ in $[\sigma_i]_{i \in I} \rightarrow \tau$ if $I = \{\}$ or there is top-level occurrence of $[]$ in τ .

The **sign** of an occurrence of $[]$ in a type τ is given by the parity of the number of nestings of this occurrence in multiset types and an occurrence of $[]$ is top-level when it is not nested. Note that $[]$ occurs both negatively in $[] \rightarrow o$ and in $[[\] \rightarrow o]$ according to the definition above. We say that $[]$ occurs positively (resp. negatively) in a context Γ if it occurs positively (resp. negatively) in $\Gamma(x)$ for some $x \in \mathcal{V}$.

5.1.2 Unforgetfulness

By Proposition 3.9, if a term t is typed by a derivation Π , then t is head normalizing. However, if a variable x is assigned a type with a top-level occurrence of $[]$, then some of the possible arguments t_k of x in $x t_1 \dots t_q$ may be left untyped. For instance, if x is assigned $[o] \rightarrow [] \rightarrow [o, o'] \rightarrow o$, then t_2 is not typed in $x t_1 t_2 t_3$, whereas t_1 is typed with o and t_3 with o and o' .

This explains why, in system \mathcal{R}_0 , typability ensures head normalization but *not* weak normalization, since, given a \mathcal{R}_0 -derivation Π typing a term t , some of the head arguments of the head normal form of t may be left untyped (modulo some subject reduction steps). As observed in Section 3.4.2, a derivation Π typing a term t can be a certificate of weak normalization only if (still modulo subject reduction steps) Π types every subterm of the normal form of t .

Thus, we must prevent that an argument of a variable (occurring in a normal form) is left untyped. This may be achieved by considering the notion of unforgetful derivation, so that the following property will hold: “a term is WN iff it is unforgetfully typable” (Proposition 5.1).

Although we introduce the vocable “unforgetful” (which is motivated in Remark 5.1), this criterion is well-known since the 80s for the characterization of weak normalization in idempotent intersection type systems (see *e.g.*, Theorem 4.13 of [68]):

Definition 5.2.

- A type τ is **unforgetful** when $[]$ does not occur negatively in τ .
- A judgment $\Gamma \vdash t : \tau$ is *unforgetful* when $[]$ occurs neither negatively in Γ nor positively in τ .
- A derivation is *unforgetful* when it concludes with an unforgetful judgment.

We make the following claims, that are narrowly related to each other:

- An unforgetful derivation is a certificate of weak normalization (*i.e.* if t is unforgetfully typable, then t is WN) and conversely, a weakly normalizing term is unforgetfully typable.
- If an unforgetful derivation types a term t , then, modulo some subject reduction steps, the normal form of t is *completely* typed (there is no untyped subterm in the NF of t).

- If t is unforgetfully typed by Π and a variable occurrence of x is typed in Π , then the arguments of x are not left untyped (at least when this variable occurrence is Böhm stable). Actually, those arguments are also unforgetfully typed.

Those claims will be proved in the next sections and addressed in Remarks 5.1, 5.2 and 5.3. This will provide a type-theoretic characterization of weak normalization and a proof of completeness of the minimal reduction strategy w.r.t. weak normalization.

Remark 5.1. If τ is an unforgetful type, then there is no top-level occurrence of $[]$ in an unforgetful type τ , so that, if the head variable x of $t = x t_1 \dots t_q$ has been assigned an unforgetful type τ , then each head argument t_k ($1 \leq k \leq q$) must be typed *at least* once, since there is no top-level occurrence of $[]$ in τ .

5.1.3 Unforgetfulness and Typing Rules

The first observation to be made is that unforgetfulness is stable under subject reduction and subject expansion, by context preservation. Let us prove now some technical lemmas. A look at the **abs**-rule in \mathcal{R}_0 yields:

Lemma 5.1. Let Π a derivation typing an abstraction $\lambda x.t_0$ and Π_0 , the subderivation of Π typing t_0 . Then Π is unforgetful iff Π_0 is unforgetful.

Proof. Π is of the form:

$$\Pi = \frac{\Pi_0 \triangleright \Gamma; x : [\sigma_i]_{i \in I} \vdash t_0 : \tau}{\Gamma \vdash \lambda x.t_0 : [\sigma_i]_{i \in I} \rightarrow \tau}$$

The statement is then a straightforward consequence of Definition 5.2. \square

Remark 5.2.

- Let us call *co-unforgetful* a type τ in which $[]$ does not occur positively. Thus, by Definition 5.2 a judgment is unforgetful when the types in the context are unforgetful and the type of the subject is co-unforgetful.
- The **abs**-rule allows a transition from $\Gamma, x : [\sigma_i]_{i \in I} \vdash t : \tau$ to $\Gamma \vdash \lambda x.t : [\sigma_i]_{i \in I} \rightarrow \tau$. Notice that a negative (resp. positive) occurrence of $[]$ in $[\sigma_i]_{i \in I}$ corresponds to a positive (resp. negative) one in $[\sigma_i]_{i \in I} \rightarrow \tau$ *i.e.* some negative (resp. positive) occurrences in the *context* of the judgment the premise will correspond to some positive (resp. negative) occurrences of $[]$ in the *type* of an abstraction. This roughly motivates Definition 5.2 by the first item of this remark and explains Lemma 5.1.

As reminded above, some parts of a typed term may not be typed, but when a head normal form is *unforgetfully* typed, all the head arguments are typed, as expected:

Lemma 5.2. If Π is an unforgetful derivation typing the head normal form $t = \lambda x_1 \dots x_p. x t_1 \dots t_q$ then there are unforgetful subderivations Π_1, \dots, Π_q of Π that respectively type t_1, \dots, t_q .

Proof. By Lemma 5.1, there is a subderivation Π_0 of Π that unforgetfully types the zero head normal form $t_0 = x t_1 \dots t_q$ – say that Π_0 concludes with $\Gamma \vdash x t_1 \dots t_q : \tau$ where $[]$ does not occur negatively in Γ or positively in τ .

Let τ_0 be the type assigned to the *unique* left subderivation typing the occurrence of x as the head variable of t_0 . By typing constraints, τ_0 is of the form $[\sigma_{1,i}]_{i \in I(1)} \rightarrow \dots \rightarrow [\sigma_{q,i}]_{i \in I(q)} \rightarrow \tau$ and the derivation Π_0 is of the form:

$$\frac{\overline{\Gamma_0 \vdash x : \tau_0} \quad (\Pi_{1,i} \triangleright \Delta_{1,i} \vdash t_1 : \sigma_{1,i})_{i \in I(1)}}{\Gamma_1 \vdash x t_1 : \tau_1} \quad \dots \quad \frac{\Gamma_{q-1} \vdash x t_1 \dots t_{q-1} : \tau_{q-1} \quad (\Pi_{q,i} \triangleright \Delta_{q,i} \vdash t_q : \sigma_{q,i})_{i \in I(q)}}{\Gamma_q \vdash x t_1 \dots t_{q-1} t_q : \tau}$$

where $\Gamma_0 = x : \tau_0$, $\Gamma_{k+1} = \Gamma_k + (+_{i \in I(k+1)} \Delta_{k+1,i})$ for $0 \leq k \leq q-1$, $\tau_q = \tau$ and $\tau_k = [\sigma_i^k] \rightarrow \tau_{k+1}$ for $0 \leq k \leq q-1$ (note that $\Gamma = \Gamma_q$).

Since the head variable is free in Γ , τ_0 appears in Γ . Then, by unforgetfulness, $[]$ cannot occur negatively in τ_q , so that:

- Neither $I(1), \dots, I(q-1)$ nor $I(q)$ is empty.
- $[]$ cannot occur positively in one of the σ_i^k .

For instance, there is a $i_1 \in I(1)$ and we set $\Pi_1 = \Pi_{i_0}^1$, $\Delta_1 = \Delta_{i_0}^1$ and $\sigma_1 = \sigma_{i_0}^1$, so that $\Pi_1 \triangleright \Delta_1 \vdash t_1 : \sigma_1$. As noted above, $[]$ does not occur positively in σ_1 . Moreover, since $\Delta_1 \leq \Gamma$, $[]$ does not occur negatively in Δ_1 , so that Π_1 is unforgetful. We reason likewise for t_2, \dots, t_q . \square

Remark 5.3. We notice that an occurrence of $[]$ in a $\sigma_{k,i}$ (for some $1 \leq k \leq q$ and $i \in I(k)$) whose nesting depth is n will correspond to an occurrence of $[]$ in the type $[\sigma_i^1]_{i \in I(1)} \rightarrow \dots \rightarrow \tau$ of the head variable x whose nesting depth is $n+1$. Moreover, since $\sigma_{k,i}$ is a type given to the head argument t_k , which is a normal form, $\sigma_{k,i}$ is co-unforgetful *i.e.* a type in which $[]$ does not occur positively (Remark 5.2). This explains why unforgetfulness is not only about the top-level occurrences of $[]$.

Remark 5.4. Actually, we notice that every subderivation of the unforgetful derivation Π that types a head argument is also unforgetful.

We recall from Sec. 2.3.4 that $b \in \text{supp}(t)$ is Böhm stable if b is reached from the root of t by visiting a series of HNF.

Lemma 5.3. Let Π an unforgetful derivation typing a term t .

- If b is Böhm stable, then b is a typed position in Π and the subderivations of Π corresponding to position b are unforgetful.
- If b is the position of a maximal head reducible subterm of t , then b is a typed position in Π and the subderivations of Π corresponding to position b are unforgetful.

Proof.

- By induction on the Böhm stability of b , using Lemmas 5.1 and 5.2 (the latter when we visit head arguments).
- Let b be the position of a maximal head reducible subterm of t . If t is head reducible, then $b = \varepsilon$ and the only minimal redex is the head redex and it is typed. Let us assume then that t is a HNF. In that case, $b = b_0 \cdot 0^p \cdot 1^k \cdot 2$ where b_0 is a Böhm stable position of t and $t|_b$ is a head argument of $t|_{b_0}$. By the first point,

there is an unforgetful subderivation Π_0 of Π that types $t|_{b_0}$. By Lemma 5.2, there is an unforgetful subderivation Π_* of Π that types $t|_b$, so that the head redex of $t|_b$ (which is a minimal redex of t) is typed.

□

Remark 5.5. We actually notice that every subderivation of the unforgetful derivation Π that corresponds to a Böhm stable position is unforgetful.

By Lemma 5.3, in the case of unforgetful typing, every minimal redex is typed and minimal reduction (Section 2.3.5) allows us to apply weighted subject reduction (Proposition 3.6), yielding:

Corollary 5.1. Let Π an unforgetful derivation concluding with $\Gamma \vdash t : \tau$. If $t \rightarrow_m t'$, then there is a derivation Π' concluding with $\Gamma \vdash t' : \tau$ such that $\text{sz}(\Pi') < \text{sz}(\Pi)$.

5.1.4 Weak Normalization

In this section, we explain how to characterize weak normalization (instead of head normalization) by means of an intersection type system and we also prove that the minimal strategy (Section 2.3.5) is complete for weak normalization. Since the leftmost outermost strategy and the Böhm reductions strategies are instances for the minimal reduction strategy, this will also prove that they are complete for weak normalization.

As in Section 3.4, we reuse the general scheme sketched in Section 3.3.1.

Proposition 5.1. Let t be a term. Then t is unforgetfully typable in system \mathcal{R}_0 iff t is weakly normalizing.

Proposition 5.2. A term t is weakly normalizing iff the minimal reduction strategy terminates on t .

Although not explicitly due to Gardner [43] or de Carvalho [22], Propositions 5.1 and 5.2 are well-known folklore, since from a type system characterizing HN, one may characterize WN just by considering unforgetful judgments. They are a simple consequence of the two below:

Proposition 5.3. If t is unforgetfully typable in system \mathcal{R}_0 , then the minimal reduction strategy applied to t terminates (*i.e.* reaches a normal form).

Proof. Assume that $\Pi \triangleright \Gamma \vdash t : \tau$ is unforgetful and we consider an instance \mathbf{rs} of the minimal reduction strategy starting from t . Let us prove that there is no such instance that is of infinite length. This will be enough to conclude.

We write t_n for the term that appears after n reduction steps in \mathbf{rs} , when it exists when it exists (*i.e.* n is lesser or equal than the length of \mathbf{rs}).

- By Corollary 5.1, there is a sequence of derivations $\Pi_0 \Pi_1, \Pi_2, \dots$ typing respectively t_0, t_1, t_2, \dots such that $\text{sz}(\Pi_0) > \text{sz}(\Pi_1) > \text{sz}(\Pi_2) \dots$
- Thus, the length of \mathbf{rs} must be lesser or equal than $\text{sz}(\Pi)$. Thus, any instance of the minimal reduction strategy is of finite length, as demanded.

□

Proposition 5.4. If t is weakly normalizing, then t is unforgetfully typable in system \mathcal{R}_0 .

Proof. Let t be a WN term. Thus, there are a normal form t' and a reduction sequence \mathbf{rs} such that $t \xrightarrow{\mathbf{rs}} t'$.

- We notice that the proof of Lemma 3.6, together with Lemma 5.1, yields an unforgetful derivation for any given normal form. Thus, there is an unforgetful derivation $\Pi' \triangleright \Gamma \vdash t' : \tau$ for some Γ, τ .
- By subject expansion (Proposition 3.3), there is a derivation Π concluding with $\Gamma \vdash t : \tau$. Since Π' is unforgetful, Π also is.

□

Remark 5.6 (Upper Bound on the Minimal Reduction Strategy). If we look at the proof of Proposition 5.3, we notice that actually, the number of steps of minimal reduction to normalize t is bounded by $\mathbf{sz}(\Pi)$. Thus, the size of an unforgetful \mathcal{R}_0 -derivation gives an upper bound for the length to complete an instance of the minimal reduction strategy, in particular, of the leftmost outermost reduction strategy and instances of the Böhm reduction strategy.

5.2 Characterizing Strong Normalization

In this section, we will characterize the set of strongly normalizing λ -terms with intersection types. Contrary to the case of weak normalization, we will actually have to modify the typing rules of \mathcal{R}_0 to achieve this purpose.

The size $|t|$ of a λ -term is the cardinal of its support (Definition 2.1). We will reuse Notation 2.2 *i.e.*, given a SN term t , $\eta(t)$ denotes the maximal length of a reduction sequence starting at t .

The reduction step $t \xrightarrow{b} t'$ is an **erasing reduction step** if $x \notin \mathbf{fv}(r)$ where $t|_b = (\lambda x.r)s$. In that case, note that $t'|_b = r$ and the argument s of the fired redex is indeed erased during reduction. For instance, $t_1 = y((\lambda x.y)z) \xrightarrow{2} yy = t'_1$ is an erasing reduction step, whereas $t_2 = y(\lambda x.xx)z \xrightarrow{2} y(zz) = t'_2$ is *non-erasing* since $t|_2 = (\lambda x.r)s$ with $r = xx$, $s = z$ and $x \in \mathbf{fv}(r)$.

5.2.1 Erasable Subterms

In Sec. 3.4.1, we noticed that, given a \mathcal{R}_0 -derivation Π typing a term t , some parts of t could be left untyped by Π *e.g.*, the unique derivation Π concluding with $x : [[] \rightarrow o] \vdash xI : o$ does not type the subterm I (where $I = \lambda x.x$), although there are also derivations typing xI such that it is typed (*e.g.*, the one concluding with $x : [[o] \rightarrow o] \vdash xI : o$).

Thus, in the term $t = xI$, the subterm I may or may not be typed, depending on the \mathcal{R}_0 -derivation typing t . However, let us notice now that there are terms t such that no argument u of t may be typed *as a subterm of tu* . For instance, in $(\lambda x.y)u$, the argument u cannot be typed, whether u is HN or not (Example 3.5 in Sec. 3.4.1). Why is that? As noted in Remark 3.2.4, $\lambda x.y$ may only be typed in \mathcal{R}_0 with a type of the form $[] \rightarrow \tau$ and thus, for all \mathcal{R}_0 -derivation Π typing $(\lambda x.y)u$, u is left untyped *i.e.* $2 \notin \hat{\Pi}$. This is because u is an erasable subterm of $(\lambda x.y)u$: indeed, $(\lambda x.y)u \rightarrow y$ and u is absent from y . By the way, we may give a general definition of erasable subterms:

Definition 5.3. Let $t \in \Lambda$ and $b \in \text{supp}(t)$. Then b is the position of an **erasable subterm** of t if there is a reduction sequence \mathbf{rs} such that $\text{Res}_{\mathbf{rs}}^t(b) = \emptyset$.

Actually, even an unforgetful derivation Π cannot type the erasable subterms of its subject (this may be proved by induction on b and \mathbf{rs} in Definition 5.3). From the type-theoretic point of view, if u is SN, then $(\lambda x.y)u$ also is, but no \mathcal{R} -derivation Π can be a certificate of strong normalization of $(\lambda x.y)u$ since a \mathcal{R}_0 -typing of $(\lambda x.y)u$ cannot “reach” u (*i.e.* cannot type u as a subterm), whether u is SN or not.

Now, if we want to characterize strong normalization in Λ , we need to consider derivations that even type erasable subterms. For that, we need to modify system \mathcal{R}_0 . Since system \mathcal{R}_0 characterizes head normalization, we refer to it as \mathcal{H} and we define a new system that we refer to as \mathcal{S} , which appears in [20] (slight variants appear in [14, 36, 60]). System \mathcal{S} has the same (multiset) types and judgments as \mathcal{H} does, but is inductively defined by the following rules:

$$\frac{}{x : [\tau] \vdash x : \tau} \text{ax} \qquad \frac{\Gamma; x : [\sigma_i]_{i \in I} \vdash t : \tau}{\Gamma \vdash \lambda x.t : [\sigma_i]_{i \in I} \rightarrow \tau} \text{abs}$$

$$\frac{\Gamma \vdash t : [\sigma_i]_{i \in I} \rightarrow \tau \quad (\Delta_i \vdash u : \sigma_i)_{i \in I} \quad I \neq \emptyset}{\Gamma +_{i \in I} \Delta_i \vdash tu : \tau} \text{app}_{[*]}$$

$$\frac{\Gamma \vdash t : [] \rightarrow \tau \quad \Delta \vdash u : \sigma}{\Gamma + \Delta \vdash tu : \tau} \text{app}[]$$

Thus, \mathcal{S} is similar to $\mathcal{H}/\mathcal{R}_0$ (same **ax**-rule, same **abs**-rule, almost the same **app**-rule) *except* when typing an application tu such that t that does not demand a typing of its argument u (*i.e.* $t : [] \rightarrow \tau$ for some τ). By induction on t , this proves that, if Π types t in system \mathcal{S} , then no subpart of t is untyped in Π , which is a good start for \mathcal{S} to characterize strong normalization!

For instance, the derivation on top of Sec. 3.4.1 is not valid in system \mathcal{S} . However, if a term u is typable in system \mathcal{S} by a derivation Π_u , then we can also type $(\lambda y.x)u$ in \mathcal{S} (compare again with Example 3.5):

$$\frac{\frac{}{x : [\tau] \vdash x : \tau} \text{ax}}{x : [\tau] \vdash \lambda y.x : [] \rightarrow \tau} \text{abs} \quad \Pi_u \triangleright_{\mathcal{S}} \Delta \vdash u : \sigma}{x : [\tau] + \Delta \vdash (\lambda y.x)u : \tau} \text{app}[]$$

Remark 5.7.

- The typing rule **app**_[] can be understood as a *controlled weakening/subtyping* because $t : [] \rightarrow \tau$ is taken as $t : [\sigma] \rightarrow \tau$. As remarked above, we need an irrelevant system to type erasable subterms and to characterize strong normalization, but system \mathcal{S} may be considered as the least irrelevant choice to obtain this.
- Another way to modify system \mathcal{R}_0 so that strong normalization is characterized by typability is the following:
 1. Forbid the empty multiset in types.
 2. Use the rules **abs**, **app** of system \mathcal{R}_0 (the argument must then be typed by modification 1).

3. Replace the **ax**-rule with the following *irrelevant* variant:

$$\frac{i_0 \in I}{x : [\sigma_i]_{i \in I} \vdash x : \sigma_{i_0}} \mathbf{ax}_w$$

We thus obtain that a type system that is far more irrelevant than \mathcal{S} is.

- See also system \mathcal{S}_w in Sec. 5.2.4.

Let us prove now that system \mathcal{S} indeed characterizes strongly normalizing λ -terms.

5.2.2 Subject Reduction and Expansion in \mathcal{S}

As discussed in Sec. 3.3, subject reduction and expansion are crucial for an intersection type system to characterize normalization. What about system \mathcal{S} ?

Notice that every redex of a typed \mathcal{S} -typed term is typed but subject reduction and expansion are not satisfied in system \mathcal{S} for *erasing* reduction steps *e.g.*, we have $(\lambda y.x)u \rightarrow x$ but $x : [\tau] + \Delta \vdash x : \tau$ is not \mathcal{S} -derivable whenever Δ is not the empty context, because of the relevance of the **ax**-rule. Conversely, if $u = z$, then $x : [\tau] \vdash x : \tau$ is \mathcal{S} -derivable, but $x : [\tau] \vdash (\lambda x.y)u : \tau$ is not.

However, weighted subject reduction and subject expansion are valid for *non-erasing* reduction steps:

Property 5.1 (Weighted Non-Erasing Subject Reduction for \mathcal{S}). If $\Pi \triangleright_{\mathcal{S}} \Gamma \vdash t : \tau$ and $t \rightarrow t'$ is a non-erasing reduction step, then there is a \mathcal{S} -derivation Π' such that $\Pi \triangleright_{\mathcal{S}} \Gamma \vdash t' : \tau$ and $\mathbf{sz}(\Pi') < \mathbf{sz}(\Pi)$.

Proof. This proposition is a particular case of Property 7.1, page 158, which is presented with a complete proof. \square

Property 5.2 (Non-Erasing Subject Expansion for \mathcal{S}). If $\triangleright_{\mathcal{S}} \Gamma \vdash t' : \tau$ and $t \rightarrow t'$ is a non-erasing reduction step, then $\triangleright_{\mathcal{S}} \Gamma \vdash t : \tau$.

Proof. This proposition is a particular case of Property 7.2, page 163, which is presented with a complete proof. \square

Figure 3.2 is valid for non-erasing reduction steps of \mathcal{S} , roughly because when reduction is not erasing, the **app**_[*]-rules typing the application of the redex are instances of the **app**-rule of system \mathcal{R}_0 .

For all contexts Γ , we define the domain $\mathbf{dom}(\Gamma)$ for Γ by $\mathbf{dom}(\Gamma) = \{x \in \mathcal{V} \mid \Gamma(x) \neq []\}$. A straightforward induction on the structure of Π shows the following result, sometimes referred to (maybe a bit abusively) as a *relevance* lemma:

Lemma 5.4 (Relevance). If $\triangleright_{\mathcal{S}} \Gamma \vdash t : \tau$, then $\mathbf{dom}(\Gamma) = \mathbf{fv}(t)$.

We also note that we may prove now that typability is stable for *erasing head*-reduction or expansion steps, provided the argument is typable. We actually do that in the particular case of the *zero* head reducible terms:

Lemma 5.5. If $\Pi \triangleright_{\mathcal{S}} \Gamma \vdash t : \tau$ and $t = (\lambda x.r)st_1 \dots t_q \rightarrow rt_1 \dots t_q = t'$ is an erasing reduction step ($x \notin \mathbf{fv}(r)$), then there is a \mathcal{S} -derivation Π' and a context Γ' such that $\Pi \triangleright_{\mathcal{S}} \Gamma' \vdash t' : \tau$ and $\mathbf{sz}(\Pi') < \mathbf{sz}(\Pi)$.

Proof. By induction on q . Only the case $q = 0$ is non-trivial. Thus, let us assume $q = 0$, so that Π is of the form:

$$\Pi = \frac{\frac{\Pi_r \triangleright \Gamma \vdash r : \tau}{\Gamma \vdash \lambda x.r : [] \rightarrow \tau} \quad \Pi_s \triangleright \Delta \vdash s : \sigma}{\Gamma + \Delta \vdash (\lambda x.r)s : \tau} \text{app}[]$$

where $x \notin \text{fv}(r)$ and, by Lemma 5.4, $x \notin \text{dom}(\Gamma)$. Then $(\lambda x.r)s \rightarrow r$ and we can set $\Pi' = \Pi_r$. \square

Lemma 5.6. If $\triangleright_{\mathcal{S}} \Gamma' \vdash t' : \tau$, u is \mathcal{S} -typable and $t = (\lambda x.r)s t_1 \dots t_q \rightarrow r t_1 \dots t_q$ is an erasing head reduction step, then there is a judgment Γ such that $\triangleright_{\mathcal{S}} \Gamma \vdash t : \tau$.

Proof. By induction on q . Only the case $q = 0$ is non-trivial. Thus, let us assume $q = 0$, so that Π' concludes with $\Gamma \vdash r : \tau$ where $x \notin \text{fv}(r)$ and, by Lemma 5.4, $x \notin \text{dom}(\Gamma)$.

By hypothesis, there is a derivation Π_u concluding with $\Delta \vdash s : \sigma$ for some Δ and σ . We then set:

$$\Pi = \frac{\frac{\Pi' \triangleright \Gamma \vdash r : \tau}{\Gamma \vdash \lambda x.r : [] \rightarrow \tau} \quad \Pi_u \triangleright \Delta \vdash s : \sigma}{\Gamma + \Delta \vdash (\lambda x.r)s : \tau} \text{app}[]$$

\square

Adapting the general proof scheme To sum up, subject reduction and expansion are not true in the general case for system \mathcal{S} . They are ensured for the *non-erasing* reduction steps only (Properties 5.1 and 5.2). We also have proved that, for erasing steps, typability is stable under head reduction and head expansion, provided, in the latter case, the created argument is typable (Lemmas 5.6 and 5.6).

The hitch is that we cannot prove *at this stage* that typability is stable under reduction and expansion (with a typable created argument) because Lemmas 5.6 and 5.6 do not ensure context preservation (see Sec. 3.3.3). Of course, this stability is a simple consequence of Proposition 5.5 to come, but again, at this stage, we cannot prove it.

Since we do not have general subject reduction and expansion for system \mathcal{S} , the general proof scheme of Sec. 3.3.1 cannot¹ be applied. But if we make the best of Properties 5.1, 5.2 and of Lemmas 5.5, 5.6, this very scheme can be adapted so that we obtain that indeed, \mathcal{S} -typability characterizes strong normalization. But this demands one thing more: we need to restate “ t is SN” as an inductive predicate, which is done in the next section.

5.2.3 Strong Normalization as an Inductive Predicate

We defined in Sec. 2.2.3 strong normalization by “a term t is SN if there is no reduction path of infinite length starting at t ”. As justified in the previous section, we must present strong normalization as an inductive predicate. We define below a new predicate, namely

¹This does not come fully as a surprise because strong normalization differs from head and weak normalizations, in that, SN is not about the existence of *at least* one reduction path leading to a final state, as noted in Sec. 2.2, but states that every reduction path is finite. This is also materialized by the fact that strong normalization is not stable under expansion in the general case: if the expansion of a SN term *creates* a non-SN argument, then the expanded term is not SN *e.g.*, y is SN, $t := (\lambda x.y)\Omega \rightarrow y$ but t is not SN.

ISN (where the letter I stands for “Inductive”) and we prove that it is actually equivalent to SN:

$$\frac{t_1, \dots, t_q \text{ are ISN}}{x t_1 \dots t_q \text{ is ISN } (q \geq 0)} \quad \frac{t \text{ is ISN}}{\lambda x.t \text{ is ISN}} \quad \frac{r[s/x] t_1 \dots t_q \text{ and } s \text{ are ISN}}{(\lambda x.r)s t_1 \dots t_q \text{ is ISN}}$$

Lemma 5.7. For all $t \in \Lambda$. Then t is SN iff t is ISN.

Proof. This Lemma is a consequence of the Claims 5.1 and 5.2 below. \square

Claim 5.1. If t is SN, then t is ISN.

Proof. Let t be a SN term. We show that t is ISN by induction on the pair $\langle \eta(t), |t| \rangle$ (using the lexicographical order).

- The base case is $\langle \eta(t), |t| \rangle = \langle 0, 1 \rangle$. Then $t = x$ is a variable and x is ISN by definition.
- If $t = \lambda x.t_0$, then the subterm t_0 is SN and $\langle \eta(t_0), |t_0| \rangle < \langle \eta(t), |t| \rangle$. The induction hypothesis gives that t_0 is ISN, and thus by definition, we get $t = \lambda x.t_0$ is ISN.
- If t is an application, then $t = x t_1 \dots t_q$ or $t = (\lambda x.r)s t_1 \dots t_q$ (with $q \geq 0$). The subterms t_k are obviously SN with $\langle \eta(t_k), |t_k| \rangle < \langle \eta(t), |t| \rangle$. It follows by the induction hypothesis that t_i is ISN for all $k \in \{1, \dots, q\}$. We now consider every possible case.
 - If $t = x t_1 \dots t_q$, then the fact that the t_k are ISN implies that $x t_1 \dots t_q$ is ISN definition.
 - If $t = (\lambda x.r)s t_1 \dots t_q$, the subterm r is SN and $\langle \eta(r), |r| \rangle < \langle \eta(t), |t| \rangle$. By the induction hypothesis it follows that r is ISN. Moreover, let

$$t = (\lambda x.r)s t_1 \dots t_q \rightarrow_{\beta} r[s/x] t_1 \dots t_q = t'$$

Since t' is SN and $\eta(t') < \eta(t)$, then $\langle \eta(t'), |t'| \rangle < \langle \eta(t), |t| \rangle$. By the induction hypothesis it follows that t' is ISN. Since s is also ISN, then “ t is ISN” holds by definition. \square

Claim 5.2. If t is ISN, then t is SN.

Proof. Let t be a ISN term. We show that t is SN by induction on the definition of ISN.

- If $t = x t_1 \dots t_q$ is ISN with t_1, \dots, t_q ISN, then by induction hypothesis we have t_1, \dots, t_q are SN, and it follows that t is SN.
- If $t = \lambda x.t_0$ with t_0 is ISN, then, by induction hypothesis, we have t_0 is SN, and it follows that t is also SN.
- If $t = (\lambda x.r)s t_1 \dots t_q$ with $s, r[s/x] t_1 \dots t_q$ ISN, then by induction hypothesis, s and $r[s/x] t_1 \dots t_q$ are SN. Moreover, the fact $r[s/x] t_1 \dots t_q$ is SN implies that $r[s/x]$ and the t_k are SN, and by observing that, if $r \rightarrow r'$ then $r[s/x] \rightarrow r'[s/x]$, we obtain that r is SN because $r[s/x]$ is. We show that t is SN by a second induction on $\eta(r) + \eta(s) + \sum_{i=1 \dots q} \eta(t_i)$.

Let us see how are all the reducts of t .

- If $t \rightarrow (\lambda x.r')s t_1 \dots t_q = t'$, where $r \rightarrow r'$ or $t \rightarrow (\lambda x.r)s' t_1 \dots t_q = t'$, where $s \rightarrow s'$, or $t \rightarrow (\lambda x.r)s t_1 \dots t'_k \dots t_q = t'$, where $t_i \rightarrow t'_i$, then t' is SN by the second induction hypothesis.
- If $t \rightarrow r[s/x]t_1 \dots t_q = t'$, then t' is SN as already remarked by the first induction hypothesis.

Since all the one-step reducts of t are SN, then t is SN.

□

5.2.4 Characterizing Strong Normalization

As explained, we first show that any typable term t is (I)SN.

Lemma 5.8. If t is \mathcal{S} -typable, then t is (I)SN.

Proof. Let $\Pi \triangleright \Gamma \vdash t : \tau$. We proceed by induction on $\mathbf{sz}(\Pi)$. When Π ends with the rule **ax** or **abs**, the proof is straightforward, so that we consider a derivation ending with **app**_[*] or **app**_[], where $t = x t_1 \dots t_q$ or $t = (\lambda x.r)s t_1 \dots t_q$.

By construction, it is not difficult to see that there are subderivation $(\Pi_{t_k})_{k \in \{1 \dots q\}}$ typing respectively t_k such that $(\mathbf{sz}(\Pi_{t_k}) < \mathbf{sz}(\Pi))_{k \in \{1 \dots q\}}$ so that the induction hypothesis implies that the t_k (with $1 \leq k \leq q$) are ISN. In the second, s is ISN for the same reason. We analyse the different subcases:

- If $t = x t_1 \dots t_q$, then the fact that the t_k ($1 \leq k \leq q$) are SN directly implies that $x t_1 \dots t_q$ is SN.
- If $t = (\lambda x.r)s t_1 \dots t_q$, there are two subcases:
 - $x \in \mathbf{fv}(r)$. Using Property 5.1, we get $\Pi' \triangleright \Gamma \vdash r[s/x]t_1 \dots t_q : \tau$ such that $\mathbf{sz}(\Pi') < \mathbf{sz}(\Pi)$. Then the induction hypothesis gives $r[s/x]t_1 \dots t_q$ is SN. This, together with the fact that s is SN, implies that t is SN.
 - $x \notin \mathbf{fv}(u)$. By Lemma 5.5, we get Π' typing $r[s/x]t_1 \dots t_q$ with $\mathbf{sz}(\Pi') < \mathbf{sz}(\Pi)$. Then the induction hypothesis gives that $r[s/x]t_1 \dots t_q$ is SN. This, together with the fact that s is SN, implies that t is SN.

□

And any SN term turns out to be typable:

Lemma 5.9. If t is (I)SN, then t is \mathcal{S} -typable.

Proof. We reason by induction on “ t is ISN”. The two first cases are straightforward.

Let $t = (\lambda x.r)s t_1 \dots t_q$ be ISN and coming from the assumption that $r[s/x]t_1 \dots t_q$ and s are ISN. By the induction hypothesis, $r[s/x]t_1 \dots t_q$ and s are both typable. We consider two subcases. If $x \in \mathbf{fv}(r)$, then $(\lambda x.r)s t_1 \dots t_q$ is typable by Property 5.2. Otherwise, $x \notin \mathbf{fv}(r)$ and we use Lemma 5.6. □

Lemma 5.8, 5.9 allow us to conclude with the equivalence between typability and strong-normalization for the λ -calculus (first part of the following theorem):

$$\begin{array}{c}
\frac{i_0 \in I}{\Gamma; x : [\tau_i]_{i \in I} \vdash x : \tau_{i_0}} \text{ax} \qquad \frac{\Gamma; x : [\sigma_i]_{i \in I} \vdash t : \tau}{\Gamma \vdash \lambda x.t : [\sigma_i]_{i \in I} \rightarrow \tau} \text{abs} \\
\\
\frac{\Gamma \vdash t : [\sigma_i]_{i \in I} \rightarrow \tau \quad (\Delta_i \vdash u : \sigma_i)_{i \in I} \quad I \neq \emptyset}{\Gamma +_{i \in I} \Delta_i \vdash t u : \tau} \text{app}_{[*]} \\
\\
\frac{\Gamma \vdash t : [] \rightarrow \tau \quad \Delta \vdash u : \sigma}{\Gamma + \Delta \vdash t u : \tau} \text{app}[]
\end{array}$$

Figure 5.1: System \mathcal{S}_w

Proposition 5.5. [19, 20] Let t be a λ -term. Then t is \mathcal{S} -typable iff t is strongly normalizing.

Moreover, if t is \mathcal{S} -typable with a derivation Π , then $\mathbf{sz}(\Pi)$ gives an upper bound to the maximal length of a reduction sequence starting at t .

Proof. The second part of the statement is proved in the next section, by first modifying system \mathcal{S} . \square

5.2.5 Obtaining an Upper Bound for Normalizing Sequence

In this section, we sketch the proof of the last part of Proposition 5.5 *i.e.* if Π is a \mathcal{S} -derivation typing t , then not only is t strongly normalizing, but $\mathbf{sz}(\Pi)$ also gives an upper bound for the length of a reduction sequence starting at t .

To prove that, the idea is the following: we embed system \mathcal{S} into a new system \mathcal{S}_w , which enjoys weighted subject reduction contrary to \mathcal{S} and preserves the size of system \mathcal{S} . System \mathcal{S}_w is given by Fig. 5.1.

The irrelevance of system \mathcal{S}_w is more precisely captured by the following lemma:

Lemma 5.10 (Weakening). If $\Pi \triangleright_{\mathcal{S}_w} \Gamma \vdash t : \tau$, then, for all $\Gamma' \geq \Gamma$, there is a \mathcal{S}_w -derivation $\Pi' \triangleright \Gamma' \vdash t : \tau$ such that $\mathbf{sz}(\Pi') = \mathbf{sz}(\Pi)$.

Proof. The proof is straightforward by induction on Π . It is true for “axiom derivations” thanks to \mathbf{ax}_w . \square

Lemma 5.10 is used in the proof of Substitution Lemma (Lemma 5.12), and again in the proof of subject reduction (Property 5.3) to handle the possible occurrence of $\mathbf{app}[]$ at the root of a derivation typing a redex.

Lemma 5.4 is of course false for \mathcal{S}_w , but we have an inclusion:

Lemma 5.11. If $\triangleright_{\mathcal{S}_w} \Gamma \vdash t : \tau$, then $\text{dom}(\Gamma) \supset \text{fv}(t)$.

Proof. Straightforward by induction on the structure of the \mathcal{S}_w -derivations. \square

Remark 5.8 (Failure of Subject Expansion). Subject expansion does not hold in system \mathcal{S}_w *e.g.*, we have $t := \lambda z.(\lambda x.y)z \rightarrow \lambda z.y =: t'$, and we may derive $\vdash \lambda x.y : [] \rightarrow \tau$ in \mathcal{S}_w (assign τ to y), but it is not possible to derive $\vdash t : [] \rightarrow \tau$: indeed, by $\mathbf{app}_{[*]}$, the occurrence of z in t must be typed and t has types of the form $[\sigma_i]_{i \in I} \rightarrow \tau$ where $I \neq \emptyset$.

We have a substitution Lemma with a quantitative flavour:

Lemma 5.12. If $\Pi \triangleright_{\mathcal{S}_w} \Gamma; x : [\sigma_i]_{i \in I} \vdash t : \tau$ and, for all $i \in I$, $\Phi_i \triangleright_{\mathcal{S}_w} \Gamma_i^u \vdash u : \sigma_i$, then, there exists a derivation $\Pi_{t[u/x]}$ concluding with $\Gamma + (+_{i \in I} \Gamma_i^u) \vdash t[u/x] : \tau$ such that $\mathbf{sz}(\Pi_{t[u/x]}) \leq \mathbf{sz}(\Pi) + (+_{i \in I} \mathbf{sz}(\Phi_i)) - |I|$.

Proof. By induction on the structure of Π . The only interesting case is \mathbf{ax}_w . The \mathbf{abs} and \mathbf{app} -rules are handled as in the relevant case. See for instance the proof of Lemma 7.4 for a system $\mathcal{S}_{\lambda_\mu}$ that extends system \mathcal{S} for λ_μ -calculus (system $\mathcal{S}_{\lambda_\mu}$ resorts to the formalism of *auxiliary* derivations introduced in Sec. 7.1). We have now two subcases:

- $t = x$: let us assume that Π is of the form:

$$\frac{i_0 \in I}{\Gamma; x : [\sigma_i]_{i \in I} \vdash t : \tau} \mathbf{ax}_w$$

Since Φ_{i_0} concludes with $\Gamma_{i_0}^u \vdash u : \sigma_{i_0}$, by Lemma 5.10, there is a derivation Φ'_{i_0} concluding with $\Gamma + (+_{i \in I} \Gamma_i^u \vdash u : \sigma_0$ such that $\mathbf{sz}(\Phi'_{i_0}) = \mathbf{sz}(\Phi_{i_0})$. We have $\mathbf{sz}(\Pi) = 1$ and $|I| - 1 \leq +_{i \in I \setminus \{i_0\}} \mathbf{sz}(\Phi_i)$ (for all i , $1 \leq \mathbf{sz}(\Phi_i)$), $\mathbf{sz}(\Phi'_{i_0}) = \mathbf{sz}(\Phi_{i_0}) = \mathbf{sz}(\Pi) - 1 + \mathbf{sz}(\Phi_{i_0}) \leq \mathbf{sz}(\Pi) - 1 + \mathbf{sz}(\Phi_{i_0}) + (+_{i \in I \setminus \{i_0\}} \mathbf{sz}(\Phi_i) - (|I| - 1)) = \mathbf{sz}(\Pi) + (+_{i \in I} \mathbf{sz}(\Phi_i)) - |I|$.

- $t = y \neq x$: let us assume that Π is of the form:

$$\frac{\tau \text{ occurs in } \Gamma(y)}{\Gamma; x : [\sigma_i]_{i \in I} \vdash y : \tau} \mathbf{ax}_w$$

Let us define Π' by

$$\frac{\tau \text{ occurs in } \Gamma(y)}{\Gamma + (+_{i \in I} \Gamma_i^u) \vdash y : \tau} \mathbf{ax}_w$$

so that $\mathbf{sz}(\Pi') = 1$. We conclude because $+_{i \in I} \mathbf{sz}(\Phi_i) \geq |I|$ (see previous case), so that $\mathbf{sz}(\Pi') = 1 = \mathbf{sz}(\Pi) \leq \mathbf{sz}(\Pi) + (+_{i \in I} \mathbf{sz}(\Phi_i)) - |I|$. \square

We can prove now weighted subject reduction for system \mathcal{S}_w :

Property 5.3. Let $\Pi \triangleright_{\mathcal{S}_w} \Gamma \vdash t : \tau$. If $t \rightarrow t'$, then there is a \mathcal{S}_w -derivation Π' concluding with $\Gamma \vdash t' : \tau$ such that $\mathbf{sz}(\Pi') < \mathbf{sz}(\Pi)$.

Proof. By induction on the relation \rightarrow . We only show the main cases of reduction at the root $t = (\lambda x.r)s \rightarrow r[s/x]$, the other ones being straightforward (note that every subterm of a \mathcal{S}_w -typed term is also typed). We have now two subcases, depending on the rule typing the application of the redex:

- $\mathbf{app}_{[*]}$: the derivation Φ has the following form:

$$\Phi = \frac{\frac{\Pi_r \triangleright \Gamma_r; x : [\sigma_i]_{i \in I} \vdash r : \tau}{\Gamma_r \vdash \lambda x.r : [\sigma_i]_{i \in I} \rightarrow \tau} \quad (\Phi_i \triangleright \Gamma_i \vdash s : \sigma_i)_{i \in I}}{\Gamma \vdash (\lambda x.r)s : \tau} \mathbf{app}_{[*]}$$

where $\Gamma = \Gamma + (+_{i \in I} \Gamma_i)$ and $I \neq \emptyset$. By Lemma 5.12 applied to $t = r$ and $u = s$, there is a derivation Π' concluding with $\Gamma \vdash r[s/x] : \tau$ and $\mathbf{sz}(\Pi') \leq \mathbf{sz}(\Pi_r) + (+_{i \in I} \mathbf{sz}(\Phi_i)) - |I|$.

Since $\mathbf{sz}(\Pi) = \mathbf{sz}(\Pi_r) + (+_{i \in I} \mathbf{sz}(\Phi_i)) + 2$, we have $\mathbf{sz}(\Pi') < \mathbf{sz}(\Pi)$ as expected.

- $\mathbf{app}[\]$: the derivation Φ has the following form:

$$\Phi = \frac{\frac{\Pi_r \triangleright \Gamma_r \vdash r : \tau}{\Gamma_r \vdash \lambda x.r : [\] \rightarrow \tau} \quad \Phi \triangleright \Gamma_s \vdash s : \sigma}{\Gamma \vdash (\lambda x.r)s : \tau} \mathbf{app}[\]$$

where $\Gamma = \Gamma_r + \Gamma_s$ and $x \notin \mathbf{dom}(\Gamma_r)$. By Lemma 5.11, $x \notin r$, so that $t' = r$. By Lemma 5.10, there is a \mathcal{S}_w -derivation $\Pi' \triangleright \Gamma_r + \Gamma_s \vdash r : \tau$ such that $\mathbf{sz}(\Pi') = \mathbf{sz}(\Pi_r)$. Since $\mathbf{sz}(\Pi) = \mathbf{sz}(\Pi_r) + \mathbf{sz}(\Pi_s) + 2$, we have $\mathbf{sz}(\Pi') < \mathbf{sz}(\Pi)$ as expected.

□

We may then prove by induction on n :

Lemma 5.13. Let $t \in \Lambda$, \mathbf{rs} a reduction sequence of length n starting at t and Π a \mathcal{S}_w -derivation typing t . Then $\mathbf{sz}(\Pi) \geq n + 1$.

Proof. The case $n = 1$ is obvious and the inductive step is a direct consequence of Property 5.3. □

The lemma entails the last part of Proposition 5.5. This concludes the presentation of the techniques that will be exported to λ_μ -calculus in Chapter 7.

Part II

Resources for Classical Natural Deduction

Presentation

Lambda-Mu calculus (λ_μ) is a computational interpretation of *classical* natural deduction, as λ -calculus is for *intuitionistic* natural deduction.

It was for a long time believed that the Curry-Howard correspondence could not be extended to classical logic until Griffin [53] observed that Feilleisen’s \mathcal{C} operator can be typed with the double-negation elimination and the control operator `call-cc` of Scheme with Peirce’s Law (see Sec. 6.1.1). There were various reasons for this belief:

- If two proofs of *e.g.*, the classical sequent calculus Π and Π' conclude with same judgment $\Gamma \vdash \Delta$, then they are equal up to cut-elimination steps (which correspond to β -reduction steps *via* the Curry-Howard isomorphism).
- Joyal’s Lemma [70]: a cartesian closed category with a dualizing object is a pre-order, which generalizes the previous fact.
- The fact that classical logic is not constructive *e.g.*, from a proof an existential proposition $\exists x. \mathcal{P}(x)$, one may not necessarily extract an existential witness as in the intuitionistic case (but only finite list of possible candidates, by Herbrand’s Theorem).

After Griffin, λ_μ -calculus was introduced by Parigot [89] as a simple term notation for classical natural deduction proofs, and the (simply typed) λ_μ -calculus is an extension of the (simply typed) λ -calculus that encodes usual *control operators* as `call-cc`. Other calculi were proposed since then, as for example Curien-Herbelin’s $\lambda\mu\tilde{\mu}$ -calculus [32] based on classical sequent calculus.

Many notions relative to λ -calculus can be extended to λ_μ , like *normalization* (*e.g.*, head, weak or strong). Parigot [91] proved that simply typable λ_μ -terms are strongly normalizing (this result is stated in Theorem 6.1). One may want to know whether types can provide the same kind of characterizations of head/weak/strong normalization in λ_μ -calculus as they do in λ -calculus (see Propositions 3.7, 5.1 and 5.5), and not just a *guarantee* of normalization (*i.e.* only the implication “typable \Rightarrow normalizing”). Laurent [73] pioneered a type system featuring **intersection** and **union** types, characterizing head and weak normalization in λ_μ -calculus. Van Bakel [109] later proposed intersection type systems characterizing strong normalization [110, 111].

Aside from the work of Laurent and van Bakel et al. cited above, intersection and union types were also studied in the framework of classical logic [38, 65], but there is no work addressing the problem from a quantitative perspective. Type-theoretical characterization of strong-normalization for classical calculi were provided both for λ_μ [110] and $\lambda\mu\tilde{\mu}$ -calculus [38, 108], but the (idempotent) typing systems do not allow constructing decreasing measures for reduction, thus a resource aware semantics cannot be extracted from those interpretations. Models for classical calculi were proposed in [4, 98, 113], thus limiting the characterization of operational properties to head-normalization. Different small step semantics for classical calculi were developed in the framework of neededness [6, 92], without resorting to any resource aware semantical argument.

Towards Non-Idempotent Types Both Laurent and van Bakel’s work feature idempotent type operators. Thus, there are still some crucial aspects of computation, like the use of resources (*e.g.*, time and space), that still need to be logically understood, as they were with non-idempotent intersection types for λ -calculus (see beginning of

Chapter 3). Extending the understanding of resource consumption outside the pure λ -calculus is a big challenge facing the programming language community. It would lead to a new understanding of programming languages and proof assistants, with a clean type-theoretic account of resource capabilities.

We want in particular to associate quantitative information to languages being able to express control operators, that can enrich the declarative programming languages with imperative features.

- One of the contribution of this thesis, presented in Chapter 7, is the definition of two resource aware type systems for the λ_μ -calculus based on non-idempotent **intersection** and **union** types, coming along with the very simple combinatorial arguments (provided by the non-idempotent approach, see Sec. 3.4.3 and Remark 3.15 for instance), only based on a decreasing *measure*, to characterize head or strongly normalizing terms by means of typability.
- The second contribution of this thesis (Chapter 8) in the field of classical calculi is the definition of a new resource aware operational semantics for λ_μ , called $\lambda_{\mu\tau}$, which is compatible with the non-idempotent typing system defined for λ_μ . Indeed, we first define a set of reduction rules –inspired from the *substitution at a distance paradigm* [3, 60], presented in Sec. 2.4 and 4.2 – that gives a small-step implementation of the λ -calculus. We then extend the typing system for λ_μ , so that the extended reduction system $\lambda_{\mu\tau}$ preserves (and decreases the size of) typing derivations. Using this arithmetical argument, we thus derive a very simple characterization of strongly normalizing terms by means of typability, thus particularly simplifying existing proofs of strong normalization for small-step operational semantics of classical calculi [93].

Before presenting these contributions, we dedicate a Chapter 6 to the basics of pure and simply λ_μ -calculus and in particular, to the relation between the operational semantics of λ_μ and cut-elimination in minimal classical natural deduction.

Chapter 6

The Lambda-Mu Calculus

In this chapter, we present the pure (Sec. 6.2.1) and simply typed (Sec. 6.2.2) λ_μ -calculus. As it turns out, strong normalizing terms can also be defined in λ_μ and simple typability is a guarantee of strong normalization as in the case of the simply typed λ -calculus. This leads us to look for type-theoretic characterizations in Chapters 7 and 8.

- Section 6.1 briefly presents a few ways to obtain classical logic from intuitionistic/minimal natural deduction *e.g.*, by adding the *elimination of the double negation*, the *excluded middle* and *Peirce's Law* in Sec. 6.1.1 or by just allowing several formulas in the right-hand sides of sequents in Sec. 6.1.2. In this latter section, we present a focussed classical natural deduction which λ_μ -calculus is designed to be the term calculus counterpart of *via* the Curry-Howard correspondence (Sec. 3.1.1).
- The λ_μ -calculus is then introduced in Sec. 6.2, first, in its pure untyped version (Sec. 6.2.1) then in its simply typed version (Sec. 6.2.2). The operational semantics of the pure λ_μ is presented in Sec. 6.2.3, just before being interpreted as a cut-elimination step (Sec. 6.2.4) in the simply typed case, still *via* the Curry-Howard correspondence. The extension of the notions of head, weak or strong normalization may be found in Sec. 6.2.5. Since every simply-typed term is strongly normalizing (Theorem 6.1, due to Parigot [90,91]), this raises the question of characterizing normalization that is handled in the two next chapters.

6.1 Classical Logic in Natural Deduction

In this section, we present different ways to obtain classical logic from intuitionistic natural deduction. The last one is used to construct the λ_μ -calculus (Sec. 6.1.2).

6.1.1 Getting Classical Logic

Intuitionistic natural deduction NJ can be enriched with the constant \perp , representing falsehood, coming along with the elimination rule *ex falso quodlibet* meaning that *anything* can be deduced from false:

$$\frac{\Gamma \vdash \perp}{\Gamma \vdash A} \text{ex falso}$$

Thus, if a collection of sequents $(\Gamma_i)_{i \in I}$ entail \perp , it also entails any formula A .

Intuitionistic logic without \perp is known as minimal intuitionistic logic. In the presence of \perp , the *negation* of A , denoted $\neg A$, may be defined by $\neg A = A \rightarrow \perp$. Indeed, by *modus*

$$\frac{}{\vdash \neg\neg A \rightarrow A} \neg\neg_e \quad \frac{}{\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A} \text{Peirce} \quad \frac{}{\Gamma \vdash A \vee \neg\neg A} \text{exmid}$$

Figure 6.1: Obtaining Classical Logic

ponens, we may deduce \perp from A and $\neg A$ ($A \rightarrow \perp$). In NJ, we may easily derive the sequent $\vdash A \rightarrow \neg\neg A$ for any formula A (and actually, $\vdash A \rightarrow ((A \rightarrow B) \rightarrow B)$ for all A, B) by starting with the two axiom rules giving $A \rightarrow B \vdash A \rightarrow B$ and $A \vdash A$.

However, the converse implication $\neg\neg A \rightarrow A$ (*i.e.* $((A \rightarrow \perp) \rightarrow \perp) \rightarrow A$), which gives the complete equivalence between any formula A and its double-negation $\neg\neg A$, is valid for classical logic but not for intuitionistic logic. This implication is known as the *elimination of double negation*.

Actually, there are several very well-known ways to obtain classical natural deduction NK from NJ. For that, we may just add to NJ either:

- The elimination of double negation $\neg\neg_e$ (discussed above),
- or Peirce's Law **Peirce**,
- or the law of *excluded middle* **exmid**, allowing to assert $A \vee \neg A$ for any formula A .

These 3 additional rules may be found in Fig. 6.1. Peirce's Law is different from $\neg\neg_e$ and **exmid** in that, it does not (explicitly or implicitly) use the symbol \perp . It can thus be used to recover a *minimal* classical logic *i.e.* classical logic without the symbol \perp and the rule **exfalse**.

Note that the law of excluded middle supposes that we also use the connective \vee (disjunction). That is why it will not be addressed much here, but it provides however a very direct obstruction to the *last rule* property for classical logic, whereas it is an emblematic feature of intuitionistic logic. For instance, in NJ, it is known (see *e.g.*, [49], 5.2.1.) that any proof Π of a disjunction $A \vee B$ can be rewritten (with cut-elimination steps) into a proof Π' of $A \vee B$ concluding with an introduction of disjunction, which shows that, if $A \vee B$ is provable, then either A or B is provable. With the excluded middle, $\vdash C \vee \neg C$ is immediately derivable, whereas $\vdash C$ and $\vdash \neg C$ could be unprovable (*e.g.*, when C is a propositional variable) or at least, very difficult to prove.

The last rule property, also valid (in NJ) for existential propositions, illustrates why intuitionistic logic is essentially constructive whereas classical logic is not. This is one of the main reasons why, until the observations of Griffin [53] (p. II), it was thought that the Curry-Howard correspondence (Sec. 3.1.1) could not be extended to classical logic, since a program is supposedly an effective/constructive computation.

6.1.2 Focused Classical Natural Deduction

An alternative way to recover classical natural deduction consists in just allowing several formulas on the right-hand sides of sequents with an axiom rule of the form:

$$\frac{}{\Gamma, A \vdash A, \Delta} \text{ax}$$

for any formula A and sequences of formulas Γ and Δ (with an implicit exchange rule). Indeed, recalling that $\neg A = A \rightarrow \perp$, excluded middle becomes easily derivable with this

formalism:

$$\frac{\overline{A \vdash A, \perp}^{\text{ax}}}{\vdash A, \neg A} \rightarrow_i$$

More subtly, we may also derive Peirce's Law, which is done in Fig. 6.2.

$$\frac{\frac{\frac{\frac{\overline{A \vdash A, B}}{\vdash A \rightarrow B, A}}{(A \rightarrow B) \rightarrow A \vdash A, A}}{(A \rightarrow B) \rightarrow A \vdash A, A}}{(A \rightarrow B) \rightarrow A \vdash A}}{\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A}$$

Figure 6.2: Inferring Peirce's Law

Simply typed λ_μ -calculus is based upon a “focused” variant of natural deduction with several formulas on the right-hand side of a sequent: a sequent is then a triple of the form $\Gamma \vdash A \mid \Delta$, in which A is called the *active* formula. In that case, implication can be introduced or eliminated (*modus ponens*) only in the active part of the sequents. But we may freely choose which formula on the right-hand side of \vdash is active, thanks to the rule:

$$\frac{\Gamma \vdash B \mid A, \Delta}{\Gamma \vdash A \mid B, \Delta}^{\text{act}}$$

Contraction is implicit in Δ , but we also resort to an explicit rule to contract the active formula if it also occurs inactively:

$$\frac{\Gamma \vdash A \mid A, \Delta}{\Gamma \vdash A \mid \Delta}^{\text{contrac}}$$

With this focused style, the proof of Peirce's Law becomes:

$$\frac{\frac{\frac{\frac{\overline{A \vdash A \mid B}}{\overline{A \vdash B \mid A}}^{\text{act}}}{\vdash A \rightarrow B \mid A}}{(A \rightarrow B) \rightarrow A \vdash A \mid A}}{(A \rightarrow B) \rightarrow A \vdash A \mid A}}{\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A \mid}^{\text{contrac}}$$

Figure 6.3: Inferring Peirce's Law (Focused Version)

6.2 The Lamda-Mu Calculus

This section gives the syntax (Sec. 6.2.1) and the operational semantics (Sec. 6.2.3) of the λ_μ -calculus [89].

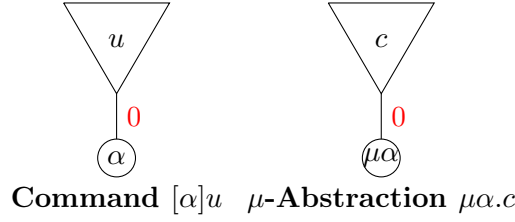


Figure 6.4: Lambda-Mu Objects as Labelled Trees

6.2.1 Lambda-Mu Terms

We consider a countable infinite set of **variables** x, y, z, \dots (resp. **continuation names** $\alpha, \beta, \gamma, \dots$). The set of **objects** ($\mathcal{O}_{\lambda\mu}$), **terms** ($\mathcal{T}_{\lambda\mu}$) and **commands** ($\mathcal{C}_{\lambda\mu}$) of the $\lambda\mu$ -calculus are given by the following grammars (see Fig. 6.4):

$$\begin{array}{lll}
 \text{(objects)} & o & ::= t \mid c \\
 \text{(terms)} & t, u, v & ::= x \mid \lambda x.t \mid tu \mid \mu\alpha.c \\
 \text{(commands)} & c & ::= [\alpha]t
 \end{array}$$

We sometimes write \mathcal{T}_λ (instead of Λ) for the set of λ -terms. Notice that the grammar extends λ -terms with two new constructors: commands $[\alpha]t$ and μ -abstractions $\mu\alpha.c$. The **size of an object** o is also denoted by $|o|$ (straightforward extension of Def. 2.1). **Free and bound variables** of objects are defined as expected, in particular $\text{fv}(\mu\alpha.c) := \text{fv}(c)$ and $\text{fv}([\alpha]t) := \text{fv}(t)$. **Free names** of objects are defined as follows:

$$\begin{array}{ll}
 \text{fn}(x) := \emptyset & \text{fn}(tu) := \text{fn}(t) \cup \text{fn}(u) \\
 \text{fn}(\lambda x.t) := \text{fn}(t) & \text{fn}([\alpha]t) := \text{fn}(t) \cup \{\alpha\} \\
 \text{fn}(\mu\alpha.c) := \text{fn}(c) \setminus \{\alpha\}
 \end{array}$$

Bound names are defined accordingly. We work with the standard notion of α -**conversion** *i.e.* renaming of bound variables and names, thus for example:

$$[\delta](\mu\alpha.[\alpha](\lambda x.x))z \equiv [\delta](\mu\beta.[\beta](\lambda y.y))z$$

Substitutions are defined as in Sec. 2.1.2 and **replacements** are (finite) functions from names to terms specified by $\{\alpha_1 // u_1, \dots, \alpha_n // u_n\}$ ($n \geq 0$). Intuitively, the operation $\{u // \alpha\}$ passes the term u as an argument to any command of the form $[\alpha]t$. Formally, the application of the **replacement** Σ to the term o , written $o\Sigma$, may require α -conversion in order to avoid the capture of free variables/names, and is defined as:

$$\begin{array}{ll}
 x\{u // \alpha\} := x & (\lambda z.t)\{u // \alpha\} := \lambda z.t\{u // \alpha\} \\
 ([\alpha]t)\{u // \alpha\} := [\alpha](t\{u // \alpha\}) & (tv)\{u // \alpha\} := t\{u // \alpha\}v\{u // \alpha\} \\
 ([\gamma]t)\{u // \alpha\} := [\gamma]t\{u // \alpha\} & (\mu\gamma.c)\{u // \alpha\} := \mu\gamma.c\{u // \alpha\}
 \end{array}$$

For example, if $I = \lambda z.z$, then $(x(\mu\alpha[\alpha]y)(\lambda z.zx))[I/x] = I(\mu\alpha[\alpha]y)(\lambda z.zI)$, and $[\alpha]x(\mu\beta.[\alpha]y)\{I // \alpha\} = [\alpha](x\mu\beta.[\alpha]yI)I$.

Substitution and replacement enjoy the following well-known interaction properties.

- Lemma 6.1.**
1. If $x \notin \text{fv}(v)$ and $x \neq y$ then $o[u/x][v/y] = o[v/y][u[v/y]/x]$.
 2. If $\alpha \notin \text{fn}(v)$ and $\alpha \neq \beta$ then $o\{u // \alpha\}\{v // \beta\} = o\{v // \beta\}\{u\{v // \beta\} // \alpha\}$.
 3. If $x \notin \text{fv}(v)$, then $o[u/x]\{v // \alpha\} = o\{v // \alpha\}[u\{v // \alpha\}/x]$.
 4. If $\alpha \notin \text{fn}(v)$, then $o\{u // \alpha\}[v/x] = o[v/x]\{u[v/x] // \alpha\}$.

$$\begin{array}{c}
\frac{}{\Gamma; x : A \vdash x : A \mid \Delta} \text{ax} \quad \frac{\Gamma; x : A \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x.t : A \rightarrow B \mid \Delta} \text{abs} \\
\\
\frac{\Gamma_t \vdash t : A \rightarrow B \mid \Delta_t \quad \Gamma_u \vdash u : A \mid \Delta_u}{\Gamma_t :: \Gamma_t \vdash t u : B \mid \Delta_t :: \Delta_u} \text{app} \\
\\
\frac{\Gamma \vdash t : B \mid \alpha : A, \Delta}{\Gamma \vdash \mu\alpha.[\beta]t : A \mid (\beta : B) :: \Delta}^{\mu_1} \\
\\
\frac{\Gamma \vdash t : A \mid \alpha : A, \Delta}{\Gamma \vdash \mu\alpha.[\beta]t : A \mid \Delta}^{\mu}
\end{array}$$

Figure 6.5: Simply Typing the λ_μ -Calculus (System $\text{Curry}_0^{\lambda_\mu}$)

6.2.2 Simply Typed Lambda-Mu Calculus

We consider the set of simple types defined in Sec. 3.1.3 and we present the Parigot's extension of system Curry_0 to λ_μ -calculus.

A (variable) context Γ , also called a **variable assignment** is a partial function from the set of term variables \mathcal{V} to the set of simple types and a (name) context Δ , also called a **name assignment** is a partial function from the set of names to the set of simple types. The domain of a name assignment Δ is also written $\text{dom}(\Delta)$. The notion of compatible name assignments is naturally extended and if Δ_1, Δ_2 are compatible, their join is also written $\Delta_1 :: \Delta_2$. If $\text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) = \emptyset$, we may also write $\Delta_1; \Delta_2$ instead of $\Delta_1 :: \Delta_2$.

The set of derivations of simply typed λ_μ -calculus is defined *inductively* by the rules of Fig. 6.5.

Those rules are licit only if the joins of variables and name contexts are defined. Forgetting about variables, terms and name, we recognize the proof calculus sketched in Sec. 6.1.2: the rule μ_1 corresponds to rule **act** and the rule μ_2 to **contrac**.

6.2.3 Operational Semantics

The λ_μ -calculus is given by the set of objects introduced in Sec. 6.2.1 and the **reduction relation** $\rightarrow_{\lambda_\mu}$, often written just \rightarrow , which is the closure by all contexts of the following rewriting rules

$$\begin{array}{l}
(\lambda x.t)u \rightarrow_\beta t[u/x] \\
(\mu\alpha.c)u \rightarrow_\mu \mu\alpha.c\{u//\alpha\}
\end{array}$$

Thus, β -reduction is defined as before and the calculus is extended with a μ -rule, which is illustrated by Fig. 6.6 (compare with Fig. 2.6). Its operational semantics of λ_μ -calculus is perhaps better understood when we see how subject reduction is processed for typed terms (Sec. 6.2.4). Note that μ -reduction does not destroy the μ -abstraction of the fired redex, whereas β -reduction destroys the λ -abstraction of the fired β -redex.

An alternative specification of the μ -rule, suggested by Andou [5], is given by $(\mu\alpha.c)u \rightarrow_\mu \mu\gamma.c\{\gamma.u//\alpha\}$, where $\{\gamma.u//\alpha\}$ denotes the **fresh replacement** meta-operation assigning $[\gamma](t\{\gamma.u//\alpha\})u$ to $[\alpha]t$ (thus changing the name of the command), in contrast to $\{u//\alpha\}$ introduced in Sec. 6.2.1 which replaces $[\alpha]t$ by $[\alpha](t\{u//\alpha\})u$. We remark however that the resulting terms $\mu\alpha.c\{u//\alpha\}$ and $\mu\gamma.c\{\gamma.u//\alpha\}$ are α -equivalent; thus *e.g.*,

$\mu\alpha.([\alpha]x)\{u//\alpha\} = \mu\alpha.[\alpha]xu \equiv \mu\gamma.[\gamma]xu = \mu\gamma.([\alpha]x)\{\gamma.u//\alpha\}$. We keep in mind this alternative formulation of the μ -rule which will justify the operational semantics of the λ_{μ} -calculus that will be introduced in Sec. 8.1.

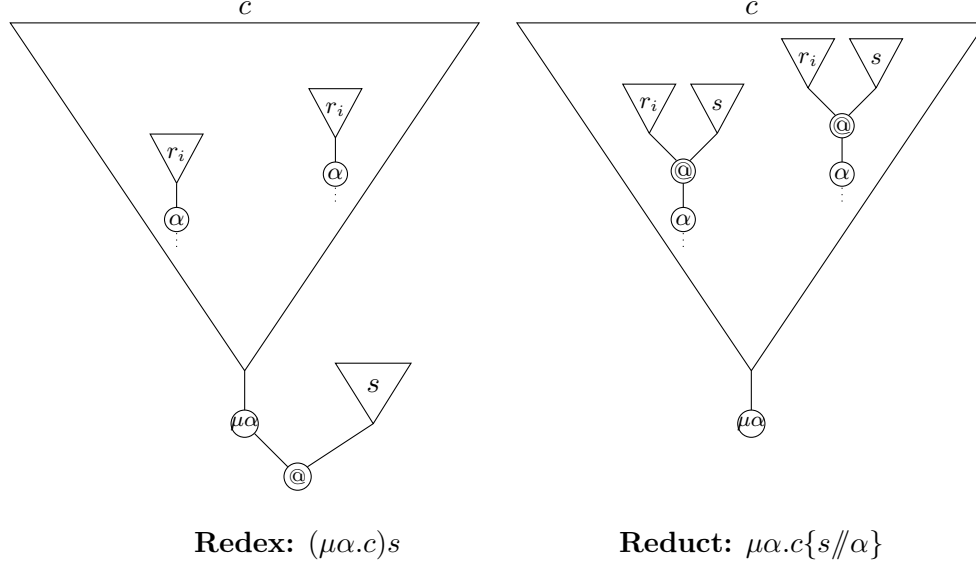


Figure 6.6: Mu-Reduction from the Tree Perspective

A typical example of expressivity in the λ_{μ} -calculus is the encoding of the control operator `call-cc` by setting `call-cc` := $\lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x)$ which gives raise to the following reduction sequence:

$$\begin{aligned} \text{call-cc } t u_1 \dots u_n &\rightarrow_{\beta} (\mu\alpha.[\alpha]t(\lambda x.\mu\beta.[\alpha]x))u_1 \dots u_n \\ &\rightarrow_{\mu} (\mu\alpha.[\alpha]t(\lambda x.\mu\beta.[\alpha]xu_1)u_1)u_2 \dots u_n \\ &\rightarrow_{\mu}^* \mu\alpha.[\alpha]t(\lambda x.\mu\beta.[\alpha]xu_1 \dots u_n)u_1 \dots u_n \end{aligned}$$

6.2.4 Subject Reduction for Simply Typed Lambda-Mu

The operational semantics of the λ_{μ} -calculus is designed so that a μ -reduction step mimicks a cut-elimination step for the focused classical natural deduction of Sec. 6.1.2:

Proposition 6.1. Simply typed λ_{μ} -calculus enjoys subject reduction: if $o \rightarrow_{\lambda_{\mu}} o'$ and $\Gamma \vdash o : B \mid \Delta$ is derivable, then $\Gamma \vdash o' \mid \Delta$ is derivable.

$$\frac{\frac{\frac{}{x : A \vdash x : A \mid \beta : B}}{x : A \vdash \mu\beta.[\alpha]x : B \mid \alpha : A}}{y : (A \rightarrow B) \rightarrow A \vdash y : (A \rightarrow B) \rightarrow A \mid \vdash \lambda x.\mu\beta.[\alpha]x : A \rightarrow B \mid \alpha : A}}{y : (A \rightarrow B) \rightarrow A \vdash y(\lambda x.\mu\beta.[\alpha]x) : A \mid \alpha : A}}{y : (A \rightarrow B) \rightarrow A \vdash \mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x) : A \mid \vdash \lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x) : (A \rightarrow B) \rightarrow A \rightarrow A \mid}$$

Figure 6.7: Simply Typing call-cc

We do not give a proof of this proposition, but it is illustrated by Fig. 6.8 by considering a derivation Π simply typing a μ -redex $(\mu\alpha.[\gamma]r)s$: we indicate only the types that are involved in the correctness of the reduction (we thus omit most part of the variable and name contexts). Obtaining a derivation Π' typing the reduct $\mu\alpha.[\gamma]r\{s//\alpha\}$ from the derivation Π typing the redex $(\mu\alpha.[\gamma]r)s$ is done by:

- Destroying the **app**-rule of the redex at the root of Π .
- Creating nested **app**-rules above each α -naming rule in Π_r . Those α -naming rule are applied to subterms r_i typed with $A \rightarrow B$ in Π .
- Duplicating the argument derivation Π_s typing the argument s of the redex with A as argument derivations of the aforementioned **app**-rules.
- Note that the derivations Π_i may be nested one in another (so that in the derivation typing the reduct, some Π'_i may be not quite equal to Π_i).

Since the r_i are typed with $A \rightarrow B$, then $r_i s$ is correctly typed with B . But now, in Π' , the α -naming rules save the type B instead of $A \rightarrow B$ in Π .

$$\begin{array}{c}
 \Pi_i (i \in I) \\
 \vdots \\
 \Pi_r \frac{r_i : A \rightarrow B \mid \gamma_i : C_i; \alpha : A \rightarrow B}{\mu\gamma_i.[\alpha]r : C_i \mid \alpha : A \rightarrow B} \\
 \vdots \\
 \frac{r : C \mid \gamma : C; \alpha : A \rightarrow B}{\mu\alpha.[\gamma]r : A \rightarrow B \mid \gamma : C} \quad \Phi \\
 \frac{\mu\alpha.[\gamma]r : A \rightarrow B \mid \gamma : C \quad s : A}{(\mu\alpha.[\gamma]r)s : B \mid \gamma : C} \text{app}
 \end{array}
 \qquad
 \begin{array}{c}
 \Pi'_i (i \in I) \\
 \vdots \\
 \frac{r_i\{s//\alpha\} : A \rightarrow B \mid \gamma_i : C_i; \alpha : B \quad s : A}{\Pi_r \frac{r_i\{s//\alpha\} s : B \mid \gamma_i : C_i; \alpha : B}{\mu\gamma_i.[\alpha]r_i\{s//\alpha\} s : C_i \mid \alpha : B}} \text{app} \\
 \vdots \\
 \frac{r\{s//\alpha\} : C \mid \gamma : C; \alpha : B}{\mu\alpha.[\gamma]r\{s//\alpha\} : B \mid \gamma : C}
 \end{array}$$

Figure 6.8: Subject Reduction for Simply Typed Λ_μ

6.2.5 Normalization in Lambda-Mu Calculus

Let us describe now a few variants of normalization¹ in λ_μ -calculus. A **normal form** of Λ_μ is a term that does not contain a redex (*i.e.* $(\lambda x.r)s$ or $(\mu\alpha.[\gamma]r)s$) and things go smoothly with weak and strong normalization:

Definition 6.1.

- A λ_μ -object o is said to be **weakly normalizing** if there is a reduction sequence starting at o reaching a normal form.
- A λ_μ -object o is said to be **strongly normalizing** if there is no infinite reduction sequence starting at o .

¹By lack of time and space, we did not study weak normalization (and its prospective type-theoretic characterisations) in λ_μ -calculus in the course of this PhD. Unforgetfulness (Definition 5.2) can probably be adapted for Λ_μ .

The following proposition, coming from Parigot [91], is an extension of Theorem 3.1:

Theorem 6.1. If a λ_μ -object is simply typable, then it is strongly normalizing.

Head normal forms and head reducible terms are a bit more complicated to define in Λ_μ . A HNF of Λ_μ is an object consisting in a redex $((\lambda x.r)s$ or $(\mu\alpha.[\gamma]r)s$) applied to a stack of arguments $t_1 \dots t_q$, then a series of possibly interwoven abstractions λx or $\mu\alpha.[\gamma]$, that can end with a naming $[\alpha]$. Formally, we define **head-contexts** \mathbf{O}_h by the following grammar:

$$\begin{aligned} \mathbf{O}_h &::= \mathbf{T}_h \mid \mathbf{C}_h \\ \mathbf{T}_h &::= \square t_1 \dots t_q (q \geq 0) \mid \lambda x.\mathbf{T}_h \mid \mu\alpha.\mathbf{C}_h \\ \mathbf{C}_h &::= [\alpha]\mathbf{T}_h \end{aligned}$$

A **head-normal form** is an object of the form $\mathbf{O}_h[x]$, where x is any variable replacing the constant \square (x may be captured in \mathbf{O}_h). Thus, for example $\mu\alpha.[\beta]\lambda y.x(\lambda z.z)$ is a head-normal form. An object $o \in \mathcal{O}_{\lambda_\mu}$ is said to be **head-normalizing**, written $o \in \text{HN}(\lambda_\mu)$, if $o \rightarrow_{\lambda_\mu}^* o'$, for some head-normal form o' . Remark that $o \in \text{HN}(\lambda_\mu)$ does not imply $o \in \text{SN}(\lambda_\mu)$ while the converse necessarily holds. We write $\text{HN}(\lambda)$ and $\text{SN}(\lambda)$ when t is restricted to be a λ -term and the reduction system is restricted to the β -reduction rule.

A redex $u = (\lambda x.r)s$ or $u = (\mu\alpha.[\gamma]r)s$ in an object of the form $t := \mathbf{O}_h[u]$ is called the **head-redex** of t . The reduction step $o \rightarrow o'$ contracting the head-redex of o is called **head-reduction**. The reduction sequence composing head-reduction steps until head-normal form is called the **head-strategy**. If the head-strategy starting at o terminates, then $o \in \text{HN}(\lambda_\mu)$, while the converse will be stated later (*cf.* Thm. 7.1), as it was proved for λ -calculus (Proposition 3.8).

A reduction step $o \rightarrow o'$ is said to be **erasing** iff $o = (\lambda x.u)v$ and $x \notin \text{fv}(u)$, or $o = (\mu\alpha.c)u$ and $\alpha \notin \text{fn}(c)$ (recall the beginning of Sec. 5.2) *e.g.*, $(\lambda x.z)y \rightarrow_\beta z$ and $(\mu\alpha.[\beta]x)I \rightarrow_\mu \mu\alpha.[\beta]x$ are erasing steps. A reduction step $o \rightarrow o'$ which is not erasing is called **non-erasing**. Reduction is stable by substitution and replacement. More precisely, if $o \rightarrow o'$, then $o[u/x] \rightarrow o'[u/x]$ and $o\{u//\alpha\} \rightarrow o'\{u//\alpha\}$. This gives the following corollary.

Corollary 6.1. If $o[u/x]$ is SN (resp. $o\{u//\alpha\}$ is SN), then o is also SN.

Chapter 7

Non-Idempotent Intersection and Union Types for Lambda-Mu

In this chapter, we present the first contribution of this thesis, that is published in [62]. We introduce two types systems $\mathcal{H}_{\lambda_\mu}$ and $\mathcal{S}_{\lambda_\mu}$ that *characterize* normalization in λ_μ -calculus (head and strong respectively), and do not just ensure it. For that, we will resort to **intersection** and **union types**, to overcome the limitations of simple typing.

In Chapter 6, we presented the pure and the simply typed λ_μ -calculus of Parigot [89] and we stated one of Parigot's theorems (Theorem 6.1), that establishes the strong normalization of simply typed λ_μ -calculus.

But note that Parigot's theorem is not a characterization and that the simple types for λ_μ suffer from the same drawbacks as they do for λ -calculus. For instance, we recall that the normal form $\Delta = \lambda x.x x$ is not a simply typable λ -term. The introduction of names in the operational semantics of λ_μ actually raises new examples of non-simply typable normal forms *e.g.*, $t := x(\mu\beta.[\alpha]I)(\mu\beta.[\alpha]K)$ (where $I = \lambda x.x$ and $K = \lambda xy.x$): indeed, $\lambda x.x$ has simple types of the form $A \rightarrow A$ and K simple types of the form $A' \rightarrow B' \rightarrow A'$ and the equality $A \rightarrow A = A' \rightarrow B' \rightarrow A'$ is impossible with simple types (because A' and $B' \rightarrow A'$ do not have the same number of symbols), so that the free name α cannot save both the type of I and that of K .

The idea behind intersection type systems was the following (introduction of Sec. 3.2): each occurrence of any variable x may be given a different type (and even several different types), what may be described as a form of “unconstrained polymorphism”. The types assigned to x are then collected by means of the intersection operator.

The idea behind (intersection and) union type systems for λ_μ -calculus [73] is the counterpart of this relaxation for names instead of just variables: a name α may be used to save as many types as we want. The types that α saved are collected by means of the **union operator**.

Concretely, in an intersection and union type system, a name may be assigned a new type (or several new ones) each time that it occurs in a command. Thus, the term t above becomes easily typable and more generally, it is easy to type any normal form of the λ_μ -calculus. With unconstrained polymorphic variables and names, it does not come as a surprise that we also recover subject expansion, which is the second ingredient (after the typing of the NF) to obtain an implication “Normalizing \Rightarrow Typable”, as it has been already noticed many times in this document (*e.g.*, Sec. 3.3.1 for head/weak normalization or end of Sec. 5.2.2 for strong normalization). Along with the direct implication “Typable \Rightarrow Normalizing”, this gives the expected characterization of normalization by typability.

By the way, why using a union operator rather than also using intersection for names ? Intuitively, because variable contexts are located on the left-hand side of \vdash and name contexts on its right-hand side. But we recall that $A_1, \dots, A_m \vdash B_1, \dots, B_n$ corresponds to $A_1 \wedge \dots \wedge A_m \vdash B_1 \vee \dots \vee B_n$. Thus, a variable x that has been assigned several formulas has morally been assigned their conjunction and a name α that saves several formulas morally saves their disjunction. More semantically, if we collected the types given to a name with the intersection \wedge instead of \vee , we would lose the Curry-Howard correspondence. For instance, instead of typing some λ_μ -term with $A \vee \neg A$, we would type it with $A \wedge \neg A$, which is a bit problematic...

Remark 7.1. There also exists types systems for λ_μ -calculus featuring only intersection (but not union) types, but still characterizing normalization [110,111]. Indeed, the use of union types for names may be bypassed by suitably resorting to subtyping (which may be seen as a form of weakening) and by restricting just a bit the polymorphism of names.

Let us understand how this works with an example and assume that the name α both saves $A \rightarrow C$ and $B \rightarrow C$ *i.e.* α saves $A \rightarrow C \vee B \rightarrow C$. Since in propositional logic, $(A \rightarrow C) \vee (B \rightarrow C)$ implies $(A \wedge B) \rightarrow C$, the name assignment $\alpha : (A \rightarrow C) \vee (B \rightarrow C)$ may be weakened into $\alpha : (A \wedge B) \rightarrow C$. Note that this works only because the arrow types $A \rightarrow C$ and $B \rightarrow C$ have the same target C and this kind of formalism supposes that the names are imposed to save types with a common target.

Thus, in order to characterize normalization in λ_μ , we will resort to intersection and union type. But moreover, we want to do that in a quantitative way, as it was announced in the introduction of Part II. Indeed, the non-idempotent approach provides very simple combinatorial arguments, only based on a decreasing *measure*, to characterize head or strongly normalizing terms by means of typability (recall Sec. 3.4 for HN and Sec. 5.2 for SN). We show that for every typable term t with type derivation Π , if t reduces to t' , then t' is typable with a type derivation Π' such that the measure of Π is strictly greater than that of Π' . In the well-known case of the λ -calculus, this measure is simply based on the structure of type tree derivations given by the number of its nodes (Definition 3.2), which strictly decreases along reduction. However, in the λ_μ -calculus, the creation of nested applications during μ -reduction may increase the number of nodes of the corresponding type derivations, so that such a naive definition of measure is not decreasing anymore. We then take into account not only the structure of derivations, but also the structure (*multiplicity* and *size*) of certain types appearing in the derivations, thus ensuring an overall decreasing of the measure during reduction.

7.1 Auxiliary Judgments and Choice Operators

As mentioned before, our results rely on typability of λ_μ -terms in suitable systems with non-idempotent types. Let us find first more readable denotations for the known systems, depending on:

- Their target calculus (for now, Λ and later, Λ_μ).
- The type of normalization they characterize (here, head or strong normalization).
- Whether they feature (or not) *auxiliary judgments*, to be presented in Sec. 7.1.

$$\frac{}{x : [\tau] \vdash x : \tau} \text{ax} \quad \frac{\Gamma \vdash t : \tau}{\Gamma \parallel x \vdash \lambda x.t : \Gamma(x) \rightarrow \tau} \text{abs}$$

$$\frac{\Gamma \vdash t : [\sigma_k]_{k \in K} \rightarrow \tau \quad (\Gamma_k \vdash u : \sigma_k)_{k \in K}}{\Gamma \wedge_{k \in K} \Gamma_k \vdash t u : \tau} \text{app}$$

Figure 7.1: System \mathcal{H}_λ

$$\frac{}{x : [\tau] \vdash x : \tau} \text{ax} \quad \frac{\Gamma \vdash t : \tau}{\Gamma \parallel x \vdash \lambda x.t : \Gamma(x) \rightarrow \tau} \text{abs}$$

$$\frac{(\Gamma_k \vdash t : \sigma_k)_{k \in K}}{\wedge_{k \in K} \Gamma_k \Vdash t : [\sigma_k]_{k \in K}} \wedge \quad \frac{\Gamma \vdash t : \mathcal{I} \rightarrow \sigma \quad \Gamma' \Vdash u : \mathcal{I}}{\Gamma \wedge \Gamma' \vdash t u : \sigma} \text{app}$$

Figure 7.2: System \mathcal{H}'_λ

System \mathcal{R}_0 (Sec. 3.2.4), that characterizes head normalization, for λ -calculus (Proposition 3.7) will be renamed \mathcal{H}_λ and system \mathcal{S} (Sec. 5.2.1), that characterizes strong normalization for λ -calculus (Proposition 5.5), will be renamed \mathcal{S}_λ .

Auxiliary judgments are a great tool to simplify the grammar of derivations of intersection type systems. We may introduce them to reformulate system \mathcal{H}_λ (*i.e.* Gardner/de Carvalho's system \mathcal{R}_0), which is recalled in Fig. 7.1. For that, we now distinguish two sorts of judgments: **regular judgments** of the form $\Gamma \vdash t : \sigma$ assign *types* to terms, and **auxiliary judgments** of the form $\Gamma \Vdash t : \mathcal{I}$ assign *intersection types* $\mathcal{I} := [\sigma_i]_{i \in I}$ to terms.

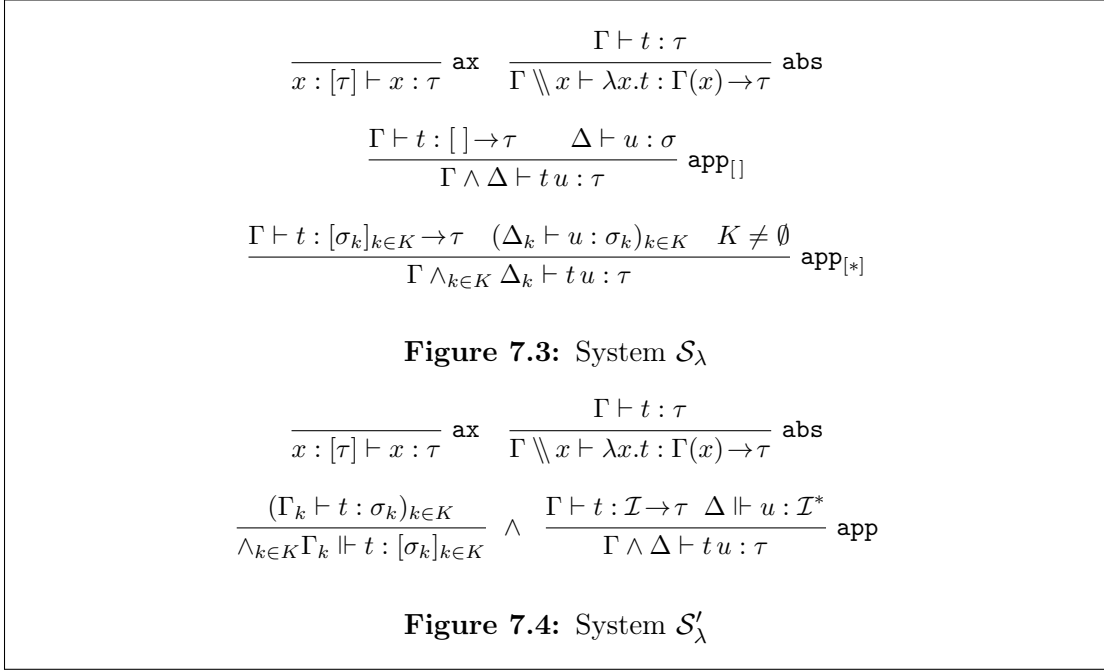
An equivalent formulation of system \mathcal{H}_λ , called \mathcal{H}'_λ , is given in Fig. 7.2. There are two inherited forms of type derivations: **regular** (resp. **auxiliary**) **derivations** are those that conclude with regular (resp. auxiliary) judgments. Notice that $I = \emptyset$ in rule (\wedge) gives $\Vdash u : []$ for *any* term u , *e.g.*, $\Vdash \Omega : []$, so that one can derive $x : [\tau] \vdash (\lambda y.x)\Omega : \tau$ in this system (see Sec. 3.4.1). Notice also that systems \mathcal{H}_λ and \mathcal{H}'_λ are *relevant*, *i.e.* they lack weakening. Equivalence between \mathcal{H}_λ and \mathcal{H}'_λ gives the following corollary of Proposition 3.7:

Corollary 7.1. Let $t \in \mathcal{T}_\lambda$. Then t is \mathcal{H}'_λ -typable iff t is HN.

Auxiliary judgments turn out to be fundamental to lighten the key tools of the forthcoming typing systems for λ_μ .

Now, let us turn our attention to system \mathcal{S}_λ , coming from Bucciarelli, Kesner and Ventura and presented in Sec. 5.2.1. This system characterizes strong normalization (Theorem 5.5) and its rules are recalled in Fig. 7.3. Applications $t u$ are typed according to one of the two following cases:

- If t is typed with $[\sigma_i]_{i \in I} \rightarrow \tau$ and I is not empty, then the argument u must be typed with σ_i for all $i \in I$ (while respecting the multiplicities of the σ_i). This corresponds to an auxiliary typing of u with $[\sigma_i]_{i \in I}$.
- If t is typed with $[] \rightarrow \tau$ (the source of the argument is empty), then the argument u must still be typed with an arbitrary type σ .



In the second case, σ is indeed arbitrary (it is unrelated to the type of t), and thus, may be chosen non-deterministically (as long as u is typable with σ): this can be handled with the following **non-deterministic choice operator** $_*$, defined on intersection types as follows:

$$[\sigma_i]_{i \in I}^* := \begin{cases} [\tau] & \text{if } I = \emptyset \text{ and } \tau \text{ is any arbitrary type} \\ [\sigma_i]_{i \in I} & \text{if } K \neq \emptyset \end{cases}$$

System \mathcal{S}_λ can then be reformulated in a system called \mathcal{S}'_λ , given in Fig. 7.4 (the metavariable \mathcal{I} still stands for a multiset type $[\sigma_i]_{i \in I}$). As before, we use regular as well as auxiliary judgments. The two rules **app** $[]$ and **app** $[*]$ are now subsumed in the unique rule **app**. Notice that $I = \emptyset$ in rule \wedge is still possible, but derivations of the form $\Vdash t : []$, representing untyped terms, will never be used. The operator $_*$ in rule **app** is used to impose an arbitrary type to an erasable term, *i.e.* when t has type $[] \rightarrow \tau$, then u still needs to be typed with an arbitrary type $[\sigma]$, as in system \mathcal{S}_λ . Thus, the auxiliary judgment typing u on the right premise of **app** cannot assign $[]$ to u . This should be understood as a sort of *controlled weakening*, as explained in Remark 5.7. Here is an example of type derivation in system \mathcal{S}'_λ , which is the transcription of the one concluding Sec. 5.2.1:

$$\frac{\frac{}{x : [\sigma] \vdash x : \sigma} \quad \frac{}{z : [\tau] \vdash z : \tau}}{x : [\sigma] \vdash \lambda y.x : [] \rightarrow \sigma} \quad z : [\tau] \Vdash z : [\tau]}{x : [\sigma], z : [\tau] \vdash (\lambda y.x)z : \sigma}$$

Since \mathcal{S}_λ and \mathcal{S}'_λ are equivalent, Theorem 5.5 gives:

Corollary 7.2. Let $t \in \Lambda$. Then t is \mathcal{S}'_λ -typable iff t is SN.

Remark 7.2 (Auxiliary Judgments and Strictness). Strict intersection types [106] are defined by forbidding intersection types as the targets of arrow types (see for instance

[107], Def. 1.12) and usually, strict intersection type systems are presented so that a term cannot have an intersection type: indeed, if we had $\Gamma; x : [\sigma_i]_{i \in I} \vdash t : [\tau_j]_{j \in J}$, then, syntactically, the **abs**-rule would give $\Gamma \vdash \lambda x.t : [\sigma_i]_{i \in I} \rightarrow [\tau_j]_{j \in J}$ and $[\sigma_i]_{i \in I} \rightarrow [\tau_j]_{j \in J}$ *i.e.* the type system would not be correctly defined.

However, our syntax with auxiliary derivations/judgments does not suffer from this problem: it is not licit to apply an **abs**-rule to an *auxiliary* judgment $\Gamma; x : [\sigma_i]_{i \in I} \Vdash t : [\tau_j]_{j \in J}$ and it is thus impossible to get a type that is not strict in \mathcal{H}'_λ and \mathcal{S}'_λ .

7.2 Quantitative Type Systems for the λ_μ -Calculus

We present in this section two quantitative systems for the λ_μ -calculus, systems $\mathcal{H}_{\lambda_\mu}$ (Sec. 7.2.2) and $\mathcal{S}_{\lambda_\mu}$ (Sec. 7.2.4), characterizing, respectively, head and strong λ_μ -normalizing objects. Since λ -calculus is embedded in the λ_μ -calculus, then the starting points to design $\mathcal{H}_{\lambda_\mu}$ and $\mathcal{S}_{\lambda_\mu}$ are, respectively, \mathcal{H}'_λ and \mathcal{S}'_λ , introduced in Sec. 7.1.

The set of head normalizing and that strongly normalizing λ_μ -objects were defined in Sec. 6.2.5. We respectively denote them $\text{HN}(\lambda_\mu)$ and $\text{SN}(\lambda_\mu)$.

7.2.1 Types

We consider again a countable set \mathcal{O} of type variables. The following categories of types are defined:

(Object Types)	\mathcal{A}	$:=$	$\mathcal{C} \mid \mathcal{U}$
(Command Type)	\mathcal{C}	$:=$	$\#$
(Union Types)	\mathcal{U}, \mathcal{V}	$::=$	$\langle \sigma_k \rangle_{k \in K}$
(Intersection Types)	\mathcal{I}	$::=$	$[\mathcal{U}_k]_{k \in K}$
(Types)	σ, τ	$::=$	$o \mid \mathcal{I} \rightarrow \mathcal{U}$

The constant $\#$ is used to type commands, union types to type terms and intersection types to type variables (thus left-hand sides of arrows). Both $[\sigma_k]_{k \in \{1..n\}}$ and $\langle \sigma_k \rangle_{k \in \{1..n\}}$ can be seen as *multisets*, representing, respectively, $\sigma_1 \cap \dots \cap \sigma_n$ and $\sigma_1 \cup \dots \cup \sigma_n$, where \cap and \cup are both associative, commutative, but *non-idempotent*. We may omit the indices in the simplest case: thus $[\mathcal{U}]$ and $\langle \sigma \rangle$ denote singleton multisets. We define the operator \wedge (resp. \vee) on intersection (resp. union) multiset types by $[\mathcal{U}_k]_{k \in K} \wedge [\mathcal{V}_\ell]_{\ell \in L} := [\mathcal{U}_k]_{k \in K} + [\mathcal{V}_\ell]_{\ell \in L}$ and $\langle \sigma_k \rangle_{k \in K} \vee \langle \tau_\ell \rangle_{\ell \in L} := \langle \sigma_k \rangle_{k \in K} + \langle \tau_\ell \rangle_{\ell \in L}$, where $+$ always means multiset union.

Remark 7.3 (Overload of Metavariable o). The metavariable o is overloaded: it may denote a type variable as well as an object of λ_μ , but this ambiguity will always be straightforwardly dissolved by the context.

The *non-deterministic choice* operator $_*$ is now defined on intersection *and* union types:

$$\begin{aligned} [\mathcal{U}_k]_{k \in K}^* &:= \begin{cases} [\mathcal{U}] & \text{if } K = \emptyset \text{ } \mathcal{U} \neq \langle \rangle \text{ is any arbitrary non-empty union type} \\ [\mathcal{U}_k]_{k \in K} & \text{if } K \neq \emptyset \end{cases} \\ \langle \sigma_k \rangle_{k \in K}^* &:= \begin{cases} \langle \sigma \rangle & \text{if } K = \emptyset \text{ and } \sigma \text{ is any arbitrary blind type} \\ \langle \sigma_k \rangle_{k \in K} & \text{if } K \neq \emptyset \end{cases} \end{aligned}$$

where a blind type is a type of the form $[\] \rightarrow \dots \rightarrow [\] \rightarrow o$. The choice operator for union type is defined so that (1) the empty union cannot be assigned to μ -abstractions (see discussion on the non-emptiness of union-types, page 150) (2) subject reduction is guaranteed in system $\mathcal{H}_{\lambda_\mu}$ for erasing steps $(\mu\alpha.c)u \rightarrow \mu\alpha.c$ ($\alpha \notin \text{fn}(c)$).

The **arity** of types (resp. union multiset types) is defined by induction: for types σ , if $\sigma = \mathcal{I} \rightarrow \mathcal{U}$, then $\mathbf{ar}(\sigma) := \mathbf{ar}(\mathcal{U}) + 1$, otherwise, $\mathbf{ar}(\sigma) := 0$; for union multiset types, $\mathbf{ar}(\langle \sigma_k \rangle_{k \in K}) := \sum_{k \in K} \mathbf{ar}(\sigma_k)$. Thus, the arity of a type is the number of *top-level* arrows that it contains. The **cardinality of multisets** is defined, as before, by $|\langle \mathcal{U}_k \rangle_{k \in K}| = |\langle \sigma_k \rangle_{k \in K}| := |K|$.

As before also, **Variable assignments** (Γ) are functions from variables to intersection multiset types. Similarly, **name assignments** (Δ), are functions from names to union multiset types. The **domain of** Δ is given by $\mathbf{dom}(\Delta) := \{\alpha \mid \Delta(x) \neq \langle \rangle\}$, where $\langle \rangle$ is the empty union multiset. When $\alpha \notin \mathbf{dom}(\Delta)$, then $\Delta(x)$ stands for $\langle \rangle$. We write $\Delta \vee \Delta'$ for $\alpha \mapsto \Delta(\alpha) + \Delta'(\alpha)$, where $\mathbf{dom}(\Delta \vee \Delta') = \mathbf{dom}(\Delta) \cup \mathbf{dom}(\Delta')$.

When $\mathbf{dom}(\Gamma)$ and $\mathbf{dom}(\Gamma')$ are disjoint, we may write $\Gamma; \Gamma'$ instead of $\Gamma \wedge \Gamma'$. We write $x : \langle \mathcal{U}_k \rangle_{k \in K}; \Gamma$, even when $K = \emptyset$, for the following variable assignment ($x : \langle \mathcal{U}_k \rangle_{k \in K}; \Gamma$)(x) = $\langle \mathcal{U}_k \rangle_{k \in K}$ and $(x : \langle \mathcal{U}_k \rangle_{k \in K}; \Gamma)(y) = \Gamma(y)$ if $y \neq x$. Similar concepts apply to name assignments, so that $\alpha : \langle \sigma_k \rangle_{k \in K}; \Delta$ and $\Delta \parallel \alpha$ are defined as expected.

We now present our typing systems $\mathcal{H}_{\lambda_\mu}$ and $\mathcal{S}_{\lambda_\mu}$, both having **regular** (resp. **auxiliary**) judgments of the form $\Gamma \vdash t : \mathcal{U} \mid \Delta$ (resp. $\Gamma \Vdash t : \mathcal{I} \mid \Delta$), together with their respective notions of regular and auxiliary derivations. An important syntactical property they enjoy is that both are **syntax directed**, *i.e.* for each (regular/auxiliary) typing judgment j there is a *unique* typing rule whose conclusion matches the judgment j (Remark 3.4, p. 83). This makes our proofs much simpler than those arising with idempotent types which are based on long generation lemmas (see also Sec. 3.2.1).

7.2.2 System $\mathcal{H}_{\lambda_\mu}$

In this section we present a quantitative typing system for λ_μ , called $\mathcal{H}_{\lambda_\mu}$, characterizing head λ_μ -normalization. It can be seen as a first intuitive step to understand the typing system $\mathcal{S}_{\lambda_\mu}$, introduced later in Sec. 7.2.4, and characterizing strong λ_μ -normalization. However, the two systems will not be described and studied in the same way: by lack of space we choose to discuss $\mathcal{H}_{\lambda_\mu}$ in a more informal and compact way, while reserving more space and discussion to system $\mathcal{S}_{\lambda_\mu}$.

The (syntax directed) rules of the typing system $\mathcal{H}_{\lambda_\mu}$ appear in Fig. 7.5. Rule **app** is to be understood as a *logical admissible* rule: if union (resp. intersection) is interpreted as the \vee (resp. \wedge) logical connective, then $\vee_{k \in K} (\mathcal{I}_k \Rightarrow \mathcal{U}_k)$ and $(\wedge_{k \in K} \mathcal{I}_k)$ implies $(\vee_{k \in K} \mathcal{U}_k)$. Intuitively, if (1) in a family of implication, *at least one* is true and (2) *all* the premises of the implications are true, then we may conclude that *at least one* of the targets of those implications is true. As in the simply typed λ_μ -calculus (Sec. 6.2.2), the **name-rule** saves a type \mathcal{U} for the name α , however, in our system, the corresponding name assignment $\Delta \vee \{\alpha : \mathcal{U}\}$, specified by means of \vee , collects *all* the types that α has been assigned during the derivation. Notice that the **restore-rule** is not deterministic since $\Delta(\alpha)^*$ denotes an arbitrary union type, a choice that is now discussed.

The use of $_*$ in the **restore-rule** can be seen as a *weakening* on the right hand-sides of sequents. However, as it will be discussed in Sec. 7.5, this does not compromise the nice features of relevant type assignments systems, leading us to question the notion of *relevance* in general.

In simply typed λ_μ , **call-cc** = $\lambda y. \mu \alpha. [\alpha] y (\lambda x. \mu \beta. [\alpha] x)$ is typed with $((a \rightarrow b) \rightarrow a) \rightarrow a$ (Peirce's Law), as seen in Fig. 6.7, so that the fact that α is *used* twice in the type derivation is not explicitly materialized (same comment applies to idempotent intersection/union types). This makes a strong contrast with the derivation in Fig. 7.6,

$$\begin{array}{c}
\frac{\mathcal{U} \neq \langle \rangle}{x : [\mathcal{U}] \vdash x : \mathcal{U} \mid \emptyset} \text{ax} \qquad \frac{\Gamma \vdash t : \mathcal{U} \mid \Delta}{\Gamma \parallel x \vdash \lambda x.t : \langle \Gamma(x) \rightarrow \mathcal{U} \rangle \mid \Delta} \text{abs} \\
\\
\frac{\Gamma \vdash t : \mathcal{U} \mid \Delta}{\Gamma \vdash [\alpha]t : \# \mid \Delta \vee \{\alpha : \mathcal{U}\}} \text{name} \qquad \frac{\Gamma \vdash c : \# \mid \Delta}{\Gamma \vdash \mu\alpha.c : \Delta(\alpha)^* \mid \Delta \parallel \alpha} \text{restore} \\
\\
\frac{(\Gamma_k \vdash t : \mathcal{U}_k \mid \Delta_k)_{k \in K}}{\wedge_{k \in K} \Gamma_k \Vdash t : [\mathcal{U}_k]_{k \in K} \mid \vee_{k \in K} \Delta_k} \wedge \\
\\
\frac{\Gamma_t \vdash t : \langle \mathcal{I}_k \rightarrow \mathcal{U}_k \rangle_{k \in K} \mid \Delta_t \quad \Gamma_u \Vdash u : \wedge_{k \in K} \mathcal{I}_k \mid \Delta_u}{\Gamma_t \wedge \Gamma_u \vdash tu : \vee_{k \in K} \mathcal{U}_k \mid \Delta_t \vee \Delta_u} \text{app}
\end{array}$$

Figure 7.5: System $\mathcal{H}_{\lambda_\mu}$

$$\frac{
\frac{
\frac{
\frac{
\frac{
\frac{
\frac{
\frac{x : [\mathcal{U}_a] \vdash x : \mathcal{U}_a \mid}{x : [\mathcal{U}_a] \vdash [\alpha]x : \# \mid \alpha : \mathcal{U}_a}
}{x : [\mathcal{U}_a] \vdash \mu\beta.[\alpha]x : \mathcal{U}_b \mid \alpha : \mathcal{U}_a}
}{\vdash \lambda x.\mu\beta.[\alpha]x : \langle [\mathcal{U}_a] \rightarrow \mathcal{U}_b \rangle \mid \alpha : \mathcal{U}_a}
}{\Phi_y \quad \Vdash \lambda x.\mu\beta.[\alpha]x : \langle [\mathcal{U}_a] \rightarrow \mathcal{U}_b \rangle \mid \alpha : \mathcal{U}_a}
}{y : [\mathcal{U}_y] \vdash y(\lambda x.\mu\beta.[\alpha]x) : \mathcal{U}_a \mid \alpha : \mathcal{U}_a}
}{y : [\mathcal{U}_y] \vdash [\alpha]y(\lambda x.\mu\beta.[\alpha]x) : \# \mid \alpha : \langle \mathcal{U}_a, \mathcal{U}_a \rangle}
}{y : [\mathcal{U}_y] \vdash \mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x) : \langle \mathcal{U}_a, \mathcal{U}_a \rangle \mid}
}{\vdash \lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x) : \langle [\mathcal{U}_y] \rightarrow \langle \mathcal{U}_a, \mathcal{U}_a \rangle \rangle \mid}
}
}
}
}
}
}
}$$

Figure 7.6: Typing call-cc

where $\mathcal{U}_a := \langle b \rangle$, $\mathcal{U}_b := \langle b \rangle$ (given two type variables a and b), $\mathcal{U}_y := \langle \langle [\mathcal{U}_a] \rightarrow \mathcal{U}_b \rangle \rightarrow \mathcal{U}_a \rangle$ and $\Phi_y \triangleright y : [\mathcal{U}_y] \vdash y : \mathcal{U}_y \mid \cdot$. Indeed, we can distinguish two different uses of names :

- The name α is **saved** twice by a **name**-rule : once for x and once for $y(\lambda x.\mu\beta.[\alpha]x)$, both times with type \mathcal{U}_a . After that, the abstraction $\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x)$ **restores** the types that were stored by the two free occurrences of α . A similar phenomenon occurs with λ -abstractions, which restore the types of the free occurrences of variables in the body of the functions.
- The name β is not free in $[\alpha]x$, so that a new union type \mathcal{U}_b is introduced to type the abstraction $\mu\beta.[\alpha]x$. From a logical point of view, this corresponds to a *weakening* on the right-hand side of the sequent. Notice, consequently, that λ and

$$\begin{aligned}
\mathbf{sz}\left(\frac{}{x : [\mathcal{U}] \vdash x : \mathcal{U} \mid \emptyset} \mathbf{ax}\right) &:= 1 \\
\mathbf{sz}\left(\frac{\Phi_t \triangleright t}{\Gamma \parallel x \vdash \lambda x.t : \langle \Gamma(x) \rightarrow \mathcal{U} \rangle \mid \Delta} \mathbf{abs}\right) &:= \mathbf{sz}(\Phi_t) + 1 \\
\mathbf{sz}\left(\frac{\Phi_t \triangleright t}{\Gamma \vdash [\alpha]t : \# \mid \Delta \vee \{\alpha : \mathcal{U}\}} \mathbf{name}\right) &:= \mathbf{sz}(\Phi_t) + \mathbf{ar}(\mathcal{U}) \\
\mathbf{sz}\left(\frac{\Phi_c \triangleright c}{\Gamma \vdash \mu\alpha.c : \Delta(\alpha)^* \mid \Delta \parallel \alpha} \mathbf{restore}\right) &:= \mathbf{sz}(\Phi_c) + 1 \\
\mathbf{sz}\left(\frac{(\Phi_k \triangleright t)_{k \in K}}{\bigwedge_{k \in K} \Gamma_k \Vdash t : [\mathcal{U}_k]_{k \in K} \mid \bigvee_{k \in K} \Delta_k} \wedge\right) &:= \sum_{k \in K} \mathbf{sz}(\Phi_k) \\
\mathbf{sz}\left(\frac{\Phi_t \triangleright t \quad \Phi_u \triangleright u}{\Gamma \vdash tu : \bigvee_{k \in K} \mathcal{V}_k \mid \Delta} \mathbf{app}\right) &:= \mathbf{sz}(\Phi_t) + \mathbf{sz}(\Phi_u) + |K|
\end{aligned}$$

Figure 7.7: Derivation Sizes in $\mathcal{S}_{\lambda\mu}$

μ -abstractions are not treated symmetrically: when x is not free in t , then $\lambda x.t$ will be typed with $[\] \rightarrow \sigma$ (where σ is the type of t), and no new intersection type is introduced for the abstracted variable x .

Thus, μ -abstractions have two uses: to restore saved types and to create new types, which explains the fact that empty union types are banned. Indeed, if $\triangleright \Gamma \vdash t : \mathcal{U} \mid \Delta$, then $\mathcal{U} \neq \langle \rangle$.

Why union types cannot be empty? Let us suppose that empty union types may be introduced by the **restore**-rule, at least when $\alpha \notin \mathbf{fn}(c)$, so that for example $t = \mu\beta.[\alpha]x$ would be typed with $\langle \rangle$ (this can be obtained by simply changing $\Delta(\alpha)^*$ to $\Delta(\alpha)$ in the **restore**-rule). Suppose also an object o containing 2 occurrences of the subcommand $[\gamma]t$, so that γ receives the union type $\langle \rangle$ twice in the corresponding name assignment. Then, the term $\mu\gamma.o$ will be typed with $\langle \rangle = \langle \rangle \vee \langle \rangle$, which does not reflect the fact that γ is used twice, thus losing the *quantitative* flavour of the system (see also a formal argument just after Lemma 7.2). Note that in Laurent's system featuring idempotent intersection and union types, commands cannot have an empty union type either (see Sec. 4.1 of [73]).

We extend now the notion of **size derivation** (Sec. 3.2) to system $\mathcal{H}_{\lambda\mu}$ (and later to $\mathcal{S}_{\lambda\mu}$), which is a natural number representing the amount of crucial information in a tree derivation. Formally, for any type derivation Φ , $\mathbf{sz}(\Phi)$ is inductively defined by the following rules, where we use an abbreviated notation for the premises. Indeed, the size of derivations typing commands takes into account the arity of their corresponding type; and this is essential to materialize a decreasing measure for μ -reduction (see Sec. 7.2.3 and 7.3 for $\mathcal{S}_{\lambda\mu}$). Notice that $\mathbf{sz}(\Phi) \geq 1$ holds for any *regular* derivation Φ , whereas, by definition, the derivation of the empty auxiliary judgment $\Vdash t : [\] \mid$ has size 0.

System $\mathcal{H}_{\lambda\mu}$ behaves as expected, in particular, typing is stable by reduction (Subject Reduction) and anti-reduction (Subject Expansion), and Subject Reduction is actually weighted, as in Proposition 3.6. Moreover,

Theorem 7.1. Let $o \in \mathcal{O}_{\lambda\mu}$. Then o is $\mathcal{H}_{\lambda\mu}$ -typable iff $o \in \mathbf{HN}(\lambda_\mu)$ iff the head-strategy terminates on o . Moreover, if o is $\mathcal{H}_{\lambda\mu}$ -typable with tree derivation Π , then $\mathbf{sz}(\Pi)$ gives an upper bound to the length of the head-reduction strategy starting at o .

We do not provide the proof of this theorem, because it uses special cases of the more general technology that we are going to develop later to deal with strong non-

malization. However, in Sec. 7.2.3, we explain how system $\mathcal{H}_{\lambda_\mu}$ was designed from a “global” perspective. Notice that Theorem 7.1 ensures that the head-strategy is complete for head-normalization in λ_μ , thus generalizing Proposition 3.8.

A last comment of this section concerns the restriction of system $\mathcal{H}_{\lambda_\mu}$ to the pure λ -calculus: union types, name assignments and rules **restore** and **name** are no more necessary, so that every union multiset takes the single form $\langle \tau \rangle$, which can be simply identified with τ . Thus, the restricted typing system $\mathcal{H}_{\lambda_\mu}$ becomes the one in Fig. 7.2.

7.2.3 Design of System $\mathcal{H}_{\lambda_\mu}$

Figure 7.8 aims to explain why subject reduction holds in system $\mathcal{H}_{\lambda_\mu}$. For that, we consider the reduction step $t = (\mu\alpha.[\gamma]r)s \rightarrow \mu\alpha.[\gamma]r\{s//\alpha\} = t'$ where t is the subject of a $\mathcal{H}_{\lambda_\mu}$ -derivation Π . We are mainly interested in the types \mathcal{F}_i , \mathcal{U}_i and the intersection types \mathcal{I}_i : each \mathcal{F}_i is a “arrow union type” such that $\text{Sc}(\mathcal{F}_i) = \mathcal{I}_i$ and $\text{Tg}(\mathcal{F}_i) = \mathcal{U}_i$ *i.e.* there are $\mathcal{I}_{i,k}$ and $\mathcal{U}_{i,k}$ (k ranging over some $K(i)$) such that $\mathcal{F}_i = \langle \mathcal{I}_{i,k} \rightarrow \mathcal{U}_{i,k} \rangle_{k \in K(i)}$, $\mathcal{I}_i = \bigwedge_{k \in K(i)} \mathcal{I}_{i,k}$ and $\mathcal{U}_i = \langle \mathcal{U}_{i,k} \rangle_{k \in K(i)}$. We set $\mathcal{F} = \bigvee_{i \in I} \mathcal{F}_i = \langle \mathcal{I}_k \rightarrow \mathcal{U}_k \rangle_{k \in K}$ with $K = \uplus_{i \in I} K_i$ (disjoint union). The letter \mathcal{F} stands for “function type” and a term t with type \mathcal{F}_i can be fed with an argument u of type \mathcal{I}_i , which yields the term tu of type \mathcal{U}_i . Likewise, if t is typed with $\bigvee_{i \in I} \mathcal{F}_i$, then it can be fed with an argument u of type $\bigwedge_{i \in I} \mathcal{I}_i$, which yields the term tu of type $\bigvee_{i \in I} \mathcal{U}_i$. Remember for the following that \mathcal{F} is a type that contain $|K|$ strict arrow types.

We associate the *union type* metavariable C to the *name* metavariable γ (although C should not be used for union type): they are involved in the μ -rules, but they are not affected by the subject reduction (compare Π and Π').

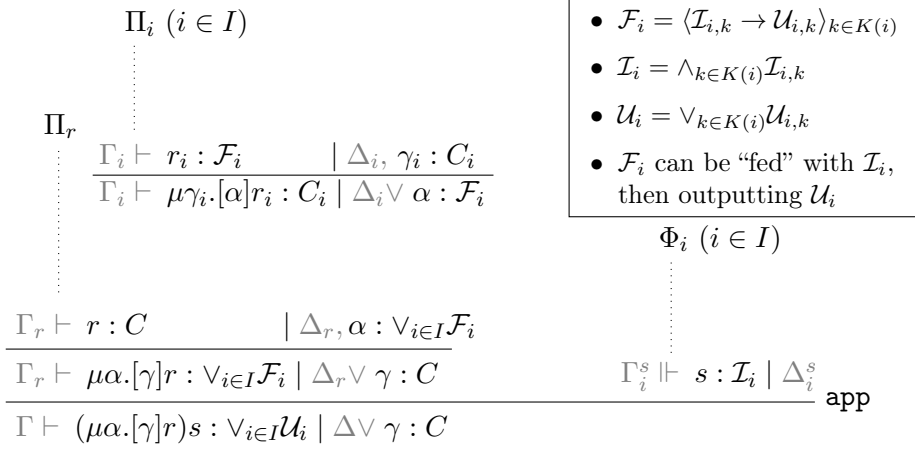
Obtaining a derivation Π' typing the reduct $\mu\alpha.[\gamma]r\{s//\alpha\}$ from the derivation Π typing the redex $(\mu\alpha.[\gamma]r)s$ is done by:

- Destroying the **app**-rule of the redex at the root of Π .
- Creating nested **app**-rules above each α -naming rule in Π_τ .
- Suitably dispatching components of the auxiliary argument derivation Ψ^s typing the argument s of the redex as auxiliary argument derivations of the aforementioned created **app**-rules.

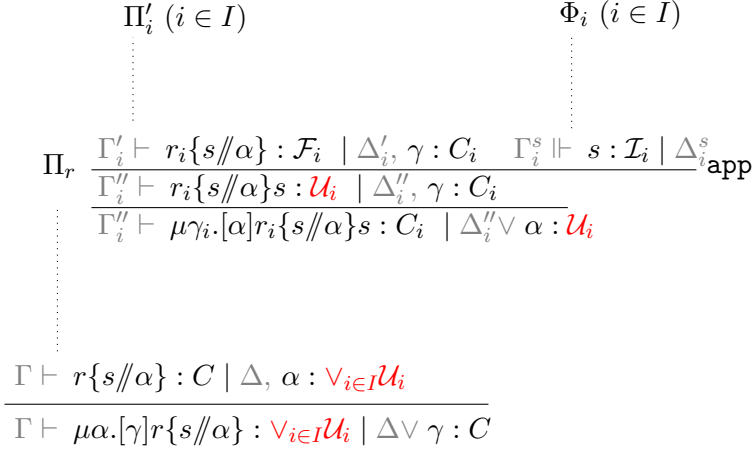
All this is represented in Fig. 7.8, in which we have put the uninvolved parts of the typing in lighter shades. The subterm $\mu\gamma_i.[\alpha]r_i\{s//\alpha\}$ of the derivation Π' typing the reduct has the same type C_i as the subterm $\mu\gamma_i.[\alpha]r_i$ of the derivation Π typing the redex: this is roughly the reason why derivation Π' is correct (*i.e.* respects the typing rules of $\mathcal{H}_{\lambda_\mu}$).

Now, notice that, while reducing the μ -redex, the **app**-rule at the root is destroyed but that there are $|I|$ new **app**-rules that appear for each α -command. That is why the **app**-rule at the root of the redex should count more than the new ones. The most natural choice is to associate to an **app**-rule the size of union type it uses on the left handside *i.e.* the number $|K|$ of strict arrow types. However, this does not decrease the total size of **app**-rules in the derivation (but this does not increase it either). That is why one needs to ensure the decrease in another way, which is done through the naming rules.

The name α is assigned the types \mathcal{F}_i ($i \in I$) in the redex and the types \mathcal{U}_i ($i \in I$) in the reduct: intuitively, α is used to save smaller types in the “reduct derivation” than in the “redex derivation”. Formally, we notice that there is a decrease in arity of the



After reducing the μ -redex at root, we obtain:



Remark 7.4 (Reading the Figure).

- The subderivations of Π_r concluding with an α -naming rule (*i.e.* the Π_i) may be nested one into another *e.g.*, some Π_i may be a subderivation of some Π_j .
- We made the implicit assumption that, for all $i \in I$, $C_i \neq \langle \rangle$. If not, we should decorate the figure with choice operators *e.g.*, writing $\mu\gamma_i.[\alpha]r_i : C_i^*$ instead of $\mu\gamma_i.[\alpha]r_i : C_i$.
- We also assume that $I \neq \emptyset$ (non-erasing red. step) but a figure can be easily drawn in that case. Note that it is vital to create h-types in this case (see the example in Sec. 7.5).
- There should be only one argument auxiliary derivation Φ typing the argument s of the redex, concluding with $+_{i \in I} \Gamma_i^s \Vdash s : +_{i \in I} \mathcal{I}_i \quad | \quad +_{i \in I} \Delta_i^s$ but we have preferred here to decompose it into several auxiliary derivations (as allowed by Lemma 7.3) to prepare subject reduction.
- Π'_i , Γ'_i and Δ'_i are not necessarily equal to Π_i , Γ_i and Δ_i , because there may be some nested α -commands. Moreover, $\Gamma''_i = \Gamma'_i + (+_{k \in K(i)} \Gamma_i)$ and $\Delta''_i = \Delta'_i \vee (\vee_{k \in K(i)} \Delta_{i,k})$.
- It may be that $r = r_i$ for some i . In that case, we need to slightly modify the above picture.

Figure 7.8: Subject Mu-Reduction

$$\begin{array}{c}
\frac{\mathcal{U} \neq \langle \rangle}{x : [\mathcal{U}] \vdash x : \mathcal{U} \mid \emptyset} \text{ ax} \qquad \frac{\Gamma \vdash t : \mathcal{U} \mid \Delta}{\Gamma \parallel x \vdash \lambda x.t : \langle \Gamma(x) \rightarrow \mathcal{U} \rangle \mid \Delta} \text{ abs} \\
\\
\frac{\Gamma \vdash t : \mathcal{U} \mid \Delta}{\Gamma \vdash [\alpha]t : \# \mid \Delta \vee \{\alpha : \mathcal{U}\}} \text{ name} \qquad \frac{\Gamma \vdash c : \# \mid \Delta}{\Gamma \vdash \mu\alpha.c : \Delta(\alpha)^* \mid \Delta \parallel \alpha} \text{ restore} \\
\\
\frac{(\Gamma_k \vdash t : \mathcal{U}_k \mid \Delta_k)_{k \in K}}{\wedge_{k \in K} \Gamma_k \Vdash t : [\mathcal{U}_k]_{k \in K} \mid \vee_{k \in K} \Delta_k} \wedge \\
\\
\frac{\Gamma_t \vdash t : \langle \mathcal{I}_k \rightarrow \mathcal{U}_k \rangle_{k \in K} \mid \Delta_t \quad \Gamma_u \Vdash u : \wedge_{k \in K} \mathcal{I}_k^* \mid \Delta_u}{\Gamma_t \wedge \Gamma_u \vdash t u : \vee_{k \in K} \mathcal{U}_k \mid \Delta_t \vee \Delta_u} \text{ app}
\end{array}$$

Figure 7.9: System $\mathcal{S}_{\lambda_\mu}$

types saved by α after reducing the subject. This is why we associated (Fig. 7.7) to a naming/saving rule the arity¹ of the saved type.

7.2.4 System $\mathcal{S}_{\lambda_\mu}$

This section presents a quantitative typing system characterizing strongly β -normalizing λ_μ -terms. The (syntax directed) typing rules of the typing system $\mathcal{S}_{\lambda_\mu}$ appear in Fig. 7.9. As in system \mathcal{S}'_λ , the operation $_*$ is used to choose arbitrary types for erasable terms, so that no subterm is untyped, thus ensuring strong λ_μ -normalization. Thus:

Lemma 7.1. If $\triangleright \Gamma \vdash t : \mathcal{U} \mid \Delta$, then $\mathcal{U} \neq \langle \rangle$.

As well as in the case of $\mathcal{H}_{\lambda_\mu}$, system $\mathcal{S}_{\lambda_\mu}$ can be restricted to the pure λ -calculus. Using the same observations at the end of Sec. 7.2.2, we obtain the typing system \mathcal{S}'_λ in Fig. 7.4 that characterizes β -strong normalization.

We also have a Relevance Lemma (extending Lemma 5.4), often used in proofs:

Lemma 7.2 (Relevance). Let $o \in \mathcal{O}_{\lambda_\mu}$. If $\Phi \triangleright \Gamma \vdash o : \mathcal{A} \mid \Delta$, then $\text{dom}(\Gamma) = \text{fv}(o)$ and $\text{dom}(\Delta) = \text{fn}(o)$.

Relevance holds thanks to the choice operator $_*$: indeed, if $\Delta(\alpha)^*$ is replaced by $\Delta(\alpha)$ in the **restore**-rule, then the following derivations gives a counter-example to the relevance property, where $\alpha \in \text{fn}([\alpha]\mu\beta.[\gamma]x)$ but $\alpha \notin \text{dom}(\gamma : \langle o \rangle)$.

$$\frac{\frac{\frac{x : [\langle o \rangle] \vdash x : \langle o \rangle \mid}{x : [\langle o \rangle] \vdash [\gamma]x : \# \mid \gamma : \langle o \rangle}}{x : [\langle o \rangle] \vdash \mu\beta.[\gamma]x : \langle \rangle \mid \gamma : \langle o \rangle}}{x : [\langle o \rangle] \vdash [\alpha]\mu\beta.[\gamma]x : \# \mid \gamma : \langle o \rangle}$$

¹Other choices could be made, such as the *size* (number of nodes) $|\mathcal{U}|$ of the saved type \mathcal{U} , inductively defined by $|o| = 1$, $|\mathcal{I} \rightarrow \mathcal{U}| = |\mathcal{I}| + |\mathcal{U}| + 1$, $|\mathcal{U}_i|_{i \in I} = +_{i \in I} |\mathcal{U}_i|$, $|\langle \tau_i \rangle_{i \in I}| = +_{i \in I} |\tau_i|$.

7.3 Typing Properties

This section shows two fundamental properties of reduction (*i.e. forward*) and anti-reduction (*i.e. backward*) of system $\mathcal{S}_{\lambda\mu}$. In Sec. 7.3.1, we analyse the *Weighted Subject Reduction* property, and we prove that reduction preserves typing and decreases the size of type derivations. The proof of this property makes use of two fundamental properties (Lemma 7.4 and 7.5) guaranteeing well-typedness of the meta-operations of substitution and replacement. Sec. 7.3.2 is devoted to *Subject Expansion*, which states that non-erasing anti-reduction preserves types. The proof uses the fact that reverse substitution (Lemma 7.7) and reverse replacement (Lemma 7.8) preserve types.

We start by stating an interesting property, to be used in our forthcoming lemmas, that allows us to split and merge auxiliary derivations typing the same term:

Lemma 7.3. Let $\mathcal{I} = \bigwedge_{k \in K} \mathcal{I}_k$. Then $\Phi \triangleright \Gamma \Vdash t : \mathcal{I} \mid \Delta$ iff $\exists(\Gamma_k)_{k \in K}, \exists(\Delta_k)_{k \in K}$ s.t. $(\Phi_k \triangleright \Gamma_k \Vdash t : \mathcal{I}_k \mid \Delta_k)_{k \in K}$, $\Gamma = \bigwedge_{k \in K} \Gamma_k$ and $\Delta = \bigvee_{k \in K} \Delta_k$. Moreover, $\mathbf{sz}(\Phi) = \sum_{k \in K} \mathbf{sz}(\Phi_k)$.

7.3.1 Forward Properties

We first state the substitution lemma, which guarantees that typing is stable by substitution. The lemma also establishes the size of the derivation tree of a substituted object from the sizes of the derivations trees of its components.

Lemma 7.4 (Substitution). Let $\Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$. If $\Phi_o \triangleright \Gamma; x : \mathcal{I} \vdash o : \mathcal{A} \mid \Delta$, then there is $\Phi_{o[u/x]}$ such that

- $\Phi_{o[u/x]} \triangleright \Gamma \wedge \Gamma_u \vdash o[u/x] : \mathcal{A} \mid \Delta \vee \Delta_u$.
- $\mathbf{sz}(\Phi_{o[u/x]}) = \mathbf{sz}(\Phi_o) + \mathbf{sz}(\Theta_u) - \#\mathcal{I}$.

Proof. We prove a more general statement, namely: Let $\Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$.

- If $\Phi_o \triangleright \Gamma; x : \mathcal{I} \vdash o : \mathcal{A} \mid \Delta$, then there is $\Phi_{o[u/x]}$ such that

$$\Phi_{o[u/x]} \triangleright \Gamma \wedge \Gamma_u \vdash o[u/x] : \mathcal{A} \mid \Delta \vee \Delta_u$$

- If $\Phi_o \triangleright \Gamma; x : \mathcal{I} \Vdash t : \mathcal{J} \mid \Delta$, then there is $\Phi_{o[u/x]}$ such that

$$\Phi_{o[u/x]} \triangleright \Gamma \wedge \Gamma_u \Vdash t[u/x] : \mathcal{J} \mid \Delta \vee \Delta_u$$

In both cases $\mathbf{sz}(\Phi_{o[u/x]}) = \mathbf{sz}(\Phi_o) + \mathbf{sz}(\Theta_u) - \#\mathcal{I}$.

We proceed by induction on the structure of Φ_o .

- **ax:**

- If $o = x$, then $\mathcal{I} := [\mathcal{U}]$ is a singleton, $\Gamma = \Delta = \emptyset$ and $o[u/x] = u$. The derivation Θ_u is necessarily of the following form

$$\frac{\Phi'_u \triangleright \Gamma' \vdash u : \mathcal{U} \mid \Delta'}{\Gamma' \Vdash u : [\mathcal{U}] \mid \Delta'} \wedge$$

We then set $\Phi_{x[u/x]} = \Phi'_u$. Then $\mathbf{sz}(\Phi_{x[u/x]}) = \mathbf{sz}(\Phi_x) + \mathbf{sz}(\Theta_u) - \#\mathcal{I}$, since $\mathbf{sz}(\Phi_x) = 1 = \#\mathcal{I}$ and $\mathbf{sz}(\Theta_u) = \mathbf{sz}(\Phi'_u)$.

- If $o = x \neq y$, then $\mathcal{I} = []$ and $o[u/x] = y$. Moreover, Θ_u is necessarily :

$$\overline{\emptyset \Vdash u : [] \mid \emptyset}^\wedge$$

We set $\Phi_{y[u/x]} = \Phi_o$. Then $\mathbf{sz}(\Phi_{y[u/x]}) = \mathbf{sz}(\Phi_y) + \mathbf{sz}(\Theta_u) - \#\mathcal{I}$ since $|\mathcal{I}| = 0$ and $\mathbf{sz}(\Theta_u) = 0$.

- **abs** : then $o = \lambda x.t$ and the derivation has the following form

$$\Phi_o = \frac{\Phi_t \triangleright \Gamma; x : \mathcal{I}; y : \mathcal{J} \vdash t : \mathcal{U}_t \mid \Delta}{\Gamma; x : \mathcal{I} \vdash \lambda y.t : \langle \mathcal{J} \rightarrow \mathcal{U}_t \rangle \mid \Delta}$$

By the *i.h.*, we have $\Phi_{t[u/x]} \triangleright (\Gamma; y : \mathcal{J}) \wedge \Gamma_u \vdash t[u/x] : \mathcal{U} \mid \Delta \vee \Delta_u$ with $\mathbf{sz}(\Phi_{t[u/x]}) = \mathbf{sz}(\Phi_t) + \mathbf{sz}(\Theta_u) - \#\mathcal{I}$. By α -conversion $y \notin \mathbf{fv}(u)$ so that $y \notin \mathbf{dom}(\Gamma_u)$ by Lemma 7.2, thus $(\Gamma; y : \mathcal{J}) \wedge \Gamma_u = (\Gamma \wedge \Gamma_u); y : \mathcal{J}$. We set then $\Phi_{(\lambda y.t)[u/x]} =$

$$\frac{\Phi_{t[u/x]}}{\Gamma \wedge \Gamma_u \vdash \lambda y.t[u/x] : \langle \mathcal{J} \rightarrow \mathcal{U}_t \rangle \mid \Delta \vee \Delta_u}$$

We have $\mathbf{sz}(\Phi_{(\lambda y.t)[u/x]}) = \mathbf{sz}(\Phi_{t[u/x]}) + 1 =_{i.h.} \mathbf{sz}(\Phi_t) + \mathbf{sz}(\Theta_u) - \#\mathcal{I} + 1 = \mathbf{sz}(\Phi) + \mathbf{sz}(\Theta_u) - \#\mathcal{I}$.

- \wedge : then o is a term t and Φ_o has the following form

$$\frac{(\triangleright \Gamma_k; x : \mathcal{I}_k \vdash t : \mathcal{U}_k \mid \Delta_k)_{k \in K}}{\Gamma; x : \mathcal{I} \Vdash t : [\mathcal{U}_k]_{k \in K} \mid \Delta}$$

where $\mathcal{I} = \wedge_{k \in K} \mathcal{I}_k$, $\Gamma = \wedge_{k \in K} \Gamma_k$ and $\Delta = \vee_{k \in K} \Delta_k$. By Lemma 7.3 there are auxiliary derivations $(\triangleright \Gamma_u^k \Vdash u : \mathcal{I}_k \mid \Delta_u^k)_{k \in K}$ such that $\Gamma_u = \wedge_{k \in K} \Gamma_u^k$ and $\Delta_u = \vee_{k \in K} \Delta_u^k$. The *i.h.* gives derivations $(\triangleright \Gamma_k \wedge \Gamma_u^k \vdash t[u/x] : \mathcal{U}_k \mid \Delta_k \wedge \Delta_u^k)_{k \in K}$ and we construct the following auxiliary derivation to conclude

$$\frac{(\triangleright \Gamma_k \wedge \Gamma_u^k \vdash t[u/x] : \mathcal{U}_k \mid \Delta_k \wedge \Delta_u^k)_{k \in K}}{\wedge_{k \in K} \Gamma_k \wedge \Gamma_u^k \Vdash t[u/x] : [\mathcal{U}_k]_{k \in K} \mid \vee_{k \in K} \Delta_k \wedge \Delta_u^k}$$

We have $\wedge_{k \in K} \Gamma_k \wedge \Gamma_u^k = \Gamma \wedge \Gamma_u$ and $\vee_{k \in K} \Delta_k \wedge \Delta_u^k = \Delta \vee \Delta_u$ as desired. The size statement trivially holds by the *i.h.*.

- **app** : then $o = t v$ and the derivation has the following form $\Phi =$

$$\frac{\Phi_t \triangleright \Gamma_t; x : \mathcal{I}_t \vdash t : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K} \mid \Delta_t \quad \Gamma_v; x : \mathcal{I}_v \Vdash v : \wedge_{k \in K} \mathcal{I}_k^* \mid \Delta_v}{\Gamma; x : \mathcal{I} \vdash t v : \vee_{k \in K} \mathcal{V}_k \mid \Delta}$$

where $\Gamma = \Gamma_t \wedge \Gamma_v$, $\Delta = \Delta_t \vee \Delta_v$ and $\mathcal{I} = \mathcal{I}_t \wedge \mathcal{I}_v$.

Moreover, by Lemma 7.3 we can split Θ_u in $\Theta_u^t \triangleright \Gamma_u^t \Vdash u : \mathcal{I}_t \mid \Delta_u^t$ and $\Theta_u^v \triangleright \Gamma_u^v \Vdash u : \mathcal{I}_v \mid \Delta_u^v$ s.t. $\mathbf{sz}(\Theta_u) = \mathbf{sz}(\Theta_u^t) + \mathbf{sz}(\Theta_u^v)$.

By the *i.h.*, there is $\Phi_{t[u/x]} \triangleright \Gamma_t' \vdash t[u/x] : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K} \mid \Delta_t'$, where $\Gamma_t' = \Gamma_t \wedge \Gamma_u^t$ and $\Delta_t' = \Delta_t \vee \Delta_u^t$ and $\mathbf{sz}(\Phi_{t[u/x]}) = \mathbf{sz}(\Phi_t) + \mathbf{sz}(\Theta_u) - \#\mathcal{I}_t$.

Also by the *i.h.*, there is $\Phi_{v[u/x]} \triangleright \Gamma_v' \Vdash v[u/x] : \wedge_{k \in K} \mathcal{I}_k^* \mid \Delta_v'$, where $\Gamma_v' = \Gamma_v \wedge \Gamma_u^v$ and $\Delta_v' = \Delta_v \vee \Delta_u^v$ and $\mathbf{sz}(\Phi_{v[u/x]}) = \mathbf{sz}(\Phi_v) + \mathbf{sz}(\Theta_u^v) - \#\mathcal{I}_v$.

We set then

$$\Phi_{o[u/x]} = \frac{\Phi_{t[u/x]} \quad \Phi_{v[u/x]}}{\Gamma' \vdash (tv)[u/x] : \bigvee_{k \in K} \mathcal{V}_k \mid \Delta'}$$

where $\Gamma' = (\Gamma_t \wedge \Gamma_u^t) \wedge (\Gamma_v \wedge \Gamma_u^v) = \Gamma \wedge \Gamma_u$ and $\Delta' = (\Delta_t \vee \Delta_u^t) \vee (\Delta_v \vee \Delta_u^v) = \Delta \vee \Delta_u$ as desired.

We conclude since

$$\begin{aligned} \mathbf{sz}(\Phi_{o[u/x]}) &= \mathbf{sz}(\Phi_{t[u/x]}) + \mathbf{sz}(\Phi_{v[u/x]}) + |K| \\ &=_{i.h.} (\mathbf{sz}(\Phi_t) + \mathbf{sz}(\Theta_u^t) - |\mathcal{I}_t|) + \\ &\quad (\mathbf{sz}(\Phi_v) + \mathbf{sz}(\Theta_u^v) - |\mathcal{I}_v|) + |K| \\ &= \mathbf{sz}(\Phi) + \mathbf{sz}(\Theta_u) - |\mathcal{I}| \end{aligned}$$

- All the other cases are straightforward. □

Typing is also stable by replacement. Moreover, we can specify the exact size of the derivation tree of the replaced object from the sizes of its components.

Lemma 7.5 (Replacement). Let $\Theta_u \triangleright \Gamma_u \Vdash u : \bigwedge_{k \in K} \mathcal{I}_k^* \mid \Delta_u$ where $\alpha \notin \mathbf{fn}(u)$. If $\Phi_o \triangleright \Gamma \vdash o : \mathcal{A} \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta$, then there is $\Phi_{o\{u//\alpha\}}$ such that :

- $\Phi_{o\{u//\alpha\}} \triangleright \Gamma \wedge \Gamma_u \vdash o\{u//\alpha\} : \mathcal{A} \mid \alpha : \bigvee_{k \in K} \mathcal{V}_k; \Delta \vee \Delta_u$.
- $\mathbf{sz}(\Phi_{o\{u//\alpha\}}) = \mathbf{sz}(\Phi_o) + \mathbf{sz}(\Theta_u)$.

Proof. We prove a more general statement, namely:

Let $\Theta_u \triangleright \Gamma_u \Vdash u : \bigwedge_{k \in K} \mathcal{I}_k^* \mid \Delta_u$ where $\alpha \notin \mathbf{fn}(u)$.

- If $\Phi_o \triangleright \Gamma \vdash o : \mathcal{A} \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta$, then there is $\Phi_{o\{u//\alpha\}}$ such that $\Phi_{o\{u//\alpha\}} \triangleright \Gamma \wedge \Gamma_u \vdash o\{u//\alpha\} : \mathcal{A} \mid \alpha : \bigvee_{k \in K} \mathcal{V}_k; \Delta \vee \Delta_u$.
- If $\Phi_o \triangleright \Gamma \Vdash o : \mathcal{J} \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta$, then there is $\Phi_{o\{u//\alpha\}}$ such that $\Phi_{o\{u//\alpha\}} \triangleright \Gamma \wedge \Gamma_u \Vdash t\{u//\alpha\} : \mathcal{J} \mid \alpha : \bigvee_{k \in K} \mathcal{V}_k; \Delta \vee \Delta_u$

In both cases, $\mathbf{sz}(\Phi_{o\{u//\alpha\}}) = \mathbf{sz}(\Phi_o) + \mathbf{sz}(\Theta_u)$.

We reason by induction on Φ_o . Let us call $\mathcal{U}_\alpha = \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}$ and $\mathcal{U}'_\alpha = \bigvee_{k \in K} \mathcal{V}_k$.

- **ax:** $o = x$, thus we have by construction

$$\Phi_o \triangleright x : [\mathcal{U}] \vdash x : \mathcal{U} \mid$$

so that $K = \emptyset$. Thus, $\bigwedge_{k \in K} \mathcal{I}_k^* = []$ and $\Gamma_u = \Delta_u = \emptyset$, then Θ_u is :

$$\frac{}{\Vdash u : [] \mid} \wedge$$

Thus, $\mathbf{sz}(\Theta_u) = 0$.

We set $\Phi_{o\{u//\alpha\}} = \Phi_o$ and the first result holds because the derivation has the desired form. We conclude since $\mathbf{sz}(\Phi_{o\{u//\alpha\}}) = \mathbf{sz}(\Phi_o) + \mathbf{sz}(\Theta_u)$ as desired.

- **abs**: then $o = \lambda x.t$, $o\{u//\alpha\} = \lambda x.(t\{u//\alpha\})$ and by construction we have

$$\Phi_{\lambda x.t} \triangleright \frac{\Phi_t \triangleright x : \mathcal{I}; \Gamma \vdash t : \mathcal{U} \mid \alpha : \mathcal{U}_\alpha; \Delta}{\Gamma \vdash \lambda x.t : \langle \mathcal{I} \rightarrow \mathcal{U} \rangle \mid \alpha : \mathcal{U}_\alpha; \Delta} \text{ abs}$$

By *i.h.*, it follows that

$$\Phi_{t\{u//\alpha\}} \triangleright (x : \mathcal{I}; \Gamma) \wedge \Gamma_u \vdash t\{u//\alpha\} : \mathcal{U} \mid \alpha : \mathcal{U}'_\alpha; \Delta \vee \Delta_u$$

with $\text{sz}(\Phi_{t\{u//\alpha\}}) = \text{sz}(\Phi_t) + \text{sz}(\Theta_u)$. By α -conversion we can assume that $x \notin \text{fv}(u)$, thus by Lemma 7.2 $x \notin \text{dom}(\Gamma_u)$, so that $(x : \mathcal{I}; \Gamma) \wedge \Gamma_u = x : \mathcal{I}; \Gamma \wedge \Gamma_u$.

We thus obtain $\Phi_{\lambda x.t\{u//\alpha\}}$ of the form:

$$\frac{\Phi_{t\{u//\alpha\}}}{\Gamma \wedge \Gamma_u \vdash \lambda x.t\{u//\alpha\} : \langle \mathcal{I} \rightarrow \mathcal{U} \rangle \mid \alpha : \mathcal{U}'_\alpha; \Delta \vee \Delta_u} \text{ abs}$$

We conclude since

$$\begin{aligned} \text{sz}(\Phi_{\lambda x.t\{u//\alpha\}}) &= \text{sz}(\Phi_{t\{u//\alpha\}}) + 1 \\ &=_{i.h.} \text{sz}(\Phi_t) + \text{sz}(\Theta_u) + 1 \\ &= \text{sz}(\Phi_{\lambda x.t}) + \text{sz}(\Theta_u) \end{aligned}$$

- **app**: then $o = tv$, $o\{u//\alpha\} = t\{u//\alpha\}v\{u//\alpha\}$ and by construction we have $\Phi_o =$

$$\frac{\Phi_t \triangleright \Gamma_t \vdash t : \mathcal{U}_t \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K_t}; \Delta_t \quad \Phi_v \triangleright \Gamma_v \Vdash v : \mathcal{J}_v \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K_v}; \Delta_v}{\Gamma \vdash o : \mathcal{U} \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta}$$

where $\mathcal{U}_t = \langle \mathcal{J}_\ell \rightarrow \mathcal{U}_\ell \rangle_{\ell \in L}$, $\mathcal{J}_v = \wedge_{\ell \in L} \mathcal{J}_\ell^*$, $\mathcal{U} = \vee_{\ell \in L} \mathcal{U}_\ell$ (those types are of no matter here, except they satisfy the typing constraint of **app**), $\Gamma = \Gamma_t \wedge \Gamma_v$, $\Delta = \Delta_t \vee \Delta_v$, $K = K_t \uplus K_v$.

Moreover, by Lemma 7.3, we can split Θ_u in $\Theta_u^t \triangleright \Gamma_u^t \Vdash u : \wedge_{k \in K_t} \mathcal{I}_k^* \mid \Delta_u^t$ and $\Theta_u^v \triangleright \Gamma_u^v \Vdash u : \wedge_{k \in K_v} \mathcal{I}_k^* \mid \Delta_u^v$ s.t. $\text{sz}(\Theta_u) = \text{sz}(\Theta_u^t) + \text{sz}(\Theta_u^v)$.

By *i.h.*, we have $\Phi_{t\{u//\alpha\}} \triangleright \Gamma_t \wedge \Gamma_u^t \vdash t\{u//\alpha\} : \mathcal{U}_t \mid \alpha : \vee_{k \in K_t} \mathcal{V}_k; \Delta_t \vee \Delta_u^t$ (since $\alpha \notin \text{fn}(u)$) with $\text{sz}(\Phi_{t\{u//\alpha\}}) = \text{sz}(\Phi_t) + \text{sz}(\Theta_u^t)$.

Also by *i.h.*, we have $\Phi_{v\{u//\alpha\}} \triangleright \Gamma_v \wedge \Gamma_u^v \Vdash v\{u//\alpha\} : \mathcal{J}_v \mid \alpha : \vee_{k \in K_v} \mathcal{V}_k, \Delta_v \vee \Delta_u^v$ with $\text{sz}(\Phi_{v\{u//\alpha\}}) = \text{sz}(\Phi_v) + \text{sz}(\Theta_u^v)$.

We can now construct the following derivation

$$\frac{\Phi_{t\{u//\alpha\}} \quad \Phi_{v\{u//\alpha\}}}{\Gamma' \vdash o\{u//\alpha\} : \mathcal{U} \mid \alpha : \vee_{k \in K} \mathcal{V}_k; \Delta'}$$

where $\Gamma' = (\Gamma_t \wedge \Gamma_u^t) \wedge (\Gamma_v \wedge \Gamma_u^v) = (\Gamma_t \wedge \Gamma_v) \wedge (\Gamma_u^t \wedge \Gamma_u^v) = \Gamma \wedge \Gamma_u$ and likewise, $\Delta' = \Delta \vee \Delta_u$ as desired.

Moreover,

$$\begin{aligned} \text{sz}(\Phi_{(tv)\{u//\alpha\}}) &= \text{sz}(\Phi_{t\{u//\alpha\}}) + \text{sz}(\Phi_{v\{u//\alpha\}}) + |L| \\ &=_{i.h.} (\text{sz}(\Phi_t) + \text{sz}(\Theta_u^t)) + (\text{sz}(\Phi_v) + \text{sz}(\Theta_u^v)) + |L| \\ &= (\text{sz}(\Phi_t) + \text{sz}(\Phi_v) + |L|) + (\text{sz}(\Theta_u^t) + \text{sz}(\Theta_u^v)) \\ &= \text{sz}(\Phi_{tv}) + \text{sz}(\Theta_u) \end{aligned}$$

- If $o = [\alpha]t$, then $o\{u//\alpha\} = [\alpha]t\{u//\alpha\}$ and by construction we have a derivation $\Phi_{[\alpha]t}$ of the form:

$$\frac{\Phi_t \triangleright \Gamma \vdash t : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K_t} \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K_\alpha}; \Delta}{\Gamma \vdash [\alpha]t : \# \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta} \text{ name}$$

where $K = K_t \uplus K_\alpha$.

Moreover, by Lemma 7.3, we can split Θ_u in $\Theta_u^t \triangleright \Gamma_u^t \Vdash u : \bigwedge_{k \in K_t} \mathcal{I}_k^* \mid \Delta_u^t$ and $\Theta_u^\alpha \triangleright \Gamma_u^\alpha \Vdash u : \bigwedge_{k \in K_\alpha} \mathcal{I}_k^* \mid \Delta_u^\alpha$ s.t. $\mathbf{sz}(\Theta_u) = \mathbf{sz}(\Theta_u^t) + \mathbf{sz}(\Theta_u^\alpha)$.

By the *i.h.*, we have $\Phi_{t\{u//\alpha\}} \triangleright \Gamma_0 \vdash t\{u//\alpha\} : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K_t} \mid \alpha : \bigvee_{k \in K_\alpha} \mathcal{V}_k; \Delta_0$ with $\Gamma_0 = \Gamma \wedge \Gamma_u^\alpha$, $\Delta_0 = \Delta \vee \Delta_u^\alpha$ and $\mathbf{sz}(\Phi_{t\{u//\alpha\}}) = \mathbf{sz}(\Phi_t) + \mathbf{sz}(\Theta_u^\alpha)$.

We can then construct the following derivation $\Phi_{[\alpha]t\{u//\alpha\}}$:

$$\frac{\frac{\Phi_{t\{u//\alpha\}} \quad \Theta_u^t}{\Gamma' \vdash t\{u//\alpha\}u : \bigvee_{k \in K_t} \mathcal{V}_k \mid \alpha : \bigvee_{k \in K_\alpha} \mathcal{V}_k; \Delta'}}{\Gamma' \vdash [\alpha]t\{u//\alpha\}u : \# \mid \alpha : \bigvee_{k \in K} \mathcal{V}_k; \Delta'}}$$

with $\Gamma' = \Gamma_0 \wedge \Gamma_u^t = \Gamma \wedge \Gamma_u^\alpha \wedge \Gamma_u^t = \Gamma \wedge \Gamma_u$ and likewise $\Delta' = \Delta \vee \Delta_u$ (since $\alpha \notin \mathbf{fn}(u)$) as expected.

We conclude since

$$\begin{aligned} \mathbf{sz}(\Phi_{[\alpha]t\{u//\alpha\}}) &= \mathbf{sz}(\Phi_{t\{u//\alpha\}}) + \mathbf{ar}(\bigvee_{k \in K_t} \mathcal{V}_k) \\ &= \mathbf{sz}(\Phi_{t\{u//\alpha\}}) + \mathbf{sz}(\Theta_u^t) + |K_t| + \mathbf{ar}(\bigvee_{k \in K_t} \mathcal{V}_k) \\ &= \text{i.h.} (\mathbf{sz}(\Phi_t) + \mathbf{sz}(\Theta_u^\alpha)) + \mathbf{sz}(\Theta_u^t) + \mathbf{ar}(\langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K_t}) \\ &= \mathbf{sz}(\Phi_t) + \mathbf{sz}(\Theta_u) + \mathbf{ar}(\langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K_t}) \\ &= \mathbf{sz}(\Phi_{[\alpha]t}) + \mathbf{sz}(\Theta_u) \end{aligned}$$

- All the other cases are straightforward. □

Notice that the type of α in the conclusion of the derivation $\Phi_{o\{u//\alpha\}}$ (which is $\bigvee_{k \in K} \mathcal{V}_k$) is strictly smaller than that of the conclusion of the derivation Φ_o (which is $\langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}$). The substitution and the replacement Lemmas are used in the proof of Weighted Subject Reduction for $\mathcal{S}_{\lambda_\mu}$ (extending Proposition 5.1, since System $\mathcal{S}_{\lambda_\mu}$ extends System \mathcal{S}_λ).

Property 7.1 (Weighted Subject Reduction for μ). Let $\Phi \triangleright \Gamma \vdash o : \mathcal{A} \mid \Delta$. If $o \rightarrow o'$ is a non-erasing step, then there exists a derivation $\Phi' \triangleright \Gamma \vdash o' : \mathcal{A} \mid \Delta$ such that $\mathbf{sz}(\Phi) > \mathbf{sz}(\Phi')$.

Proof. By induction on the relation \rightarrow . We only show the main cases of reduction at the root, the other ones being straightforward.

- If $o = (\lambda x.t)u$, then $o' = t[u/x]$ and $x \in \mathbf{fv}(t)$. The derivation Φ has the following form:

$$\Phi = \frac{\frac{\Phi_t \triangleright \Gamma_t; x : \mathcal{I} \vdash t : \mathcal{U} \mid \Delta_t}{\Gamma_t \vdash \lambda x.t : \langle \mathcal{I} \rightarrow \mathcal{U} \rangle \mid \Delta_t} \quad \Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I}^* \mid \Delta_u}{\Gamma \vdash o : \mathcal{U} \mid \Delta}$$

where $\Gamma = \Gamma_t \wedge \Gamma_u$, $\Delta = \Delta_t \vee \Delta_u$. Indeed, $x \in \text{fv}(t)$ implies by Lemma 7.2 that $\mathcal{I} \neq []$ so that $\mathcal{I}^* = \mathcal{I} = [\mathcal{U}_k]_{k \in K}$ for some $K \neq \emptyset$ and some $(\mathcal{U}_k)_{k \in K}$.

Lemma 7.4 yields a derivation $\Phi'_{t[u/x]} \triangleright \Gamma_t \wedge \Gamma_u \vdash t[u/x] : \mathcal{U} \mid \Delta_t \vee \Delta_u$ with $\text{sz}(\Phi'_{t[u/x]}) = \text{sz}(\Phi_t) + \text{sz}(\Theta_u) - |K|$ ($|\mathcal{I}| = |K|$). We set $\Phi' = \Phi'_{t[u/x]}$ so that $\text{sz}(\Phi) = \text{sz}(\Phi_t) + 1 + \text{sz}(\Theta_u) + 1 > \text{sz}(\Phi')$.

- If $o = (\mu\alpha.c)u$, then $o' = \mu\alpha.c\{u//\alpha\}$ and $\alpha \in \text{fn}(c)$.

The derivation Φ has the following form:

$$\Phi = \frac{\frac{\Phi_c \triangleright \Gamma_c \vdash c : \# \mid \alpha : \mathcal{V}_c; \Delta_c}{\Gamma_c \vdash \mu\alpha.c : \mathcal{V}_c \mid \Delta_c} \quad \Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I}_u \mid \Delta_u}{\Gamma_c \wedge \Gamma_u \vdash (\mu\alpha.c)u : \mathcal{U} \mid \Delta_c \vee \Delta_u}$$

where $\mathcal{V}_c = \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}$, $\mathcal{I}_u = \wedge_{k \in K} \mathcal{I}_k^*$, $\mathcal{U} = \vee_{k \in K} \mathcal{V}_k$, $\Gamma = \Gamma_c \wedge \Gamma_u$ and $\Delta = \Delta_c \vee \Delta_u$. Lemma 7.5 then gives the derivation $\Phi_{c\{u//\alpha\}} \triangleright \Gamma_c \wedge \Gamma_u \vdash c\{u//\alpha\} : \# \mid \alpha : \vee_{k \in K} \mathcal{V}_k; \Delta_c \vee \Delta_u$. Since $\alpha \in \text{fn}(c)$ by hypothesis, then $K \neq \emptyset$ by Lemma 7.2 so that we construct the following derivation:

$$\phi' \triangleright \frac{\Phi_{c\{u//\alpha\}}}{\Gamma_c \wedge \Gamma_u \vdash \mu\alpha.c\{u//\alpha\} : \vee_{k \in K} \mathcal{V}_k \mid \Delta_c \vee \Delta_u}$$

We conclude since

$$\begin{aligned} \text{sz}(\Phi') &= \text{sz}(\phi_{c\{u//\alpha\}}) + 1 \\ &=_{L. 7.5} \text{sz}(\Phi_c) + \text{sz}(\Theta_u) + 1 \\ &< \text{sz}(\Phi_c) + 1 + \text{sz}(\Theta_u) + |K| \\ &= \text{sz}(\Phi_{\mu\alpha.c}) + \text{sz}(\Theta_u) + |K| = \text{sz}(\Phi) \end{aligned}$$

The step $<$ is justified by $K \neq \emptyset$.

□

Actually, for non-erasing reduction steps, Fig. 7.8 is still valid, which intuitively explains (without the painful inductions above) why Property 7.1 holds.

Discussion A first remark about the property above is that the whole discussion concluding Sec. 5.2.2 is still valid for $\mathcal{S}_{\lambda_\mu}$ and that variable and name assignments are not necessarily preserved by erasing reductions. Thus, for example, consider $t = (\lambda y.x)z \rightarrow x = t'$. The term t is typed with a variable assignment whose domain is $\{x, z\}$, while t' can only be typed with an assignment whose domain is $\{x\}$. Concretely, starting from a derivation of $x : [\langle a \rangle], z : [\langle b \rangle] \vdash (\lambda x.y)z : \langle a \rangle$ (the simplified type derivation of this term in the \mathcal{S}'_λ system appears on page 146), we can only construct a derivation of $x : [\langle a \rangle] \vdash x : \langle a \rangle$, so that the type is preserved while the variable assignment is not. Actually, our restricted form of subject reduction (*i.e.* for non-erasing steps only) is sufficient for our purpose, along with an equivalent for $\mathcal{S}_{\lambda_\mu}$ of Lemma 5.5, dealing with erasing head-reduction steps. This result is easy to prove and stated below:

Lemma 7.6. If $\Pi \triangleright_{\mathcal{S}_{\lambda\mu}} \Gamma \vdash t : \mathcal{U} \mid \Delta$ and $t = (\lambda x.r)st_1 \dots t_q \rightarrow rt_1 \dots t_q = t'$ or $t = (\mu\alpha.c)st_1 \dots t_q \rightarrow \mu\alpha.ct_1 \dots t_q$ is an erasing reduction step (i.e. $x \notin \text{fv}(r)$ or $\alpha \notin \text{fn}(c)$ respectively), then there is a $\mathcal{S}_{\lambda\mu}$ -derivation Π , a variable context Γ' and a name context Δ' such that $\Pi' \triangleright_{\mathcal{S}_{\lambda\mu}} \Gamma' \vdash t' : \mathcal{U} \mid \Delta'$ and $\text{sz}(\Pi') < \text{sz}(\Pi)$.

Note that, as in Sec. 5.2.2, *at this stage*, we cannot prove the implication “if $o \rightarrow o'$ and o is typable, then so is o' ”: for now, we have this implication for non-erasing steps and for erasing head reduction steps only. See how we manage to get from this the proof of Lemma 7.11.

A second remark is that the consideration of arities of names in the definition of the size of derivations (third case **name**) is crucial to guarantee that μ -reduction decreases $\text{sz}(_)$, a matter that was already (informally) addressed in the conclusion of Sec. 7.2.3. This is perfectly reflected in Lemma 7.5, where the type of α in the conclusion of the derivation $\Phi_{o\{u//\alpha\}}$ is strictly smaller than that of the conclusion of the derivation Φ_o .

Furthering the discussion of page 7.2.2, a third point is again about the use of the choice operator in the typing rule **restore**, which does not allow for the type $\langle \rangle$ to be assigned to α when $\alpha \notin \text{fn}(c)$. More precisely, assume, just temporarily, that the **restore** rule does not use the choice operator, so that a μ -abstraction can be typed with $\langle \rangle$. Set $u := \mu\beta.[\gamma]y$ and $c := [\alpha]\mu\delta.[\alpha]u$ so that u , $\mu\delta.[\alpha]u$ and $\mu\alpha.c$ are typed with $\langle \rangle$. The resulting type derivation $\Phi_c \triangleright \Gamma \vdash c : \# \mid \Delta$ contradicts the Relevance Lemma 7.2, simply because $\alpha \notin \text{fn}(\Delta)$ but α has two free occurrences in c . This has heavy consequences that can be illustrated by the reduction sequence $t = (\mu\alpha.c)x \rightarrow \mu\alpha.[\alpha](\mu\delta.[\alpha](\mu\beta.[\gamma]y)x)x \rightarrow^* \mu\alpha.c = t'$. Indeed, the type of $\mu\alpha.c$, which is $\langle \rangle$, holds no information capturing the number of free occurrences of α in c , so that there is no local way to know how many times the argument x should be typed in the whole derivation of the term $(\mu\alpha.c)x$. This prevents the reduction relation to decrease any reasonable measure associated to type derivations.

7.3.2 Backward Properties

Subject expansion is based on two technical properties: the first one, called reverse substitution, allows us to extract type information for an object o and a term u from the type derivation of $o[u/x]$; similarly, the second one, called reverse replacement, gives type information for a command c and a term u from the type derivation of $c\{u//\alpha\}$. Formally,

Lemma 7.7 (Reverse Substitution). Let $\Phi' \triangleright \Gamma' \vdash o[u/x] : \mathcal{A} \mid \Delta'$ Then $\exists \Gamma, \exists \Delta, \exists \mathcal{I}, \exists \Gamma_u, \exists \Delta_u$ such that:

- $\Gamma' = \Gamma \wedge \Gamma_u$,
- $\Delta' = \Delta \vee \Delta_u$,
- $\triangleright \Gamma; x : \mathcal{I} \vdash o : \mathcal{A} \mid \Delta$
- $\triangleright \Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$.

Proof. We prove a more general statement, namely:

- If $\Phi' \triangleright \Gamma' \vdash o[u/x] : \mathcal{A} \mid \Delta'$, then $\triangleright \Gamma; x : \mathcal{I} \vdash o : \mathcal{A} \mid \Delta$, $\Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$, $\triangleright \Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$, $\Gamma' = \Gamma \wedge \Gamma_u$, $\Delta' = \Delta \vee \Delta_u$ for some \mathcal{I} , Γ , Γ_u , Δ , Δ_u .

- If $\Phi' \triangleright \Gamma' \Vdash t[u/x] : \mathcal{J} \mid \Delta'$, then $\triangleright \Gamma; x : \mathcal{I} \Vdash t : \mathcal{J} \mid \Delta$, $\Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$, $\triangleright \Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$, $\Gamma' = \Gamma \wedge \Gamma_u$, $\Delta' = \Delta \vee \Delta_u$ for some \mathcal{I} , Γ , Γ_u , Δ , Δ_u .

We proceed by induction on the structure of Φ' .

- **ax:**

- If $o = y \neq x$, then $y[u/x] = y$. By construction, one has that $\Gamma' = y : [\mathcal{U}]$ and $\mathcal{A} = \mathcal{U}$. The result thus holds for $\mathcal{I} = []$, $\Gamma = \Gamma'$, $\Delta = \Delta'$, $\Gamma_u = \emptyset$ and $\Delta_u = \emptyset$ as $\Vdash u : [] \mid$ is derivable by the \wedge rule.
- **ax:** If $o = x$, then $x[u/x] = u$. By construction, one has that $\mathcal{A} = \mathcal{U}$. We type x with the axiom rule:

$$\frac{}{x : [\mathcal{U}] \Vdash x : \mathcal{U} \mid}$$

so that the property holds for $\Gamma = \Delta = \emptyset$, $\mathcal{I} = [\mathcal{U}]$, $\Gamma_u = \Gamma'$, $\Delta_u = \Delta'$, where $\triangleright \Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$ is obtained by the rule \wedge from $\Gamma' \Vdash u : \mathcal{U} \mid \Delta'$.

- **abs:** $o = \lambda y.t$ and $(\lambda y.t)[u/x] = \lambda y.t[u/x]$. Then Φ' is of the form

$$\frac{\Phi'_t \triangleright \Gamma'; y : \mathcal{J} \Vdash t[u/x] : \mathcal{V} \mid \Delta'}{\Gamma' \Vdash \lambda y.t[u/x] : \langle \mathcal{J} \rightarrow \mathcal{V} \rangle \mid \Delta'}$$

where $\mathcal{U} = \langle \mathcal{J} \rightarrow \mathcal{V} \rangle$.

By the *i.h.* $\Gamma'; y : \mathcal{I} = \Gamma_t \wedge \Gamma_u$ and $\Delta' = \Delta \vee \Delta_u$, $\triangleright \Gamma_t; x : \mathcal{I} \Vdash t : \mathcal{V} \mid \Delta$ and $\triangleright \Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$. By α -conversion we can assume that $y \notin \text{fv}(u)$, so that $y \notin \text{dom}(\Gamma_u)$ by Lemma 7.2 and thus $\Gamma_t = \Gamma; y : \mathcal{J}$ and $\Gamma' = \Gamma \wedge \Gamma_u$. Hence, we obtain $\Gamma; x : \mathcal{I} \Vdash \lambda y.t : \mathcal{U} \mid \Delta$ by the rule **abs**.

- **app:** $o = tv$ and $(tv)[u/x] = t[u/x]v[u/x]$. By construction, we have that $\Gamma' = \Gamma'_t \wedge \Gamma'_v$ and $\Delta' = \Delta'_t \vee \Delta'_v$ and $\triangleright \Gamma'_t \Vdash t[u/x] : \mathcal{U}_t \mid \Delta'_t$, $\triangleright \Gamma'_v \Vdash v : \mathcal{J}_v \mid \Delta'_v$ with $\mathcal{U}_t = \langle \mathcal{J}_k \rightarrow \mathcal{U}_k \rangle_{k \in K}$, $\mathcal{J}_v = \wedge_{k \in K} \mathcal{J}_k^*$ (those types are of no matter here, except they satisfy the typing constraint of **app**). By the *i.h.* there are:

- $\Gamma_t, \mathcal{I}_t, \Delta_t, \Gamma_u^t, \Delta_u^t$ s.t. $\Gamma'_t = \Gamma_t \wedge \Gamma_u^t$, $\Delta'_t = \Delta_t \vee \Delta_u^t$, $\triangleright \Gamma_t; x : \mathcal{I}_t \Vdash t : \mathcal{U}_t \mid \Delta_t$ and $\triangleright \Gamma_u^t \Vdash u : \mathcal{I}_t \mid \Delta_u^t$.
- $\Gamma_v, \mathcal{I}_v, \Delta_v, \Gamma_u^v, \Delta_u^v$ s.t. $\Gamma'_v = \Gamma_v \wedge \Gamma_u^v$, $\Delta'_v = \Delta_v \vee \Delta_u^v$, $\triangleright \Gamma_v; x : \mathcal{I}_v \Vdash v : \mathcal{J}_v \mid \Delta_v$ and $\triangleright \Gamma_u^v \Vdash u : \mathcal{I}_v \mid \Delta_u^v$.

Thus, we can type tv with :

$$\frac{\triangleright \Gamma_t; x : \mathcal{I}_t \Vdash t : \mathcal{U}_t \mid \Delta_t \quad \triangleright \Gamma_v; x : \mathcal{I}_v \Vdash v : \mathcal{J}_v \mid \Delta_v}{\Gamma; x : \mathcal{I} \Vdash tv : \mathcal{U} \mid \Delta}$$

where $\Gamma = \Gamma_t \wedge \Gamma_u$, $\Delta = \Delta_t \vee \Delta_u$, $\mathcal{I} = \mathcal{I}_t \vee \mathcal{I}_v$.

We obtain $\triangleright \Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$ with $\Gamma_u = \Gamma_u^t \wedge \Gamma_u^v$, $\Delta_u = \Delta_u^t \vee \Delta_u^v$ by Lemma 7.3.

- The other cases are similar.

□

Lemma 7.8 (Reverse Replacement). Let $\Phi' \triangleright \Gamma' \vdash o\{u//\alpha\} : \mathcal{A} \mid \alpha : \mathcal{V}; \Delta'$, where $\alpha \notin \text{fn}(u)$. Then $\exists \Gamma, \exists \Delta, \exists \Gamma_u, \exists \Delta_u, \exists (\mathcal{I}_k)_{k \in K}, \exists (\mathcal{V}_k)_{k \in K}$ such that:

- $\Gamma' = \Gamma \wedge \Gamma_u$,
- $\Delta' = \Delta \vee \Delta_u$,
- $\mathcal{V} = \bigvee_{k \in K} \mathcal{V}_k$,
- $\triangleright \Gamma \vdash o : \mathcal{A} \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta$, and
- $\triangleright \Gamma_u \Vdash u : \bigwedge_{k \in K} \mathcal{I}_k^* \mid \Delta_u$

Proof. We prove a more general statement, namely :

- If $\Phi' \triangleright \Gamma' \vdash o\{u//\alpha\} : \mathcal{A} \mid \alpha : \mathcal{V}; \Delta'$, then $\triangleright \Gamma \vdash o : \mathcal{A} \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta$, $\triangleright \Gamma_u \Vdash u : \bigwedge_{k \in K} \mathcal{I}_k^* \mid \Delta_u$ where $\Gamma' = \Gamma \wedge \Gamma_u$, $\Delta' = \Delta \vee \Delta_u$, $\mathcal{V} = \bigvee_{k \in K} \mathcal{V}_k$ for some $\Gamma, \Gamma_u, \Delta, \Delta_u, (\mathcal{V}_k)_{k \in K}, (\mathcal{I}_k)_{k \in K}$.
- If $\Phi' \triangleright \Gamma' \Vdash t\{u//\alpha\} : \mathcal{J} \mid \alpha : \mathcal{V}; \Delta'$, then $\triangleright \Gamma \Vdash t : \mathcal{J} \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta$, $\triangleright \Gamma_u \Vdash u : \bigwedge_{k \in K} \mathcal{I}_k^* \mid \Delta_u$ where $\Gamma' = \Gamma \wedge \Gamma_u$, $\Delta' = \Delta \vee \Delta_u$, $\mathcal{V} = \bigvee_{k \in K} \mathcal{V}_k$ for some $\Gamma, \Gamma_u, \Delta, \Delta_u, (\mathcal{V}_k)_{k \in K}, (\mathcal{I}_k)_{k \in K}$.

We proceed by induction on the structure of Φ' .

- **ax:** $o = x$ and $o\{u//\alpha\} = x$. Then Φ' is of the form $x : [\mathcal{U}] \vdash x : \mathcal{U} \mid$ and we have $\mathcal{V} = \langle \rangle$ so that we set $\Gamma = x : [\mathcal{U}], \Gamma_u = \Delta_u = \Delta = \emptyset, K = \emptyset$. Notice that $\Vdash u : [] \mid$ always holds.
- **abs:** $o = \lambda y.t$ and $(\lambda y.t)\{u//\alpha\} = \lambda y.t\{u//\alpha\}$. Then Φ' is of the form

$$\frac{\Gamma'; y : \mathcal{J} \vdash t\{u//\alpha\} : \mathcal{U}_t \mid \alpha : \mathcal{V}; \Delta'}{\Gamma' \vdash \lambda y.t\{u//\alpha\} : \langle \mathcal{I} \rightarrow \mathcal{U}_t \rangle \mid \alpha : \mathcal{V}; \Delta'}$$

The *i.h.* gives $\Gamma'; y : \mathcal{J} = \Gamma_t \wedge \Gamma_u, \mathcal{V} = \bigvee_{k \in K} \mathcal{V}_k, \Delta' = \Delta \vee \Delta_u, \triangleright \Gamma_t \vdash t : \mathcal{U}_t \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta$ and $\triangleright \Gamma_u \Vdash u : \bigwedge_{k \in K} \mathcal{I}_k^* \mid \Delta_u$. By α -conversion we can assume that $y \notin \text{fv}(u)$, so that $y \notin \text{dom}(\Gamma_u)$ holds by Lemma 7.2 and thus $\Gamma_t = \Gamma; y : \mathcal{J}$. Hence, we obtain

$$\frac{\triangleright \Gamma; y : \mathcal{J} \vdash t : \mathcal{U}_t \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta}{\triangleright \Gamma \vdash \lambda y.t : \langle \mathcal{J} \rightarrow \mathcal{U}_t \rangle \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta}$$

From that, the desired conclusion is straightforward.

- $o = [\alpha]t$ and $o\{u//\alpha\} = [\alpha]t\{u//\alpha\}u$. Then Φ' has the following form

$$\frac{\frac{\Gamma'_t \vdash t\{u//\alpha\} : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K_t} \mid \alpha : \mathcal{V}_\alpha; \Delta'_t \quad \Gamma'_u \Vdash u : \bigwedge_{k \in K_t} \mathcal{I}_k^* \mid \alpha : \mathcal{V}_u; \Delta_u^t}{\Gamma'_t \wedge \Gamma'_u \vdash t\{u//\alpha\}u : \bigvee_{k \in K_t} \mathcal{V}_k \mid \alpha : \mathcal{V}_\alpha; \Delta'}}{\Gamma' \vdash [\alpha]t\{u//\alpha\}u : \# \mid \alpha : \bigvee_{k \in K_t} \mathcal{V}_k \vee \mathcal{V}_\alpha; \Delta'}$$

where $\Gamma' = \Gamma'_t \wedge \Gamma'_u, \Delta' = \Delta_t \vee \Delta_u^t$ and $\mathcal{V} = \bigvee_{k \in K_t} \mathcal{V}_k \vee \mathcal{V}_\alpha \vee \mathcal{V}_u$. Moreover, the hypothesis $\alpha \notin \text{fn}(u)$ implies $\mathcal{V}_u = \langle \rangle$ by Lemma 7.2.

The *i.h.* gives $\triangleright \Gamma \vdash t : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K_t} \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K_\alpha}; \Delta$, $\triangleright \Gamma_u^\alpha \Vdash u : \wedge_{k \in K_\alpha} \mathcal{I}_k^* \mid \Delta_u^\alpha$ where $\Gamma'_t = \Gamma \wedge \Gamma_u^\alpha$, $\Delta'_t = \Delta \vee \Delta_u^\alpha$, and $\mathcal{V}_\alpha = \vee_{k \in K_\alpha} \mathcal{V}_k$. W.l.o.g we can assume $K_\alpha \cap K_t = \emptyset$. We then set $K = K_\alpha \uplus K_t$ and we define :

$$\frac{\triangleright \Gamma \vdash t : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K_t} \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K_\alpha}; \Delta}{\Gamma \vdash [\alpha]t : \# \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta}$$

By Lemma 7.3, we also have $\triangleright \Gamma_u \Vdash u : \wedge_{k \in K} \mathcal{I}_k^* \mid \Delta_u$ with $\Gamma_u = \Gamma_u^t \wedge \Gamma_u^\alpha$, $\Delta_u = \Delta_u^t \vee \Delta_u^\alpha$. We can then conclude since $\Gamma \wedge \Gamma_u = \Gamma \wedge (\Gamma_u^\alpha \wedge \Gamma_u^t) = \Gamma'_t \wedge \Gamma_u^t = \Gamma'$ and likewise $\Delta \vee \Delta_u = \Delta'$.

- $o = tv$ so that $o\{u//\alpha\} = t\{u//\alpha\}v\{u//\alpha\}$. Then Φ has the following form:

$$\frac{\triangleright \Gamma'_t \vdash t\{u//\alpha\} : \mathcal{U}_t \mid \alpha : \mathcal{V}_t; \Delta'_t \quad \triangleright \Gamma'_v \Vdash v\{u//\alpha\} : \mathcal{J}_v \mid \alpha : \mathcal{V}_v; \Delta'_v}{\Gamma' \vdash t\{u//\alpha\}v\{u//\alpha\} : \mathcal{U} \mid \alpha : \mathcal{V}; \Delta'}$$

where $\mathcal{V} = \mathcal{V}_t \vee \mathcal{V}_\alpha$, $\Gamma' = \Gamma'_t \wedge \Gamma'_v$, $\Delta' = \Delta'_t \vee \Delta'_v$, $\mathcal{U}_t = \langle \mathcal{J}_k \rightarrow \mathcal{U}_k \rangle_{k \in K}$ and $\mathcal{J}_v = \wedge_{k \in K} \mathcal{J}_k^*$ (those types are of no matter here, except they satisfy the typing constraint of `app`).

The property then trivially holds by the *i.h.* (we proceed as in the complete proof of Lemma 7.7, case `app`).

- The other cases are similar. □

We will make use of the following property (which extends Property 5.2) in Sec. 2.2.3 to show that normalization implies typability.

Property 7.2 (Subject Expansion for λ_μ). Assume $\Phi' \triangleright \Gamma' \vdash o' : \mathcal{A} \mid \Delta'$. If $o \rightarrow o'$ is a non-erasing step, then there is $\Phi \triangleright \Gamma' \vdash o : \mathcal{A} \mid \Delta'$.

Proof. By induction on the reduction relation. We only show the main cases of reduction at the root, the other ones being straightforward by induction. We can then assume $\mathcal{A} = \mathcal{U}$ for some union type \mathcal{U} .

- If $o = (\lambda x.t)u$, then $o' = t[u/x]$ with $x \in \text{fv}(t)$. The Reverse Substitution Lemma 7.7 yields

- $\Gamma' = \Gamma \wedge \Gamma_u$,
- $\Delta' = \Delta \wedge \Delta_u$,
- $\triangleright \Gamma; x : \mathcal{I} \vdash t : \mathcal{U} \mid \Delta$, and
- $\triangleright \Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$.

Moreover, $x \in \text{fv}(t)$ implies by Lemma 7.2 that $\mathcal{I} \neq []$, so that $\mathcal{I}^* = \mathcal{I}$. We can then set :

$$\Phi = \frac{\frac{\triangleright \Gamma; x : \mathcal{I} \vdash t : \mathcal{U} \mid \Delta}{\Gamma \vdash \lambda x.t : \langle \mathcal{I} \rightarrow \mathcal{U} \rangle \mid \Delta} \quad \triangleright \Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u}{\Gamma' \vdash (\lambda x.t)u : \mathcal{U} \mid \Delta'}$$

- If $o = (\mu\alpha.c)u$, then $o' = \mu\alpha.c\{u//\alpha\}$ with $\alpha \in \mathbf{fn}(c)$. Moreover, $\alpha \in \mathbf{fn}(c\{u//\alpha\})$ and Φ' has the following form:

$$\frac{\Gamma' \vdash c\{u//\alpha\} : \# \mid \alpha : \mathcal{U}; \Delta'}{\Gamma' \vdash \mu\alpha.c\{u//\alpha\} : \mathcal{U} \mid \Delta'}$$

where $\mathcal{U} \neq \langle \rangle$ holds by Lemma 7.2, since $\alpha \in \mathbf{fn}(c\{u//\alpha\})$, so that the **restore** rule is correctly applied. Then the Reverse Replacement Lemma 7.8 yields:

- $\Gamma' = \Gamma_c \wedge \Gamma_u$,
- $\Delta' = \Delta_c \vee \Delta_u$,
- $\mathcal{U} = \vee_{k \in K} \mathcal{V}_k$,
- $\triangleright \Gamma_c \vdash c : \# \mid \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta_c$, and
- $\triangleright \Gamma_u \Vdash u : \wedge_{k \in K} \mathcal{I}_k^* \mid \Delta_u$.

Moreover, $\mathcal{U} \neq \langle \rangle$ implies $K \neq \emptyset$, thus $\langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}^* = \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}$ and we conclude by constructing the following derivation:

$$\frac{\frac{\triangleright \Gamma_c \vdash c : \# \mid \alpha : \mathcal{V}_c; \Delta_c}{\Gamma_c \Vdash \mu\alpha.c : \mathcal{V}_c \mid \Delta_c} \quad \triangleright \Gamma_u \Vdash u : \mathcal{I}_u \mid \Delta_u}{\Gamma' \Vdash (\mu\alpha.c)u : \mathcal{U} \mid \Delta'}$$

where $\mathcal{V}_c = \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}$, $\mathcal{I}_u = \wedge_{k \in K} \mathcal{I}_k^*$

□

We may also prove easily that typability is stable by “creating head expansion” provided the created argument of the redex is typable. This is an extension Lemma 5.6 (and it is useful for the same reasons):

Lemma 7.9. If $\Pi' \triangleright_{\mathcal{S}_{\lambda_\mu}} \Gamma' \vdash t' : \mathcal{U} \mid \Delta'$, $t = (\lambda x.r)st_1 \dots t_q \rightarrow rt_1 \dots t_q = t'$ or $t = (\mu\alpha.c)st_1 \dots t_q \rightarrow (\mu\alpha.c)t_1 \dots t_q$ is an erasing reduction step (*i.e.* $x \notin \mathbf{fv}(r)$ or $\alpha \notin \mathbf{fn}(c)$ respectively) and s is $\mathcal{S}_{\lambda_\mu}$ -typable, then there is a $\mathcal{S}_{\lambda_\mu}$ -derivation Π , a variable context Γ and a name context Δ' such that $\Pi \triangleright_{\mathcal{S}_{\lambda_\mu}} \Gamma \vdash t : \mathcal{U} \mid \Delta$.

7.4 Strongly Normalizing λ_μ -Objects

In this section we show the characterization of strongly-normalizing objects of the $\ell\mu$ -calculus by means of the typing system introduced in Section 7.2, *i.e.* we show that an object o is strongly-normalizing iff t is typable.

Since (1) we have not proved yet that $\mathcal{S}_{\lambda_\mu}$ -typability is stable under reduction (but we have weighted SR for non-erasing steps by Property 7.1 and Lemma 7.6 for erasing head reduction steps) (2) $\mathcal{S}_{\lambda_\mu}$ -typability is not stable under expansion (but we have subject expansion for non-erasing steps by Property 7.2 and Lemma 7.6), we must use the general scheme exposed at the end of Sec. 5.2.2 (used for strong normalization in λ -calculus) instead of that of Sec. 3.3.1. To be able to make something out of the properties and lemmas quoted above, we must reformulate strong normalization in an inductive way (Lemma 7.10), as it was done in Sec. 5.2.3.

To sum up, the proof of the main contribution of this chapter (Theorem 7.2) relies on the following three steps:

- We define an inductive set $\text{ISN}(\lambda_\mu)$ which turns to be equivalent to the set of λ_μ -strongly normalizing objects $\text{SN}(\lambda_\mu)$ (Lemma 7.10).
- We show that every typable object is in $\text{ISN}(\lambda_\mu)$ (Lemma 7.11).
- We show that every o in $\text{ISN}(\lambda_\mu)$ is typable (Lemma 7.12).

We start by defining the inductive set $\text{ISN}(\lambda_\mu)$ as is the smallest subset of $\mathcal{O}_{\lambda_\mu}$ satisfying the following closure properties:

- (1) If t_1, \dots, t_q ($q \geq 0$) $\in \text{ISN}(\lambda_\mu)$, then $x t_1 \dots t_q \in \text{ISN}(\lambda_\mu)$.
- (2) If $t \in \text{ISN}(\lambda_\mu)$, then $\lambda x.t \in \text{ISN}(\lambda_\mu)$.
- (3) If $c \in \text{ISN}(\lambda_\mu)$, then $\mu \alpha.c \in \text{ISN}(\lambda_\mu)$.
- (4) If $t \in \text{ISN}(\lambda_\mu)$, then $[\alpha]t \in \text{ISN}(\lambda_\mu)$.
- (5) If $s, r[s/x] t_1 \dots t_q$ ($q \geq 0$) $\in \text{ISN}(\lambda_\mu)$, then $(\lambda x.r) s t_1 \dots t_q \in \text{ISN}(\lambda_\mu)$.
- (6) If $s, \mu \alpha.c\{s//\alpha\} t_1 \dots t_q$ ($q \geq 0$) $\in \text{ISN}(\lambda_\mu)$, then $(\mu \alpha.c) s t_1 \dots t_q \in \text{ISN}(\lambda_\mu)$.

We can formally relate $\text{SN}(\lambda_\mu)$ and $\text{ISN}(\lambda_\mu)$:

Lemma 7.10. $\text{SN}(\lambda_\mu) = \text{ISN}(\lambda_\mu)$.

Proof. This Lemma is a consequence of the Claims 7.1 and 7.2 below. \square

Claim 7.1. Let $o \in \text{SN}(\lambda_\mu)$. Then $o \in \text{ISN}(\lambda_\mu)$.

Proof. We show $o \in \text{ISN}(\lambda_\mu)$ by induction on $\langle \eta(o), |o| \rangle$. We handle only the new cases compared to those of Claim 5.1 (which are directly adapted to λ_μ).

- If $o = [\alpha]t$, then the subterm t verifies $t \in \text{SN}(\lambda_\mu)$ and $\langle \eta(t), |t| \rangle < \langle \eta(o), |o| \rangle$. The induction hypothesis gives $t \in \text{ISN}(\lambda_\mu)$, and thus by definition we get $[\alpha]t \in \text{ISN}(\lambda_\mu)$.
- The case $t = \mu \alpha.c$ is similar.
- If $o = (\mu \alpha.c) s t_1 \dots t_q$. The sub-objects c, s and the t_k satisfy $c, s, t_k \in \text{SN}(\lambda_\mu)$ with $\langle \eta(c), |c| \rangle, \langle \eta(s), |s| \rangle, \langle \eta(t_k), |t_k| \rangle < \langle \eta(o), |o| \rangle$. It follows by the induction hypothesis that c, s and t_k (for all $k \in \{1, \dots, q\}$) are in $\text{ISN}(\lambda_\mu)$. Moreover, let

$$o = (\mu \alpha.c) s t_1 \dots t_q \rightarrow_\mu (\mu \alpha.c\{s//\alpha\}) t_1 \dots t_q = o'$$

We have $o' \in \text{SN}(\lambda_\mu)$ and $\eta(o') < \eta(o)$, so that $\langle \eta(o'), |o'| \rangle < \langle \eta(o), |o| \rangle$. By the induction hypothesis, it follows that $o' \in \text{ISN}(\lambda_\mu)$. Since also $s \in \text{ISN}(\lambda_\mu)$, hence by definition, $o \in \text{ISN}(\lambda_\mu)$. \square

Claim 7.2. Let $o \in \text{ISN}(\lambda_\mu)$. Then $o \in \text{SN}(\lambda_\mu)$.

Proof. We show $o \in \text{SN}(\lambda_\mu)$ by induction on the definition of $\text{ISN}(\lambda_\mu)$. We handle only the cases that cannot be straightforwardly adapted from the proof of Claim 5.2.

- If $o = [\alpha]t$ with $t \in \text{ISN}(\lambda_\mu)$, then, by induction hypothesis, we have $t \in \text{SN}(\lambda_\mu)$ and it follows that $o \in \text{SN}(\lambda_\mu)$.
- The case $o = \mu\alpha.c$ with $c \in \text{ISN}(\lambda_\mu)$ is similar.
- If $o = (\mu\alpha.c)s t_1 \dots t_q$ with $s, \mu\alpha.c\{s//\alpha\} t_1 \dots t_q \in \text{ISN}(\lambda_\mu)$, then by *i.h.* c and $\mu\alpha.c\{s//\alpha\} t_1 \dots t_q$ are in $\text{SN}(\lambda_\mu)$. Moreover, the fact $\mu\alpha.c\{s//\alpha\} t_1 \dots t_q$ is SN implies that $c\{s//\alpha\}$ and the t_k are SN, and by observing that, if $c \rightarrow c'$ then $c\{s//\alpha\} \rightarrow c'\{s//\alpha\}$, we obtain that c is SN because $c\{s//\alpha\}$ is. We show that o is SN by a second induction on $\eta(c) + \eta(s) + \sum_{i=1 \dots q} \eta(t_i)$. Let us see how are all the reducts of o .
 - If $o \rightarrow (\mu\alpha.c')s t_1 \dots t_q = o'$, where $c \rightarrow c'$ or $o \rightarrow (\mu\alpha.c)s' t_1 \dots t_q = o'$, where $s \rightarrow s'$, or $o \rightarrow (\mu\alpha.c)s t_1 \dots t'_i \dots t_q = o'$, where $t_i \rightarrow t'_i$, then $o' \in \text{SN}(\lambda_\mu)$ by the second induction hypothesis.
 - If $o \rightarrow \mu\alpha.c\{s//\alpha\} t_1 \dots t_q = o'$, then $o' \in \text{SN}(\lambda_\mu)$ as already remarked by the first induction hypothesis.

Since all the one-step reducts of o are in $\text{SN}(\lambda_\mu)$, then $o \in \text{SN}(\lambda_\mu)$. □

As explained, we first show that any typable object o belongs to $\text{ISN}(\lambda_\mu)$.

Lemma 7.11. If o is $\mathcal{S}_{\lambda_\mu}$ -typable, then $o \in \text{ISN}(\lambda_\mu)$.

Proof. Let $\Phi \triangleright \Gamma \vdash o : \mathcal{A} \mid \Delta$. We proceed by induction on $\text{sz}(\Phi)$. When Φ does not end with the rule (**app**) the proof is straightforward, so that we consider a derivation ending with (**app**), where $\mathcal{A} = \mathcal{U}$ and $o = x t_1 \dots t_q$ or $o = (\lambda x.r) s t_1 \dots t_q$ or $o = (\mu\alpha.c) s t_1 \dots t_q$, where $q \geq 0$. The two first cases are handled as in the proof of Lemma 5.8, so let us deal only with the third one.

By construction, it is not difficult to see that there are subderivations Φ_s and $(\Phi_{t_k})_{k \in \{1 \dots q\}}$ typing s and t_k respectively such that $\text{sz}(\Phi_s) < \text{sz}(\Phi)$, $(\text{sz}(\Phi_{t_k}) < \text{sz}(\Phi))_{k \in \{1 \dots q\}}$ so that the induction hypothesis gives $s \in \text{ISN}(\lambda_\mu)$ and $(t_k \in \text{ISN}(\lambda_\mu))_{k \in \{1 \dots q\}}$. There are two subcases:

- $\alpha \in \text{fn}(c)$. Using Proposition 7.1, we get $\Phi' \triangleright \Gamma \vdash \mu\alpha.c\{s//\alpha\} t_1 \dots t_q \mid \Delta$ such that $\text{sz}(\Phi') < \text{sz}(\Phi)$. Then the induction hypothesis gives $\mu\alpha.c\{s//\alpha\} t_1 \dots t_q \in \text{ISN}(\lambda_\mu)$. This, together with $s \in \text{ISN}(\lambda_\mu)$ gives $o \in \text{ISN}(\lambda_\mu)$.
- $\alpha \notin \text{fn}(c)$. Then, by Lemma 7.6, we have a $\mathcal{S}_{\lambda_\mu}$ -derivation Π' typing $\mu\alpha.c t_1 \dots t_q$ such that $\text{sz}(\Pi') < \text{sz}(\Phi)$. Then $\mu\alpha.c t_1 \dots t_q \in \text{ISN}(\lambda_\mu)$ holds by the induction hypothesis. This, together with $s \in \text{ISN}(\lambda_\mu)$, gives $o \in \text{ISN}(\lambda_\mu)$. □

And any object $o \in \text{ISN}(\lambda_\mu)$ turns out to be typable.

Lemma 7.12. If $o \in \text{ISN}(\lambda_\mu)$, then o is $\mathcal{S}_{\lambda_\mu}$ -typable.

Proof. We reason by induction $o \in \text{ISN}(\lambda_\mu)$. The four first cases are straightforward. The fifth one ($o = (\lambda x.r)s t_1 \dots t_q$ etc) is handled as in the proof of Lemma 5.9.

Let $o = (\mu\alpha.c)s t_1 \dots t_q \in \text{ISN}(\lambda_\mu)$ coming from $\mu\alpha.c\{s//\alpha\} t_1 \dots t_q$, $s \in \text{ISN}(\lambda_\mu)$. By the induction hypothesis, $\mu\alpha.c\{s//\alpha\} t_1 \dots t_q$ and s are both typable. We consider two subcases. If $\alpha \in \text{fn}(c)$, then $(\mu\alpha.c)s t_1 \dots t_q$ is typable by Proposition 7.2. Otherwise, we use Lemma 7.9. \square

Lemma 7.11, 7.12 and the equality stated by Lemma 7.10 allow us to conclude with the equivalence between $\mathcal{S}_{\lambda_\mu}$ -typability and strong-normalization for the λ_μ -calculus.

Theorem 7.2. Let $o \in \mathcal{O}_{\lambda_\mu}$. Then o is typable in system $\mathcal{S}_{\lambda_\mu}$ iff $o \in \text{SN}(\lambda_\mu)$. Moreover, if o is $\mathcal{S}_{\lambda_\mu}$ -typable with a derivation Π , then $\text{sz}(\Pi)$ gives an upper bound to the maximal length of a reduction sequence starting at o .

To prove the second statement it is sufficient to endow the system with non-relevant axioms for variables and names, as we did for λ -calculus and system \mathcal{S}_λ in Sec. 5.2.5. This modification, which does not preserve subject expansion (see Remark 5.8), is however sufficient to extend Property 5.3 and to guarantee *weighted* subject reduction in all the cases (erasing and non-erasing steps) without changing the original measure of the derivations in system $\mathcal{S}_{\lambda_\mu}$.

1

7.5 Relevance (an Inquiry)

Before concluding this chapter, let us have a prospective and informal discussion on relevance. In the course of this PhD, relevance/irrelevance quickly appeared as very important features of intersection type systems. Basically, relevance simply means the prohibition of weakening (Sec. 3.3.5).

The author had discussions with several researchers about this notion. The only consensus that emerged was that “relevance was a matter of philosophy”. For instance, Gardner/de Carvalho’s system \mathcal{R}_0 is presented as relevant in this thesis. However, one participant argued that $\lambda x.y$ could be typed (with $[] \rightarrow \tau$) and that relevance concerns only λI terms (*i.e.* terms t such that, if $t|_b = \lambda x.u$, then $x \in \text{fv}(u)$), so that \mathcal{R}_0 was actually not relevant.

The author would say that a system is relevant as long as subject reduction just clones β -reduction. We iterate the following sentence (from Sec. 4.1.1): relevant derivations behave like λ -terms more than irrelevant ones do. A bit more precisely, perhaps a type system could be considered as relevant as long as weakenings do not add a layer of non-determinism. For instance:

- In system \mathcal{D}_0 (idempotent intersection, relevant): subject reduction is deterministic (Sec. 3.3.2).
- In system \mathcal{R}_0 (non-idempotent intersection, relevant): subject reduction is not deterministic (Sec. 4.1.2), but it just consists in destroying axiom leaves and moving argument derivations (Sec. 3.3.2).
- In system $\mathcal{D}_{0,w}$ (idempotent intersection, irrelevant): we may discard some argument derivation during reduction (see left-part of Fig. 3.3 in Sec. 3.3.4), which can entail a loss of context preservation. This loss of context preservation can be corrected by using weakenings (see the proof sketch following 3.5).

- In system \mathcal{S}_w (non-idempotent intersection, irrelevant): subject reduction is also ensured by weakening for some erasing steps (see *e.g.*, the proof of Property 5.3).

In the λ_μ -calculus, not only for the characterization of strong normalization ($\mathcal{S}_{\lambda_\mu}$) but also for that of head normalization (system $\mathcal{H}_{\lambda_\mu}$), the presence of the choice operator $_*$ in the **restore**-rule can be seen as a *weakening* (on the right-hand sides of sequents), as noted in Sec. 7.2.2.

However, this rule does not compromise the operational behaviour of subject reduction: if $t = (\mu\alpha.c)s \rightarrow \mu\alpha.c\{s//\alpha\} = t'$ is an erasing step ($\alpha \notin \mathbf{fn}(c)$) and the $\mathcal{H}_{\lambda_\mu}$ -derivation Π types t , then Π is of the form:

$$\Pi = \frac{\frac{\Phi_c \triangleright \Gamma \vdash c : \# \mid \Delta}{\Gamma \vdash \mu\alpha.c : \langle [] \rightarrow \sigma \rangle \mid \Delta}}{\Gamma \vdash (\mu\alpha.c)s : \langle \sigma \rangle \mid \Delta}$$

where $\alpha \notin \mathbf{dom}(\Delta)$. The derivation Π' obtained from Π by subject reduction is the following:

$$\Pi' = \frac{\Phi_c \triangleright \Gamma \vdash c\{s//\alpha\} : \# \mid \Delta}{\Gamma \vdash \mu\alpha.c\{s//\alpha\} : \langle \sigma \rangle \mid \Delta}$$

In Π , the choice operator was instantiated with the blind type $[] \rightarrow \sigma$ and Π' , we just have to instantiate it with $\langle \sigma \rangle$. Thus, to obtain Π' , no additional weakening (compared to Π) is needed (*e.g.*, contrary to the proofs of Lemma 5.12, or Property 5.3, p. 129). Thus, one may say that system $\mathcal{H}_{\lambda_\mu}$ is **dynamically relevant** and indeed, in Sec. 7.2.3, we saw that subject reduction in $\mathcal{H}_{\lambda_\mu}$ for the μ -rule consisted in not much more than moving argument derivations (and destroying the **app**-rule of the μ -redex). Another way to understand this consists in going back to the discussion on the typing of **call-cc** in $\mathcal{H}_{\lambda_\mu}$, p. 7.2.2. When $\Delta(\alpha)$ is not empty, the μ -abstraction *restores* the type $\Delta(\alpha)$ whereas when $\Delta(\alpha)$ is empty, $\mu\alpha$ *creates* a type *via* the choice operator. In the latter case, the **restore** rule almost behaves like an axiom rule (on names instead of variable). Distinguishing explicitly those two rules could help to think of $\mathcal{H}_{\lambda_\mu}$ as a relevant system (and more generally, the fact that even in Laurent's system [73], union types cannot be empty). For now, these considerations are only prospective: one still needs to find a satisfactory formal definition (including its dynamical aspects) and relevance remains to be better understood.

Chapter 8

A Resource Aware Semantics for the Lambda-Mu-Calculus

In this chapter, we present $\lambda_{\mu x}$ a resource aware semantics for λ_{μ} -calculus, that processes both substitution and replacements *linearly i.e.* one occurrence after the other. This calculus $\lambda_{\mu x}$ is an extension of Accattoli and Kesner calculus with Explicit Substitution (Sec. 2.4). As the previous chapter, this is common work with Delia Kesner.

We then endow the $\lambda_{\mu x}$ -calculus with a type system that both extends (1) the quantitative type system $\mathcal{S}_{\lambda_{\mu}}$ presented in Sec. 7.2, and (2) the variant of system \mathcal{R}_{ex} (Sec. 4.2) that would characterize Strong Normalization in Λ_{ex} , which is not presented here by lack of time.

The main point is to define the operational semantics (so that replacement is linearly processed) and the size measure associated to the new typing rules (so that we obtain a weighted subject reduction property). The latter point is tricky since we need to use half-integers instead of just the plain integers as for the other quantitative system that were hitherto presented.

Apart from that, the proof and the progression exactly follows that of Chapter 7 *e.g.*, we do not use the general scheme of Sec. 3.3.1 but that at the end of Sec. 5.2.2 (see also the discussion following Property 7.1): we only have *weighted* subject reduction and subject expansion in the non-erasing cases and we must therefore reformulate strong normalization in the calculus as an inductive predicate.

For those reasons, not as many details and explanations as in Chapter 7 will be given, except in the first section (operational semantics, typing rules and size).

We could also probably define linear head reduction for this calculus and also obtain a type-theoretic characterization, but this was not investigated yet, by lack of time.

8.1 The $\lambda_{\mu x}$ -calculus

This section introduces the syntax (Sec. 8.1.1) and the operational semantics (Sec. 8.1.2) of the $\lambda_{\mu x}$ -calculus, a resource aware model of λ_{μ} . The $\lambda_{\mu x}$ -calculus is an extension of the (intuitionistic) *linear substitution calculus* [2], deeply studied in rewriting theory and implicit complexity and that is briefly presented in Sec. 2.4.

8.1.1 Syntax

The set of **objects** ($\mathcal{O}_{\lambda_{\mu r}}$), **terms** ($\mathcal{T}_{\lambda_{\mu r}}$) and **commands** ($\mathcal{C}_{\lambda_{\mu r}}$) of the $\lambda_{\mu r}$ -calculus are given by the following grammars

$$\begin{array}{lll} \text{(objects)} & o & ::= t \mid c \\ \text{(terms)} & t, u & ::= x \mid \lambda x.t \mid tu \mid \mu\alpha.c \mid t\langle x \setminus u \rangle \\ \text{(commands)} & c & ::= [\alpha]t \mid c\langle \alpha \setminus \beta.u \rangle \end{array}$$

The construction $\langle x \setminus u \rangle$ (resp. $\langle \alpha \setminus \beta.u \rangle$) is called an **explicit substitution** (resp. **explicit replacement**). Remark that explicit substitutions do not apply to commands and explicit replacements do not apply to terms and that explicit replacements and substitutions both resort to angle brackets: we distinguish explicit replacement from explicit substitution by the metavariables and the use of \setminus instead of just \setminus .

An explicit substitution $\langle x \setminus u \rangle$ implements the *meta-substitution* operator $[u/x]$ while an explicit replacement $\langle \alpha \setminus \beta.u \rangle$ implements the *fresh* replacement meta-operator $\{\beta.u // \alpha\}$ introduced in Sec. 6.2.3, *i.e.* the small step computation of $c\langle \alpha \setminus \beta.u \rangle$ replaces only one occurrence of $[\alpha]t$ inside c by $[\beta]t\langle \alpha \setminus \beta.u \rangle$ [5]. As in Sec. 6.2.1, the **size of an object** o is denoted by $|o|$.

The notions of **free** and **bound variables** and **names** are extended as expected, in particular $\mathbf{fv}(t\langle x \setminus u \rangle) := (\mathbf{fv}(t) \setminus \{x\}) \cup \mathbf{fv}(u)$ and $\mathbf{fn}(c\langle \alpha \setminus \beta.u \rangle) := (\mathbf{fn}(c) \setminus \{\alpha\}) \cup \{\beta\} \cup \mathbf{fn}(u)$. The derived notion of α -conversion (*i.e.* renaming of bound variables and names) will be assumed in the rest of the paper. The **number of free occurrences** of the variable x (resp. the name α) in o is denoted by $|o|_x$ (resp. $|o|_\alpha$). For instance, $([\gamma]x[x/y])\langle \gamma \setminus \beta.z \rangle =_\alpha ([\gamma']x'[x'/y])\langle \gamma' \setminus \beta.z \rangle$.

List (L), **term (TT, CT, OT)**, and **command (TC, CC, OC) contexts** are respectively defined by the following grammars:

$$\begin{array}{l} \mathbf{L} ::= \square \mid \mathbf{L}\langle x \setminus u \rangle \\ \mathbf{TT} ::= \square \mid \lambda x.\mathbf{TT} \mid \mathbf{TT}t \mid t\mathbf{TT} \mid \mu\alpha.\mathbf{CT} \mid \mathbf{TT}\langle x \setminus t \rangle \mid t\langle x \setminus \mathbf{TT} \rangle \\ \mathbf{CT} ::= [\alpha]\mathbf{TT} \mid \mathbf{CT}\langle \alpha \setminus \beta.u \rangle \mid c\langle \alpha \setminus \beta.\mathbf{TT} \rangle \\ \mathbf{OT} ::= \mathbf{TT} \mid \mathbf{CT} \\ \mathbf{TC} ::= \lambda x.\mathbf{TC} \mid \mathbf{TC}t \mid t\mathbf{TC} \mid \mu\alpha.\mathbf{CC} \mid \mathbf{TC}\langle x \setminus t \rangle \mid t\langle x \setminus \mathbf{TC} \rangle \\ \mathbf{CC} ::= \square \mid [\alpha]\mathbf{TC} \mid \mathbf{CC}\langle \alpha \setminus \beta.u \rangle \mid c\langle \alpha \setminus \beta.\mathbf{TC} \rangle \\ \mathbf{OC} ::= \mathbf{TC} \mid \mathbf{CC} \end{array}$$

The hole \square (resp. \square) can be replaced by a term (resp. a command); indeed, $\mathbf{L}[t]$ denotes the replacement of \square in \mathbf{L} by the term t (similarly for $\mathbf{TT}[t]$, $\mathbf{CT}[t]$ and \mathbf{OT}), while $\mathbf{CC}[c]$ denotes the replacement of \square in \mathbf{CC} by the command c (similarly for \mathbf{TC} and \mathbf{OC}). The meta-expressions can be read like this: \mathbf{TC} denotes a context that takes a command (\mathbf{C} on the right) and outputs a term (\mathbf{T} on the left).

Let \mathcal{S} be a set of objects. Proceeding as in Sec. 2.2, we write $\mathbf{OT}^{\mathcal{S}}$ for a term context \mathbf{OT} which does not capture the free variables and names of any object in \mathcal{S} , *i.e.* there are no abstractions and substitutions in the context that bind a symbol that occurs free in an objects in \mathcal{S} . For instance $\mathbf{TT} = \lambda y.\square$ (resp. $\mu\beta.\square$) can be specified as $\mathbf{TT}^{\mathcal{S}}$ (resp. $\mathbf{TC}^{\mathcal{S}}$) while $\mathbf{TT} = \lambda x.\square$ (resp. $\mu\alpha.\square$) cannot. In order to emphasize this particular property we may write $\mathbf{TT}^{\mathcal{S}}[[t]]$ (resp. $\mathbf{OC}^{\mathcal{S}}[[c]]$) instead of $\mathbf{TT}^{\mathcal{S}}[t]$ (resp. $\mathbf{OC}^{\mathcal{S}}[c]$), and we may omit \mathcal{S} when it is clear from the context. Same concepts apply to command contexts, *i.e.* $\mathbf{OC}^{\mathcal{S}}$ does not capture the variables and names in \mathcal{S} and the notation used for that is $\mathbf{OC}^{\mathcal{S}}[[c]]$.

8.1.2 Operational Semantics

The reduction rules of the $\lambda_{\mu x}$ -calculus aim to give a resource aware semantics to the λ_{μ} -calculus, based on the *substitution/replacement at a distance* paradigm [2,3]. Indeed, the reduction relation $\lambda_{\mu x}$ of the calculus is given by the context closure of the following rewriting rules.

$$\begin{array}{lll}
L[(\lambda x.t)u] & \rightarrow_{\mathbf{B}} & L[t\langle x \setminus u \rangle] \\
\mathbf{TT}[[x]\langle x \setminus u \rangle] & \rightarrow_{\mathbf{c}} & \mathbf{TT}[[u]\langle x \setminus u \rangle] \quad \text{if } |\mathbf{TT}[[x]]|_x > 1 \\
\mathbf{TT}[[x]\langle x \setminus u \rangle] & \rightarrow_{\mathbf{d}} & \mathbf{TT}[[u]] \quad \text{if } |\mathbf{TT}[[x]]|_x = 1 \\
t\langle x \setminus u \rangle & \rightarrow_{\mathbf{w}} & t \quad \text{if } x \notin \mathbf{fv}(t) \\
L[(\mu \alpha.c)u] & \rightarrow_{\mathbf{M}} & L[\mu \gamma.c\langle \alpha \setminus \gamma.u \rangle] \quad \text{if } \gamma \text{ is fresh} \\
\mathbf{CC}[[\alpha]t]\langle \alpha \setminus \gamma.u \rangle & \rightarrow_{\mathbf{c}_n} & \mathbf{CC}[[\gamma]tu]\langle \alpha \setminus \gamma.u \rangle \quad \text{if } |\mathbf{CC}[[\alpha]t]]_{\alpha} > 1 \\
\mathbf{CC}[[\alpha]t]\langle \alpha \setminus \gamma.u \rangle & \rightarrow_{\mathbf{d}_n} & \mathbf{CC}[[\gamma]tu] \quad \text{if } |\mathbf{CC}[[\alpha]t]]_{\alpha} = 1 \\
c\langle \alpha \setminus \gamma.u \rangle & \rightarrow_{\mathbf{w}_n} & c \quad \text{if } \alpha \notin \mathbf{fn}(c)
\end{array}$$

where \mathbf{TT} is to be understood as \mathbf{TT}^x and \mathbf{CC} as $\mathbf{CC}^{\alpha, \gamma}$.

We use $\rightarrow_{\mathbf{w}}$ for the reduction relation generated by the rules $\rightarrow_{\mathbf{w}}$ and $\rightarrow_{\mathbf{w}_n}$ and $\rightarrow_{\lambda_{\mu x}}$ for the **non-erasing reduction relation** $\rightarrow_{\lambda_{\mu x}} \setminus \rightarrow_{\mathbf{w}}$. For instance, the big step reduction

$$(\mu \alpha.[\alpha]x(\mu \beta.[\alpha]\lambda x.xx))u \rightarrow_{\mu} \mu \gamma.[\gamma]x(\mu \beta.[\gamma](\lambda x.xx)u)u$$

where α has been alpha-renamed to γ , can be now emulated by 3 small steps :

$$\begin{array}{l}
(\mu \alpha.[\alpha]x(\mu \beta.[\alpha]\lambda x.xx))u \\
\rightarrow_{\mathbf{M}} \mu \gamma.([\alpha]x(\mu \beta.[\alpha]\lambda x.xx))\langle \alpha \setminus \gamma.u \rangle \\
\rightarrow_{\mathbf{c}_n} \mu \gamma.([\alpha]x(\mu \beta.[\gamma](\lambda x.xx)u))\langle \alpha \setminus \gamma.u \rangle \\
\rightarrow_{\mathbf{d}_n} \mu \gamma.[\gamma]x(\mu \beta.[\gamma](\lambda x.xx)u)u
\end{array}$$

Notice that the occurrences of α are (arbitrarily) replaced by γ one after another, thus replacement is *linearly* processed. When there is just one occurrence of α left, the small reduction step \mathbf{d}_n performs the last replacement and erases the remaining ER $\langle \alpha \setminus \gamma.u \rangle$ to complete the operation.

More generally, not only the syntax of the $\lambda_{\mu x}$ -calculus can be seen as a refinement of the λ_{μ} -calculus, but also its operational semantics. Formally,

Lemma 8.1. If $o \in \mathcal{O}_{\lambda_{\mu}}$, then $o \rightarrow_{\lambda_{\mu}} o'$ implies $o \rightarrow_{\lambda_{\mu x}}^+ o'$.

Moreover, we can project $\lambda_{\mu x}$ -reduction sequences into λ_{μ} -reduction sequences. Indeed, consider the projection function $\mathbf{P}(_)$ computing all the explicit substitutions and replacements of an object, thus in particular $\mathbf{P}(t\langle x \setminus u \rangle) := \mathbf{P}(t)[\mathbf{P}(u)/x]$ and $\mathbf{P}(c\langle \alpha \setminus \alpha'.u \rangle) := \mathbf{P}(c)\{\alpha'.\mathbf{P}(u)\}/\alpha\}$. Then,

Lemma 8.2. If $o \in \mathcal{O}_{\lambda_{\mu x}}$, then $o \rightarrow_{\lambda_{\mu x}} o'$ implies $\mathbf{P}(o) \rightarrow_{\lambda_{\mu}}^* \mathbf{P}(o')$.

8.1.3 Typing System

In this section we extend the (quantitative) typing system $\mathcal{S}_{\lambda_{\mu}}$ in order to capture the $\lambda_{\mu x}$ -calculus, the aim being to characterize the set of strongly $\lambda_{\mu x}$ -normalizing objects by using quantitative arguments.

More precisely, system $\mathcal{S}_{\lambda_{\mu}}$ is enriched with the two typing rules in Fig. 8.1. Rule (**exs**) is inspired by the derivation tree typing the term $(\lambda x.t)$: indeed, any derivation

$$\frac{\Gamma_t; x : \mathcal{I} \vdash t : \mathcal{U} \mid \Delta_t \quad \Gamma_u \Vdash u : \mathcal{I}^* \mid \Delta_u}{\Gamma_t \wedge \Gamma_u \vdash t\langle x \setminus u \rangle : \mathcal{U} \mid \Delta_t \vee \Delta_u} \text{ (exs)}$$

$$\frac{\Gamma_c \vdash c : \# \mid \Delta_c; \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K} \quad \Gamma_u \Vdash u : (\wedge_{k \in K} \mathcal{I}_k^*)^* \mid \Delta_u}{\Gamma_c \wedge \Gamma_u \vdash c\langle \alpha \setminus \alpha'.u \rangle : \# \mid \Delta_c \vee \Delta_u \vee \alpha' : \vee_{k \in K} \mathcal{V}_k} \text{ (exr)}$$

Figure 8.1: System $\mathcal{S}_{\lambda_{\mu r}}$

$\triangleright \Gamma \vdash (\lambda x.t)u : \mathcal{V} \mid \Delta$ induces two derivations $\triangleright \Gamma_t, x : \mathcal{I} \vdash t : \mathcal{V} \mid \Delta_t$ and $\triangleright \Gamma_u \Vdash u : \mathcal{I}^* \mid \Delta_u$, from which we can type $t\langle x \setminus u \rangle$. Likewise, the rule **(exr)** is motivated by the derivation tree typing a μ -redex. In particular, when $K = \emptyset$ (i.e. when $\alpha \notin \text{fn}(c)$), then $(\wedge_{k \in \emptyset} \mathcal{I}_k^*)^* = []^*$, so that the outer star in $(\wedge_{k \in K} \mathcal{I}_k^*)^*$ gives an arbitrary multiset $[\sigma]$ ensuring the typing (and thus the *SN* property) of the replacement argument u . Notice that Lemma 7.1 still holds for $\mathcal{S}_{\lambda_{\mu r}}$.

As one may expect, system $\mathcal{S}_{\lambda_{\mu r}}$ encodes a non-idempotent and relevant system for intuitionistic logic with explicit substitutions [60]. More precisely, restricting rule **(exs)** to λ -terms with explicit substitutions gives the following rule:

$$\frac{\Gamma; x : \mathcal{I} \vdash t : \sigma \quad \Gamma' \Vdash u : \mathcal{I}^*}{\Gamma \wedge \Gamma' \vdash t\langle x \setminus u \rangle : \sigma}$$

A Relevance Lemma also holds for $\lambda_{\mu r}$:

Lemma 8.3 (Relevance). Let $o \in \mathcal{O}_{\lambda_{\mu r}}$. If $\triangleright \Gamma \vdash o : \mathcal{A} \mid \Delta$ (resp. $\triangleright \Gamma \Vdash t : \mathcal{I} \mid \Delta$ with $\mathcal{I} \neq []$), then $\text{fv}(o) = \text{dom}(\Gamma)$ and $\text{fn}(o) = \text{dom}(\Delta)$ (resp. $\text{fv}(t) = \text{dom}(\Gamma)$ and $\text{fn}(t) = \text{dom}(\Delta)$).

We now extend the function $\text{sz}(_)$ introduced in Sec. 7.2.4 by adding the following cases:

$$\text{sz}\left(\frac{\Phi_t \triangleright t \quad \Phi_u \triangleright u}{\Gamma_t \wedge \Gamma_u \vdash t\langle x \setminus u \rangle : \mathcal{U} \mid \Delta_t \vee \Delta_u} \text{ (exs)}\right) := \text{sz}(\Phi_t) + \text{sz}(\Phi_u)$$

$$\text{sz}\left(\frac{\Phi_c \triangleright c \quad \Phi_u \triangleright u}{\Gamma_c \wedge \Gamma_u \vdash c\langle \alpha \setminus \alpha'.u \rangle : \# \mid \Delta_c \vee \Delta_u} \text{ (exr)}\right) := \text{sz}(\Phi_c) + \text{sz}(\Phi_u) + |K| - \frac{1}{2}$$

Notice that $\text{sz}(\Phi) \geq 1$ still holds for any *regular* derivation Φ .

As explained in Sec. 7.3.1, *weighted* subject reduction holds for μ -reduction steps like $t = (\mu\alpha.c)u \rightarrow_{\mu} \mu\gamma.c\{\gamma.u/\alpha\} = t'$ because γ is typed in t' with smaller arity than that of α in t . The (big) step above is emulated in the $\lambda_{\mu r}$ -calculus by the (small) steps $t \rightarrow_{\mathbb{M}} \mu\gamma.c\langle \alpha \setminus \gamma.u \rangle \rightarrow_{\mathbf{c}_n, \mathbf{d}_n, \mathbf{w}_n}^+ t'$, where \mathbf{c}_n and \mathbf{d}_n perform linear replacements, so they are also naturally expected to decrease the size of type derivations. However, for the first step $t = (\mu\alpha.c)u \rightarrow_{\mathbb{M}} \mu\gamma.c\langle \alpha \setminus \gamma.u \rangle = t'$, even if no real replacement has taken place yet, we should still have a quantifiable decrease of the form $\text{sz}(\Phi_t) > \text{sz}(\Phi_{t'})$. This is the reason we use " $-\frac{1}{2}$ " when defining the size of explicit replacements, which does not compromise the forthcoming weighted subject reduction property.

One may naively think that the " $-\frac{1}{2}$ " component in the size definition of an explicit replacement can compromise the decrease of the size for a step $t = \mu\gamma.c\langle \alpha \setminus \gamma.u \rangle \rightarrow_{\mathbf{a}_n}$

$\mu\alpha.c\{\gamma.u//\alpha\} = t'$, when c holds exactly one occurrence of α : indeed, removing the explicit replacement $\langle\alpha//\gamma.u\rangle$ induces an *increase* of the measure equal to $\frac{1}{2}$. However, the arity contribution of (the unique occurrence of) α in t is greater than that of the new occurrence of γ in t' : the replacement operation then induces a decrease of the measure which is equal to some $k \geq 1$; and thus the overall decrease of the measure is in the worst case $k - \frac{1}{2} > 0$, which still grants $\mathbf{sz}(\Phi) > \mathbf{sz}(\Phi')$. The decrease of the measure for a w_n -step is more evident. Last, but not least, the fact that $\mathbf{sz}(\Phi)$ is a half-integer ≥ 1 ensures that the measure is still well-founded.

8.2 Typing Properties

As in the case of the λ_μ -calculus, we show that the refined $\lambda_{\mu x}$ -calculus is well-behaved w.r.t. the extended typing system $\mathcal{S}_{\lambda_{\mu x}}$. This is done by means of forward (Sec. 8.2.1) and backward (Sec. 8.2.2) properties.

8.2.1 Forward Properties

Weighted Subject reduction for the $\lambda_{\mu x}$ -calculus (Lemma 8.6) is based on the fact that linear substitution (Lemma 8.4) and linear replacement (Lemma 8.5) preserve types.

Lemma 8.4 (Linear Substitution). Let $\Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$. If $\Phi_{\text{OT}[x]} \triangleright \Gamma; x : \mathcal{I} \vdash \text{OT}[x] : \mathcal{A} \mid \Delta$, then $\exists \mathcal{I}_1, \exists \mathcal{I}_2, \exists \Gamma_u^1, \exists \Gamma_u^2, \exists \Delta_u^1, \exists \Delta_u^2$ s.t.

- $\mathcal{I} = \mathcal{I}_1 \wedge \mathcal{I}_2$, where $\mathcal{I}_1 \neq []$,
- $\Gamma_u = \Gamma_u^1 \wedge \Gamma_u^2$ and $\Delta_u = \Delta_u^1 \vee \Delta_u^2$,
- $\Theta_u^1 \triangleright \Gamma_u^1 \Vdash u : \mathcal{I}_1 \mid \Delta_u^1$,
- $\Theta_u^2 \triangleright \Gamma_u^2 \Vdash u : \mathcal{I}_2 \mid \Delta_u^2$,
- $\Phi_{\text{T}[u]} \triangleright \Gamma \wedge \Gamma_u^1; x : \mathcal{I}_2 \vdash \text{OT}[u] : \mathcal{A} \mid \Delta \vee \Delta_u^1$, and
- $\mathbf{sz}(\Phi_{\text{OT}[u]}) = \mathbf{sz}(\Phi_{\text{OT}[x]}) + \mathbf{sz}(\Theta_u^1) - |\mathcal{I}_1|$.

Proof. The proof is by induction on the context OT so we need to prove the statement of the lemma for regular derivations simultaneously with the following one for *non-empty* auxiliary derivations: if $\Phi_{\text{TT}[x]} \triangleright \Gamma; x : \mathcal{I} \Vdash \text{TT}[x] : \mathcal{J} \mid \Delta$ and $\mathcal{J} \neq []$, then $\exists \mathcal{I}_1, \exists \mathcal{I}_2, \exists \Gamma_u^1, \exists \Gamma_u^2, \exists \Delta_u^1, \exists \Delta_u^2$ s.t.

- $\mathcal{I} = \mathcal{I}_1 \wedge \mathcal{I}_2$, where $\mathcal{I}_1 \neq []$,
- $\Gamma_u = \Gamma_u^1 \wedge \Gamma_u^2$ and $\Delta_u = \Delta_u^1 \vee \Delta_u^2$,
- $\Theta_u^1 \triangleright \Gamma_u^1 \Vdash u : \mathcal{I}_1 \mid \Delta_u^1$,
- $\Theta_u^2 \triangleright \Gamma_u^2 \Vdash u : \mathcal{I}_2 \mid \Delta_u^2$,
- $\Phi_{\text{TT}[u]} \triangleright \Gamma \wedge \Gamma_u^1; x : \mathcal{I}_2 \Vdash \text{TT}[u] : \mathcal{J} \mid \Delta \vee \Delta_u^1$, and
- $\mathbf{sz}(\Phi_{\text{TT}[u]}) = \mathbf{sz}(\Phi_{\text{TT}[x]}) + \mathbf{sz}(\Theta_u^1) - |\mathcal{I}_1|$.

Notice that $\mathcal{I} \neq []$ by Lemma 8.3, since $x \in \mathbf{fv}(\mathbf{OT}[\![x]\!])$ (resp. $x \in \mathbf{fv}(\mathbf{TT}[\![x]\!])$). We only show the case $\mathbf{OT} = \square$ since all the other ones are straightforward. So assume $\mathbf{OT} = \square$. Then $\mathcal{I} = [\mathcal{U}]$ for some \mathcal{U} and the derivation Φ_x has the following form :

$$\Phi_x = \frac{}{x : [\mathcal{U}] \vdash x : \mathcal{U} \mid \emptyset}$$

Thus, $\mathbf{sz}(\Phi_x) = 1$. We set then $\Theta_u^1 = \Theta_u$ and $\Theta_u^2 = \vdash u : [] \mid$. We have $\mathbf{sz}(\Phi_u) = \mathbf{sz}(\Theta_u^1) = \mathbf{sz}(\Phi_x) + \mathbf{sz}(\Theta_u) - |\mathcal{I}_1|$ since $|\mathcal{I}_1| = 1$. \square

Lemma 8.5 (Linear Replacement). Let $\Theta_u \triangleright \Gamma_u \Vdash u : \bigwedge_{\ell \in L} \mathcal{I}_\ell^* \mid \Delta_u$ s.t. $\alpha \notin \mathbf{fv}(u)$. If $\Phi_{\mathbf{OC}[\![\alpha]t\!] } \triangleright \Gamma \vdash \mathbf{OC}[\![\alpha]t\!] : \mathcal{A} \mid \alpha : \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L}; \Delta$, then $\exists L_1, \exists L_2, \exists \Gamma_u^1, \exists \Gamma_u^2, \exists \Delta_u^1, \exists \Delta_u^2, \exists \Phi_{\mathbf{OC}[\![\alpha']tu]}$ s.t.

- $L = L_1 \uplus L_2$, where $L_1 \neq \emptyset$.
- $\Gamma_u = \Gamma_u^1 \wedge \Gamma_u^2$ and $\Delta_u = \Delta_u^1 \vee \Delta_u^2$,
- $\Theta_u^1 \triangleright \Gamma_u^1 \Vdash u : \bigwedge_{\ell \in L_1} \mathcal{I}_\ell^* \mid \Delta_u^1$,
- $\Theta_u^2 \triangleright \Gamma_u^2 \Vdash u : \bigwedge_{\ell \in L_2} \mathcal{I}_\ell^* \mid \Delta_u^2$,
- $\Phi_{\mathbf{OC}[\![\alpha']tu]} \triangleright \Gamma \wedge \Gamma_u^1 \vdash \mathbf{OC}[\![\alpha']tu] : \mathcal{A} \mid \alpha : \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L_2}; \alpha' : \bigvee_{\ell \in L_1} \mathcal{V}_\ell \vee \Delta \vee \Delta_u^1$, and
- $\mathbf{sz}(\Phi_{\mathbf{OC}[\![\alpha']tu]}) = \mathbf{sz}(\Phi_{\mathbf{OC}[\![\alpha]t]}) + \mathbf{sz}(\Theta_u^1)$.

Proof. The proof is by induction on the context \mathbf{OC} so we need to prove the statement of the lemma for regular derivations simultaneously with the following one for *non-empty* auxiliary derivations: if $\Phi_{\mathbf{TC}[\![\alpha]t\!] } \triangleright \Gamma \Vdash \mathbf{TC}[\![\alpha]t\!] : \mathcal{J} \mid \alpha : \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L}; \Delta$ and $\mathcal{J} \neq []$, then $\exists L_1, \exists L_2, \exists \Gamma_u^1, \exists \Gamma_u^2, \exists \Delta_u^1, \exists \Delta_u^2, \exists \Phi_{\mathbf{TC}[\![\alpha']tu]}$ s.t.

- $L = L_1 \uplus L_2$, where $L_1 \neq \emptyset$.
- $\Gamma_u = \Gamma_u^1 \wedge \Gamma_u^2$ and $\Delta_u = \Delta_u^1 \vee \Delta_u^2$,
- $\Theta_u^1 \triangleright \Gamma_u^1 \Vdash u : \bigwedge_{\ell \in L_1} \mathcal{I}_\ell^* \mid \Delta_u^1$,
- $\Theta_u^2 \triangleright \Gamma_u^2 \Vdash u : \bigwedge_{\ell \in L_2} \mathcal{I}_\ell^* \mid \Delta_u^2$,
- $\Phi_{\mathbf{TC}[\![\alpha']tu]} \triangleright \Gamma \wedge \Gamma_u^1 \Vdash \mathbf{TC}[\![\alpha']tu] : \mathcal{J} \mid \alpha : \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L_2}; \alpha' : \bigvee_{\ell \in L_1} \mathcal{V}_\ell \vee \Delta \vee \Delta_u^1$, and
- $\mathbf{sz}(\Phi_{\mathbf{TC}[\![\alpha']tu]}) = \mathbf{sz}(\Phi_{\mathbf{TC}[\![\alpha]t]}) + \mathbf{sz}(\Theta_u^1)$.

Notice that $L \neq \emptyset$ by Lemma 8.3, since $\alpha \in \mathbf{fn}(\mathbf{OC}[\![\alpha]t\!])$ (resp. $\alpha \in \mathbf{fn}(\mathbf{TC}[\![\alpha]t\!])$). We only show the case $\mathbf{OC} = \square$ since all the other ones are straightforward.

So assume $\mathbf{OC} = \square$. Then the derivation $\Phi_{[\alpha]t}$ has the following form, where $K \neq \emptyset$ holds by Lemma 7.1:

$$\frac{\Phi_t \triangleright \Gamma \vdash t : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K} \mid \alpha : \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L \setminus K}; \Delta}{\Gamma \vdash [\alpha]t : \# \mid \alpha : \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L}; \Delta}$$

Thus, $\mathbf{sz}(\Phi_{[\alpha]t}) = \mathbf{sz}(\Phi_t) + \mathbf{ar}(\langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}) = \mathbf{sz}(\Phi_t) + |K| + \mathbf{ar}(\bigvee_{k \in K} \mathcal{V}_k)$. We set $L_1 = K$ and $L_2 = L \setminus K$ and we write $\bigwedge_{\ell \in L} \mathcal{I}_\ell^*$ as $(\bigwedge_{\ell \in L_1} \mathcal{I}_\ell^*) \wedge (\bigwedge_{\ell \in L_2} \mathcal{I}_\ell^*)$. Then by Lemma 7.3 there are $\Theta_u^1 \triangleright \Gamma_u^1 \Vdash u : \bigwedge_{\ell \in L_1} \mathcal{I}_\ell^* \mid \Delta_u^1$, $\Theta_u^2 \triangleright \Gamma_u^2 \Vdash u : \bigwedge_{\ell \in L_2} \mathcal{I}_\ell^* \mid \Delta_u^2$ s.t.

$\Gamma_u^1 \wedge \Gamma_u^2 = \Gamma_u$, $\Delta_u^1 \vee \Delta_u^2 = \Delta_u$. We set $\mathcal{V} = \vee_{\ell \in L_1} \mathcal{V}_\ell$ and then construct the following derivation $\Phi_{[\alpha']tu}$:

$$\frac{\frac{\Phi_t \quad \Theta_u^1}{\Gamma \wedge \Gamma_u^1 \vdash tu : \vee_{\ell \in L_1} \mathcal{V}_\ell \mid \alpha : \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L_2}; \Delta \vee \Delta_u^1}}{\Gamma \wedge \Gamma_u^1 \vdash [\alpha']tu : \# \mid \alpha : \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L_2}; \alpha' : \mathcal{V} \vee \Delta \vee \Delta_u^1}}$$

We have:

$$\begin{aligned} \mathbf{sz}(\Phi_{[\alpha']tu}) &= \mathbf{sz}(\Phi_{tu}) + \mathbf{ar}(\vee_{k \in K} \mathcal{V}_k) \\ &= \mathbf{sz}(\Phi_t) + \mathbf{sz}(\Theta_u^1) + |K| + \mathbf{ar}(\vee_{k \in K} \mathcal{V}_k) \\ &= \mathbf{sz}(\Phi_t) + \mathbf{sz}(\Theta_u^1) + \mathbf{ar}(\langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}) \\ &= \mathbf{sz}(\Phi_{[\alpha]t}) + \mathbf{sz}(\Theta_u^1) \end{aligned}$$

□

Lemma 8.6 (Weighted Subject Reduction for $\lambda_{\mu\alpha}$). Let $\Phi \triangleright \Gamma \vdash o : \mathcal{A} \mid \Delta$. If $o \rightarrow o'$ is a non-erasing step, then $\Phi' \triangleright \Gamma \vdash o' : \mathcal{A} \mid \Delta$ and $\mathbf{sz}(\Phi) > \mathbf{sz}(\Phi')$.

Proof. By induction on the reduction relation \rightarrow . We only show the main cases of reduction at the root, the other ones being straightforward.

- If $o = \mathbf{L}[(\lambda x.t)]u \rightarrow \mathbf{L}[t[x/u]] = o'$: we proceed by induction on \mathbf{L} , by detailing only the case $\mathbf{L} = \square$ as the other one is straightforward.

The derivation Φ has the following form:

$$\Phi = \frac{\frac{\Phi_t \triangleright \Gamma_t; x : \mathcal{I} \vdash t : \mathcal{U} \mid \Delta_t}{\Gamma_t \vdash \lambda x.t : \langle \mathcal{I} \rightarrow \mathcal{U} \rangle \mid \Delta_t} \quad \Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I}^* \mid \Delta}{\Gamma \vdash (\lambda x.t)u : \mathcal{U} \mid \Delta}}$$

where $\Gamma = \Gamma_t \wedge \Gamma_u$, $\Delta = \Delta_t \vee \Delta_u$ and $\mathcal{A} = \mathcal{U}$. We then construct the following derivation Φ' :

$$\frac{\Phi_t \triangleright \Gamma_t; x : \mathcal{I} \vdash t : \mathcal{U} \mid \Delta_t \quad \Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I}^* \mid \Delta_u}{\Gamma_t \wedge \Gamma_u \vdash t[x/u] : \mathcal{U} \mid \Delta_t \vee \Delta_u}}$$

We have:

$$\begin{aligned} \mathbf{sz}(\Phi) &= \mathbf{sz}(\Phi_t) + \mathbf{sz}(\Theta_u) + 2 \\ &> \mathbf{sz}(\Phi_t) + \mathbf{sz}(\Theta_u) = \mathbf{sz}(\Phi') \end{aligned}$$

- If $o = \mathbf{L}[\mu\alpha.c]u \rightarrow \mathbf{L}[\mu\alpha'.c\langle\alpha \parallel \alpha'.u\rangle] = o'$: we proceed by induction on \mathbf{L} , by detailing only the case $\mathbf{L} = \square$ as the other one is straightforward. The derivation Φ has the following form:

$$\Phi = \frac{\frac{\Phi_c \triangleright \Gamma_c \vdash c : \# \mid \alpha : \mathcal{V}_c; \Delta_c}{\Gamma_c \vdash \mu\alpha.c : \mathcal{V}_c \mid \Delta_c} \quad \Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I}_u \mid \Delta_u}{\Gamma_c \wedge \Gamma_u \vdash (\mu\alpha.c)u : \mathcal{U} \mid \Delta_u}}$$

where $\mathcal{V}_c = \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L}$, $\mathcal{I}_u = \wedge_{\ell \in L} \mathcal{I}_\ell^*$, $\mathcal{U} = \vee_{\ell \in L} \mathcal{V}_\ell$, $\Gamma = \Gamma_c \wedge \Gamma_u$ and $\Delta = \Delta_c \vee \Delta_u$. Moreover, Lemma 7.1 gives $L \neq \emptyset$, so that $\wedge_{\ell \in L} \mathcal{I}_\ell^* = (\wedge_{\ell \in L} \mathcal{I}_\ell^*)^*$.

We then construct the following derivation Φ' :

$$\frac{\frac{\Phi_c \quad \Theta_u}{\Gamma' \wedge \Gamma_u \vdash c\langle\alpha \parallel \alpha'.u\rangle : \# \mid \Delta' \vee \Delta_u; \alpha' : \mathcal{U}}{\Gamma' \wedge \Gamma_u \vdash \mu\alpha'.c\langle\alpha \parallel \alpha'.u\rangle : \mathcal{U} \mid \Delta' \vee \Delta_u}}$$

We conclude since $|L| \geq 1$ in the following equation:

$$\begin{aligned} \mathbf{sz}(\Phi') &= \mathbf{sz}(\Phi_{c(\alpha \setminus \alpha'.u)}) + 1 \\ &= \mathbf{sz}(\Phi_c) + \mathbf{sz}(\Theta_u) + |L| - \frac{1}{2} + 1 \\ &= \mathbf{sz}(\Phi_{\mu\alpha.c}) + \mathbf{sz}(\Theta_u) + |L| - \frac{1}{2} \\ &< \mathbf{sz}(\Phi_{\mu\alpha.c}) + \mathbf{sz}(\Theta_u) + |L| = \mathbf{sz}(\Phi) \end{aligned}$$

- If $o = \mathbb{T}\mathbb{T}[x][x/u] \rightarrow \mathbb{T}\mathbb{T}[u][x/u] = o'$, with $|\mathbb{T}\mathbb{T}[x]|_x > 1$. The derivation Φ has the following form:

$$\frac{\Phi'_{\mathbb{T}\mathbb{T}[x]} \triangleright \Gamma'; x : \mathcal{I} \vdash \mathbb{T}\mathbb{T}[x] : \mathcal{U} \mid \Delta' \quad \Theta_u \triangleright \Gamma_u \vdash u : \mathcal{I}^* \mid \Delta_u}{\Gamma' \wedge \Gamma_u \vdash \mathbb{T}\mathbb{T}[x][x/u] : \mathcal{U} \mid \Delta' \vee \Delta_u}$$

Moreover, $|\mathbb{T}\mathbb{T}[x]|_x > 1$ so that Lemma 8.3 applied to $\Phi_{\mathbb{T}\mathbb{T}[x]}$ gives $\mathcal{I} \neq []$ and thus $\mathcal{I}^* = \mathcal{I}$. We can then apply Lemma 8.4 which gives a derivation

$$\Phi_{\mathbb{T}\mathbb{T}[u]} \triangleright \Gamma' \wedge \Gamma_u^1; x : \mathcal{I}_2 \vdash \mathbb{T}\mathbb{T}[u] : \mathcal{U} \mid \Delta' \vee \Delta_u^1$$

where $\mathcal{I} = \mathcal{I}_1 \wedge \mathcal{I}_2$ and $\mathcal{I}_1 \neq []$ and $\Gamma_u = \Gamma_u^1 \wedge \Gamma_u^2$ and $\Delta_u = \Delta_u^1 \vee \Delta_u^2$. Moreover $\Theta_u^1 \triangleright \Gamma_u^1 \vdash u : \mathcal{I}_1 \mid \Delta_u^1$, $\Theta_u^2 \triangleright \Gamma_u^2 \vdash u : \mathcal{I}_2 \mid \Delta_u^2$, and $\mathbf{sz}(\Phi_{\mathbb{T}\mathbb{T}[u]}) = \mathbf{sz}(\Phi_{\mathbb{T}\mathbb{T}[x]}) + \mathbf{sz}(\Theta_u^1) - |\mathcal{I}_1|$.

The hypothesis $|\mathbb{T}[x]|_x > 1$ implies $|\mathbb{T}[u]|_x > 0$, then $\mathcal{I}_2 \neq []$ by Lemma 8.3 applied to $\Phi_{\mathbb{T}\mathbb{T}[u]}$ so that $\mathcal{I}_2^* = \mathcal{I}_2$. We can then construct the derivation Φ' as follows:

$$\frac{\Phi_{\mathbb{T}\mathbb{T}[u]} \quad \Theta_u^2}{\Gamma' \wedge \Gamma \vdash \mathbb{T}\mathbb{T}[u][x/u] : \mathcal{U} \mid \Delta' \wedge \Delta} \text{ (exs)}$$

We conclude since $\mathbf{sz}(\Phi') = \mathbf{sz}(\Phi_{\mathbb{T}\mathbb{T}[u]}) + \mathbf{sz}(\Theta_u^2) =_{L. 8.4} \mathbf{sz}(\Phi_{\mathbb{T}\mathbb{T}[x]}) + \mathbf{sz}(\Theta_u^1) - |\mathcal{I}_1| + \mathbf{sz}(\Theta_u^2) = \mathbf{sz}(\Phi_{\mathbb{T}\mathbb{T}[x]}) + \mathbf{sz}(\Theta_u) - |\mathcal{I}_1| < \mathbf{sz}(\Phi)$.

The step $<$ is justified by $\mathcal{I}_1 \neq []$.

- If $o = \mathbb{T}[x][x/u] \rightarrow \mathbb{T}[u] = o'$, with $|\mathbb{T}[x]|_x = 1$. The derivation Φ has the following form:

$$\frac{\Phi_{\mathbb{T}[x]} \triangleright \Gamma'; x : \mathcal{I} \vdash \mathbb{T}[x] : \mathcal{U} \mid \Delta' \quad \Theta_u \triangleright \Gamma_u \vdash u : \mathcal{I}^* \mid \Delta_u}{\Gamma' \wedge \Gamma_u \vdash \mathbb{T}[x][x/u] : \mathcal{U} \mid \Delta' \vee \Delta_u}$$

Lemma 8.3 applied to $\Phi_{\mathbb{T}[x]}$ gives $\mathcal{I} \neq []$ and thus $\mathcal{I}^* = \mathcal{I}$. We can then apply Lemma 8.4 which gives a derivation

$$\Phi_{\mathbb{T}[u]} \triangleright \Gamma' \wedge \Gamma_u^1; x : \mathcal{I}_2 \vdash \mathbb{T}[u] : \mathcal{U} \mid \Delta' \vee \Delta_u^1$$

where $\mathcal{I} = \mathcal{I}_1 \wedge \mathcal{I}_2$ and $\mathcal{I}_1 \neq []$ and $\Gamma_u = \Gamma_u^1 \wedge \Gamma_u^2$ and $\Delta_u = \Delta_u^1 \vee \Delta_u^2$. Moreover $\Theta_u^1 \triangleright \Gamma_u^1 \vdash u : \mathcal{I}_1 \mid \Delta_u^1$, $\Theta_u^2 \triangleright \Gamma_u^2 \vdash u : \mathcal{I}_2 \mid \Delta_u^2$, and $\mathbf{sz}(\Phi_{\mathbb{T}[u]}) = \mathbf{sz}(\Phi_{\mathbb{T}[x]}) + \mathbf{sz}(\Theta_u^1) - |\mathcal{I}_1|$. By hypothesis $|\mathbb{T}[x]|_x = 1$ so that $|\mathbb{T}[u]|_x = 0$, then $\mathcal{I}_2 = \emptyset$ by Lemma 8.3 applied to $\Phi_{\mathbb{T}[u]}$. Thus, $\mathcal{I} = \mathcal{I}_1$. We then set $\Phi' = \Phi_{\mathbb{T}[u]}$ and conclude since

$$\begin{aligned} \mathbf{sz}(\Phi') &= \mathbf{sz}(\Phi_{\mathbb{T}[u]}) \\ &=_{L. 8.4} \mathbf{sz}(\Phi_{\mathbb{T}[x]}) + \mathbf{sz}(\Theta_u^1) - |\mathcal{I}_1| \\ &= \mathbf{sz}(\Phi_{\mathbb{T}[x]}) + \mathbf{sz}(\Theta_u) - |\mathcal{I}| < \\ &= \mathbf{sz}(\Phi_{\mathbb{T}[x]}) + \mathbf{sz}(\Theta_u) = \mathbf{sz}(\Phi) \end{aligned}$$

The step $<$ is justified by $\mathcal{I} = \mathcal{I}_1 \neq []$.

- If $o = \mathbb{CC}[[\alpha]t]\langle\alpha\backslash\alpha'.u\rangle \rightarrow \mathbb{CC}[[\alpha']tu]\langle\alpha\backslash\alpha'.u\rangle = o'$, with $|\mathbb{CC}[[\alpha]t]|_\alpha > 1$. Then Φ has the following form

$$\frac{\Phi_c \triangleright \Gamma_c \vdash \mathbb{CC}[[\alpha]t] : \# \mid \Delta_c; \alpha : \mathcal{V}' \quad \Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I}_u \mid \Delta_u}{\Gamma_c \wedge \Gamma_u \vdash \mathbb{CC}[[\alpha]t]\langle\alpha\backslash\alpha'.u\rangle : \# \mid \Delta_c \vee \Delta_u \vee \alpha' : \vee_{\ell \in L} \mathcal{V}_\ell}$$

where $c = \mathbb{CC}[[\alpha]t]$, $\mathcal{V}' = \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L}$, $\mathcal{I}_u = (\wedge_{\ell \in L} \mathcal{I}_\ell^*)^*$, $\mathcal{A} = \#$, $\Gamma = \Gamma_c \wedge \Gamma_u$ and $\Delta = \Delta_c \vee \Delta_u \vee \alpha' : \vee_{\ell \in L} \mathcal{V}_\ell$. Since $|\mathbb{CC}[[\alpha]t]|_\alpha > 1$ implies $L \neq \emptyset$ by Lemma 8.3, we have that $\mathcal{I}_u = \wedge_{\ell \in L} \mathcal{I}_\ell^*$. By Lemma 8.5 there are $L_1, L_2, \Gamma_u^1, \Gamma_u^2, \Delta_u^1, \Delta_u^2, \Phi_{\mathbb{CC}[[\alpha']tu]}$ s.t.

- $L = L_1 \uplus L_2$, where $L_1 \neq \emptyset$.
- $\Gamma_u = \Gamma_u^1 \wedge \Gamma_u^2$ and $\Delta_u = \Delta_u^1 \vee \Delta_u^2$,
- $\Theta_u^1 \triangleright \Gamma_u^1 \Vdash u : \wedge_{\ell \in L_1} \mathcal{I}_\ell^* \mid \Delta_u^1$,
- $\Theta_u^2 \triangleright \Gamma_u^2 \Vdash u : \wedge_{\ell \in L_2} \mathcal{I}_\ell^* \mid \Delta_u^2$,
- $\Phi_{\mathbb{CC}[[\alpha']tu]} \triangleright \Gamma_c \wedge \Gamma_u^1 \vdash \mathbb{CC}[[\alpha']tu] : \mathcal{A} \mid \alpha : \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L_2}; \alpha' : \vee_{\ell \in L_1} \mathcal{V}_\ell \vee \Delta_c \vee \Delta_u^1$,
and
- $\mathbf{sz}(\Phi_{\mathbb{CC}[[\alpha']tu]}) = \mathbf{sz}(\Phi_{\mathbb{CC}[[\alpha]t]}) + \mathbf{sz}(\Theta_u^1)$.

Moreover, $|\mathbb{CC}[[\alpha]t]|_\alpha > 1$ implies $|\mathbb{CC}[[\alpha']tu]|_\alpha > 0$ so that $L_2 \neq \emptyset$ holds by Lemma 8.3 and thus $\wedge_{\ell \in L_2} \mathcal{I}_\ell^* = (\wedge_{\ell \in L_2} \mathcal{I}_\ell^*)^*$. Then we can build the following derivation Φ' :

$$\frac{\Phi_{\mathbb{CC}[[\alpha']tu]} \quad \Theta_u^2}{\Gamma' \vdash \mathbb{CC}[[\alpha']tu]\langle\alpha\backslash\alpha'.u\rangle : \# \mid \Delta'}$$

where $\Gamma' = (\Gamma_c \wedge \Gamma_u^1) \wedge \Gamma_u^2 = \Gamma$, $\Delta' = (\alpha' : \vee_{\ell \in L_1} \mathcal{V}_\ell \vee \Delta_c \vee \Delta_u^1) \vee \Delta_u^2 \vee (\alpha' : \vee_{\ell \in L_2} \mathcal{V}_\ell) = \Delta$.

We conclude since

$$\begin{aligned} \mathbf{sz}(\Phi') &= \mathbf{sz}(\Phi_{\mathbb{CC}[[\alpha']tu]}) + \mathbf{sz}(\Theta_u^2) + |L_2| - \frac{1}{2} \\ &=_{L. 8.5} \mathbf{sz}(\Phi_{\mathbb{CC}[[\alpha]t]}) + \mathbf{sz}(\Theta_u^1) + \mathbf{sz}(\Theta_u^2) + |L_2| - \frac{1}{2} \\ &= \mathbf{sz}(\Phi_{\mathbb{CC}[[\alpha]t]}) + \mathbf{sz}(\Theta_u) + |L_2| - \frac{1}{2} \\ &< \mathbf{sz}(\Phi_{\mathbb{CC}[[\alpha]t]}) + \mathbf{sz}(\Theta_u) + |L| - \frac{1}{2} = \mathbf{sz}(\Phi) \end{aligned}$$

The step $<$ is justified because $L_1 \neq \emptyset$ and thus $|L_2| < |L|$.

- If $o = \mathbb{CC}[[\alpha]t]\langle\alpha\backslash\alpha'.u\rangle \rightarrow \mathbb{CC}[[\alpha']tu] = o'$, with $|\mathbb{CC}[[\alpha]t]|_\alpha = 1$. The derivation Φ has the following form

$$\frac{\Phi_c \triangleright \Gamma_c \vdash \mathbb{CC}[[\alpha]t] : \# \mid \Delta_c; \alpha : \mathcal{V}' \quad \Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I}_u \mid \Delta_u}{\Gamma_c \wedge \Gamma_u \vdash \mathbb{CC}[[\alpha]t]\langle\alpha\backslash\alpha'.u\rangle : \# \mid \Delta_c \vee \Delta_u \vee \alpha' : \vee_{\ell \in L} \mathcal{V}_\ell}$$

where $c = \mathbb{CC}[[\alpha]t]$, $\mathcal{V}' = \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L}$, $\mathcal{I}_u = (\wedge_{\ell \in L} \mathcal{I}_\ell^*)^*$, $\mathcal{A} = \#$, $\Gamma = \Gamma_c \wedge \Gamma_u$ and $\Delta = \Delta_c \vee \Delta_u \vee \alpha' : \vee_{\ell \in L} \mathcal{V}_\ell$. Since $|\mathbb{CC}[[\alpha]t]|_\alpha = 1$ implies $L \neq \emptyset$ by Lemma 8.3, we have that $\mathcal{I}_u = \wedge_{\ell \in L} \mathcal{I}_\ell^*$. By Lemma 8.5 there are $L_1, L_2, \Gamma_u^1, \Gamma_u^2, \Delta_u^1, \Delta_u^2, \Phi_{\mathbb{CC}[[\alpha']tu]}$ s.t.

- $L = L_1 \uplus L_2$, where $L_1 \neq \emptyset$.
- $\Gamma_u = \Gamma_u^1 \wedge \Gamma_u^2$ and $\Delta_u = \Delta_u^1 \vee \Delta_u^2$,

- $\Theta_u^1 \triangleright \Gamma_u^1 \Vdash u : \wedge_{\ell \in L_1} \mathcal{I}_\ell^* \mid \Delta_u^1$,
- $\Theta_u^2 \triangleright \Gamma_u^2 \Vdash u : \wedge_{\ell \in L_2} \mathcal{I}_\ell^* \mid \Delta_u^2$,
- $\Phi_{\text{CC}[[\alpha']tu]} \triangleright \Gamma_c \wedge \Gamma_u^1 \vdash \text{CC}[[\alpha']tu] : \mathcal{A} \mid \alpha : \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L_2}; \alpha' : \vee_{\ell \in L_1} \mathcal{V}_\ell \vee \Delta_c \vee \Delta_u^1$,
and
- $\text{sz}(\Phi_{\text{CC}[[\alpha']tu]}) = \text{sz}(\Phi_{\text{CC}[[\alpha]t]}) + \text{sz}(\Theta_u^1)$.

Moreover, $|\mathcal{C}[[\alpha]t]|_\alpha = 1$ implies $|\mathcal{C}[[\alpha']tu]|_\alpha = 0$ so that $L_2 = \emptyset$ and $L = L_1$ holds by Lemma 8.3. Thus, $\Theta_u^1 = \Theta_u$ and so on. We then set $\Phi' = \Phi_{\text{CC}[[\alpha']tu]}$ and conclude since

$$\begin{aligned} \text{sz}(\Phi') &= \text{sz}(\Phi_{\text{CC}[[\alpha']tu]}) \\ &=_{L. 8.5} \text{sz}(\Phi_{\text{CC}[[\alpha]t]}) + \text{sz}(\Theta_u) \\ &< \text{sz}(\Phi_{\text{CC}[[\alpha]t]}) + \text{sz}(\Theta_u) + |L| - \frac{1}{2} = \text{sz}(\Phi) \end{aligned}$$

The step $<$ is justified because $L \neq \emptyset$, so that $|L| \geq 1$ implies $|L| - \frac{1}{2} > 0$. □

8.2.2 Backward Properties

As in the implicit case (Sec. 7.3.2), subject expansion for non-erasing $\lambda_{\mu x}$ -step relies on **(Linear) Reverse Substitution** and **(Linear) Reverse Replacement Lemmas**: if $\Phi' \triangleright \Gamma \vdash o' : \mathcal{A} \mid \Delta$ and o' has been obtained from o by substituting one occurrence of x by u (or one subcommand $[\alpha]t$ by $[\alpha']tu$), then, informally speaking, it is possible to decompose Φ' into a regular derivation Φ_0 typing o and an auxiliary derivation Θ_u typing u .

Lemma 8.7 (Reverse Partial Substitution). Let $\Phi \triangleright \Gamma \vdash \text{OT}[[u]] : \mathcal{A} \mid \Delta$, where $x \notin \text{fv}(u)$. Then, $\exists \Gamma_0, \exists \Delta_0, \exists \mathcal{I}_0 \neq [], \exists \Gamma_u, \exists \Delta_u$ such that

- $\Gamma = \Gamma_0 \wedge \Delta_u$,
- $\Delta = \Delta_0 \vee \Delta_u$,
- $\Phi_{\text{OT}[[x]]} \triangleright \Gamma_0 \wedge x : \mathcal{I}_0 \vdash \text{OT}[[x]] : \mathcal{A} \mid \Delta_0$
- $\triangleright \Gamma_u \Vdash u : \mathcal{I}_0 \mid \Delta_u$.

Proof. The proof is by induction on the context OT . For this induction to work, we need as usual to adapt the statement for auxiliary derivations. We only show the case $\text{OT} = \square$ since all the other ones are straightforward and rely on suitable partitions of the contexts in the premises. So assume $\text{OT} = \square$, then $\mathcal{A} = \mathcal{U}$ for some \mathcal{U} . We set $\Gamma_0 = \Delta_0 = \emptyset$, $\mathcal{I}_0 = [\mathcal{U}]$, $\Gamma_u = \Gamma$, $\Delta_u = \Delta$ (so that $\triangleright \Gamma_u \Vdash u : \mathcal{I}_0 \mid \Delta_u$ holds by using the (\wedge) rule), and

$$\Phi_x = \frac{}{x : [\mathcal{U}] \vdash x : \mathcal{U} \mid \emptyset}$$

The claimed set and context equalities trivially hold. □

Lemma 8.8 (Reverse Partial Replacement). Let $\Gamma \vdash \text{OC}[[\alpha']tu] : \mathcal{A} \mid \alpha' : \mathcal{V}; \Delta$, where $\alpha, \alpha' \notin \text{fn}(u)$. Then $\exists \Gamma_0, \exists \Delta_0, \exists \mathcal{V}_0, \exists K \neq \emptyset, \exists (\mathcal{I}_k)_{k \in K}, \exists (\mathcal{V}_k)_{k \in K}, \Gamma_u$, and Δ_u such that

- $\Gamma = \Gamma_0 \wedge \Gamma_u$,
- $\Delta = \Delta_0 \vee \Delta_u$,
- $\mathcal{V} = \mathcal{V}_0 \vee_{k \in K} \mathcal{V}_k$,
- $\triangleright \Gamma_0 \vdash \mathbf{OC}[[\alpha]t] : \mathcal{A} \mid \alpha' : \mathcal{V}_0; \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K} \vee \Delta_0$, and
- $\triangleright \Gamma_u \Vdash u : \wedge_{k \in K} \mathcal{I}_k^* \mid \Delta_u$

Proof. The proof is by induction on the context \mathbf{OC} . For this induction to work, we need as usual to adapt the statement for auxiliary derivations. Notice that $\mathcal{V} \neq \langle \rangle$ by Lemma 8.3, since $\alpha' \in \mathbf{fn}(\mathbf{OC}[[\alpha']tu])$. We only show the case $\mathbf{OC} = \square$ since all the other ones are straightforward. So assume $\mathbf{OC} = \square$. Then the derivation of $[\alpha']tu$ has the following form, where $K \neq \emptyset$:

$$\frac{\frac{\Phi_t \triangleright \Gamma_0 \vdash t : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K} \mid \alpha' : \mathcal{V}_0; \Delta_0 \quad \triangleright \Gamma_u \Vdash u : \wedge_{k \in K} \mathcal{I}_k^* \mid \Delta_u}{\Gamma_0 \wedge \Gamma_u \vdash tu : \vee_{k \in K} \mathcal{V}_k \mid \alpha' : \mathcal{V}_0; \Delta_0 \vee \Delta_u}}{\Gamma_0 \wedge \Gamma_u \vdash [\alpha']tu : \# \mid \alpha' : \mathcal{V}_0 \vee_{k \in K} \mathcal{V}_k; \Delta_0 \vee \Delta_u}$$

where $\Gamma = \Gamma_0 \wedge \Gamma_u$, and $\Delta = \Delta_0 \vee \Delta_u$ and $\mathcal{V} = \mathcal{V}_0 \vee_{k \in K} \mathcal{V}_k$.

We then construct the following derivation :

$$\frac{\Phi_t}{\Gamma \vdash [\alpha]t : \# \mid \alpha' : \mathcal{V}_0; \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K} \vee \Delta_0}$$

Thus, we have all the claimed set and context equalities. \square

Lemma 8.9 (Subject Expansion for $\lambda_{\mu x}$). Let $\Phi' \triangleright \Gamma \vdash o' : \mathcal{A} \mid \Delta$. If $o \rightarrow_{\lambda_{\mu x}} o'$ (i.e. a non-erasing $\lambda_{\mu x}$ -step), then $\Phi \triangleright \Gamma \vdash o : \mathcal{A} \mid \Delta$.

Proof. By induction on the non erasing reduction relation $\rightarrow_{\lambda_{\mu x}}$. We only show the main cases of non-erasing reduction at the root, the other ones being straightforward.

- If $o = \mathbf{L}[[\lambda x.t]u] \rightarrow \mathbf{L}[t\langle x \setminus u \rangle] = o'$, we proceed by induction on \mathbf{L} , by detailing only the case $\mathbf{L} = \square$ as the other one is straightforward.

The derivation Φ' has the following form :

$$\frac{\Phi_t \triangleright \Gamma_t; x : \mathcal{I} \vdash t : \mathcal{U} \mid \Delta_t \quad \Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I}^* \mid \Delta_u}{\Gamma \vdash t\langle x \setminus u \rangle : \mathcal{U} \mid \Delta}$$

We then construct the following derivation Φ :

$$\frac{\frac{\Phi_t \triangleright \Gamma_t; x : \mathcal{I} \vdash t : \mathcal{U} \mid \Delta_t}{\Gamma_t \vdash \lambda x.t : \mathcal{I} \rightarrow \mathcal{U} \mid \Delta_t} \quad \Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I}^* \mid \Delta_u}{\Gamma \vdash (\lambda x.t)u : \mathcal{U} \mid \Delta}$$

- If $o = \mathbf{L}[[\mu \alpha.c]u] \rightarrow \mathbf{L}[\mu \alpha'.c\langle \alpha \setminus \alpha'.u \rangle] = o'$, where α' is fresh, then we proceed by induction on \mathbf{L} , by detailing only the case $\mathbf{L} = \square$ as the other one is straightforward. Then Φ' has the following form :

$$\frac{\frac{\Phi_c \triangleright \Gamma_c \vdash c : \# \mid \alpha : \mathcal{V}_\alpha; \Delta_c \quad \Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I}_u \mid \Delta_u}{\Gamma_c \wedge \Gamma_u \vdash c\langle \alpha \setminus \alpha'.u \rangle : \# \mid \Delta_c \vee \Delta_u; \alpha' : \mathcal{V}_{\alpha'}}}{\Gamma_c \wedge \Gamma_u \vdash \mu \alpha'.c\langle \alpha \setminus \alpha'.u \rangle : (\mathcal{V}_{\alpha'})^* \mid \Delta_c \vee \Delta_u}$$

where $\mathcal{V}_\alpha = \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L}$, $\mathcal{I}_u = (\wedge_{\ell \in L} \mathcal{I}_\ell^*)^*$, $\mathcal{V}_{\alpha'} = \vee_{\ell \in L} \mathcal{V}_\ell$, $\mathcal{A} = (\mathcal{V}_{\alpha'})^* = (\vee_{\ell \in L} \mathcal{V}_\ell)^*$, $\Gamma = \Gamma_c \wedge \Gamma_u$ and $\Delta = \Delta_c \vee \Delta_u$. Notice that the name assignment of the judgment typing $c \langle \alpha \backslash \alpha'. u \rangle$ has the form $\Delta_c \vee \Delta_u; \alpha' : \mathcal{V}_{\alpha'}$ since α' is a fresh name by hypothesis, so that $\alpha' \notin \text{dom}(\Delta_c \vee \Delta_u)$ holds by Lemma 8.3. We now consider two cases:

If $L \neq \emptyset$, then $\langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L}^* = \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L}$, $(\wedge_{\ell \in L} \mathcal{I}_\ell^*)^* = \wedge_{\ell \in L} \mathcal{I}_\ell^*$, $\mathcal{A} = (\vee_{\ell \in L} \mathcal{V}_\ell)^* = \vee_{\ell \in L} \mathcal{V}_\ell$, so that we construct the following derivation Φ :

$$\frac{\Phi_c \triangleright \Gamma_c \vdash \mu\alpha.c : \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L} \mid \Delta_c \quad \Theta_u \triangleright \Gamma_u \Vdash u : \wedge_{\ell \in L} \mathcal{I}_\ell^* \mid \Delta_u}{\Gamma_c \wedge \Gamma_u \vdash (\mu\alpha.c)u : \vee_{\ell \in L} \mathcal{V}_\ell \mid \Delta_c \vee \Delta_u}$$

If $L = \emptyset$, then let $(\wedge_{\ell \in L} \mathcal{I}_\ell^*)^*$ (resp. $(\vee_{\ell \in L} \mathcal{V}_\ell)^*$) be of the form $[\mathcal{U}]$ (resp. $\langle \sigma \rangle$) for some arbitrary \mathcal{U} (resp. σ). Then we choose $\langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L}^*$ to be $\langle [\mathcal{U}] \rightarrow \langle \sigma \rangle \rangle$. We then construct the following derivation Φ :

$$\frac{\Phi_c \triangleright \Gamma_c \vdash \mu\alpha.c : \langle [\mathcal{U}] \rightarrow \langle \sigma \rangle \rangle \mid \Delta_c \quad \Theta_u \triangleright \Gamma_u \Vdash u : [\mathcal{U}] \mid \Delta_u}{\Gamma_c \wedge \Gamma_u \vdash (\mu\alpha.c)u : \langle \sigma \rangle \mid \Delta_c \vee \Delta_u}$$

We conclude since $\mathcal{A} = \langle \sigma \rangle$.

- If $o = \text{OT}[x][x/u] \rightarrow \text{OT}[u][x/u] = o'$, with $|\text{OT}[x]|_x > 1$. The derivation Φ' has the following form:

$$\frac{\Phi_{\text{OT}[u]} \triangleright \Gamma_*; x : \mathcal{I} \vdash \text{OT}[u] : \mathcal{A} \mid \Delta_* \quad \Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I}^* \mid \Delta_u}{\Gamma_* \wedge \Gamma_u \vdash \text{OT}[u][x/u] : \mathcal{A} \mid \Delta_* \vee \Delta_u}$$

where $x \in \text{fv}(\text{OT}[u])$ implies $\mathcal{I} \neq []$ by Lemma 8.3, so that $\mathcal{I}^* = \mathcal{I}$.

By Lemma 8.7 applied to $\Phi_{\text{OT}[u]}$, we have $\Gamma'_0, \Delta_0, \mathcal{I}_0 \neq [], \Gamma'_u, \Delta'_u$ such that

- $\Gamma_*; x : \mathcal{I} = \Gamma'_0 \wedge \Gamma'_u$,
- $\Delta_* = \Delta_0 \vee \Delta'_u$,
- $\Phi_{\text{OT}[x]} \triangleright \Gamma'_0 \wedge x : \mathcal{I}_0 \vdash \text{OT}[x] : \mathcal{A} \mid \Delta_0$
- $\triangleright \Gamma'_u \Vdash u : \mathcal{I}_0 \mid \Delta'_u$.

We set $\mathcal{I}^+ = \mathcal{I} \wedge \mathcal{I}_0$, $\Gamma_u^+ = \Gamma_u \wedge \Gamma'_u$, $\Delta_u^+ = \Delta_u \vee \Delta'_u$. Thus, in particular, $(\mathcal{I}^+)^* = \mathcal{I}^+$. By Lemma 8.3, $x \notin \text{dom}(\Gamma'_u)$, so that $\Gamma'_0 = \Gamma_0; x : \mathcal{I}$ for some Γ_0 and thus $\Gamma'_0 \wedge x : \mathcal{I}_0 = \Gamma_0; x : \mathcal{I}^+$. By Lemma 7.3 there is a derivation $\Theta_u^+ \triangleright \Gamma_u^+ \Vdash u : \mathcal{I}^+ \mid \Delta_u^+$. We then construct the following derivation Φ :

$$\frac{\Phi_{\text{OT}[x]} \quad \Theta_u^+ \triangleright \Gamma_u^+ \Vdash u : \mathcal{I}^+ \mid \Delta_u^+}{\Gamma_0 \wedge \Gamma_u^+ \vdash \text{OT}[x][x/u] : \mathcal{A} \mid \Delta_0 \vee \Delta_u^+} \text{ (exs)}$$

We conclude since $\Gamma_0 \wedge \Gamma_u^+ = \Gamma_0 \wedge \Gamma'_u \wedge \Gamma_u = \Gamma_* \wedge \Gamma_u = \Gamma$ and $\Delta_0 \vee \Delta_u^+ = \Delta_0 \vee \Delta'_u \vee \Delta_u = \Delta_* \vee \Delta_u = \Delta$.

- If $o = \text{OT}[x][x/u] \rightarrow \text{OT}[u] = o'$, with $|\text{OT}[x]|_x = 1$. The derivation Φ' ends with $\Gamma \vdash \text{OT}[u] : \mathcal{A} \mid \Delta$ where $x \notin \text{dom}(\Gamma)$ by Lemma 8.3. By Lemma 8.7 applied to Φ' , we have $\Gamma_0, \Delta_0, \mathcal{I}_0 \neq [], \Gamma_u, \Delta_u$ such that

- $\Gamma = \Gamma_0 \wedge \Gamma_u$,

- $\Delta = \Delta_0 \vee \Delta_u$,
- $\Phi_{\text{OT}[x]} \triangleright \Gamma_0 \wedge x : \mathcal{I}_0 \vdash \text{OT}[x] : \mathcal{A} \mid \Delta_0$
- $\triangleright \Gamma_u \Vdash u : \mathcal{I}_0 \mid \Delta_u$.

Thus, in particular, $\mathcal{I}_0^* = \mathcal{I}_0$. Since $x \notin \text{dom}(\Gamma)$, $x \notin \text{dom}(\Gamma_0)$, so that $\Gamma_0 \wedge x : \mathcal{I}_0 = \Gamma_0; x : \mathcal{I}_0$. We then construct the following derivation Φ :

$$\frac{\Phi_{\text{OT}[x]} \quad \triangleright \Gamma_u \Vdash u : \mathcal{I}_0 \mid \Delta_u}{\Gamma_0 \wedge \Gamma_u \vdash \text{OT}[x][x/u] : \mathcal{U} \mid \Delta_0 \vee \Delta_u}$$

We conclude since $\Gamma = \Gamma_0 \wedge \Gamma_u$ and $\Delta = \Delta_0 \vee \Delta_u$.

- If $o = \text{OC}[[\alpha]t]\langle \alpha \backslash \alpha'.u \rangle \rightarrow \text{OC}[[\alpha']tu]\langle \alpha \backslash \alpha'.u \rangle = o'$, with $|\text{OC}[[\alpha]t]|_\alpha > 1$. Then Φ' has the following form :

$$\frac{\Phi'_0 \triangleright \Gamma_* \vdash \text{OC}[[\alpha']tu] : \mathcal{A} \mid \Delta_*; \alpha' : \mathcal{V}_{\alpha'}; \alpha : \mathcal{V}_\alpha \quad \Theta_u \triangleright \Gamma_u \Vdash u : \mathcal{I}_u \mid \Delta_u}{\Gamma_* \wedge \Gamma_u \vdash \text{OC}[[\alpha']tu]\langle \alpha \backslash \alpha'.u \rangle : \mathcal{A} \mid (\Delta_*; \alpha' : \mathcal{V}_{\alpha'}) \vee \Delta_u \vee \alpha' : \mathcal{V}}$$

where $\mathcal{V}_\alpha = \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L}$, $\mathcal{I}_u = (\wedge_{\ell \in L} \mathcal{I}_\ell^*)^*$, $\mathcal{V} = \vee_{\ell \in L} \mathcal{V}_\ell$, $\Gamma = \Gamma_* \wedge \Gamma_u$ and $\Delta = (\Delta_*; \alpha' : \mathcal{V}_{\alpha'}) \vee \Delta_u \vee \alpha' : \mathcal{V} = (\Delta_* \vee \Delta_u; \alpha' : \mathcal{V}_{\alpha'} \vee \mathcal{V})$ since $\alpha' \notin \text{fn}(u)$ implies $\alpha' \notin \text{dom}(\Delta_u)$. Since $\alpha \in \text{fn}(\text{OC}[[\alpha']tu])$, then $L \neq \emptyset$ by Lemma 8.3, so that $\mathcal{I}_u = \wedge_{\ell \in L} \mathcal{I}_\ell^*$.

By Lemma 8.8 applied to Φ'_0 , we have $\Gamma_0, \Delta'_0, \Phi_0, \mathcal{V}_0, K \neq \emptyset, (\mathcal{I}_k)_{k \in K}, (\mathcal{V}_k)_{k \in K}, \Gamma'_u, \Delta'_u$, and Θ'_u such that

- $\Gamma_* = \Gamma_0 \wedge \Gamma'_u$,
- $\Delta_*; \alpha : \mathcal{V}_\alpha = \Delta'_0 \vee \Delta'_u$,
- $\mathcal{V}_{\alpha'} = \mathcal{V}_0 \vee_{k \in K} \mathcal{V}_k$,
- $\Phi_0 \triangleright \Gamma_0 \vdash \text{OC}[[\alpha]t] : \mathcal{A} \mid \alpha' : \mathcal{V}_0; \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K} \vee \Delta'_0$ and
- $\triangleright \Gamma'_u \Vdash u : \wedge_{k \in K} \mathcal{I}_k^* \mid \Delta'_u$

We set $L^+ = L \uplus K$, $\Gamma_u^+ = \Gamma_u \wedge \Gamma'_u$, $\Delta_u^+ = \Delta_u \vee \Delta'_u$ and $\mathcal{I}_u^+ = \wedge_{\ell \in L^+} \mathcal{I}_\ell^*$. By Lemma 8.3, $\alpha \notin \text{dom}(\Delta'_u)$, so that $\Delta'_0 = \Delta_0; \alpha : \mathcal{V}_\alpha$ for some Δ_0 and $\alpha' : \mathcal{V}_0; \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K} \vee \Delta'_0 = \alpha' : \mathcal{V}_0; \alpha : \langle \mathcal{I}_\ell \rightarrow \mathcal{V}_\ell \rangle_{\ell \in L^+}; \Delta_0$ since $\alpha' \notin \text{dom}(\Delta'_0)$. By Lemma 7.3, there is $\Theta_u^+ \triangleright \Gamma_u^+ \Vdash u : \mathcal{I}_u^+ \mid \Delta_u^+$. We then construct the following derivation Φ :

$$\frac{\Phi_0 \quad \Theta_u^+}{\Gamma_0 \wedge \Gamma_u^+ \vdash \text{OC}[[\alpha]t]\langle \alpha \backslash \alpha'.u \rangle : \mathcal{A} \mid \alpha' : \mathcal{V}_0 \vee_{\ell \in L^+} \mathcal{V}_\ell; \Delta_0 \vee \Delta_u^+}$$

We conclude since $\Gamma_0 \wedge \Gamma_u^+ = \Gamma_0 \wedge \Gamma'_u \wedge \Gamma_u = \Gamma_* \wedge \Gamma_u = \Gamma$, $\Delta_0 \vee \Delta_u^+ = \Delta_0 \vee \Delta'_u \vee \Delta_u = \Delta_* \vee \Delta_u$ and $\mathcal{V}_0 \vee_{\ell \in L^+} \mathcal{V}_\ell = \mathcal{V}_0 \vee_{k \in K} \mathcal{V}_k \vee_{\ell \in L} \mathcal{V}_\ell = \mathcal{V}_{\alpha'} \vee_{\ell \in L} \mathcal{V}_\ell = \mathcal{V}_{\alpha'} \vee \mathcal{V}$.

- If $o = \text{OC}[[\alpha]t]\langle \alpha \backslash \alpha'.u \rangle \rightarrow \text{OC}[[\alpha']tu] = o'$, with $|\text{OC}[[\alpha]t]|_\alpha = 1$, then the derivation Φ' necessarily ends with the judgment $\Gamma \vdash \text{OC}[[\alpha']tu] : \mathcal{A} \mid \Delta_*; \alpha' : \mathcal{V}$, where $\Delta = \Delta_*; \alpha' : \mathcal{V}$.

By Lemma 8.8 applied to Φ' , we have $\Gamma_0, \Delta_0, \mathcal{V}_0, K \neq \emptyset, (\mathcal{I}_k)_{k \in K}, (\mathcal{V}_k)_{k \in K}, \Gamma_u, \Delta_u$, and Θ_u such that

- $\Gamma = \Gamma_0 \wedge \Gamma_u$,
- $\Delta_* = \Delta_0 \vee \Delta_u$,
- $\mathcal{V} = \mathcal{V}_0 \vee_{k \in K} \mathcal{V}_k$,
- $\Phi_0 \triangleright \Gamma_0 \vdash \mathbf{OC}[[\alpha]t] : \mathcal{A} \mid \alpha' : \mathcal{V}_0; \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K} \vee \Delta_0$ and
- $\Theta_u \triangleright \Gamma_u \Vdash u : \wedge_{k \in K} \mathcal{I}_k^* \mid \Delta_u$

Notice that $K \neq \emptyset$ implies $(\wedge_{k \in K} \mathcal{I}_k^*)^* = \wedge_{k \in K} \mathcal{I}_k^*$. Moreover, by Lemma 8.3, since $\alpha \notin \mathbf{fn}(\mathbf{OC}[[\alpha]t])$, then $\alpha \notin \mathbf{dom}(\Delta)$, thus $\alpha \notin \mathbf{dom}(\Delta_0)$ and $\alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K} \vee \Delta_0 = \alpha : \langle \mathcal{I}_k \rightarrow \mathcal{V}_k \rangle_{k \in K}; \Delta_0$. We then construct Φ :

$$\frac{\Phi_0 \quad \Theta_u}{\Gamma \vdash \mathbf{OC}[[\alpha]t] \langle \alpha \backslash \alpha'.u \rangle : \mathcal{A} \mid (\Delta_0; \alpha' : \mathcal{V}_0) \vee \Delta_u \vee \alpha' : \vee_{k \in K} \mathcal{V}_k}$$

We conclude since $\alpha' \notin \mathbf{fn}(u)$ implies $\alpha' \notin \mathbf{dom}(\Delta_u)$ by Lemma 8.3 so that $(\Delta_0; \alpha' : \mathcal{V}_0) \vee \Delta_u \vee \alpha' : \vee_{k \in K} \mathcal{V}_k = \Delta_0 \vee \Delta_u; \alpha' : \mathcal{V}_0 \vee_{k \in K} \mathcal{V}_k = \Delta_*; \alpha' : \mathcal{V} = \Delta$ as desired. \square

8.3 Strongly Normalizing $\lambda_{\mu\mathbf{x}}$ -Objects

In this section we show a characterization of the set of strongly $\lambda_{\mu\mathbf{x}}$ -normalizing terms by means of typability. The proof is done in several steps. The first key point is the characterization of the set of strongly $\lambda_{\overline{\mu\mathbf{x}}}$ -normalizing terms (instead of $\lambda_{\mu\mathbf{x}}$ -normalizing terms). For that, SR and SE lemmas for the type system are used, and an inductive characterization of the set $\mathbf{SN}(\lambda_{\overline{\mu\mathbf{x}}})$ turns out to be helpful to obtain them. The second key point is the equivalence between strongly $\lambda_{\overline{\mu\mathbf{x}}}$ and $\lambda_{\mu\mathbf{x}}$ -normalizing terms. While the inclusion $\mathbf{SN}(\lambda_{\mu\mathbf{x}}) \subseteq \mathbf{SN}(\lambda_{\overline{\mu\mathbf{x}}})$ is straightforward, the fact that every \mathbf{w} -reduction step can be *postponed* w.r.t. any $\lambda_{\overline{\mu\mathbf{x}}}$ -step (Lemma 8.11) turns out to be crucial to show $\mathbf{SN}(\lambda_{\overline{\mu\mathbf{x}}}) \subseteq \mathbf{SN}(\lambda_{\mu\mathbf{x}})$.

These technical tools are now used to prove that $\mathbf{SN}(\lambda_{\overline{\mu\mathbf{x}}})$ coincides exactly with the set of typable terms. To close the picture, *i.e.* to show that also $\mathbf{SN}(\lambda_{\mu\mathbf{x}})$ coincides with the set of typable terms, we establish an equivalence between $\mathbf{SN}(\lambda_{\overline{\mu\mathbf{x}}})$ and $\mathbf{SN}(\lambda_{\mu\mathbf{x}})$. This is done constructively thanks to the use of an inductive definition for $\mathbf{SN}(\lambda_{\overline{\mu\mathbf{x}}})$. Indeed, the inductive set $\mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$ is the smallest subset of $\mathcal{O}_{\lambda_{\overline{\mu\mathbf{x}}}}$ that satisfies the following properties:

- (1) If $t_1, \dots, t_n \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$ ($n \geq 0$), then $xt_1 \dots t_n \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$.
- (2) If $t \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$, then $\lambda x.t \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$.
- (3) If $c \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$, then $\mu\alpha.c \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$.
- (4) If $t \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$, then $[\alpha]t \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$.
- (5) If $t, s \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$ and $|t|_x = 0$, then $t[x/s] \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$.
- (6) If $c, s \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$ and $|c|_\alpha = 0$, then $c\langle \alpha \backslash \alpha'.s \rangle \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$.
- (7) If $u\langle x \backslash v \rangle \vec{t} \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$, then $(\lambda x.u)v\vec{t} \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$.
- (8) If $(\mu\alpha'.c\langle \alpha \backslash \alpha'.v \rangle)\vec{t} \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$, then $(\mu\alpha.c)v\vec{t} \in \mathbf{ISN}(\lambda_{\overline{\mu\mathbf{x}}})$.

- (9) If $\mathbb{TT}[[u]]\langle x \setminus u \rangle \in \text{ISN}(\lambda_{\mu\bar{x}})$, and $|\mathbb{TT}[[x]]|_x > 1$ then $\mathbb{TT}[[x]]\langle x \setminus u \rangle \in \text{ISN}(\lambda_{\mu\bar{x}})$.
- (10) If $\mathbb{CC}[[\alpha']tv]\langle \alpha \setminus \alpha'.v \rangle \in \text{ISN}(\lambda_{\mu\bar{x}})$, and $|\mathbb{CC}[[\alpha]t]|_\alpha > 1$ then $\mathbb{CC}[[\alpha]t]\langle \alpha \setminus \alpha'.v \rangle \in \text{ISN}(\lambda_{\mu\bar{x}})$.
- (11) If $\mathbb{TT}[[u]] \in \text{ISN}(\lambda_{\mu\bar{x}})$, and $|\mathbb{TT}[[x]]|_x = 1$ then $\mathbb{TT}[[x]][x/u] \in \text{ISN}(\lambda_{\mu\bar{x}})$.
- (12) If $\mathbb{CC}[[\alpha']tv] \in \text{ISN}(\lambda_{\mu\bar{x}})$, and $|\mathbb{CC}[[\alpha]t]|_\alpha = 1$ then $\mathbb{CC}[[\alpha]t]\langle \alpha \setminus \alpha'.v \rangle \in \text{ISN}(\lambda_{\mu\bar{x}})$.
- (13) If $(tu)\langle x \setminus s \rangle \in \text{ISN}(\lambda_{\mu\bar{x}})$ and $|u|_x = 0$, then $t\langle x \setminus s \rangle u \in \text{ISN}(\lambda_{\mu\bar{x}})$.

It is not surprising that $\text{ISN}(\lambda_{\mu\bar{x}})$ turns out to be equivalent to $\text{SN}(\lambda_{\mu\bar{x}})$, a property which considerably simplifies the proof of Theorem 8.1.

Lemma 8.10. $\text{SN}(\lambda_{\mu\bar{x}}) = \text{ISN}(\lambda_{\mu\bar{x}})$

Proof. Given $o \in \text{SN}(\lambda_{\mu\bar{x}})$, we show $o \in \text{ISN}(\lambda_{\mu\bar{x}})$ by induction on $\langle \eta_{\lambda_{\mu\bar{x}}}(o), |o| \rangle$. The converse uses induction on the definition of $\text{ISN}(\lambda_{\mu\bar{x}})$ and the fact that $\eta_{\lambda_{\mu\bar{x}}}(t[x/s]u) = \eta_{\lambda_{\mu\bar{x}}}((tu)[x/s])$ for the last case (so that $(tu)[x/s] \in \text{SN}(\lambda_{\mu\bar{x}})$ iff $t[x/s]u \in \text{SN}(\lambda_{\mu\bar{x}})$). \square

In order to infer $\text{SN}(\lambda_{\mu\bar{x}}) \subseteq \text{SN}(\lambda_{\mu x})$, the following postponement property is crucial.

Lemma 8.11 (Postponement). Let $o \in \mathcal{O}_{\lambda_{\mu x}}$. If $o \rightarrow_{\mathfrak{w}}^+ \rightarrow_{\lambda_{\mu\bar{x}}} o'$ then $o \rightarrow_{\lambda_{\mu\bar{x}}} \rightarrow_{\mathfrak{w}}^+ o'$.

Proof. We first show by cases $o \rightarrow_{\mathfrak{w}} \rightarrow_{\lambda_{\mu\bar{x}}} o'$ implies $o \rightarrow_{\lambda_{\mu\bar{x}}} \rightarrow_{\mathfrak{w}}^+ o'$. Then, the statement holds by induction on the number of \mathfrak{w} -steps from o . \square

Lemma 8.12 (From $\lambda_{\mu\bar{x}}$ to $\lambda_{\mu x}$). Let $o \in \mathcal{O}_{\lambda_{\mu x}}$. If $o \in \text{SN}(\lambda_{\mu\bar{x}})$, then $o \in \text{SN}(\lambda_{\mu x})$.

Proof. We show that any reduction sequence $\rho : o \rightarrow_{\lambda_{\mu x}} \dots$ is finite by induction on the pair $\langle o, n \rangle$, where n is the maximal integer such that ρ can be decomposed as $\rho : o \rightarrow_{\mathfrak{w}}^n o' \rightarrow_{\lambda_{\mu\bar{x}}} o'' \rightarrow \dots$ (this is well-defined since $\rightarrow_{\mathfrak{w}}$ is trivially terminating). We compare the pair $\langle o, n \rangle$ using $\rightarrow_{\lambda_{\mu\bar{x}}}$ for the first component (this is well-founded since $o \in \text{SN}(\lambda_{\mu\bar{x}})$ by hyp.) and the standard order on natural numbers for the second one. When the reduction sequence starts with at least one \mathfrak{w} -step we conclude by Lemma 8.11. All the other cases are straightforward. \square

We conclude with the main theorem of this section:

Theorem 8.1. Let $o \in \mathcal{O}_{\lambda_{\mu x}}$. Then $o \in \text{SN}(\lambda_{\mu x})$ iff o is typable. Moreover, if o is $\mathcal{S}_{\lambda_{\mu x}}$ -typable with a derivation Π , then $\mathbf{sz}(\Pi)$ gives an upper bound to the maximal length of a reduction sequence starting at o .

Proof. Let $\Phi \triangleright \Gamma \vdash o : \tau \mid \Delta$. Assume $o \notin \text{SN}(\lambda_{\mu\bar{x}})$ so that $\exists \infty$ sequence $o = o_0 \rightarrow_{\lambda_{\mu\bar{x}}} o_1 \rightarrow_{\lambda_{\mu\bar{x}}} o_2 \rightarrow_{\lambda_{\mu\bar{x}}} \dots$. By Lemma 8.6 $\Phi_i \triangleright \Gamma \vdash o_i : \tau \mid \Delta$ for every i , and there exists an infinite sequence $\mathbf{sz}(\Phi_0) > \mathbf{sz}(\Phi_1) > \mathbf{sz}(\Phi_2) > \dots$, which leads to a contradiction because $\mathbf{sz}(_)$ is a half-integer ≥ 1 . Therefore, $o \in \text{SN}(\lambda_{\mu\bar{x}}) \subseteq_{L. 8.12} \text{SN}(\lambda_{\mu x})$.

For the converse, $o \in \text{SN}(\lambda_{\mu x}) \subseteq \text{SN}(\lambda_{\mu\bar{x}})$ because $\rightarrow_{\lambda_{\mu\bar{x}}} \subseteq \rightarrow_{\lambda_{\mu x}}$. We then show that $o \in \text{SN}(\lambda_{\mu\bar{x}})$ implies o is typable. For that, we use the equality $\text{SN}(\lambda_{\mu\bar{x}}) =_{L. 8.10} \text{ISN}(\lambda_{\mu\bar{x}})$ to reason by induction on $t \in \text{ISN}(\lambda_{\mu\bar{x}})$. The cases (1)-(6) and (13) are straightforward while the cases (7)-(12) uses Lemma 8.9 (Partial Subject Expansion). \square

It is worth noticing that the proof of Theorem 8.1 is self-contained: we do not use at all the previous characterization of strongly-normalizing objects in the λ_{μ} -calculus that we have developed in Sec. 7.4. We remark however that an alternative proof of this theorem can be given in terms of the projection function defined in Sec. 8.1.2, an appropriate PSN-like property [58], and Theorem 7.2.

8.4 Conclusion

Chapter 7 provides two quantitative type assignment systems $\mathcal{H}_{\lambda_\mu}$ and $\mathcal{S}_{\lambda_\mu}$ for λ_μ , characterizing, respectively, head and strongly normalizing terms. We have shown that whenever o is typable in system $\mathcal{H}_{\lambda_\mu}$, then we can extract a measure from its type derivation which provides an upper bounds to the length of the head-reduction strategy starting at o . The same happens with system $\mathcal{S}_{\lambda_\mu}$ with respect to the maximal length of a reduction sequence starting at o : indeed, the system $\mathcal{S}_{\lambda_\mu}$ endowed with weakening axioms enjoys full subject reduction (on erasing and non-erasing steps), and $\mathcal{S}_{\lambda_\mu}$ can be embedded in such a system by preserving the size of derivations.

The construction of these typing systems suggests the definition of a resource aware calculus, coming along with the corresponding extensions of the typing systems presented in Chapter 7. This leads us to implement in Chapter 8 a small step operational semantics for classical natural deduction. Such a calculus can be seen as an extension of the *substitution at a distance paradigm* [2, 3] to the classical case. From the type-theoretic point of view, $\lambda_{\mu\mathbf{x}}$ can be shown to be compatible with a natural extension of the quantitative typing systems defined for λ_μ . Moreover, strong normalization in $\lambda_{\mu\mathbf{x}}$ can be characterized by means of typability in the extended typing systems. In both cases (λ_μ and $\lambda_{\mu\mathbf{x}}$), the characterization proofs are obtained by simple arithmetical (quantitative) arguments.

Quantitative types are a powerful tool to provide relational models for λ -calculus [4, 22]. The construction of such models for λ_μ should be investigated, particularly to understand in the classical case the collapse relation between quantitative and qualitative models [41].

We expect to be able to transfer the ideas in this paper to a classical sequent calculus system, as was already done for focused intuitionistic logic [61].

The fact that idempotent types were already used to show observational equivalence between call-by-name and call-by-need [59] in intuitionistic logic suggests that our typing system $\mathcal{S}_{\lambda_{\mu\mathbf{x}}}$ could be used in the future to understand from a semantical point of view the fact that classical call-by-name and classical call-by-need are not observationally equivalent [92].

Moreover, it is possible to obtain *exact* bounds (as in [13]) for the lengths of the head-reduction and the perpetual reduction sequences in the case of the λ_μ -calculus. For that, it is necessary to integrate some additional typing rules being able to type the constructors appearing in the normal forms of the terms. Although this concrete development remains as future work, the difficult and conceptual part of the technique stays in finding the decreasing measure for reduction. We may thus have good hope to obtain as well exact bounds for λ_μ or $\lambda_{\mu\mathbf{x}}$.

The inhabitation problem for λ -calculus is known to be undecidable for idempotent intersection types [105], but decidable for the non-idempotent ones [18]. We may conjecture that inhabitation is also decidable for $\mathcal{H}_{\lambda_\mu}$.

Part III

Infinitary Normalization and Sequential Intersection

Presentation

Böhm trees (Chapter 10 of [8]) were originally suggested by the proof of Böhm Separability Theorem. The Böhm tree of a λ -term corresponds to its whole execution w.r.t. the Böhm reduction strategy (2nd variant). Böhm trees provide a natural semantics for the λ -calculus, meaning that if $t \equiv_{\beta} u$, then $\text{BT}(t) = \text{BT}(u)$. More generally, they are deeply involved in the study of observational equivalence [76].

The Böhm tree of a term is its (possibly infinite) normal form, in which every sub-head argument that is not head normalizing is replaced by the special constant symbol \perp (indicating a meaningless computation) *e.g.*, $\text{BT}(t) = \perp$ if t is not HN or $\text{BT}(x \Omega y) = x \perp y$. Moreover, $\text{BT}(Y_f) = f^{\omega}$ will hold, where Y_f is the term from Sec. 2.1.4 satisfying $Y_f \rightarrow_{\beta} f(Y_f)$ and f^{ω} is the infinite tree introduced in Sec. 2.3.3, informally written $f(f(f(\dots)))$.

The infinitary λ -calculi were introduced by Kennaway, Klop, and de Vries [57] in the 90s to capture several infinitary semantics of λ -calculus. They presented eight calculi, denoted Λ^{abc} (with $a, b, c \in \{0, 1\}$), the variables a, b, c indicating which constructions are allowed to build infinite terms. For instance, the normal forms of Λ^{001} correspond to the Böhm trees that *do not* contain \perp . Likewise, Λ^{111} and Λ^{101} respectively capture the Berarducci trees and the Lévy-Longo trees as normal forms (also up to the rewriting of “computationally meaningless terms” into \perp). The calculus Λ^{000} is just the finite λ -calculus Λ .

The infinitary λ -calculi have many applications [10] in relation *e.g.*, with Berry’s Sequentiality Theorem and relative computability. Incidentally, Klop *et al.* proved that the calculi Λ^{111} , Λ^{001} and Λ^{101} were the only infinite ones satisfying (a partial form of) confluence (*i.e.* having an interesting behaviour).

Our first interest in this thesis lies in the calculus Λ^{001} . We will just sketch the construction of Λ^{111} (also denoted Λ^{∞}) because it contains all the other infinite calculi.

Since a Böhm tree that does not contain \perp is a normal form of Λ^{001} , a term t whose Böhm tree does not contain \perp (*i.e.* the Böhm reduction strategies never outputs from t an argument that is not head normalizing) can be considered as *weakly normalizing* w.r.t. the calculus Λ^{001} .

Klop’s Problem is to find out whether the terms whose Böhm trees does not contain \perp (also called the **Hereditary Head Normalizing (HHN) terms**) can be characterized with an intersection type system. One of the contribution of this thesis is to provide a positive answer to this question in the infinitary case (*i.e.* when resorting to an infinitary type system) whereas it was proved impossible in the finite case (see the introduction of Chapter 10 for more details).

Hereditary head normalization is characterized with the intersection type system **S**, that we introduce in this thesis. System **S** uses **sequences** to represent intersection types, instead of sets or multisets (compare with Sec. 3.2.2). As system \mathcal{R}_0 , system **S** has a non-idempotent flavour *e.g.*, assigning a variable once or twice the same variable

is not the same. However, system \mathbf{S} is rigid in the sense of Sec. 2.1.1, whereas system \mathcal{R}_0 is not (remember Sec. 4.1.2). Actually, it is easy to see that coinductive type grammar usually give birth to *unsound* derivations (see Sec. 10.1.3), that type for instance mute terms (Sec. 2.3.2). We resort to a *validity criterion*, called **approximability**, to discard the unsound derivations that could make the characterization of HHN fail. Rigidity is a necessary feature to express this validity criterion, so that we cannot use an infinitary version of system \mathcal{R}_0 and we need to build system \mathbf{S} instead (for more details, see also the introduction of Chapter 10).

Hereditary head normalization is a particular case of *infinitary weak normalization* (corresponding to the first variant of the Böhm reduction strategy, Sec. 2.3.5). This strategy is known to be complete for weak normalization in the case of the finite λ -calculus (Sec. 5.1.4). Our type-theoretic characterization of the HHN terms gives a semantical proof that the hereditary head reduction strategy is (asymptotically) complete for infinitary weak normalization.

Contents of Part III

- Chapter 9 just gives some background on the infinitary calculus and Böhm trees. We present the notions of *infinite trees*, *coinductive grammars*, then we formally define the Böhm trees and the calculi Λ^{111} and Λ^{101} , as well as their infinitary operational semantics, given by the *strongly converging reduction sequences*.
- Chapter 10 presents our contributions to Klop’s Problem. Moreover, we introduce type system \mathbf{S} , featuring sequences as infinitary intersection types (instead of infinite multisets). We explain how to perform infinitary subject expansion by *truncating* infinitary derivations into finite ones and then taking their join. We then show that approximability cannot be defined by means of multiset constructions. We formally define the approximability criterion in system \mathbf{S} , allowing to discard unsound derivations. We conclude by proving all the properties necessary to the type-theoretic characterization of infinitary weak normalization: infinitary subject reduction and subject expansion, the typing of the infinite normal forms and an “infinitary normalization property”, meaning that every suitably typed term will asymptotically produce an infinitary normal form.

Chapter 9

The Infinitary Lambda-Calculus

We saw in Sec. 2.3.3 that some terms are not normalizing but nevertheless produce an unbounded number of stable positions, suggesting infinite λ -terms and in particular, infinite normal forms. We formalize those intuitions in this chapter:

- We introduce **Böhm trees**: the Böhm tree of a (normalizing or not) term corresponds to its whole execution w.r.t. the 2nd variant of the Böhm reduction strategy.
- We present two **infinitary extensions of the λ -calculus**, namely Λ^∞ and Λ^{001} . These calculi both feature infinite terms and an infinitary operational semantics.
- We explain how Böhm trees (almost) identify to the normal forms of Λ^{001} .
- Before introducing Λ^∞ and Λ^{001} , we explain the mechanisms of coinductive grammars.

We first give an intuitive account of Böhm trees (Sec. 9.1). As it turns out, many usual terms have a Böhm tree that is infinite. This leads us to present the formalism to be used in this thesis for infinite trees (Sec. 9.2.1), then we quickly describe how sets of (possibly) infinite trees can be defined with *coinductive* grammars (Sec. 9.2.3), which are a key tool of Part 10 and IV, both for defining the infinitary calculus and for defining infinitary types (*e.g.*, Sec. 10.2 in Chapter 10). In Sec. 9.3.1 and 9.3.2 respectively, we define and discuss the infinitary calculi Λ^∞ and Λ^{001} . We conclude this chapter with a formal presentation of Böhm trees as normal forms of the infinitary calculus Λ_{\perp}^{001} , a variant of Λ^{001} with an oracle deciding whether a term is head normalizing or not (Sec. 9.3.3).

9.1 Böhm Trees

We recall from Lemma 2.5 and Sec. 2.3.5 that a *Hereditarily Nested Head Normal Form (HNHNF)* of a term t is a subterm of t that is a HNF nested in a series of HNF (starting with t itself). The positions of the HNHNF of t are said to be *Böhm stable*. The second variant of the Böhm reduction strategy will output more and more Böhm stable positions. By Remark 2.8 p. 71, the 2nd variant of the Böhm reduction strategy will unfold from t all the possible HNHNF one after the other. This suggests the notion of Böhm trees: the **Böhm tree** $BT(t)$ of a term t is then the datum of all the HNHNF that can be outputted from t , with the additional convention that every maximal subterm

that is not head normalizing is replaced by the fresh constant symbol \perp . For instance, remembering some terms presented in Sec. 2.1.4:

- If t is weakly normalizing, then $\text{BT}(t)$ can be identified with the normal form of t by Proposition 3.8.
- We have $\text{BT}(\Omega) = \text{BT}(\Omega_3) = \text{BT}(Y'_f) = \text{BT}(Y_\lambda) = \perp$ since those terms are not head normalizing.
- If $t = \lambda x.y\Omega x\Omega_3$, then $\text{BT}(t) = \lambda x.y\perp x\perp$: the non-HN arguments Ω and Ω_3 are replaced by \perp in the Böhm tree.
- We have $\text{BT}(Y_f) = f^\omega$, where f^ω is the infinite tree defined in Sec. 2.3.3. Indeed, recall that $Y_f \rightarrow f(Y_f) \rightarrow \dots \rightarrow f^n(Y_f) \rightarrow \dots$. Note that $f^n(Y_f)$ is a nesting¹ of n head normal forms and, intuitively, one can output from Y_f infinitely many HNHNF, yielding $f(f(f(\dots))) =: f^\omega$ as expected.

This naive definition immediately brings up two remarks:

- Since head normalization is undecidable², the computation of Böhm trees demands an oracle to know when and where the constant \perp should pop up.
- The example of $\text{BT}(Y_f)$ above show that a finite term can produce infinitely many Böhm stable positions, and thus have an infinite Böhm tree.

The second point demands a formal description of infinite labelled trees, which can be found in Sec. 9.2.1.

A Glimpse at Coinduction Alternatively, the computation of Böhm trees can be described by a *coinductive* grammar:

$$\text{BT}(t) := \begin{cases} \perp & \text{if } t \text{ is not HN} \\ \lambda x_1 \dots x_p.x\text{BT}(t_1) \dots \text{BT}(t_q) & \text{if } t \text{ (head-)reduces to } \lambda x_1 \dots x_p.x t_1 \dots t_q \end{cases}$$

Roughly speaking, coinduction means that there can be infinitely many “calls” to the grammar to compute $\text{BT}(t)$. This will be made clearer in Sec. 9.2.3. Meanwhile, since $Y_f \rightarrow_\beta f(Y_f)$, this definition entails that $\text{BT}(Y_f) = f(\text{BT}(Y_f))$ and thus, $\text{BT}(Y_f) = f(\text{BT}(Y_f)) = f(f(\text{BT}(Y_f))) = f(\dots f(\text{BT}(Y_f)))$, so that after infinity many “calls”, we obtain $\text{BT}(Y_f) = f^\omega$.

Representation of Böhm Trees There are two ways to represent *e.g.*, the Böhm tree $\lambda xyz.y\text{BT}_1\text{BT}_2$ (with BT_1, BT_2 Böhm trees), given in Fig. 9.1.

The one on the left-hand side is traditional and more compact. The right one represents the same Böhm tree as λ -term. We favour this second choice though, since Böhm trees will be recovered as the normal forms of an infinitary λ -calculus (Sec. 9.3.3).

Note that the trees representing $\text{NF}_\infty(Y_\lambda)$, $\text{NF}_\infty(\Omega_3)$ and $\text{NF}_\infty(Y'_f)$ are not Böhm trees (Fig. 2.11, p. 67), since neither Y_λ , Ω_3 or Y'_f are head normalizing. Moreover, as suggested in Sec. 2.3.3, Böhm trees intuitively correspond to infinite normal forms (Sec. 9.3.3).

¹We have $\text{stab}_B(f^n(Y_f)) = \{2^k \mid k < n\} \cup \{2^k \cdot 1 \mid k < n\}$ (with $f^n(Y_f)(2^k) = @$, $f^n(Y_f)(2^k \cdot 1) = f$ for $k < n$)

²The λ -calculus endowed with head-reduction is Turing complete, see for instance Chapter 2 of [68]

A is a sequence $b = (a_k)_{k \in \mathbb{N}}$ such that every finite prefix of b is in A e.g., 2^ω is an infinite branch of the ∞ -tree $A = \{2^k \mid k \in \mathbb{N}\} \cup \{2^k \cdot 1 \mid b \in \mathbb{N}\}$.

A *rigid labelled ∞ -tree* T on the *signature* Σ is a function whose *support* (domain) $\text{supp}(T)$ is a rigid ∞ -tree. The terms *position*, *occurrence*, *subtree* rooted at w are straightforwardly extended to ∞ -trees.

9.2.2 Smallest and Biggest Invariant Subsets

In order to present inductive and coinductive grammars, we need to explain, given a set \mathcal{X} , how to build the smallest subset (resp. the biggest subset) of \mathcal{X} that is *invariant* under a given family of functions \mathcal{F} (**\mathcal{F} -invariance**). This is a classical tool of the theory of partially ordered sets (posets). By the way, the **power set** of \mathcal{X} is denoted $\mathbf{P}(\mathcal{X})$.

With these notations, the **arity** of a function $f \in \mathcal{F}$ is denoted $\text{ar}(f)$ i.e. $\text{ar}(f) = n$ means that f is a function from \mathcal{X}^n to \mathcal{X} . When $\text{ar}(f) = 0$, f is a constant function. Given $Y \subseteq \mathcal{X}$ and $f \in \mathcal{F}$ such that $\text{ar}(f) = n$, we abusively write $f(Y)$ for $f(Y^n)$ (i.e. $\{f(x_1, \dots, x_n) \mid x_1, \dots, x_n \in Y^n\}$).

We then build a function $F_{\mathcal{F}}$ (or just F) from $\mathbf{P}(\mathcal{X})$ to itself by setting, for all $Y \subseteq \mathcal{X}$:

$$F(Y) = \cup_{f \in \mathcal{F}} f(Y)$$

Thus, a subset $Y \subseteq \mathcal{X}$ is \mathcal{F} -stable iff $F(Y) \subseteq Y$ and Y is \mathcal{F} -invariant iff Y is a **fixpoint** of F (i.e. $F(Y) = Y$). Note that \mathcal{F} -invariance entails \mathcal{F} -stability.

The function F is monotonic i.e. for all $Y, Y' \subseteq \mathcal{X}$, if $Y \subseteq Y'$, then $F(Y) \subseteq F(Y')$. The set \mathcal{X} is obviously \mathcal{F} -stable and if the $(Y_i)_{i \in I}$ (with I finite or infinite set) are \mathcal{F} -stable, then $\cap_{i \in I} Y_i$ is \mathcal{F} -stable.

- We set $\mu Y.F(Y) = \cup_{n \in \mathbb{N}} F^n(\emptyset)$. Using the monotonicity of F , it is straightforward to check that (1) $F(\mu Y.F(Y)) = \mu Y.F(Y)$ so that $\mu Y.F(Y)$ is \mathcal{F} -invariant and that moreover (2) $\mu Y.F(Y)$ is the **smallest \mathcal{F} -invariant subset** of \mathcal{X} (if $Z \subseteq \mathcal{X}$ is \mathcal{F} -invariant, then $\mu Y.F(Y) \subseteq Z$). Observe that $\mu Y.F(Y)$ is empty iff there are no nullary (0-ary) functions in \mathcal{F} .
- We set $\nu Y.F(Y) = \cap_{n \in \mathbb{N}} F^n(\mathcal{X})$. Using the monotonicity of F , we check that (1) $F(\nu Y.F(Y)) = \nu Y.F(Y)$, so that $\nu Y.F(Y)$ is also \mathcal{F} -invariant and that (2) $\nu Y.F(Y)$ is the **biggest \mathcal{F} -invariant subset** of \mathcal{X} (if $Z \subseteq \mathcal{X}$ is \mathcal{F} -invariant, then $\nu Y.F(Y) \supseteq Z$).

9.2.3 Inductive vs. Coinductive Grammars

We shall not present the general framework of **induction** and **coinduction**, but just discuss the example of the *inductive* grammar defining the (finite) λ -calculus and that of its *coinductive* counterpart defining the *infinite* λ -calculus. See [34, 42] for a very elegant framework allowing to interweave induction and coinduction and define the infinitary λ -calculus.

Inductive grammars are used to define sets of strings of characters but they can be used as well to define sets of labelled trees. This does not make much of a difference e.g., λ -terms can be both seen as strings of characters or as labelled trees (Sec. 2.1.2).

But note that it is a bit problematic to find a canonical formalism for *infinite* strings of characters: intuitively, as a string of characters, an infinite word may be infinite on its right-hand side (if we apply the rewriting rule $\mathcal{S} \rightarrow a\mathcal{S}$ infinitely many times, we obtain

aaa...) or on in left-hand side (with $\mathcal{S} \rightarrow \mathcal{S}a$, we obtain ...aaa) or both (with $\mathcal{S} \rightarrow a\mathcal{S}a$, we obtain ...aaa...). Of course, one may want to concatenate several words of infinite length... This soon becomes messy and this is one of the reasons why we do not want to define infinite λ -terms as infinite strings of characters. Note that infinite trees (Sec. 9.2.1) seem easier to handle: they can have several infinite branches and they are not just infinite on one side and/or the other.

Thus, although the formalisms of λ -terms-as-strings and of λ -terms as trees can coexist in the finite case, in the infinite one, only the latter can be conveniently used. Let \mathcal{T}_λ be the set of rigid labelled trees on the signature $\Sigma_\lambda = \mathcal{V} \cup \{\lambda x \mid x \in \mathcal{V}\} \cup \{@\}$. To say that Λ is the set of trees defined by the inductive grammar below (cf. Sec. 2.1.2) just means that Λ is the smallest subset of \mathcal{T}_λ that is invariant under the constructors $(x)_{x \in \mathcal{V}}$, $(t \mapsto \lambda x.t)_{x \in \mathcal{V}}$ and $(t, u) \mapsto (tu)$ (Sec. 9.2.2).

$$t, u := x \in \mathcal{V} \mid (\lambda x.t) \mid (tu)$$

The fact that the metavariables t and u occur on the left-hand side of $:=$ means that they are the **inductive variables** i.e. are bound in the grammar (they are the actual variables of the constructors and not parameters).

What happens if we consider Λ' , the biggest set of \mathcal{T}_λ that is invariant under the same constructors? Actually nothing, and it is easy to check that $\Lambda' = \Lambda$.

Now, let $\mathcal{T}_\lambda^\omega$, the set of the ∞ -labelled trees t on the signature Σ_λ , so that $\mathcal{T}_\lambda \subseteq \mathcal{T}_\lambda^\omega$ such that $\text{supp}(t) \subseteq \{0, 1, 2\}^*$. The smallest subset of $\mathcal{T}_\lambda^\omega$ that is invariant under the above constructors is still Λ . On the other hand, the biggest invariant subset, that we denote Λ^∞ , is bigger than Λ . For instance, Λ^∞ contains the labelled trees f^ω , $\text{NF}_\infty(Y_\lambda)$, $\text{NF}_\infty(\Omega_3)$ and $\text{NF}_\infty(Y'_f)$ presented in Fig. 9.4 and Fig. 9.2 below. Note that those labelled trees can be respectively defined by the following coinductive grammars $t := ft$ (for f^ω), $t := \lambda x.t$, $t = t\omega_3$ and $t := tf$ (for Y'_f).

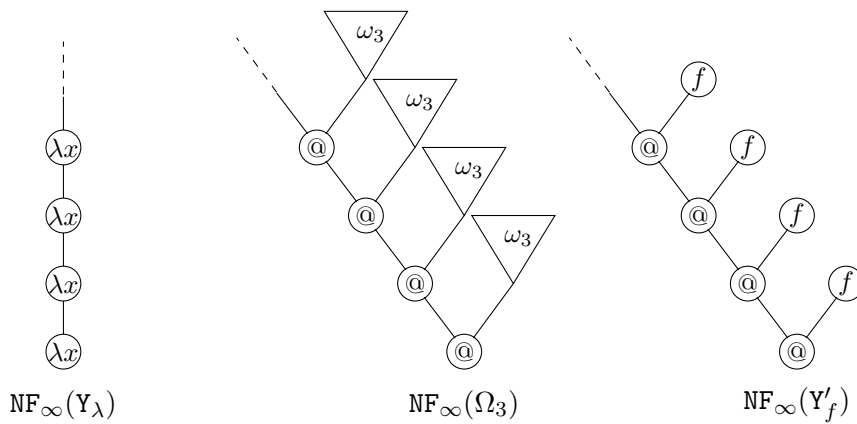


Figure 9.2: Infinite Normal Forms (copy of Fig. 2.11, p. 67)

The set Λ^∞ is called the set of **infinite λ -terms** and its operational semantics will be defined and briefly studied in Sec. 9.3.1.

Given $t \in \mathcal{T}_\lambda^\omega$, the predicate $t \in \Lambda^\infty$ can be checked node-wise:

Lemma 9.1. Let $t \in \mathcal{T}_\lambda^\omega$. Then $t \in \Lambda^\infty$ iff t satisfies the following conditions, for all $b \in \text{supp}(t)$:

- If $t(b) = x$ for some $x \in \mathcal{V}$, then b is a leaf of $\text{supp}(t)$.
- If $t(b) = \lambda x$ for some $x \in \mathcal{V}$, then $b \cdot 0 \in \text{supp}(t)$ and $b \cdot 1, b \cdot 2 \notin \text{supp}(t)$.
- If $t(b) = @$, then $b \cdot 1, b \cdot 2 \in \text{supp}(t)$ and $b \cdot 0 \notin \text{supp}(t)$.

9.3 The Infinitary Lambda Calculi

With the help of coinductive definitions, we may now present two infinitary λ -calculi, Λ^∞ (also denoted Λ^{111}) and Λ^{001} . The calculus Λ^{001} is a restriction of the full infinitary λ -calculus. They both come along with their infinitary operational semantics, embodied by the *strongly converging reduction sequences (s.c.r.s.)*, which are a special kind of reduction sequence of (possibly) infinite length, regarded as *productive*: a s.c.r.s. of infinite length will still output a term (asymptotically), to be called the *limit* of the s.c.r.s.. Both calculi feature infinitary normal forms, and thus, give rise to a notion a infinitary normalization (this last point is addressed only for weak normalization and for Λ^{001}). We conclude this section by explaining the relation between the normal forms of Λ^{001} and Böhm trees.

9.3.1 The Full Infinitary Calculus

In this section, we study the full infinitary λ -calculus and briefly discuss it as a framework.

For that, it is better to assume that the set \mathcal{V} of term variable is *uncountable* (this is explained at the end of this section, p. 198):

Definition 9.1. The set of **infinitary λ -terms**, denoted Λ^∞ , is defined by the *coinductive* grammar (Sec. 9.2.3):

$$t, u := x \in \mathcal{V} \mid (\lambda x.t) \mid (tu)$$

Notation 9.1. We use the same notation conventions as those for the finite λ -calculus (see Notation 2.1, p. 51).

Alpha-Equivalence and Capture-Free Substitution Let $(z_b)_{b \in \{0,1,2\}^*}$ be a family of fresh variables (for all $b \in \{0,1,2\}^*$, $z_b \notin \mathcal{V}$) and we set $\mathcal{V}_* = \mathcal{V} \cup \{z_b \mid b \in \{0,1,2\}^*\}$. We define the calculus Λ_*^∞ by the coinductive grammar:

$$t, u := x \in \mathcal{V}_* \mid (\lambda x.t) \mid (tu)$$

We denote Λ_B^∞ the set of terms $t \in \Lambda_*^\infty$ such that no variable z_b occurs in t freely. Note that $\Lambda^\infty \subseteq \Lambda_B^\infty$. Let $[\cdot]$ be an injection from \mathbb{N}^* to \mathbb{N}

Then, for all $t \in \Lambda_B^\infty$, we define t^B as the term obtained from t by

- replacing by $z_{[b]}$ any variable $y \in \mathcal{V}$ occurring at position b' and that is bound at position $b \in \{0,1,2\}^*$ (i.e. $t(b') = y$ and b is the longest prefix of b' s.t. $t(b) = \lambda y$)...
- ... and replacing λy (at pos. b) by $\lambda z_{[b]}$.

The letter \mathcal{B} stands for “Barendregt ” and $t^{\mathcal{B}}$ satisfies Barendregt convention by construction. We say t and u are α -**equivalent** when $t^{\mathcal{B}} = u^{\mathcal{B}}$, and write simply $t = u$.

The notion of binding position and the notation $\lambda^t(b)$ from Sec. 2.1.2 naturally extend to infinite λ -terms, as well as α -equivalence (t and t' are α -equivalent if

- (1) $\text{supp}(t) = \text{supp}(t')$
- (2) a variable x occurs freely in t at position b iff it occurs freely in t' at position b
- (3) a position b is bound in t' at position b_λ iff b is bound in t' at position b_λ .

The **capture free substitution** $t[u/x]$ is defined as the term v obtain from x by replacing (in the term t) every free occurrence of x by the term u .

We write $t[u/x]$ for the **capture-free substitution** of x by u inside t , meaning that $t[u/x]$ is the term obtained from t by replacing (in t) every *free* occurrence of x by u and by α -*renaming* the bound variables of t so that no abstraction of t binds a variable occurring free in u . Doing so is possible because (1) \mathcal{V}^+ is uncountable, and (2) $\text{supp}(t) \subset \{0, 1, 2\}^*$ implies that $\text{supp}(t)$ is countable, so that t contains a countable number of variables.

A *redex* of Λ^∞ is also a term of the form $(\lambda x.r)s$ with $r, s \in \Lambda^\infty$ and the (*root-*)*reduct* of $(\lambda x.r)s$ is $r[s/x]$. Pointed β -reduction $t \xrightarrow{b}_\beta t'$ is defined as in for all $b \in \{0, 1, 2\}^*$ by *induction* on b . The (pointed) relation $t \xrightarrow{b}_\beta t'$ is defined by *induction* on $b \in \{0, 1, 2\}^*$ as in Sec. 2.1.3: $(\lambda x.r)s \xrightarrow{\varepsilon}_\beta r[s/x]$, $\lambda x.t \xrightarrow{0 \cdot b}_\beta \lambda x.t'$ if $t \xrightarrow{b}_\beta t'$, $t_1 t_2 \xrightarrow{1 \cdot b}_\beta t'_1 t_2$ if $t_1 \xrightarrow{b}_\beta t'_1$, $t_1 t_2 \xrightarrow{2 \cdot b}_\beta t_1 t'_2$ if $t_2 \xrightarrow{b}_\beta t'_2$.

The full β -reduction is then defined by $\rightarrow_\beta = \cup_{b \in \{0,1,2\}^*} \xrightarrow{b}_\beta$. The notion of *reduction sequence* is defined as in Sec. 2.1.6.

Strong Convergence in Λ^∞ A reduction sequence of infinite length can be **productive**: assume that $t \xrightarrow{b_0}_\beta t_1 \xrightarrow{b_1}_\beta \dots \xrightarrow{b_{k-1}}_\beta t_k \xrightarrow{b_k}_\beta \dots$ and that $\lim_{k \rightarrow \infty} |b_k| = \infty$. Thus, the depth of the positions of the reduced redexes converges towards infinity. Such a reduction sequence is said to be **strongly converging**. The main intuition is the following: when a reduction is performed at depth n , the term is not affected below depth n . Thus, the reduction sequence stabilizes the term at any fixed depth. It allows us to define the *limit* t' of this reduction sequence: the truncation t' at depth n is equal to that of any t_k such that no reduction at depth $\leq n$ is performed after rank k . For instance, in Fig. 9.3 representing a strongly converging reduction sequence, the parts that have been stabilized are put in gray: after 50 reduction steps, no reduction step is performed under depth 10 (if $k \geq 50$, then $|b_k| > 10$), so that the term of the sequence are stabilized under depth 10: the depth 10 truncation of the limit is computed. After 10^9 reduction steps, the term is stabilized under depth 10^6 (if $k \geq 10^9$, then $|b_k| > 10^6$): the depth 10^6 truncation of the limit is computed. Asymptotically, the whole term is stabilized, which gives the limit of the reduction sequence.

Formally, a reduction sequence $\mathbf{rs} = (b_k)_{k < \kappa}$ (with $\kappa \leq \omega$) is strongly converging if \mathbf{rs} is of finite length ($\kappa = n \in \mathbb{N}$) or if $\lim_{k \rightarrow \infty} |b_k| = \infty$. In the latter case, we can define the **limit** of \mathbf{rs} as follows. First, we denote t_k term of rank k in \mathbf{rs} *i.e.* $t \xrightarrow{b_0}_\beta t_1 \xrightarrow{b_1}_\beta \dots \xrightarrow{b_{k-1}}_\beta t_k \xrightarrow{b_k}_\beta \dots$. Moreover, for all $\ell \in \mathbb{N}$, we denote by $N_{\mathbf{rs}}(\ell)$ the smallest

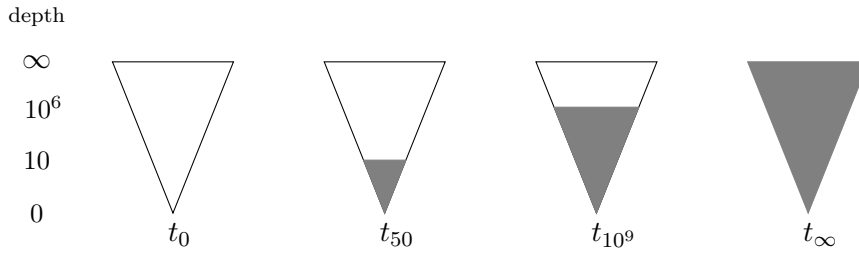


Figure 9.3: Stabilization in a Strongly Converging Sequence (Example)

integer N such that, for all $k \geq N$, $|b_k| > \ell$. Then we define t' as the following tree of $\mathcal{T}_\lambda^\omega$:

- $\text{supp}(t') = \{b \in \{0, 1, 2\}^* \mid b \in \text{supp}(t_N) \text{ with } N = N_{\text{rs}}(|b|)\}$
- For all $b \in \text{supp}(t')$, $t'(b) = t_N(b)$ with $N = N_{\text{rs}}(|b|)$.

By definition of $N_{\text{rs}}(\ell)$, we notice that (1) for all $b \in \{0, 1, 2\}^*$, $b \in \text{supp}(t')$ iff, for all $k \geq N_{\text{rs}}(|b|)$, $b \in \text{supp}(t_k)$ and (2) for all $b \in \text{supp}(t')$ and for all $k \geq N_{\text{rs}}(|b|)$, $t'(b) = t_k(b)$. From this and Lemma 9.1, we easily deduce that the labelled tree t' is an infinite λ -term. To conclude, we write

$$t \rightarrow^\infty t'$$

to mean that there is a strongly converging reduction sequence from t to t' . See [57] for an in-depth study of *strongly converging reduction sequences*.

Example 9.1. With the terms of Sec. 2.3.3, $Y_f \rightarrow^\infty f^\omega$ (so that $Y \rightarrow^\infty \lambda f.f^\omega$), $Y_\lambda \rightarrow^\infty \lambda x^\omega$, $\Omega_3 \rightarrow^\infty ((\dots)\omega_3)\omega_3$ and $Y'_f \rightarrow^\infty ((\dots)f)f$. As counter-example, the infinite reduction sequence $\Omega \rightarrow \Omega \rightarrow \dots$ is not strongly converging, since reduction occurs at depth 0 (which does converge towards infinity...).

Remark 9.1 (Head Normalization in Λ^∞).

- A head normal form of Λ^∞ is a term $t \in \Lambda^\infty$ of the form $\lambda x_1 \dots x_p.x t_1 \dots t_q$ ($p, q \geq 0$) with $t_1, \dots, t_q \in \Lambda^\infty$. A term $t \in \Lambda^\infty$ is head reducible if it is of the form $\lambda x_1 \dots x_p.(\lambda x.r)s t_1 \dots t_q$ ($p, q \geq 0$) with r, s, t_1, \dots, t_q . A term is **head-formed** if it is a HNF or if it is head reducible. If not, we say that it is **headless**. For instance, the terms $\text{NF}_\infty(Y_\lambda)$, $\text{NF}_\infty(\Omega_3)$ and $\text{NF}_\infty(\Omega_3)$ (Fig. 9.2) are headless and f^ω (Fig. 9.4) is an infinite head-formed term. A finite term is head-formed.
- A term $t \in \Lambda^\infty$ is head normalizing if there is a HNF t' such that $t \rightarrow^\infty t'$. We write $t \rightarrow_{\text{h}} t'$ (head reduction) if $t = \lambda x_1 \dots x_p.(\lambda x.r)s t_1 \dots t_q$, $t' = \lambda x_1 \dots x_p.r[s/x] t_1 \dots t_q$ for some $r, s, t_1, \dots, t_q \in \Lambda^\infty$.
- For the finite terms t , it has been proved (Proposition 3.8) that $t \rightarrow^* \lambda x_1 \dots x_p.x t_1 \dots t_q$ iff $h \rightarrow_{\text{h}}^* \lambda x_1 \dots x_p.x t_1 \dots t_q$ by using Type System \mathcal{R}_0 . For now, nothing guarantees this equivalence for infinite terms. However, we briefly explain in Sec. 10.1.2, p. 207 why Proposition 3.8 still applies and ensure that, for all $t \in \Lambda^\infty$, $t \rightarrow^\infty \lambda x_1 \dots x_p.x t_1 \dots t_q$ iff the head reduction strategy terminates (in a finite number of steps) on t . See also Remark 10.1 to compare with Λ^{001} .
- Note that if t is headless, then the head reduction strategy terminates (in 0 steps!) but t is nevertheless neither a HNF nor even HN.

Partial Confluence in Λ^∞ Neither the finite reduction nor the infinite are confluent in Λ^∞ :

- If $t = (\lambda f.f^\omega)(Ix)$, then $t \rightarrow (Ix)^\omega =: t_1$ and $t \rightarrow (\lambda f.f^\omega)x \rightarrow x^\omega$, but x^ω cannot be obtained from $(Ix)^\omega$ in a finite number of steps, so \rightarrow is not confluent.
- We have $\Upsilon I \rightarrow \Omega$ and $\Upsilon I \rightarrow^\infty I^\omega$. Since Ω (resp. I^ω) is the only reduct of Ω (resp. I^ω), Ω and I^ω do not have a common reduct, so \rightarrow^∞ is not confluent:

However, Λ^∞ satisfies a *partial* form of confluence. More precisely, it is confluent when all the mute terms (Sec. 2.3.2) are equated. Formally, we define the binary relation \sim_∞ on Λ^∞ by coinduction:

$$\frac{t \text{ and } u \text{ are mute}}{t \sim_\infty u} \quad \frac{}{x \sim_\infty x} \quad \frac{t \sim_\infty u}{\lambda x.t \sim_\infty \lambda x.u} \quad \frac{t_1 \sim_\infty u_1 \quad t_2 \sim_\infty u_2}{t_1 t_2 \sim_\infty u_1 u_2}$$

Note that the premise of the leftmost rule is not decidable (the set of mute terms is not recursively enumerable): it also depends on an oracle.

Proposition 9.1. Let $t, t', u, u' \in \Lambda^\infty$. If $t \sim_\infty u$, $t \rightarrow^\infty t'$, $u \rightarrow^\infty u'$, then there exists $t'', u'' \in \Lambda^\infty$ such that $t'' \sim_\infty u''$, $t' \rightarrow^\infty t''$, $u' \rightarrow^\infty u''$.

Proof. This proposition is the main theorem of [34], originally a particular case of Theorem 57 of [57]. \square

Discussion on the Infinitary Lambda-Calculus: The role that plays Λ^∞ for Λ is similar to the role that the set of real numbers \mathbb{R} plays for \mathbb{Q} , the set of rational numbers.

- Λ^∞ (and later Λ^{001}) are metric completions of Λ w.r.t. some *ad hoc* distances (see [57]).
- Intuitively, Λ^∞ contains too many objects *i.e.* there are terms $u \in \Lambda^\infty$ such that no *finite* term $t \in \Lambda$ satisfies $t \rightarrow^\infty u$ (*i.e.* u is not a limit of a finite term). For instance:
 - If x_0, x_1, \dots is a sequence of pairwise distinct variables, then the infinite term $u = x_0(x_1(x_2(\dots)))$ is not the limit of a finite term, since (1) $\text{fv}(u)$ is an infinite set (2) $t \rightarrow^\infty u$ implies that $\text{fv}(t) \supseteq \text{fv}(u)$ (3) if $t \in \Lambda$, then $\text{fv}(t)$ is finite.
 - If $g : \mathbb{N}$ is a non-computable total function, then the infinite term $t = \lambda x.x x^{g(0)} \lambda x.x x^{g(1)} \lambda x \dots$ is not the limit of a finite term. If it were, we could compute g .

Likewise, \mathbb{R} is very useful to define number such as $\sqrt{2}$, π and e , but it also contain numbers that are not even definable (since \mathbb{R} is uncountable).

Thus, Λ^∞ is a very general framework, but it provides an interesting semantics for the finite λ -calculus with the Berarducci trees [11], that could not be studied in this thesis.

In the finite case, the confluence of the λ -calculus is a very powerful property since it holds for any finite λ -term and has nothing to do with normalization. In particular, it allows us to conclude that a normalizing term has exactly one normal form or that a head normalizing term has a unique prefix $\lambda x_1 \dots x_p.x$ (modulo α -equivalence). On

the other hand, Proposition 9.1 only guarantees confluence for the normalizing parts⁴ of a term. However, this is enough to conclude with the unicity of the normal form of a term, when it exists.

Last, observe that although an *uncountable* set of term variables is not needed when studying the infinite terms that are the limits of some finite λ -terms, it is necessary for the full calculus Λ^∞ . Assume *ad absurdum* that \mathcal{V} is countable and that $(x_n)_{n \in \mathbb{N}}$ is an (injective) enumeration of \mathcal{V} . Let then $t = \lambda x_1.x_1 x_0$ and $u = x_0 x_1 x_2 \dots$. Then there is no way to substitute x_0 with u in t without capture, since all the variables should occur free in $t[u/x]$.

9.3.2 The Infinitary Calculus of Böhm Trees

The calculus Λ^∞ is too general for specifying Böhm trees (Sec. 9.1). For instance, it accepts the normal forms of Y_λ or Ω_3 (represented in Sec. 2.3.3), which are not Böhm trees. Indeed, Y_λ and Ω_3 are not HN (so that their Böhm tree is \perp) and if we look at $\text{NF}_\infty(Y_\lambda)$ and $\text{NF}_\infty(\Omega_3)$, we observe that they *do not* have a head variable! The term $\text{NF}_\infty(Y_\lambda)$ is a series of abstractions $\lambda x.\lambda x.\dots$ that does not end with a head variable (it is infinite) and $\text{NF}_\infty(\Omega_3)$ has a leftward infinite branch starting from the root, which does not end with some x either. This can be related to the facts that the positions of $\text{NF}_\infty(Y_\lambda)$ and $\text{NF}_\infty(\Omega_3)$ are stable but *not* Böhm stable (remember Remark 2.7).

Terms of Λ^{001} The calculus Λ^{001} is defined by restricting Λ^∞ to those terms t such that the paths (in the parent-to-child direction) visiting only application left-hand sides or abstractions should always stop, so that a term $t \in \Lambda^\infty$ is in Λ^{001} iff all its subterms are head-formed (see Remark 9.1), contrary to $\text{NF}_\infty(Y_\lambda)$ and $\text{NF}_\infty(\Omega_3)$.

We extend **applicative depth** for infinite words. We recall (Sec. 2.3.5) that, if $b \in \text{supp}(t)$, then $t|_b$, the subterm of t rooted at b , is nested in $\text{ad}(b)$ application arguments. Then, if $b = (b_i)_{i \in \mathbb{N}}$ is a word of *infinite* length, we set $\text{ad}(b) = \#\{i \in \mathbb{N} \mid b_i \geq 2\}$.

Let $t \in \Lambda^\infty$. Then, since t is a labelled tree, t can have infinite branches in the sense of Sec. 2.1.1. If b is an infinite branch of a term $t \in \Lambda^\infty$, $\text{ad}(b) = \infty$ means that b visit infinitely many times *arguments* of applications. We then define:

Definition 9.2. Let $t \in \Lambda^\infty$. Then t is said to be a **001-term** (denoted $t \in \Lambda^{001}$) if, for all infinite branches b of t , $\text{ad}(b) = \infty$.

Thus, for 001-terms, infinity is allowed, provided we visit arguments infinitely many times. Thus, no subterm of a 001-term is deprived of a head redex or a head variable, as expected above. For instance, f^ω is in Λ^{001} , because 2^ω is its unique infinite branch and $\text{ad}(2^\omega) = \infty$. However, the term $u = \text{NF}_\infty(Y'_f) \in \Lambda^{111}$, represented in Fig. 9.2 and coinductively defined by $u = u f$ (where f is a term variable) is not a 001-term: it also has a unique infinite branch, 1^ω but $\text{ad}(1^\omega) = 0$.

Strong Convergence in Λ^{001} Now, we can define *strongly converging reduction sequences* as for Λ^∞ (Sec. 9.3.1), although the definition varies a bit:

Definition 9.3. Let $t \in \Lambda^{001}$ and $\mathbf{rs} = (b_k)_{k \in \kappa}$ a reduction sequence starting at t . Then the reduction sequence \mathbf{rs} is said to be **strongly converging (w.r.t. Λ^{001})** if the length of \mathbf{rs} is finite (*i.e.* $\kappa = n \in \mathbb{N}$) or, in the other case, if $\lim_{k \rightarrow \infty} \text{ad}(b_k) = \infty$.

⁴Indeed, every non-mute term t is considered as normalizing in Λ^∞ , even when t is not HN [57]

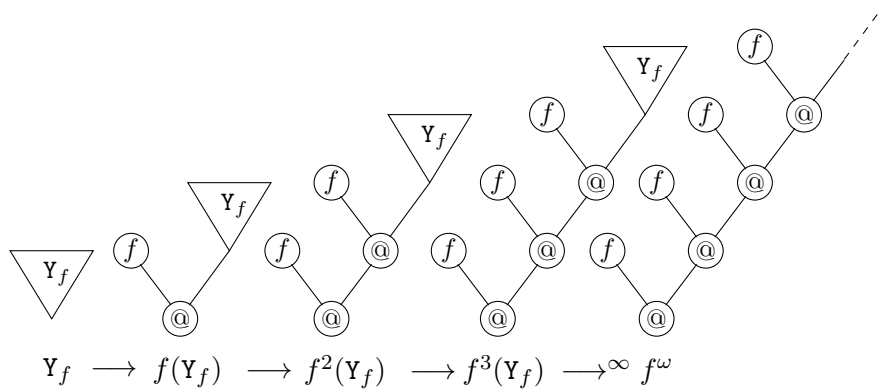


Figure 9.4: Strong Convergence of Y_f (copy of Fig. 2.10, p. 67)

Thus, an infinite reduction sequence is strongly converging if the applicative depth of the reduced redexes converges toward infinity. We shorten *strongly converging reduction sequences* into **s.c.r.s.**. As in the the case of Λ^∞ , we can define the limit of a s.c.r.s.. Assume that $t_0 \xrightarrow{b_0} t_1 \xrightarrow{b_1} \dots$ is an infinite s.c.r.s.. Let $b \in \{0, 1, 2\}^*$ and $N \in \mathbb{N}$ s.t. $\forall n \geq N, \text{ad}(b_n) > \text{ad}(b)$. Then, either $\forall n \geq N, b \notin \text{supp}(t_n)$ or $\forall n \geq N, b \in \text{supp}(t_n)$ and $t_n(b) = t_N(b)$. Let B' be the set of all $b \in \mathbb{N}^*$ in the latter case and t' the labelled tree defined by $\text{supp}(t') = B'$ and $t'(b) = t_N(b)$ for any N large enough. We notice that $t' \in \Lambda^\infty$. Actually, t' is a 001-term (because under fixed applicative depth d , t' must be identical to a t_N , for N some large enough) and we call t' the *limit* of the s.c.r.s. The notation $t \rightarrow^\infty t'$ means that there is a s.c.r.s. starting from t , whose limit is t' . In general, along with the obvious case of the finite reduction sequences:

Lemma 9.2.

- Let $t = t_0 \xrightarrow{b_0} t_1 \xrightarrow{b_1} \dots$ be a strongly converging reduction sequence of Λ^{001} . Then, there is a unique term $t' \in \Lambda^{001}$ such that, for all $d \geq 0$, t' and t_n are equal under applicative depth d , provided⁵ n is great enough. The term t' is called the **limit** of this s.c.r.s..
- We write $t \rightarrow^\infty t'$ to mean that there is a s.c.r.s. starting at t whose limit is t' .

For instance, $f^n(Y_f) \xrightarrow{2^n} f^{n+1}(Y_f)$ and $\text{ad}(2^n) = n \rightarrow \infty$, so that $Y_f \xrightarrow{\varepsilon} f(Y_f) \xrightarrow{2} f^2(Y_f) \xrightarrow{2^2} \dots$ is a s.c.r.s.. We check that its limit w.r.t. the definition above is f^ω as expected, since f^ω was taken as an example of infinite Böhm tree in Sec. 9.1. This is represented by Fig. 9.4. The redex occurs at applicative depth 0 in Y_f , then at applicative depth 1 in $f(Y_f)$, then at applicative depth 2 in $f(f(Y_f))$. Asymptotically, the redex is “swallowed” at infinite applicative depth and we obtain the 001-normal form f^ω on the right-hand side.

Remark 9.2. • The vocables “strongly converging reduction sequence” and “limit” the notation are common to Λ^∞ and Λ^{001} , as well as the notation $t \rightarrow^\infty t'$, but they do not have the same meaning. In fact, if t strongly converges towards t' in Λ^{001} , then so does it in Λ^∞ , but the converse is not true *e.g.*, $\Omega_3 \rightarrow^\infty \text{NF}_\infty(\Omega_3)$ in

⁵Formally, for all $d \geq 0$, there exists N such that, for all $n \geq N$, t' and t_n induces the same labelled tree of $\{b \in \mathbb{N}^* \mid \text{ad}(b) \leq d\}$.

Λ^∞ , but Ω_3 does not strongly converge towards $\text{NF}_\infty(\Omega_3)$ in Λ^{001} ($\text{NF}_\infty(\Omega_3)$ is not even in Λ^{001}).

- A **compression property** [57] allows us to consider only sequences of length $\leq \omega$ without loss of generality, although a more general definition could be given.
- The whole discussion concluding Sec. 9.3.1 is still valid for Λ^{001} : Λ^{001} contains terms that are not limits of finite terms and actually, all the examples of this discussion are 001-terms.
- Henceforth, “strong convergence”, “limit” and \rightarrow^∞ will only be understood w.r.t. Λ^{001} .

Weak Normalization in Λ^{001} A **001-Normal Form** *i.e.* a normal form of Λ^{001} is a term that does not have a redex. Thus, the set of 001-NF is generated by the coinductive grammar:

$$t, t_k ::= \lambda x_1 \dots x_p. x t_1 \dots t_q \quad (p, q \geq 0)$$

This is exactly the coinductive counterpart of Lemma 2.2, p. 62: we may also say that 001-NF are coinductive assemblages of head normal forms. Note that a 001-NF really look like a Böhm tree, except for the constant \perp , which is yet absent.

By analogy with weak normalization in the finite case (Definition 2.4):

Definition 9.4. A 001-term t is **001-weakly normalizing (001-WN)** iff t can be reduced to a NF by means of a s.c.r.s..

Thus, Y_f is WN as expected, and f^ω is its unique normal form, according to Corollary 9.1 below.

Hereditary head normalization plays a similar role for 001-WN than that the predicates “The head reduction strategy terminates on t ” and “The Minimal reduction strategy terminates for t ” respectively play for HN and WN in the finite case. The definition of HHN originally held for finite λ -terms, but it naturally extends to 001-terms.

Definition 9.5. Let $t \in \Lambda$ (or more generally, $t \in \Lambda^{001}$). Then t is said to be **Hereditary Head Normalizing (HHN)** if $t \rightarrow_h^* \lambda x_1 \dots x_p. x t_1 \dots t_q$ and the t_1, \dots, t_q are HHN.

Thus, a term is HHN if, coinductively, t is head normalizing and all its head arguments are themselves HHN. This means that the *first* variant of the Böhm reduction strategy (Sec. 2.3.5) gives⁶ strongly converging reduction sequences. Equivalently, t is HHN when the Böhm tree $\text{BT}(t)$ of t does not contain \perp .

As a reminder, the first variant of the Böhm reduction strategy on a term t starts with a finite number of head reductions (applicative depth = 0), stopping only in a HNF $\lambda x_1 \dots x_p. x t_1 \dots t_q$ ($p, q \geq 0$) is reached. The argument t_1, \dots, t_q are then head-reduced (applicative depth 1) until some HNF t'_1, \dots, t'_q are reached. If some t_k is not HN, then the strategy loops. Then head reductions are performed at applicative depth 2 till the head arguments of the t'_1, \dots, t'_q reach their HNF. The strategy goes on like this. When t is HHN, no non-HN subterm will be outputted and either the strategy ends in

⁶This reduction strategy is not deterministic.

a finite number of steps or it produces a strongly converging sequence that eliminates every redex. Thus, the 1st variant of the Böhm reduction strategy could also be referred to as the **Hereditary Head Reduction Strategy**. This strategy only computes the Böhm tree of a term when it does not contain \perp (whereas the 2nd variant of the Böhm reduction strategy is never “captive” of a non-HN subterm).

Remark 9.3.

- Later on (Theorem 10.4), we will prove that that hereditary head normalization is equivalent to 001-weak normalization *i.e.* that the Böhm reduction strategy is (asymptotically) *complete* for 001-WN, using type-theoretic methods as for Propositions 3.8 and 5.2.
- In contrast with Proposition 5.2, the Leftmost Outermost reduction strategy (and thus, the Minimal reduction strategy) is not complete for 001-WN *e.g.*, applied to $Y_f((\lambda x.x)x)$, it will output $f^\omega((\lambda x.x)x)$, which is not normal.

Partial Confluence and Unicity of the Normal Form The calculus Λ^{001} satisfies the same form of partial confluence as Λ^∞ does. This time, the subterms considered to be matterless are not only the mute ones, but more generally, those that are not head normalizing. Formally, we define by coinduction the binary relation \sim_{001} on Λ^{001} identifying two terms that are equal up to their non-HN subterms (occurring at the same positions):

$$\frac{t \text{ and } u \text{ are not HN}}{t \sim_{001} u} \quad \frac{}{x \sim_{001} x} \quad \frac{t \sim_{001} u}{\lambda x.t \sim_{001} \lambda x.u} \quad \frac{t_1 \sim_{001} u_1 \quad t_2 \sim_{001} u_2}{t_1 t_2 \sim_{001} u_1 u_2}$$

Proposition 9.2 is a particular case of [57], Theorem 57:

Proposition 9.2. Let $t, t', u, u' \in \Lambda^{001}$. If $t \sim_{001} u$, $t \rightarrow^\infty t'$, $u \rightarrow^\infty u'$, then there exists $t'', u'' \in \Lambda^{001}$ such that $t'' \sim_{001} v''$, $t' \rightarrow^\infty t''$, $u' \rightarrow^\infty u''$.

This partial confluence is also enough to guarantee the unicity of the normal form of a term, when it exists:

Corollary 9.1. Let $t \in \Lambda^{001}$ and u_1, u_2 be two 001-normal forms such that $t \rightarrow^\infty u_1$ and $t \rightarrow^\infty u_2$. Then $u_1 = u_2$.

Note that if $t \in \Lambda$ (t is a finite term), the confluence of the finite λ -calculus easily implies the unicity of the possible 001-NF of t .

9.3.3 Böhm Trees Revisited

Böhm trees can be recovered as the normal forms of a variant of Λ^{001} , once again sketching contents of [57].

First, let \perp be a new constant symbol (*i.e.* \perp cannot be bound). We define Λ_\perp^∞ by the following coinductive grammar:

$$t, u := x \in \mathcal{V} \mid \perp \mid (\lambda x.t) \mid (tu)$$

Then we define Λ_\perp^{001} as the set of terms $t \in \Lambda_\perp^\infty$, such that any infinite branch b of t satisfies $\text{ad}(b) = \infty$. The β -reduction \rightarrow_β is straightforwardly defined on Λ_\perp .

We define the relation \rightarrow_{\perp} as the contextual closure of:

$$\frac{t \text{ is not HN}}{t \rightarrow_{\perp} \perp} \quad \lambda x. \perp \rightarrow_{\perp} \perp \quad \perp u \rightarrow_{\perp} \perp$$

Thus, \rightarrow_{\perp} allows us to rewrite any non-HN term into \perp , with the additional convention that $\perp t$ and $\lambda x. \perp$ are considered as non-HN. The reduction \rightarrow_{\perp} allows us to bypass the relation \sim_{001} of Sec. 9.3.2: for $t, u \in \Lambda^{001}$, $t \sim_{001} u$ just means that $t, u \rightarrow_{\perp} \perp$.

We then set $\rightarrow_{\beta\perp} = \rightarrow_{\beta} \cup \rightarrow_{\perp}$. The set of normal forms of Λ_{\perp}^{001} i.e. the terms $t \in \Lambda^{001}$ such that $t \rightarrow_{\beta\perp} u$ implies $u = t$ is generated by the following coinductive grammar:

$$t, t_k ::= \perp \mid \lambda x_1 \dots x_p. x t_1 \dots t_q \quad (p, q \geq 0)$$

A normal form of Λ_{\perp}^{001} is called a **Böhm tree**.

Strongly converging reductions sequences are easy to define for Λ_{\perp}^{001} , $t \rightarrow_{\beta\perp}^{\infty} t'$ then means that t reduces to t' by means of a s.c.r.s. of Λ_{\perp}^{001} . Moreover, the calculus is confluent:

Proposition 9.3. Let $t_1, t_2, t_3 \in \Lambda_{\perp}^{001}$ such that $t_1 \rightarrow_{\beta\perp}^{\infty} t_2$, $t_1 \rightarrow_{\beta\perp}^{\infty} t_3$. Then there exists $t_4 \in \Lambda_{\perp}^{001}$ such that $t_2 \rightarrow_{\beta\perp}^{\infty} t_4$, $t_3 \rightarrow_{\beta\perp}^{\infty} t_4$.

Proof. This is a particular case of Theorem 61 of [57]. □

The calculus Λ_{\perp}^{001} is weakly normalizing: any term $t \in \Lambda_{\perp}^{001}$ reduces to a normal form (possibly asymptotically). Indeed, if t is not HN, then t reduces into \perp (in one step). If not, t reduces to some $\lambda x_1 \dots x_p. x t_1 \dots t_q$. In turn, the t_k reduce either to \perp or to a HNF. Asymptotically, this gives a Böhm tree.

Chapter 10

Klop's Problem

We saw that intersection type systems enable the characterization of many classes of normalizing terms, such as the *Head Normalizing terms* (Proposition 3.7) or the *Weakly Normalizing (WN) terms* (Proposition 5.1). A term is WN if it can be reduced to a normal form *i.e.* a term without *redexes*. Via the leftmost reduction strategy, Weak normalization can be restated as follows: a term is WN if it is HN and all the arguments of its head variable are WN (Corollary 2.1, which is a direct consequence of Proposition 5.2).

Hereditary Head Normalization (HHN) (Definition 9.5) corresponds to the *coinductive* counterpart of this latter predicate: a λ -term t is HHN if, it is HN and, coinductively, all the arguments of its head variable are themselves HHN. The hereditary head normalization is equivalent to say that the Böhm tree [8] of the term does not hold any occurrence of \perp (see Sec. 9.3.2).

According to Tatsuta [101], the question of finding out a type system characterizing *Hereditary Head Normalizing (HHN) terms* was raised by Klop in a private exchange with Dezani. This question is known as **Klop's Problem**. Klop's Problem was also addressed in [69, 95].

Tatsuta focused his study on *finitary* type systems and showed Klop's problem's answer was negative for them, by (1) noticing that the set of typing derivations in an inductive type system is recursively enumerable and, then proving that the set of HHN terms was not recursively enumerable .

This leaves the question open as to whether an *infinitary* type system is able to characterize HHN terms. One of the contribution of this thesis is proving that Klop's Problem has a positive answer with the right type system, which was published in [114]. We present this result in this chapter.

Hereditary Head Normalization vs. Infinitary Weak Normalization Before presenting our results, let us recall that intersection types systems are also useful to prove that a reduction strategy is complete for normalization *e.g.*, head reduction for head normalization (Proposition 3.8) or the minimal reduction (and thus in particular, the leftmost outermost and the Böhm reduction strategies) for weak normalization (Proposition 5.2)

We now give a brief summary of the main elements of the preliminary chapter on the infinitary λ -calculus (Chapter 9), along with pointers: the Böhm trees without \perp can be seen as the set of normal forms of the infinitary calculus Λ^{001} , presented in Sec. 9.3.2. The fact that the Böhm tree $BT(t)$ does not contain \perp is equivalent to the

strong convergence of first variant of the Böhm reduction strategy¹ (also called the *Hereditary Head Reduction Strategy*) on t . In particular, this implies that t is weakly normalizing w.r.t. Λ^{001} . Indeed, an infinite term is WN (Definition 9.4) if it can be reduced to a NF by at least one *strongly converging reduction sequence* (Definition 9.3) – for short, s.c.r.s.–, which constitute a special kind of reduction sequence of (possibly) infinite length, regarded as *productive* (recall the discussion at the beginning of Sec. 9.3).

A very simple example of strongly converging reduction sequence, which corresponds to a hereditary head normalization, is then following: let $\Delta_f = \lambda x.f(xx)$ and $Y_f = \Delta_f \Delta_f$. Then $Y_f \rightarrow f(Y_f)$, so $Y_f \rightarrow^k f^k(Y_f)$. Intuitively, if k tends toward ∞ , the redex disappear and we get $Y_f \rightarrow^\infty f^\omega$ where f^ω is the (infinite) term $f(f(f(\dots)))$, satisfying $f^\omega = f(f^\omega)$ (Fig. 10.1) and containing a rightward infinite branch. Since f^ω does not contain any redex, f^ω can be seen as the NF of Y_f and Y_f as a HHN term. Equivalently, f^ω is the Böhm tree of Y_f .

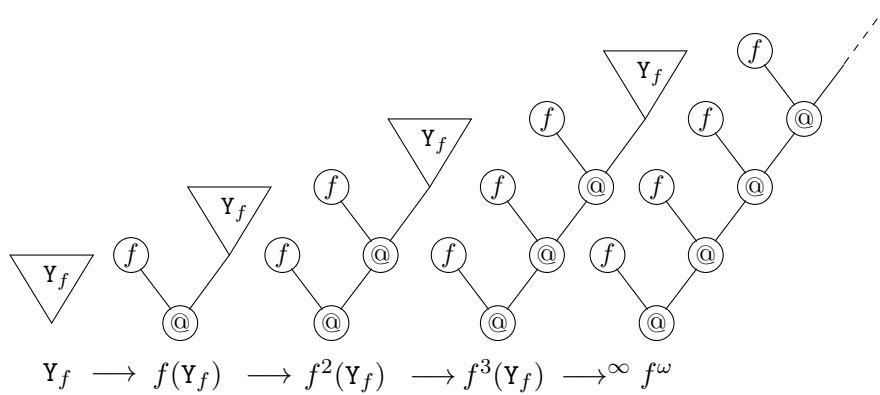


Figure 10.1: Strong Convergence of Y_f (copy of Fig. 9.4, p. 199)

But many s.c.r.s. are not instances of the Böhm reduction strategy. In the finite case, Böhm reduction strategy is complete for weak normalization. This is also known in the case of Λ^{001} (this is a not too difficult consequence of Lemma 54 of [57]), but actually, the type system to be presented gives a semantical proof of this fact.

Tools and Difficulties

We use a quantitative, resource-aware type system to achieve this goal, inspired by that of Gardner and de Carvalho's system \mathcal{R}_0 [22,43] (presented in Sec. 3.2.4). In system \mathcal{R}_0 , intersection types may be represented by *multisets* and typability implies normalization by a very simple (variant of the same) argument *i.e.* reducing a typed redex inside a derivation decreases some non-negative integer measure (Sec. 3.4.3), which entails that the reduction must stop at some point (*cf.* proof of Proposition 3.9) This is unlikely to be adapted in an infinitary framework. However, quantitative type derivations do have very simple and readable combinatorial features *e.g.*, reduction inside a derivation almost comes down to *moving* parts of the initial derivation (see Fig. 3.2 and discussion in Sec. 3.3).

¹We recall from Sec. 9.3.2 that this strategies consists in keeping in reducing at applicative depth 0 (*i.e.* head reduction steps) then, if a head normal form $\lambda x_1 \dots x_p. x t_1 \dots t_q$ is reached, keeping in reducing at a.d. 1 (*i.e.* head-reducing the head arguments t_1, \dots, t_q), then, when there is no more redexes at a.d. 1, reducing at a.d. 2 and so son.

However, it turns out that a direct coinductive adaptation of \mathcal{R}_0 cannot work for two reasons :

- It would lead to the possibility of typing some non-HN terms, like $\Omega = \Delta \Delta$ (with $\Delta = \lambda x.x x$). That is why a validity criterion is needed to discard irrelevant derivations, as in other infinitary frameworks [96]. This validity criterion relies on the idea of **approximability** (see Sec. 10.1.3).
- Moreover, as it turns out, multisets are not fit to formally express such a notion (see Sec. 10.3.4), because multiset intersection is intrinsically non-rigid (Sec. 4.1.2) meaning for instance that we cannot distinguish the two occurrences of σ in $[\sigma, \tau, \sigma]$. This motivates the need for rigid constructions: multisets are then (coinductively) replaced by **sequences**, *i.e.* families indexed by (non necessarily consecutive) integers called **tracks** (Sec. 2.1.1) .

This leads us to define a type system that we call system **S**, in which intersection is represented by sequences of types (intersection is said to be sequential). Tracks constitute the main feature of system **S** presented here. and they allow defining pointers (called **bipositions**) to any symbol nested in a derivation. With those pointers, any type can be tracked through the rules of a whole typing derivation. Our framework is deterministic, *e.g.*, there is a *unique* way to produce a derivation from another one when reducing a redex, contrary to system \mathcal{R}_0 (Sec. 4.1.2).

Once system **S** is properly defined, we can give a positive answer to Klop's Problem: the main result of this chapter (Theorem 10.4) provides a type-theoretical characterization of hereditary head normalizing terms, by suitably adapting the notion of unforgetfulness (Definition 5.2) in order to obtain an infinitary counterpart of Proposition 3.7.

Outline

We try to adapt in an infinitary framework a characterization of WN with Gardner/de Carvalho's system \mathcal{R}_0 . This characterization, which is recalled in Sec. 10.1.1, deeply relies upon the condition of unforgetfulness, controlling the occurrences of the empty type so that no subpart of a normal form can be left untyped. It turns out (Sec. 10.1.3) that coinductive typing raises the notion of *approximable derivation*, both for ensuring soundness and infinitary subject expansion. System **S**, featuring sequential intersection, is introduced in Sec. 10.2. By studying subject reduction in system **S** and comparing it with that of system \mathcal{R}_0 , we observe (Sec. 10.3.4) that multiset intersection is unfit to express approximability, whereas it can be formally done (Sec. 10.4) in system **S**. We formally define approximability in **S** and prove an infinitary subject reduction property in Sec. 10.4, as well as the first part of characterization: if a term t is approximably and unforgetfully typable, then it is hereditary head normalizing. In the last section, we type infinite normal forms and formally prove an infinitary subject expansion property. This allows us to give a type-theoretic characterization of infinitary WN (Theorem 10.4), following the general proof scheme of Sec. 3.3.1. Namely, the following circular implications will be proved: $\langle\langle t \text{ is infinitarily WN} \rangle\rangle \Rightarrow \langle\langle t \text{ is approximably and unforgetfully typable} \rangle\rangle \Rightarrow \langle\langle t \text{ is HHN} \rangle\rangle \Rightarrow \langle\langle t \text{ is infinitarily WN} \rangle\rangle$.

10.1 How to Answer Positively to Klop's Problem

The method of system \mathcal{R}_0 are recalled in Sec. 10.1.1, in particular how weak normalization is characterized by considering unforgetful derivations. In Sec. 10.1.2, we explain how system \mathcal{R}_0 can still be applied to infinite terms. We meet a first use of subject substitution *i.e.* replacing the subject t of a derivation Π by another u which identifies with t in the typed parts of Π . We present the main ideas to solve Klop's Problem in Sec. 10.1.3: we explain (1) how infinitary subject expansion *could* be performed by truncating derivations and taking the joins of directed families (2) how coinduction gives rise *unsound* derivations *e.g.*, derivations typing the non-WN term Ω and how soundness can be retrieved with *approximability*. It will be proved only later (Sec. 10.2) that all this cannot be formally done with multiset intersection.

10.1.1 The Finitary Type System \mathcal{R}_0 and Unforgetfulness

In this section, we give a summary of the methods and ideas of Chapter 3) and of Sec. 5.1 to characterize weak normalization with \mathcal{R}_0 in the finite case, so that Chapter 10 may be read independently (we also give more precise pointers).

System \mathcal{R}_0 (Sec. 3.2.4) features *non-idempotent* intersection types [22, 43], given by the following *inductive* grammar:

$$\sigma, \tau ::= o \mid [\sigma_i]_{i \in I} \rightarrow \tau$$

where the constructor $[\]$ is used for *finite* multisets (I is finite), and the type variable o ranges over a countable set \mathcal{O} . We write $[\sigma]_n$ to denote the multiset containing σ with multiplicity n . The multiset $[\sigma_i]_{i \in I}$ is meant to be the intersection of the types σ_i , taking into account their *multiplicity* (Sec. 3.2.2). In idempotent intersection type systems, the intersections $A \wedge B \wedge A$ and $A \wedge B$ are *de facto* equal, whereas in \mathcal{R}_0 , the multiset types $[\sigma, \tau, \sigma]$ and $[\sigma, \tau]$ are not.

In system \mathcal{R}_0 , a *judgment* is a triple $\Gamma \vdash t : \sigma$, where Γ is a context, *i.e.* a *total* function from the set \mathcal{V} of term variables to the set of multiset types $[\sigma_i]_{i \in I}$, t is a term and σ is a type. The context $x : [\sigma_i]_{i \in I}$ is the context Γ such that $\Gamma(x) = [\sigma_i]_{i \in I}$ and $\Gamma(y) = [\]$ for all $y \neq x$. The multiset union $+$ is extended point-wise on contexts. The set of derivations is defined inductively by the rules below:

$$\frac{}{x : [\tau] \vdash x : \tau} \text{ ax} \qquad \frac{\Gamma; x : [\sigma_i]_{i \in I} \vdash t : \tau}{\Gamma \vdash \lambda x.t : [\sigma_i]_{i \in I} \rightarrow \tau} \text{ abs}$$

$$\frac{\Gamma \vdash t : [\sigma_i]_{i \in I} \rightarrow \tau \quad (\Delta_i \vdash u : \sigma_i)_{i \in I}}{\Gamma +_{i \in I} \Delta_i \vdash t u : \tau} \text{ app}$$

We write $\Pi \triangleright \Gamma \vdash t : \tau$ to mean that the (finite) derivation Π concludes with the judgment $\Gamma \vdash t : \tau$ and $\triangleright \Gamma \vdash t : \tau$ to mean that $\Gamma \vdash t : \tau$ is derivable. No weakening is allowed (**relevance**) *e.g.*, $\lambda x.x$ (resp. $\lambda x.y$) can be typed with $[\tau] \rightarrow \tau$ (resp. $[\] \rightarrow \tau$), but *not* with $[\tau, \sigma] \rightarrow \tau$ (resp. $[\tau] \rightarrow \tau$). System \mathcal{R}_0 enjoys both **subject reduction** and **expansion**, meaning that types are invariant under (anti)reduction (if $t \rightarrow t'$, then $\triangleright \Gamma \vdash t : \tau$ iff $\triangleright \Gamma \vdash t' : \tau$). The following result was developed and proved in Sec. 3.4:

Theorem 10.1 (Proposition 3.7). A term is HN iff it is typable in system \mathcal{R}_0 .

The implication *Typable* \Rightarrow *Head Normalizable* is based on a simple arithmetical argument, similar to the one we will use to prove Lemma 10.7. The converse implication is proven by first, typing the HNF in \mathcal{R}_0 , then, using subject expansion.

Let us now recall the characterization of weak normalization in system \mathcal{R}_0 (presented throughout in Sec. 5.1). Notice that if x is assigned $[\] \rightarrow \tau$, then xt is typable with type τ for any term t – which is left untyped as a subterm of xt – even if t is not HN. In order to characterize WN, we must guarantee somehow that every subterm (that cannot be erased during a reduction sequence *i.e.* not as t in $(\lambda x.y)t \rightarrow y$) is typed: $[\]$ should not occur at bad positions in a derivation Π . Actually, it is enough to only look at the judgment concluding Π . We recall that $[\]$ occurs negatively in $[\] \rightarrow \tau$ and that $[\]$ occurs positively (resp. negatively) in $[\sigma_i]_{i \in I} \rightarrow \tau$ if $[\]$ occurs positively (resp. negatively) in τ or negatively (resp. positively) in some σ_i . We say here that judgment $\Gamma \vdash t : \tau$ is **unforgetful** when $[\]$ occurs neither negatively in Γ nor positively in τ (Definition 5.2). Here is the non-idempotent counterpart of [68], Theorem 4.13:

Theorem 10.2 (Proposition 5.1). A term t is WN iff it is typable in \mathcal{R}_0 inside an **unforgetful** judgment.

A straightforward induction on the structure of t shows that, if t is a NF and $\Pi \triangleright \Gamma \vdash t : \tau$ is unforgetful, then every subterm of t is typed in Π . Thus, no part of the normal form of a term t is “invisible” in an unforgetful derivation. This roughly justifies why the above theorem holds (see Sec. 5.1.2 for more explanations on unforgetfulness).

For now, it is enough to keep in mind that a sufficient condition of unforgetfulness is to be **$[\]$ -free**: t is WN as soon as $\triangleright \Gamma \vdash t : \tau$, where Γ and τ do not contain $[\]$.

Throughout this article, we will frequently invoke system \mathcal{R}_0 and this section to illustrate or motivate our choices.

10.1.2 Finitarily Typing the Infinite Terms

In this section, we explain how system \mathcal{R}_0 can still apply to infinite terms. This is the only section of this chapter in which infinite terms of Λ^∞ (and not just Λ^{001}) are considered. Incidentally, we use an argument of subject Substitution, that will be a key tool of Sec. 10.1.3 and 10.5.4.

If we allow the judgments of system \mathcal{R}_0 to have infinite subjects $t \in \Lambda^\infty$ (the types and the derivations being still *inductively* defined), we obtain a variant of system \mathcal{R}_0 – let us call it \mathcal{R}_0^∞ . system \mathcal{R}_0^∞ is essentially finite, but infinite terms can appear as *untyped* arguments of applications *e.g.*, f^ω in:

$$\Pi'_1 = \frac{\overline{f : [[\] \rightarrow o] \vdash f : [\] \rightarrow o}}{f : [[\] \rightarrow o] \vdash f^\omega : o}$$

Note that Π'_1 is finite (it contains only two judgments, and finite types and contexts).

All the important properties of system \mathcal{R}_0 still hold for system \mathcal{R}_0^∞ :

- Weighted subject reduction (Proposition 3.6), subject expansion (Proposition 3.3). Indeed, those propositions are proven by induction on the structure of the derivations (*not* by induction on the terms).
- Head normal forms of Λ^∞ are easily $\mathcal{R}_{0,w}$ -typable: take the derivation in the upper left corner of Fig. 3.4, p. 100.
- If a \mathcal{R}_0^∞ -typed term $t \in \Lambda^\infty$ is head reducible, then its head redex is typed as in Remark 3.13. This point is crucial to prove that if a term t is \mathcal{R}_0^∞ -typed, then the

head reduction strategy terminates on t (remember the proof of Proposition 3.9). From Remark 9.1, some term $t \in \Lambda^\infty$ can be *headless* (i.e. deprived of a head redex and of a head variable): since a headless term is not head reducible while being not head normal, this could *a priori* make this argument of termination fail. However, Lemma 10.1 entails that no headless term is \mathcal{R}_0^∞ -typable (see below), so that Proposition 3.9 still holds!

Lemma 10.1. Let Π a \mathcal{R}_0^∞ -derivation typing a term $t \in \Lambda^\infty$. For all $b \in \text{supp}(t)$, if $b \in \{0, 1\}^*$, then $b \in \hat{\Pi}$.

Proof. By induction on b , using the fact that only the argument u of a typed application $t u$ can be left untyped. \square

If $t \in \Lambda^\infty$ is headless, then it has a (unique) infinite branch b such that $\text{ad}(b) = 0$ (i.e. $b \in \{0, 1\}^\omega$). If t were \mathcal{R}_0^∞ -typable with a derivation Π , Lemma 10.1 would entail that contains all the finite prefixes of b i.e. Π would be infinite. This impossible. Thus, if t is \mathcal{R}_0^∞ -typable, then t is head-formed.

All the discussion above entails that, for all term $t \in \Lambda^\infty$, t is HN iff t is \mathcal{R}_0^∞ -typable iff the head reduction strategy terminates on t (in a finite number of steps). The proof of this equivalence follows the scheme $\langle t \text{ is } \mathcal{R}_0^\infty\text{-typable} \rangle \Rightarrow \langle \text{the head reduction strategy Terminates on } t \rangle \Rightarrow \langle t \text{ is HN} \rangle \Rightarrow \langle t \text{ is } \mathcal{R}_0^\infty\text{-typable} \rangle$ outlined in Sec. 3.3.1.

Let us explain now why this is still true while considering infinitary reduction \rightarrow^∞ . This relies upon a **subject Substitution** property, which means that we can freely modify the untyped parts of the subject a derivation Π without compromising the correctness of Π . The notion of typed position (Definition 3.1) straightforwardly extends to \mathcal{R}_0^∞ -derivations:

Lemma 10.2 (Subject Substitution). Let $t, u \in \Lambda^\infty$. If $\Pi \triangleright_{\mathcal{R}_0^\infty} \Gamma \vdash t : \tau$, $\hat{\Pi} \subseteq \text{supp}(u)$ and, for all $b \in \hat{\Pi}$, $u(b) = t(b)$, then there exists a \mathcal{R}_0^∞ -derivation Ψ such that $\Psi \Gamma \vdash u : \tau$ and $\hat{\Psi} = \hat{\Pi}$.

Proof. Straightforward by induction on Π . \square

This Lemma is illustrated in Fig. 10.2. Note that if Π types t and t and u differs only for positions $b \in \{0, 1, 2\}^*$ such that $|b| > \text{sz}(\Pi)$, then Lemma 10.2 can be applied on Π .

Now, assume that $t \rightarrow^\infty t' = \lambda x_1 \dots x_p. x t_1 \dots t_q$ (in the sense of Sec. 9.3.1) and more precisely, that $t = t_0 \xrightarrow{b_0} \dots t_n \xrightarrow{b_n} \dots \rightarrow^\infty t'$. Then, let Π' a \mathcal{R}_0^∞ -derivation typing t' . By strong convergence, there is a rank N , such that, for all $n \geq N$, $|b_n| > \text{sz}(\Pi')$. In particular, t' and t_N do not differ for positions $b \in \{0, 1, 2\}^*$ such that $|b| \leq \text{sz}(\Pi')$ and we can apply Lemma 10.2, thus obtaining a \mathcal{R}_0^∞ -derivation Π_N typing t_N . After N steps of expansion, we obtain from Π_N a derivation Π typing t . Thus, if t is infinitarily HN, then it is $\mathcal{R}_{0,w}$ -typable, so that we have proven the equivalence between $\langle t \text{ strongly converges to a HNF} \rangle$, $\langle t \text{ is } \mathcal{R}_0^\infty\text{-typable} \rangle$ and $\langle \text{the head reduction strategy terminates on } t \rangle$.

Remark 10.1. The equivalence $\langle t \text{ strongly converges to a HNF iff } t \text{ reduces (in a finite number of steps) to a HNF} \rangle$ is obvious when considering s.c.r.s. of Λ^{001} . Indeed:

- The head redex of a term t , when it exists, is the unique redex of null applicative depth.

- Moreover, if t is head reducible, $t \xrightarrow{b} t'$ with $\text{ad}(b) \geq 1$, then t' is also head reducible.
- In a s.c.r.s.of Λ^{001} , only a finite number of steps can be head reductions (contrary to Λ^∞ , cf. $\Omega_3 \rightarrow \Omega_3 \omega_3 \dots$).

Actually, with Proposition 9.2, this easily entails² that HHN is equivalent with 001-WN. But we will give an alternative and purely semantical proof (not using confluence) in Theorem 10.4.

10.1.3 Infinitary Subject Expansion by Means of Truncation

In order to adapt the proof of Proposition 10.2, the idea is to type NF (in this section, f^ω) in unforgetful judgments, and then proceed by (possibly infinite) expansion to get a typing derivation of the expanded term (in this section, Y_f). We try to give a few intuitions about how this may be performed. This will allow us to present the key notions of **truncation** and **approximability**. For that, Fig. 10.2 illustrates the main ideas of this section.

For that, we admit temporarily that there is an infinitary version of \mathcal{R}_0 , referred here as \mathcal{R} (system \mathcal{R} will be presented in Sec. 13.1.3). System \mathcal{R} allows infinite multisets (e.g., $[o]_\omega$ is the multiset in which o occurs with an infinite multiplicity, s.t. $[o]_\omega = [o] + [o]_\omega$) and proofs of infinite depth.

Let us consider the following \mathcal{R} -derivation Π' (presented as fixpoint). It is also represented on top of Fig. 10.2.

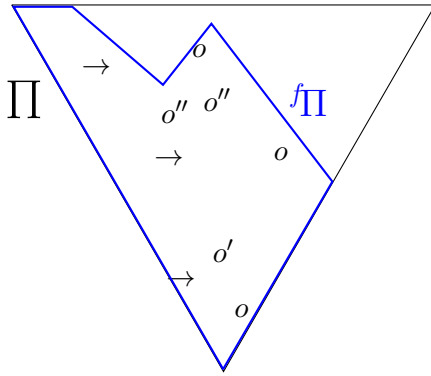
$$\Pi' = \frac{\frac{f : [[o] \rightarrow o] \vdash f : [o] \rightarrow o \quad \text{ax}}{f : [[o] \rightarrow o] \vdash f^\omega : o} \quad \frac{\Pi'}{f : [[o] \rightarrow o]_\omega \vdash f^\omega : o}}{f : [[o] \rightarrow o]_\omega \vdash f^\omega : o} \text{app}$$

Now, Π' is an $[]$ -free derivation (see end of Sec. 10.1.1) typing f^ω . This the kind of derivation we want to expand in order to get a derivation Π typing Y_f and proving that it is WN. Since $Y_f \rightarrow^\infty f^\omega$ (infinite number of reduction steps), we are stuck. But notice that Π' can be truncated into the derivation Π'_n below for any $n \geq 1$ (we set $\Gamma_n = f : [[o] \rightarrow o]_{n-1} + [[] \rightarrow o]$ and write $t : \tau$ instead of $x : [\tau] \vdash x : \tau$ for **ax**-rules):

$$\Pi'_n = \frac{\frac{\frac{\frac{\Gamma_1 \vdash f : [] \rightarrow o}{f : [o] \rightarrow o \quad \Gamma_1 \vdash f^\omega : o}}{\Gamma_2 \vdash f^\omega : o}}{\vdots}}{\frac{f : [o] \rightarrow o \quad \Gamma_{n-1} \vdash f^\omega : o}{\Gamma_n \vdash f^\omega : o}}$$

Derivations Π'_3 and Π'_4 are represented in the middle of Fig. 10.2. By **truncation**, we mean that the finite derivation Π'_n can be (informally) obtained from the infinite one Π' by erasing some elements from the infinite multisets appearing in the derivation. Conversely, we see that Π' is the graphical **join** of the Π'_n : Π' is obtained by superposing suitably all the derivations Π'_n on the same (infinite) sheet of paper.

²Indeed, if t is 001-WN and $t \rightarrow^* \lambda x_1 \dots x_p. x t_1 \dots t_q$, then this proposition implies that t_1, \dots, t_q are also 001-WN



- Π is an infinite approximable derivation.
- A finite number of symbols (arrows or type variables) have been selected in the derivation, in various judgments.
- By approximability, there is a finite derivation $f\Pi$, that is a truncation of Π and contains all the selected symbols.

Figure 10.3: Approximability

However, we are still stuck: we do not know how to expand Π'_n because, although being finite, it still types the ∞ -reduced term f^ω . But notice that we can replace f^ω by $f^k(Y_f)$ inside Π'_k whenever $k \geq n$, because those two terms do not differ in the typed parts of Π' (notion of **subject substitution**, Sec. 10.1.2). This yields a derivation $\Pi_n^k \triangleright \Gamma_n \vdash f^k(Y_f) : o$. We have represented Π_3^4 and Π_4^4 at the bottom of Fig. 10.2. This time, Π_n^k is a derivation typing $f^k(Y_f)$, the rank k reduct of Y_f , so we can expand it k times, yielding a derivation Π_n (Π_n does not depend on k). Then, we can rebuild a derivation Π such that each Π_n is a truncation of Π the same way Π'_n is of Π' (Π can be seen as the “graphical” join of the Π_n).

Thus, the ideas of truncation, subject substitution and join guides us about how to perform ∞ -subject expansion (Sec. 10.5.4). The particular form of Π_n and Π does not matter (but they are given in Appendix A.1 for the curious reader). Let us just say here that the Π_n involve a family of types $(\rho)_{n \geq 1}$ inductively defined by $\rho_1 = [] \rightarrow o$ and $\rho_{n+1} = [\rho_k]_{1 \leq k \leq n} \rightarrow o$ and Π involves an infinite type ρ satisfying $\rho = [\rho]_\omega \rightarrow o$: if t and u are typed with ρ , then tu may be typed with o .

Unfortunately, it is not difficult to see that the type ρ also allows the non HN term $\Delta\Delta$ to be typed. Indeed, $x : [\rho]_\omega \vdash xx : o$ is derivable, so $\vdash \Delta : \rho$ and $\vdash \Delta\Delta : o$ also are.

This last observation shows that the naive extension of the standard non-idempotent type system to infinite terms is unsound as non HN terms can be typed (actually, every term is typable in system \mathcal{R} , which is the main contribution of Chapter 12). Therefore, we need to discriminate between sound derivations (like Π typing Y_f) and unsound ones. For that, we define an infinitary derivation Π to be **valid** or **approximable** when Π admits finite truncations, generally denoted by $f\Pi$ – that are finite derivations of \mathcal{R}_0 –, so that any fixed finite part of Π is contained in some truncation $f\Pi$ (for now, a finite part of Π informally denotes a finite selection of graphical symbols of Π , a formal definition is given in Sec. 10.4.1). This informal definition is illustrated by Fig. 10.3.

10.2 Intersection by means of Sequences

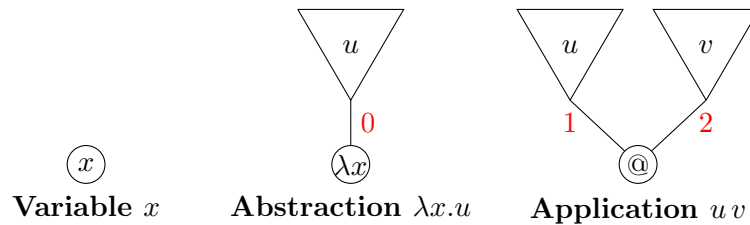
We recall from Sec. 4.1.2 that derivations of system \mathcal{R} are not *rigid* labelled trees. The whole point of system \mathbf{S} is to define a rigid counterpart of system \mathcal{R} . This section is dedicated to the presentation of system \mathbf{S} . We first explain why we should want a *rigid* type system (Sec. 10.2.1) and a few directions to construct system \mathbf{S} , including

the way positions of system \mathbf{S} will naturally *collapse* onto positions of λ -terms. We also define (labelled or not) forests. In Sec. 10.2.2, we define the types of system \mathbf{S} as (rigid) labelled trees and develop the formalism of the *sequences* *i.e.* families of objects indexed by families on integers. The derivations of system \mathbf{S} are finally defined in Sec. 10.2.3, also as rigid labelled trees, along with pointers called *bipositions*, that allow pointing to the type symbols nested in a derivation.

10.2.1 Towards System \mathbf{S}

In this section, we recall the notions of rigid labelled trees and that of track, from Sec. 2.1.1. We reuse the notations of this section. We also define (labelled or not) forests, that were not needed until now. Tracks are a key tool to later define a “deterministic” intersection type system.

First, remember that “ λ -terms can be seen as labelled trees following this pattern” (Figure 2.4):



Nodes are labelled by x , λx (x ranging over \mathcal{V} , a countable set of term variables) or $@$. The integers that label edges are called **tracks** *e.g.*, 0 is dedicated to abstractions, 1 to application left-hand sides, 2 to application arguments.

As it will turn out, we will need to define a rigid intersection type system (ITS). This will be system \mathbf{S} (Sec. 10.2.3). Rigidity will allow defining pointers called *bipositions*. It is easier to explain why multiset intersection (as in system \mathcal{R}_0) is unfit to characterize WN in the infinitary case *after* having studied Subject Reduction in system \mathbf{S} . We will see to that in Sec. 10.3.4.

In system \mathbf{S} , derivations and types will also be rigid trees *i.e.* trees with labelled edges (*i.e.* edges labelled with tracks). In ITS, the argument u of an application $t u$ may be typed several times and so, an **app**-rule may have several argument derivations. Thus, more than one track must be dedicated to arguments derivations (not only track 2). We allow then an argument derivation to be placed on *any* track ≥ 2 . A track ≥ 2 is then called **argument track**.

For instance, a subderivation on track 9 will be a subderivation typing the argument u of the underlying λ -term $t u$. As a subterm, u is on track 2. Thus, the subderivation on track 9 will type the subterm on track 2. This motivates the notion of **collapse** (written \bar{k}) of a track k , setting $\bar{k} = \min(k, 2)$.

A position of $t \in \Lambda^{001}$ is concatenation of tracks *i.e.* a word b on the alphabet $\{0, 1, 2\}$ ($b \in \{0, 1, 2\}^*$). A position of \mathbf{S} will be a word $a \in \mathbb{N}^*$ *i.e.* a word on the alphabet \mathbb{N} . The notion of collapse can be extended letter-wise on \mathbb{N}^* *e.g.*, $\overline{0 \cdot 5 \cdot 1 \cdot 3 \cdot 2} = 0 \cdot 2 \cdot 1 \cdot 2 \cdot 2$. The **applicative depth** $\text{ad}(a)$ of a is the number of argument tracks it holds *e.g.*, $\text{ad}(0 \cdot 3 \cdot 2 \cdot 1 \cdot 1) = 2$ and $\text{ad}(0 \cdot 1 \cdot 0 \cdot 0 \cdot 1) = 0$.

We recall that **tree** A of \mathbb{N}^* is a non-empty subset of \mathbb{N}^* that is downward-closed for the prefix order ($a \leq a' \in A$ implies $a \in A$). A **forest** is a set of the form $A \setminus \{\varepsilon\}$ for some tree A such that $0, 1 \notin A$. Formally, a **labelled tree** T (resp. **labelled forest** F) is a function to a set Σ , whose domain, called its **support** $\text{supp}(T)$ (resp. $\text{supp}(F)$),

is a tree (resp. a forest). If $U = T$ or $U = F$, then $U|_a$ is the function defined on $\{a_0 \in \mathbb{N}^* \mid a \cdot a_0 \in \text{supp}(U)\}$ and $U|_a(a_0) = U(a \cdot a_0)$. If U is a labelled tree (resp. a labelled forest and $a \neq \varepsilon$), then $U|_a$ is a tree.

10.2.2 Rigid Types

We start now to implement the ideas of Sec. 10.2.1 by defining rigid types. Every edge inside a type or a derivation of system \mathbf{S} must receive a track for label.

Let A be a set. A **(partial) sequence** over A is a family $F = (a_k)_{k \in K}$ s.t. $K \subseteq \mathbb{N} \setminus \{0, 1\}$ and $a_k \in A$ for all $k \in K$. We say a_k is *placed on track k* inside $F = (a_k)_{k \in K}$ and K is the set of **roots** of F : we write $K = \text{Rt}(F)$. If $a, b \in A$, then $(3 \cdot a, 5 \cdot b, 8 \cdot a)$ is the sequence $F = (a_k)_{k \in \{3,5,8\}}$ s.t. $a_3 = a, a_5 = b, a_8 = a$. We define the **disjoint union** of two sequences when their roots are disjoint. For instance, $(2 \cdot a, 3 \cdot b, 8 \cdot a) \uplus (4 \cdot a, 9 \cdot c) = (2 \cdot a, 3 \cdot b, 4 \cdot a, 8 \cdot a, 9 \cdot c)$. But notice that $(2 \cdot a, 3 \cdot b, 8 \cdot a) \uplus (3 \cdot b, 9 \cdot c)$ is not defined, because track 3 is in the roots of both sequences (**track conflict**). Thus, \uplus is not a total operator, but it is associative and commutative.

Let \mathcal{O} be a countable set of types variables (metavariable o). The sets of (rigid) types Typ^{111} (metavariables T, S_i, \dots) is coinductively defined by:

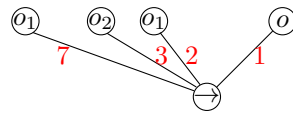
$$T, S_k ::= o \parallel (S_k)_{k \in K} \rightarrow T$$

Remark 10.2. The **sequence type** $F = (S_k)_{k \in K}$ is a sequence of types in the above meaning and is seen as an intersection of the types S_k .

The equality between two types (resp. sequence types) may be defined by mutual coinduction: $F \rightarrow T = F' \rightarrow T'$ if $F = F'$ and $T = T'$ and $(T_k)_{k \in K} = (T'_k)_{k \in K'}$ if $K = K'$ and for all $k \in K, T_k = T'_k$. It is a **syntactic equality** (unlike multiset equality). Intuitively, \mathbf{S} -type can only be written in one way.

The *support of a type* (resp. *a sequence type*), which is a tree of \mathbb{N}^* (resp. a forest), is defined by mutual coinduction: $\text{supp}(o) = \{\varepsilon\}$, $\text{supp}(F \rightarrow T) = \{\varepsilon\} \cup \text{supp}(F) \cup 1 \cdot \text{supp}(T)$ and $\text{supp}((T_k)_{k \in K}) = \cup_{k \in K} k \cdot \text{supp}(T_k)$.

For instance, $(7 \cdot o_1, 3 \cdot o_2, 2 \cdot o_1) \rightarrow o$ is represented by:



Thus, for types, track 1 is dedicated to the target of arrows and \mathbf{S} -types really are *rigid* labelled trees in the sense of Sec. 2.1.1 *i.e.* trees whose nodes and edges are both labelled, contrary to system \mathcal{R}_0 .

Example 10.1.

- Let $S = (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o'$. Then $\text{supp}(S) = \{\varepsilon, 1, 2, 3, 8\}$. We have $S(\varepsilon) = \rightarrow$, $S(1) = o'$ (since $S(1)$ is the target of the arrow), $S(2) = o$, $S(3) = o'$ and $S(8) = o$.
- Let $F = (2 \cdot o', 4 \cdot S)$. Then F is a sequence type such that $\text{Rt}(F) = \{2, 4\}$, $F|_2 = o'$ and $F|_4 = S$. Thus, $\text{supp}(F) = 2 \cdot \text{supp}(o') \cup 4 \cdot \text{supp}(S)$, so that $\text{supp}(F) = \{2, 4, 4 \cdot 1, 4 \cdot 2, 4 \cdot 3, 4 \cdot 8\}$. We have $F(2) = o'$, $F(4) = S(\varepsilon) = \rightarrow$, $F(4 \cdot 1) = S(1) = o'$, $F(4 \cdot 2) = S(2) = o$, $F(4 \cdot 3) = o'$ and $F(4 \cdot 8) = o$.

A type of \mathbf{Typ}^{111} is in the set \mathbf{Typ}^{001} if its support does not hold an infinite branch ending by 1^ω . This restriction means that we may only have finite series of arrows in a type. Indeed, 001-NF contain finite series of abstraction nodes only.

Henceforth, in this chapter, we just write \mathbf{Typ} for \mathbf{Typ}^{001} and consider only sequence types which hold types from \mathbf{Typ} . We write $()$ for the sequence type whose support is empty and a sequence type of the form $(k \cdot T)$ is called a *singleton sequence type*.

We say that a family of sequence types $(F^i)_{i \in I}$ is **disjoint** if the $\mathbf{Rt}(F^i)$ (i ranging over I) are pairwise disjoint. This means that there is no overlapping of typing information between the F^i . In that case, we define the **join** of $(F^i)_{i \in I}$ as the sequence type F s.t. $\mathbf{Rt}(F) = \uplus_{i \in I} \mathbf{Rt}(F^i)$ and, for all $k \in \mathbf{Rt}(F)$, $F|_k = F^i|_k$ where i the unique index s.t. $k \in \mathbf{Rt}(F^i)$.

10.2.3 Rigid Derivations

A **(rigid) context** C is a *total* function from \mathcal{V} to the set of sequence types. The **domain** of C is $\text{dom}(C) = \{x \in \mathcal{V} \mid C(x) \neq ()\}$. We define the join of contexts pointwise. If $\text{dom}(C) \cap \text{dom}(D) = \emptyset$, we may write $C; D$ instead of $C \uplus D$. A **judgment** is a triple of the form $C \vdash t : T$, where C is a context, t a 001-term and $T \in \mathbf{Typ}$. A **sequence judgment** is a sequence of judgments $(C_k \vdash t : T_k)_{k \in K}$. For instance, if $5 \in K$, then the judgment on track 5 is $C_5 \vdash t : S_5$.

The set \mathbf{Deriv} of **(rigid) derivations** (metavariable P) is defined coinductively by the following rules:

$$\frac{}{x : (k \cdot T) \vdash x : T} \text{ax} \qquad \frac{C; x : (S_k)_{k \in K} \vdash t : T}{C \vdash \lambda x. t : (S_k)_{k \in K} \rightarrow T} \text{abs}$$

$$\frac{C \vdash t : (S_k)_{k \in K} \rightarrow T \quad (D_k \vdash u : S_k)_{k \in K}}{C \uplus_{k \in K} D_k \vdash t u : T} \text{app}$$

In the axiom rule, k is called an **axiom track**. In the **app**-rule, the contexts must be disjoint, so that no track conflict occurs. Otherwise, **app**-rule cannot be applied. It is not difficult to find an equivalent of Lemma 9.1 for derivations of system \mathbf{S} , that describe them “node-wise”.

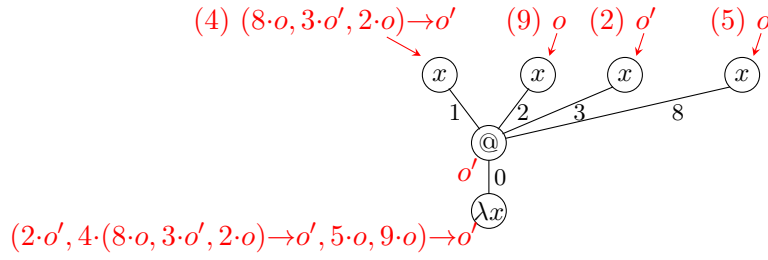
System \mathbf{S} is very low-level. Indeed, the **app** rule can be restated as follows:

$$\frac{C \vdash t : (S_k)_{k \in K} \rightarrow T \quad (D_k \vdash u : S'_k)_{k \in K'} \quad (S_k)_{k \in K} = (S'_k)_{k \in K'}}{C \uplus_{k \in K} D_k \vdash t u : T} \text{app}$$

We notice that the domain $(S_k)_{k \in K}$ of the type of t must be equal to the sequence of the types given to the argument u . The equality between sequence types is *syntactic* and is far more constraining than multiset equality used for system \mathcal{R} , which works modulo “nested permutation” e.g., $[o, o', [o', o'', o_0] \rightarrow o] = [[o'', o', o_0] \rightarrow o, o', o]$ (this observation is formalized in Sec. 13.1.1).

Example 10.2.

- In order to gain space, we do not write right-hand sides of axiom rules. We indicate the track of argument derivations between brackets e.g., $x : (2 \cdot o')$ [3] means that judgment $x : (2 \cdot o') \vdash x : o'$ is on track 3.

Figure 10.4: The Derivation P_{ex}

$$P_{\text{ex}} = \frac{\frac{x : (4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o')}{x : (2 \cdot o', 4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o', 5 \cdot o, 9 \cdot o) \vdash xx : o'} \quad \frac{x : (9 \cdot o) \quad [2]}{x : (2 \cdot o') \quad [3]} \quad \frac{x : (5 \cdot o) \quad [8]}{x : (2 \cdot o', 4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o', 5 \cdot o, 9 \cdot o) \vdash xx : o'}}{\vdash \lambda x.xx : (2 \cdot o', 4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o', 5 \cdot o, 9 \cdot o) \rightarrow o'}$$

In the **ax**-rule concluding with $x : (5 \cdot o) \vdash x : o$, the axiom track is 5.

- The derivation P_{ex} is also represented in Fig. 10.4 (in the style of Fig. 4.1, p. 106). Axiom tracks (of **ax**-rules) are indicated before the assigned type *e.g.*, $(9) o$ on x represents the **ax**-rule concluding with $x : (9 \cdot o) \vdash x : o$ (the axiom track is 9). The edges between nodes correspond to edges between judgment *e.g.*, the black track 8 indicates that the **app**-rule has an argument premise on track 8 (which is $x : (5 \cdot o) \vdash x : o$).

We can define the **support** of a derivation $P \triangleright C \vdash t : T$: $\text{supp}(P) = \varepsilon$ if P is an axiom rule, $\text{supp}(P) = \{\varepsilon\} \cup 0 \cdot \text{supp}(P_0)$ if $t = \lambda x.t_0$ and P_0 is the subderivation typing t_0 , $\text{supp}(P) = \{\varepsilon\} \cup 1 \cdot \text{supp}(P_1) \cup_{k \in K} k \cdot \text{supp}(P_k)$ if $t = t_1 t_2$, P_1 is the left subderivation typing t_1 and P_k the subderivation typing t_2 on track k . The P_k ($k \in K$) are called **argument derivations**.

In Example 10.2 (and Fig. 10.4), $\text{supp}(P_{\text{ex}}) = \{\varepsilon, 0 \cdot 1, 0 \cdot 2, 0 \cdot 3, 0 \cdot 8\}$ and we have $P_{\text{ex}}(0 \cdot 8) = x : (5 \cdot o) \vdash x : o$. We say that this judgment is on the **argument track 8** of the **app**-rule at position 0.

When we forget about tracks, a sequence naturally collapses on a multiset *e.g.*, $(3 \cdot a, 5 \cdot b, 8 \cdot a)$ collapses on $[a, b, a]$.

If we perform this collapse coinductively, then the types of **S** will collapse on types of **R** and the derivations of **S** will collapse on derivations of **R**. For instance, the derivation P_{ex} above collapses on the derivation of Sec. 4.1.1:

$$\Pi_{\text{ex}} = \frac{\frac{x : [o, o', o] \rightarrow o' \quad x : [o] \quad x : [o'] \quad x : [o]}{x : [o', [o, o', o] \rightarrow o', o, o] \vdash xx : o'}}{\vdash \lambda x.xx : [o', [o, o', o] \rightarrow o', o, o] \rightarrow o'}$$

As noticed above, the **app**-rule of system **S** is far more constraining than that of system **R**. For this reason, one may think that system **S** is less expressive than system **R**. However, it may be shown that any derivation of system **R** can be represented by a derivation of system **S** (Theorem 13.1 in Chapter 13).

For the proofs of Sec. 10.5, it will be useful to have the following notations:

Notation 10.1. Assume $a \in P$ and $a = a_* \cdot 1^n$. Then $t|_{a_*}$ is of the form $t|_a t_1 \dots t_n$. Let then $\text{ArgTr}^i(a)$ (for $1 \leq i \leq n$) be the set of the tracks of argument derivations typing the i -th argument t_i below a and $\text{ArgPos}(a)$ is the set of positions of those subderivations. For instance, in P_{ex} , $\text{ArgTr}^1(01) = \{2, 3, 8\}$ and $\text{ArgPos}^1(01) = \{02, 03, 08\}$ (with $a_0 = 0$). Formally, we set $1 \leq i \leq n$, $\text{ArgTr}^i(a) = \{k \geq 2 \mid a_* \cdot 1^{n-i} \cdot k \in P\}$ and $\text{ArgPos}^i(a) = a_* \cdot 1^{n-i} \cdot \text{ArgTr}^i(a) = \{a_* \cdot 1^{n-i} \cdot k \in P \mid k \geq 2\}$.

10.3 Statics and Dynamics

10.3.1 Bipositions and Bisupport

Thanks to rigidity, we can identify and point to every part of a derivation (as suggested in Sec. 2.1.1), thus allowing to formulate many useful notions.

If $a \in \text{supp}(P)$, then a points to a judgment inside P typing $t|_{\bar{a}}$. We write this judgment $\mathbb{C}(a) \vdash t|_{\bar{a}} : \mathbb{T}(a)$: we say a is an **(outer) position** of P . The context $\mathbb{C}(a)$ and the type $\mathbb{T}(a)$ should be written $\mathbb{C}^P(a)$ and $\mathbb{T}^P(a)$ but we often omit P . From now on, we shall also write $t|_a$ and $t(a)$ instead of $t|_{\bar{a}}$ and $t(\bar{a})$.

Example 10.3. Let us have a second look at Example 10.2, in which the right-hand sides of **ax**-rules were not written:

$$P_{\text{ex}} = \frac{\frac{x : (4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o')}{x : (9 \cdot o) \text{ [2]}} \quad \frac{x : (2 \cdot o') \text{ [3]}}{x : (5 \cdot o) \text{ [8]}}}{x : (2 \cdot o', 4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o', 5 \cdot o, 9 \cdot o) \vdash xx : o'} \frac{}{\vdash \lambda x.xx : (2 \cdot o', 4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o', 5 \cdot o, 9 \cdot o) \rightarrow o'}$$

We write \mathbb{C} and \mathbb{T} instead of $\mathbb{C}^{P_{\text{ex}}}$ and $\mathbb{T}^{P_{\text{ex}}}$ respectively, and 01 instead of $0 \cdot 1$, etc. Example 10.1 may be of some use, since it also features the type $S = (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o$, present in P_{ex} .

- $P_{\text{ex}}(01) = x : (4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o) \vdash x : (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o$, so that $\mathbb{C}(01) = x : (4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o)$ *i.e.* $\mathbb{C}(01)(x) = (4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o)$. So, for instance, $\mathbb{C}(01)(x)(4) = \rightarrow$, $\mathbb{C}(01)(x)(43) = o'$, $\mathbb{T}(01)(\varepsilon) = \rightarrow$, $\mathbb{T}(01)(1) = o'$.
- Likewise, $P_{\text{ex}}(03) = x : (2 \cdot o') \vdash o'$, so that $\mathbb{C}(03) = x : (2 \cdot o')$ and $\mathbb{T}(03) = o'$. Thus, $\mathbb{C}(03)(x)(2) = o'$ and $\mathbb{T}(03)(\varepsilon) = o'$.
- We also have $\mathbb{C}(0)(x) = (2 \cdot o', 4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o', 5 \cdot o, 9 \cdot o)$. Thus, $\mathbb{C}(0)(x)(2) = o'$ and $\mathbb{C}(0)(x)(42) = o$.

This example motivates the notion of **bipositions**: a biposition (metavariable p) is a pointer into a type nested in a judgment of a derivation. A pair (a, c) is a **right biposition** of P if $a \in \text{supp}(P)$ and $c \in \text{supp}(\mathbb{T}(a))$. A triple $(a, x, k \cdot c)$ is a **left biposition** if $a \in \text{supp}(P)$, $x \in \mathcal{V}$ and $k \cdot c \in \text{supp}(\mathbb{C}(a)(x))$.

Definition 10.1. The **bisupport** of a derivation P , written $\text{bisupp}(P)$, is the set of its (right or left) bipositions.

We consider a derivation as a *function* from its bisupport to the set $\mathcal{O} \cup \{\rightarrow\}$ and write now $P(a, c)$ for $\mathbb{T}^P(a)(c)$ and $P(a, x, k \cdot c)$ for $\mathbb{C}^P(a)(x)(k \cdot c)$ *e.g.*, with P_{ex} from Example 10.3, $P_{\text{ex}}(01, \varepsilon) = \rightarrow$, $P_{\text{ex}}(01, 1) = o'$, $P_{\text{ex}}(01, x, 43) = o'$, $P_{\text{ex}}(03, x, 2) = o'$.

10.3.2 Quantitativity and Coinduction

If $a \in A := \text{supp}(P)$ and $x \in \mathcal{V}$, we set³ $\text{Ax}_a^P(x) = \{a_0 \in A \mid a \leq a_0, t(a) = x, \nexists a'_0, a \leq a'_0 \leq a_0, t(a'_0) = \lambda x\}$ (positions of **ax**-rules in P above a typing occurrences of x that are not bound w.r.t. a). If $a_0 \in A$ is an axiom, we write $\text{tr}^P(a_0)$ for its associated axiom track *e.g.*, $\text{tr}^P(08) = 5$ in the example above. Usually, P is implicit and we just write $\text{Ax}_a(x)$ and $\text{tr}(a)$.

The presence of an infinite branch inside a derivation makes it possible that a type in a context is not created in an axiom rule. For instance, we set, for all $k \geq 2$, $j_k = f : (i \cdot (2 \cdot o) \rightarrow o)_{i \geq k}$, $x : (8 \cdot o') \vdash f^\omega : o$ and we coinductively define a family $(P_k)_{k \geq 2}$ of **S**-derivations by:

$$P_k = \frac{\frac{f : (k \cdot (2 \cdot o) \rightarrow o) \vdash f : (2 \cdot o) \rightarrow o \quad P_{k+1} \triangleright}{f : (i \cdot (2 \cdot o) \rightarrow o)_{i \geq k+1}, x : (8 \cdot o') \vdash f^\omega : o}}{f : (i \cdot (2 \cdot o) \rightarrow o)_{i \geq k}, x : (8 \cdot o') \vdash f^\omega : o}}$$

We observe that the P_k are indeed correct derivations of **S**. However, notice that x is typed (using track 8) whereas x does not appear in the typed term f^ω and the part of the context assigned to x cannot be traced back to any axiom rule typing x with o' (using axiom track 8). Intuitively, we have used an infinite branch to perform a weakening. This yields the notion of **quantitative derivation**, in which this does not happen:

Definition 10.2. A derivation P is **quantitative** when, for all $a \in A$ and $x \in \mathcal{V}$, $\mathcal{C}^P(a)(x) = \uplus_{a' \in \text{Ax}_a^P(x)} (\text{tr}^P(a') \cdot \mathbf{T}^P(a'))$.

Now, assume P is quantitative. Then $\text{Rt}(\mathcal{C}(a)(x)) = \{\text{tr}(a_0) \mid a_0 \in \text{Ax}_a(x)\}$ and for all $a \in A$, $x \in \mathcal{V}$ and $k \in \text{Rt}(\mathcal{C}(a)(x))$, we write $\text{pos}^P(a, x, k)$ or simply $\text{pos}(a, x, k)$ for the unique position $a' \in \text{Ax}_a(x)$ such that $\text{tr}(a') = k$.

Actually, $\text{pos}(a, x, k)$ can be defined by a downward induction on a as follows:

- If $a \in \text{Ax}$, then actually $a \in \text{Ax}(x)$ and $\text{tr}(a) = k$ and we set $\text{pos}(a, x, k) = a$.
- If $a : 1 \in A$, we set $\text{pos}(a, x, k) = \text{pos}(a : \ell, x, k)$, where ℓ is the necessarily unique (by typing constraint) positive integer s.t. $k \in \text{Rt}(\mathcal{C}(a \cdot \ell)(x))$.
- If $a : 0 \in A$, we set $\text{pos}(a, x, k) = \text{pos}(a : 0, x, k)$

10.3.3 One Step Subject Reduction and Expansion

System **S** enjoys both subject reduction and expansion, meaning that types are invariant under (anti)reduction. Indeed, if $t \rightarrow^* t'$, then $\triangleright C \vdash t : T$ iff $\triangleright C \vdash t' : T$:

Proposition 10.1 (One Step Subject Reduction). Assume $t \rightarrow_\beta t'$ and $P \triangleright C \vdash t : T$. Then there exists a derivation P' s.t. $P' \triangleright C \vdash t' : T$.

Proposition 10.2 (One Step Subject Expansion). Assume $t \rightarrow_\beta t'$ and $P' \triangleright C \vdash t' : T$. Then there exists a derivation P s.t. $P \triangleright C \vdash t : T$.

³An alternative definition of this notation is the following: as in Sec. 2.1.2, a *bound occurrence* of a variable $x \in \mathcal{V}$ in a trivial derivation P is a position $a \in \text{supp}(P)$ such that $t(a) = x$ and there exists $a_* < a$ with $t(a_*) = \lambda x$. If a is a bound occurrence of variable x in P , the *binding position* of this occurrence is the maximal $a_* \leq a$ such that $t(a_*) = \lambda x$. We then say that a is bound by position a_* in P and we write $a_* = \lambda^P(a)$. Then we set $\text{Ax}_a^P(x) = \{a_0 \in \text{supp}(P) \mid \lambda^P(a_0) = a\}$.

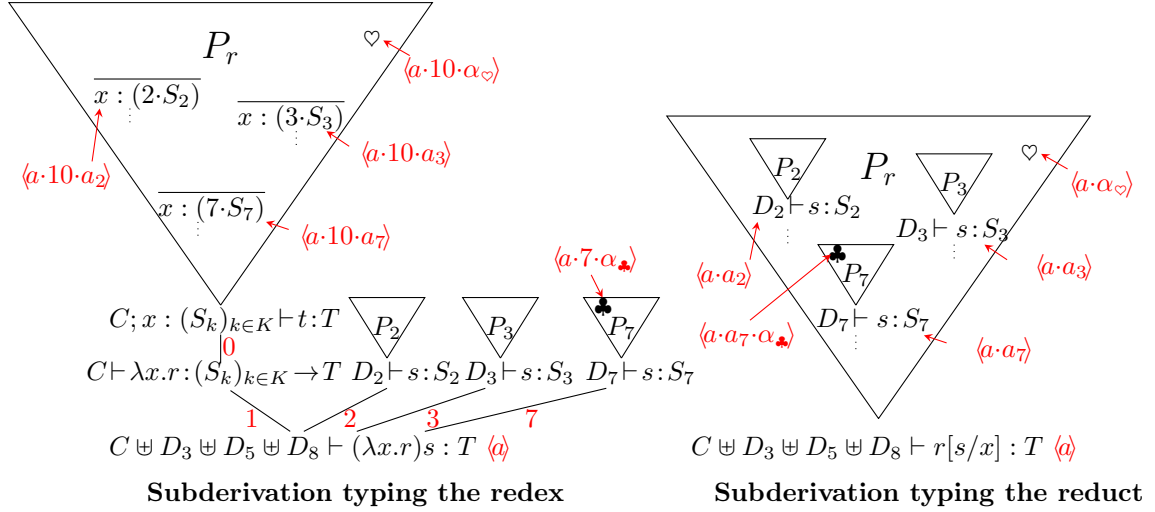


Figure 10.5: Subject Reduction and Residuals

Those propositions may be proven using coinduction. We give an alternative proof using *residuals* (that are quickly presented below p. 219) and defining directly derivation P' in Sec. 10.3.5. We explain now why subject reduction may be seen as a *deterministic* process in system \mathcal{S} . Subject expansion is *not* deterministic, but it may be processed *uniformly*.

All this is illustrated by Fig. 10.5 (compare with Fig. 3.2, p. 91 representing subject reduction in system \mathcal{R}_0): we assume that $t|_b = (\lambda x.r)s$ and $t \xrightarrow{b} t'$, P is a quantitative derivation concluding with $C \vdash t:T$. We also assume that $a \in P$ is such that $\bar{a} = b$ (thus, a is the position of a judgment typing the redex to be fired) and that there are exactly 3 **ax**-rules typing x above a , using axiom tracks 2, 3 and 7. Notice that **ax**-rule typing x on track 7 must be above $a \cdot 10$, so that its position is of the form $a \cdot 10 \cdot a_7$. Likewise, for the two other axioms.

Now, let us have a look at how reduction is performed inside P . We omit **ax**-rules right-hand sides. As in Sec. 2.1.5, we also indicate the position of a judgment between angle brackets *e.g.*, $\langle a \cdot 10 \cdot a_3 \rangle$ means that judgment $x : (3 \cdot S_3) \vdash x : S_3$ is at position $a \cdot 10 \cdot a_3$.

Notice how this transformation is *deterministic*: for instance, assume $7 \in K$. There must be an axiom rule typing x using axiom track 7 *e.g.* $x : (7 \cdot S_7) \vdash x : S_7$ at position $a \cdot 10 \cdot a_7$ and also a subderivation at argument track 7, namely, P_7 concluded by $s : S_7$ at position $a \cdot 7$. Then, when we fire the redex at position b , the subderivation P_7 *must* replace the axiom rule on track 7, even if there may be several $k \neq 7$ such that $S_k = S_7$, in contrast to system \mathcal{R} (see Sec. 10.3.4).

Thus, Proposition 10.1 implicitly yields only one derivation P' typing t' , so that we may also write $P \xrightarrow{b} P'$.

Now, we observe that subject expansion cannot be deterministic in the same sense. When we pass from a derivation typing $r[s/x]$ to a derivation typing $(\lambda x.r)s$, we create new axiom rules that will type x . Those axiom rules must be assigned axiom tracks. But if for instance, 3 axioms rules are created above position a , there is no more reason to choose tracks 2, 3 and 7 than the tracks 8, 4, 38: the axiom tracks may be chosen arbitrarily, as long as they do not raise track conflicts.

However, as it will turn out in Sec. 10.5.4, we will need to expand simultaneously *families* of derivations — and this, infinitely many times. For that, we should find a way to perform subject expansion uniformly. Let then $[\cdot]$ be any *injection* from \mathbb{N}^* to $\mathbb{N} \setminus \{0, 1\}$. We write $\text{Exp}_b(P', [\cdot], t)$ for the *unique* expansion of P' such that $P \xrightarrow{b} P'$ and, for all $a_0 \in \mathbb{N}^*$, if there is an **ax**-rule typing x created at position a_0 , then the axiom track that has been assigned is $[a_0]$. Since $[\cdot]$ is injective, no track conflict may occur. The term t must be indicated in the expression, because, for one t' and one b , there may be several t s.t. $t \xrightarrow{b} t'$.

Remark 10.3. If P is quantitative, then $\text{Red}_b(P)$ is quantitative. If P' is quantitative, then any $\text{Exp}_b(P', [\cdot], t)$ is quantitative.

With the above notations, we also write $P \xrightarrow{b} P'$. The subject-expansion property hold for quantitative derivations. Namely, we build a derivation $P \triangleright C \vdash t : T$ from a derivation $P' \triangleright C \vdash t' : T$, so that $P \xrightarrow{b} P'$ by using a converse method. There are several possibilities to build such a P , because we have to choose an axiom track k for each occurrence of x inside P (in that case, x is *quantitatively* typed). For instance, we can fix an injection $[\cdot]$ from \mathbb{N}^* to $\mathbb{N} - \{0, 1\}$ and to choose the track $[\alpha]$ for any axiom rule created at position α .

Proposition 10.3 (Subject Expansion). If $t \xrightarrow{b} t'$ and $C \vdash t' : T$ is derivable, then so is $C \vdash t : T$.

A Glimpse at Residuals

Deterministic subject reduction allows us to define the residuals of positions and right bipositions after reduction, extending the notion of residuals for position in λ -terms (presented in Sec. 2.1.5). In particular, Fig. 10.5 can be compared with Fig. 2.8, in which the symbols \heartsuit and \clubsuit play a similar role, to be explained below in the case of system **S**.

In Fig. 10.5, \heartsuit represents a judgment nested in P_7 . Thus, its position must be of the form $a \cdot 10 \cdot \alpha_{\heartsuit}$. After reduction, the **app**-rule and **abs**-rule at positions a and $a \cdot 0$ have been destroyed and the position of this judgment \heartsuit will be $a \cdot \alpha_{\heartsuit}$. We set then $\text{Res}_b(a \cdot 10 \cdot \alpha_{\heartsuit}) = a \cdot \alpha_{\heartsuit}$.

Likewise, \clubsuit represents a judgment nested in the argument derivation P_7 on track 7 w.r.t. a . Thus, its position must be of the form $a \cdot 7 \cdot \alpha_{\clubsuit}$, where $a \cdot 7$ is the root of P_7 . After reduction, P_7 will replace the **ax**-rule typing x on track 7, so its root will be at $a \cdot a_7$ (by definition of a_7). Thus, after reduction, the position of judgment \clubsuit will be $a \cdot a_7 \cdot \alpha_{\clubsuit}$. We set then $\text{Res}_b(a \cdot 7 \cdot \alpha_{\clubsuit}) = a \cdot a_7 \cdot \alpha_{\clubsuit}$.

We can thus define the **residuals** of most positions, but not all *e.g.*, $a \cdot 1$, that corresponds to the abstraction of the redex, is destroyed during reduction and does not have a residual. For *right* bipositions, when $(a, c) \in \text{bisupp}(P)$ and $a' = \text{Res}_b(a)$ is defined, we set $\text{Res}_b(a, c) = (a', c)$.

Note that defining residuals in system \mathcal{R} would be impossible: system \mathcal{R} lacks pointers and is not deterministic (Sec. 4.1.2). Residuals (as well as quasi-residuals) are defined in Sec. 10.3.5.

10.3.4 Safe Truncations of Typing Derivations

We can explain now why system \mathcal{R} and multiset intersection are unfit to express the notion of **approximability**, informally introduced at the end of Sec. 10.1.3.

Let us consider a redex $t = (\lambda x.r)s$ and its reduct $t' = r[s/x]$. If a \mathcal{R} -derivation Π types t then r has been given some type τ in some context Γ ; $x : [\sigma_i]_{i \in I}$ through a subderivation Π_r . Also, for each $i \in I$, s has been given the type σ_i through some subderivation Π_i . We can obtain a derivation Π' typing the term t' by replacing the axiom rule yielding $x : [\sigma_i] \vdash x : \sigma_i$ by the derivation Π_i . The construction of such a Π' from Π relies generally on a result referred as the “substitution lemma” (e.g., Lemma 7.4, p. 154 in the case of λ_μ). We then call Π' a **derivation reduct**.

If a type σ occurs several times in $[\sigma_i]_{i \in I}$ – say n times – there must be n axiom leaves in Π typing x with type σ , but also n argument derivations Π_i concluding with $s : \sigma$. When an axiom rule typing x and an argument derivation Π_i are concluded with the same type σ , we shall informally say that we can **associate** them (see also Remark 4.1). It means that this axiom rule can be substituted with that argument derivation Π_i when we reduce t to produce a reduct derivation Π' typing t' . There is not only one way to associate the Π_i to the axiom leaves typing x (there can be as many as $n!$) and possibly many different derivation reducts. This is to be related to the possibility of *reduction choices* in system \mathcal{R}_0 , which is addressed in Sec. 4.1.2, and the non-rigidity of this system.

This makes a sharp difference with (the rigid) system \mathbf{S} : assume that $S_2 = S_3 = S_7 =: S$ so that argument derivation P_2, P_3, P_7 type s with the same type S . Then each P_k ($k \in K$) will replace axiom rule at position $a \cdot 10 \cdot a_k$ (see Fig. 10.5) without other choice. In system \mathbf{S} , there is a unique derivation reduct.

Observe the following independent situations in system \mathcal{R} :

- Assume Π_1 and Π_2 (typing s), both concluded with the same type $\sigma = \sigma_1 = \sigma_2$. Thus, we also have two axiom leaves #1 and #2 concluded by $x : [\sigma] \vdash x : \sigma$, where #1 can be associated with Π_1 or Π_2 . When we truncate Π into a finite ${}^f\Pi$, the subderivations Π_1 and Π_2 are also cut into two derivations ${}^f\Pi_1$ and ${}^f\Pi_2$. In each ${}^f\Pi_i$, σ can be cut into a type ${}^f\sigma_i$. When Π_1 and Π_2 are different, it is possible that ${}^f\sigma_2 \neq {}^f\sigma_1$ for every finite truncation of Π . Thus, it is possible that, for every truncation ${}^f\Pi$, the axiom leaf #1 *cannot* be associated to ${}^f\Pi_2$: indeed, an association that is possible in Π could be impossible for any of its truncations.
- Assume this time $\sigma_1 \neq \sigma_2$. When we truncate Π into a finite ${}^f\Pi$, both σ_1 and σ_2 can be truncated into the same finite type ${}^f\sigma$. In that case, we can associate ${}^f\Pi_1$ with axiom #2 and ${}^f\Pi_2$ with axiom #1 inside ${}^f\Pi$ in ${}^f\Pi$ (which is impossible in Π), thus producing a reduct derivation ${}^f\Pi'$ typing t' , which has no meaning w.r.t. Π (${}^f\Pi'$ would not be a truncation of any derivation reduct of Π).

That is why we need the *deterministic* association between the argument derivations and the axiom rules typing each in system \mathbf{S} (thanks to tracks), so that the associations between them are preserved even when we truncate derivations. System \mathcal{R} does not makes it possible to formulate a well-fit notion of approximability for derivations that would be stable under (anti)reduction and hereditary for subterms. Actually, there exists a \mathcal{R} -derivation Π and two \mathbf{S} -derivations P_1 and P_2 that both collapse on Π such that P_1 is approximable (Def. 10.4 to come) and P_2 is not. This counter-example can be found in Appendix A.6.2.

10.3.5 A Proof of the Subject Reduction Property

Fig. 10.5 rather represents a quantitative case but the following construction does not assume P to be quantitative. We formalize now the notion of residuals and use them to give a proof of the subject reduction property for system \mathbf{S} .

We assume again that $t|_b = (\lambda x.r)s$ and $t \xrightarrow{b} t'$ and we consider a derivation P s.t. $P \triangleright C \vdash t : T$. The letter a will stand for a representative of b and the letter α for other positions. We set $\mathbf{Ax}_\lambda(a) = \mathbf{Ax}_{a \cdot 10}(x)$ and $\mathbf{Tr}_\lambda(a) = \{\mathbf{tr}(\alpha_0) \mid \alpha_0 \in \mathbf{Ax}_\lambda(a)\}$. Thus, $\mathbf{Ax}_\lambda(a)$ is the set of positions of the redex variable (to be substituted) above a and $\mathbf{Tr}_\lambda(a)$ is the set of the axiom tracks that have been used for them. For instance, in the Figure, $\mathbf{Ax}_\lambda(a) = \{a \cdot 10 \cdot a_2, a \cdot 10 \cdot a_3, a \cdot 10 \cdot a_7\}$ and $\mathbf{Tr}_\lambda(a) = \{2, 3, 7\}$.

Since P is quantitative, $\mathbf{C}(a \cdot 10)(x)$ must be of the form $(S_k)_{k \in K}$ where $K = \mathbf{Tr}_\lambda(a)$.

For $k \in \mathbf{Tr}_\lambda(a)$, we write a_k for the unique $a_k \in \mathbb{N}^*$ such that $\mathbf{pos}(a \cdot 10, x, k) = a \cdot 10 \cdot a_k$: thus, $a \cdot 10 \cdot a_k$ is the position of the axiom rule typing x above a using axiom track k

Assume $\alpha \in A$, $\bar{a} \neq a$, $a \cdot 1$, $a \cdot 10 \cdot a_k$ for no $a \in A$ such that $\bar{a} = b$ and $k \in \mathbf{RedTr}(a)$.

- If $\alpha \geq a \cdot k \cdot \alpha_0$ with $\bar{a} = b$ and $k \geq 2$ (paradigm \clubsuit), then $\mathbf{Res}_b(\alpha) = a \cdot a_k \cdot \alpha_0$
- If $\alpha = a \cdot 10 \cdot \alpha_0$ with $\bar{a} = b$ and $\alpha_0 \neq a_k$ (paradigm \heartsuit), then $\mathbf{Res}_b(\alpha) = a \cdot \alpha_0$
- If $\bar{a} \not\geq b$, $\mathbf{Res}_b(\alpha) = a$.

The **residual position** of α , written $\mathbf{Res}_b(\alpha)$, is defined as follows:

- *Out of the redex:* If $\alpha \not\geq a$, then α is not in the redex. We set $\mathbf{Res}_b(\alpha) = \alpha$.
- *Inside r :* Position $a \cdot 10 \cdot \alpha \in \mathbb{B}$ (paradigm \heartsuit) has a residual (except when $\alpha = a_k$ for some k) and should become $a \cdot \alpha$ after reduction: we set $\mathbf{Res}_b(a \cdot 10 \cdot \alpha) = a \cdot \alpha$ for $\alpha \neq a_k$.
- *Inside some argument derivations:* Assume $k \in \mathbf{Tr}_\lambda(a)$. Argument derivation at $a \cdot k$ will replace **ax**-rule typing at position $a \cdot 10 \cdot a_k$ (which is destroyed). So its position after reduction will be $a \cdot a_k$. More generally, the $a \cdot k \cdot \alpha \in \mathbb{B}$ (paradigm \clubsuit) will be found at $a \cdot a_k \cdot \alpha$ after reduction. We set then $\mathbf{Res}_b(a \cdot k \cdot \alpha) = a \cdot a_k \cdot \alpha$ when $k \in \mathbf{Tr}_\lambda(a)$.

In the next chapter (Sec. 12.4.1), an extension of the notion of quasi-residuation (Sec. 2.1.5) to system \mathbf{S} will be needed.

Remark 10.4.

- In the case of the pure λ -calculus (Sec. 2.1.5), residuation gives a *binary relation* between the positions of a λ -term t and those of some reduct t' of t .
- For system \mathbf{S} , residuation gives a (partial) *function* from the position of a \mathbf{S} -derivation P and those of some reduct derivation P' of t .
- Due to the possible duplication of the argument in the pure λ -calculus, residuation cannot be functional (a position can have several residuals w.r.t. the same reduction step) whereas system \mathbf{S} is a linearization of λ -calculs and forbids duplication, so that a position $a \in \mathbf{supp}(P)$ can have at most *one* residual w.r.t. a reduction sequence. This is the reason why \mathbf{Res}_b is defined as a function in this section.

By case analysis, we notice that residuation for positions preserves labelling, as it did for pure λ -calculus (first point of Lemma 2.1):

Lemma 10.3. Assume that P types t , $t \xrightarrow{b}$ and $P \xrightarrow{b} P'$. $t'(\text{Res}_b(\alpha)) = t(\alpha)$ for all $\alpha \in \text{dom}(\text{Res}_b)$.

We set $A' = \text{codom}(\text{Res}_b)$ (**residual support**). Now, whenever $\alpha' := \text{Res}_b(\alpha)$ is defined, the **residual biposition** of $\mathbf{p} := (\alpha, c) \in \text{bisupp}(P)$ is $\text{Res}_b(\mathbf{p}) = (\alpha', c)$.

We notice that Res_b is a *partial injective* function both for positions and right bipositions (compare with the second point of Lemma 2.1). In particular, Res_b is a bijection from $\text{dom}(\text{Res}_b)$ to A' and we write Res_b^{-1} for its inverse.

For any $\alpha' \in A'$, let $\mathcal{C}'(\alpha')$ be the context defined by $\mathcal{C}'(\alpha') = (\mathcal{C}(\alpha) - x) \uplus (\uplus_{k \in K(\alpha)} \mathcal{C}(\alpha \cdot k))$, where $\alpha = \text{Res}_b^{-1}(\alpha')$ and $K(a) = \text{Rt}(\mathcal{C}(a)(x))$.

Notice that $\mathcal{C}'(\alpha) = \mathcal{C}(\alpha)$ for any $\alpha \in A$ s.t. $\bar{\alpha} \not\neq b$, e.g., $\mathcal{C}'(\varepsilon) = \mathcal{C}(\varepsilon)$.

Now, let P' be the labelled tree such that $\text{supp}(P') = A'$ and $P'(\alpha')$ is $\mathcal{C}'(\alpha') \vdash t'|_{\alpha'} : T'(\alpha')$ with $\alpha' = \text{Res}_b(\alpha)$. We claim that P' is a correct derivation concluded by $C \vdash t' : T$: indeed, $\bar{A}' \subset \text{supp}(t')$ stems from $\bar{A} \subset \text{supp}(t)$. Then, for any $\alpha' \in A'$ and $\alpha = \text{Res}_b^{-1}(\alpha')$, $t'(\bar{\alpha}') = t(\bar{\alpha})$ and the rule at position α' is correct in P' because the rule at position α in P is correct (for the abstraction case, we notice that $t'(\bar{\alpha}') = \lambda y$ implies $\mathcal{C}'(\alpha')(y) = \mathcal{C}(\alpha)(y)$).

Thus, P' is a derivation concluding with $C \vdash t' : T$. This concludes the proof of the subject reduction property for \mathbf{S} .

10.4 Approximable Derivations and Unforgetfulness

In Sec. 10.1.3, we saw that Klop's Problem demands to consider “joins” of derivations and an approximability criterion. In this section, we exhibit a lattices structures underlying set of \mathbf{S} -derivations (Sec. 10.4.1). This allows us to formally define directed sets of derivations, as well as their join. In Sec. 10.4.2, we formally define approximability using the notion of bipositions and that of bisupport. Then, in Sec. 10.4.3, we adapt the unforgetfulness condition (Sec. 10.1.1) to \mathbf{S} -derivation. We also prove a few dynamics properties: approximability is stable under one step reduction/expansion. Moreover, approximable unforgetful typing ensures hereditary head normalization (Proposition 10.4).

10.4.1 The Lattice of Approximation

From last section, we know that \mathcal{R} is unfit to recover soundness through approximability. Let us now work with system \mathbf{S} only and formalize the intuitive notions seen before.

As seen in Sec. 10.1.3, we must be able to truncate derivations (notion of approximation) and define the join of some families of derivations. This can be properly defined in system \mathbf{S} .

Definition 10.3. • Let P and P_* be two derivations typing a same term t . We say P_* is an **approximation** of P , and we write $P_* \leq_{\infty} P$, if $\text{bisupp}(P_*) \subseteq \text{bisupp}(P)$ and for all $\mathbf{p} \in \text{bisupp}(P_*)$, $P_*(\mathbf{p}) = P(\mathbf{p})$.

- We write $\text{Approx}_{\infty}(P)$ for the set of approximations of a derivation P and $\text{Approx}(P)$ for the set of *finite* approximations of P .

Thus, $P_* \leq_{\infty} P$ if P_* is a *correct* restriction of P on a subset of $\mathbf{bisupp}(P)$ (*i.e.* a restriction that respects the typing rules of \mathbf{S}). We usually write fP for a *finite* approximation of P (*i.e.* $\mathbf{bisupp}({}^fP)$ is finite) and in that case only, we write ${}^fP \leq P$ instead of ${}^fP \leq_{\infty} P$. Actually, \leq_{∞} and \leq are associated to lattice structures induced by the set-theoretic inclusion, union and intersection on bisupports :

Theorem 10.3. The set of derivations typing a same term t endowed with \leq_{∞} is a directed complete semi-lattice.

- If D is a directed set of derivations typing t :
 - The **join** $\vee D$ of D is the function P defined by $\mathbf{dom}(P) = \cup_{P_* \in D} \mathbf{bisupp}(P_*)$ and $P(\mathbf{p}) = P_*(\mathbf{p})$ (for any $P_* \in D$ s.t. $\mathbf{p} \in \mathbf{bisupp}(P_*)$), which also is a derivation.
 - The **meet** $\wedge D$ of D is the function P defined by $\mathbf{dom}(P) = \cap_{P_* \in D} \mathbf{bisupp}(P_*)$ and $P(\mathbf{p}) = P_*(\mathbf{p})$ (for all $P_* \in D$), which also is a derivation.
- If P is a derivation typing t , $\mathbf{Approx}_{\infty}(P)$ is a complete lattice and $\mathbf{Approx}(P)$ is a lattice.

Proof. See Appendix A.3. □

Approximation is compatible with reduction:

Lemma 10.4.

- Reduction is monotonic: if $P_* \leq_{\infty} P$, $P_* \xrightarrow{b} P'_*$ and $P \xrightarrow{b} P'$, then $P'_* \leq_{\infty} P'$.
- Moreover, if $P \xrightarrow{b} P'$, then, for any $P'_* \leq_{\infty} P'$, there is a unique $P_* \leq_{\infty} P$ s.t. $P_* \xrightarrow{b} P'_*$.
- $[\cdot]$ -expansion is monotonic: if P' types t' , $t \xrightarrow{b} t'$ and $P'_* \leq_{\infty} P'$, then $P_* \leq_{\infty} P$ with $P_* = \mathbf{Exp}_b(P'_*, [\cdot], t)$, $P = \mathbf{Exp}_b(P', [\cdot], t)$.

10.4.2 Approximability

We define here our validity condition *i.e.* approximability, suggested in Sec. 10.1.3 and illustrated by Fig 10.3. Morally, a derivation P is approximable if all its bipoositions are meaningful *i.e.* can be part of a *finite* derivation fP approximating P .

Definition 10.4. A derivation P is **approximable** if, for all finite ${}^0B \subseteq \mathbf{bisupp}(P)$, there exists ${}^fP \leq P$ s.t. ${}^0B \subseteq \mathbf{bisupp}({}^fP)$.

Remark 10.5 (Reformulating the Approximability Condition).

- Equivalently, a derivation P is approximable when it is the join of its finite approximations.
- A derivation P is approximable iff P is quantitative (Sec. 10.2) and, for all finite set of *right* bipoositions ${}^0B \subseteq \mathbf{bisupp}(P)$, there exists ${}^fP \leq P$ s.t. ${}^0B \subseteq \mathbf{bisupp}({}^fP)$. Indeed, quantitativity is necessary for approximability (Lemma 10.5) below. Moreover, when P is quantitative, every left bipoosition can be tracked back to an axiom rule. And every left bipoosition in the conclusion of an **ax**-rule is “equinecessary” to a right bipoosition. See Appendix A.2 for more details.

- Let P a *quantitative* \mathbf{S} -derivation. It may be felt that every biposition in $\mathbf{bissupp}(P)$ is somehow “related” (*via* the typing rules) to a biposition located in the root judgment of P , to be called “depth 0 biposition”. In that case, approximability could be restated as follows: “ P is quantitative and, for all finite set ${}^0B \subseteq \mathbf{bissupp}(P)$ of depth 0 bipositions, there exists ${}^fP \leq P$ such that ${}^0B \subseteq \mathbf{bissupp}({}^fP)$ ”. However, this condition is not equivalent to approximability. A counterexample can be found in Appendix A.2.3.

Lemma 10.5.

1. If P is not quantitative, then P is not approximable.
2. If P is quantitative and $P \xrightarrow{b} P'$, then P is approximable iff P' is approximable.

Proof sketch.

1. If P is not quantitative, then P contains some left biposition $\mathbf{p} := (a, x, k \cdot c)$ that does not come from an \mathbf{ax} -rule: this implies that there are infinitely many $a' \geq a$ s.t. $(a', x, k \cdot c) \in \mathbf{bissupp}(P)$. An approximation $P_* \leq P$ that contains \mathbf{p} has to contain all those $(a', x, k \cdot c)$ and thus, cannot be finite. So P cannot be approximable.
2. Assume P approximable. Let us show that P' is also approximable. Then, let ${}^0B' \subseteq \mathbf{bissupp}(P')$ be a finite set of bipositions. We can find a *finite* set of ${}^0B \subseteq \mathbf{bissupp}(P)$ s.t. ${}^0B' \subseteq \mathbf{Res}_b({}^0B)$.

Since P is approximable, there is ${}^fP \leq P$ s.t. ${}^0B \subseteq {}^fP$. We set ${}^fP' = \mathbf{Res}_b({}^fP)$. By Lemma 10.4, ${}^fP' \leq P'$. This is enough to conclude.

The converse implication is proven likewise. However, \mathbf{Res}_b is not defined for every biposition (*e.g.*, left ones) and our argument is faulty. It is not hard to avoid this problem (it is done in Appendix A.2), using a suitable notion of *interdependencies* between bipositions.

□

10.4.3 Unforgetfulness

We remember from Sec. 10.1.1 that weak normalization for the finite calculus is characterized in system \mathcal{R}_0 by means of *unforgetful* derivations. In order to characterize weak normalizability in Λ^{001} (Definition 9.4), we want to adapt Theorem 10.2 to system \mathbf{S} . This will yield Theorem 10.4, the main result of this paper, stated as follows:

Theorem. A term t is weakly-normalizing in Λ^{001} if and only if t is typable by means of an approximable unforgetful derivation.

To state and prove this theorem, we must first adapt the definition of unforgetfulness. We recall that the targets of arrows are regarded as positive and their sources as negative. The following definitions are straightforward adaptations from system \mathcal{R}_0 (Definitions 5.1 and 5.2).

Definition 10.5. Inductively:

- For all type T , $()$ occurs negatively in $() \rightarrow T$.

- $()$ occurs positively (resp. negatively) in $(S_k)_{k \in K}$ if there exists $k \in K$ s.t. $()$ occurs positively (resp. negatively) in S_k .
- $()$ occurs positively (resp. negatively) in $(S_k)_{k \in K} \rightarrow T$ if $()$ occurs positively (resp. negatively) in T or negatively (resp. positively) in $(S_k)_{k \in K}$.

Definition 10.6.

- A judgment $C \vdash t : T$ is **unforgetful** when, $()$ does not occur positively in T and for all $x \in \mathcal{V}$, $()$ does not occur negatively in $C(x)$.
- A derivation is **unforgetful** when it concludes with an unforgetful judgment.

As in Sec. 10.1.1, we can check by induction on t that if t is a NF and $P \triangleright C \vdash t : T$ is unforgetful, then every subterm of t is typed in P (the induction is performed on the position of the considered subterm).

Lemma 10.6.

- If $P \triangleright C \vdash t : T$ is an unforgetful derivation typing a HNF $t = \lambda x_1 \dots x_p. x t_1 \dots t_q$, then, there are unforgetful subderivations P_1, \dots, P_q of P typing t_1, t_2, \dots, t_q .
- Moreover, if P is approximable, so are they.

Proof.

- The arguments of Lemma 5.2 straightforwardly adapt to system **S**: we first observe that the depth p subderivation P_0 of P typing $t_0 := x t_1 \dots t_q$ is also unforgetful, and then, that there arguments t_1, \dots, t_q are *not* left untyped (since $()$ cannot occur negatively in the type of the head variable x). It is not difficult to check that the subderivations typing the t_k are also unforgetful, by using the typing rules and Definition 10.6.
- Since P' are approximable, the subderivations P_k are themselves approximable by Definition 10.4.

□

Lemma 10.7. If $P \triangleright C \vdash t : T$ is a *finite* derivation, then t is head normalizable. Actually, the head reduction strategy terminates at t .

Proof. The proof of Proposition 3.9 straightforwardly adapts:

- By the typing rules, the head redex – if it exists *i.e.* if t is not already in HNF – must be typed.
- When we reduce a typed redex, the number of rules of the derivation must strictly decrease (at least one **app**-rule and one **abs**-rule disappear). See Fig. 10.5.
- Since there is no infinite decreasing sequence of integers, the head-reduction strategy must halt at some point, meaning that a HNF is reached.

□

Proposition 10.4. If a term t is typable by an unforgetful approximable derivation, then it is hereditary head normalizing.

Proof. We assume that $P \triangleright C \vdash t : T$ is an unforgetful and approximable derivation

- By Lemma 10.7, t head-normalizes to a HNF $t' := \lambda x_1 \dots x_p. x t_1 \dots t_q$.
- By subject reduction (Proposition 10.1), there is a derivation $P' \triangleright C \vdash t' : T$. Derivation P' is obviously unforgetful. Moreover, by Lemma 10.5, P' is approximable.
- By Lemma 10.6, the t_1, \dots, t_q are themselves approximably and unforgetfully typable by subderivation P_1, \dots, P_q of P' .

This proves that t is hereditary head normalizing, by Definition 9.5. \square

10.4.4 The infinitary Subject Reduction Property

In this section, we prove subject reduction for strongly converging reduction sequences. The initial derivation may be approximable or not.

Thus, we have to define a derivation P' typing t' from a derivation typing a term t that strongly converges towards t' . The main intuition is again the following (*cf.* Sec. 9.3.1): when a reduction is performed at applicative depth n , the contexts and types are not affected below depth n . Thus, a s.c.r.s. stabilizes contexts and types at any fixed applicative depth. This allows us to define a derivation typing the limit t' . Figure 9.3, p. 196 can be adapted for s.c.r.s. of Λ^{001} (instead of Λ^∞) and \mathbf{S} -derivations (instead of terms) by replacing “depth” by “applicative depth”.

The following **Subject Substitution Lemma** (compare with Lemma 10.2) is very useful while working with strong convergence. It states that we can freely change the untyped parts of a term in a typing derivation, as suggested in Sec. 10.1.2 and 10.1.3.

Lemma 10.8. Assume $P \triangleright C \vdash t : T$ and for all $a \in \text{supp}(P)$, $t(a) = t'(a)$ (no approximability condition).

Let $P[t'/t]$ be the *labelled tree* obtained from P by replacing t by t' (more precisely, $P[t'/t]$ is the labelled tree P' s.t. $\text{supp}(P') = \text{supp}(P)$ and, for all $a \in \text{supp}(P)$, $P(a) = \mathbf{C}(a) \vdash t'|_a : \mathbf{T}(a)$).

Then $P[t'/t]$ is a correct derivation.

Now, let us formally prove the infinitary subject reduction property. For that, we assume:

- $t \rightarrow^\infty t'$ is a s.c.r.s.. Say that this sequence is $t = t_0 \xrightarrow{b_0} t_1 \xrightarrow{b_1} \dots \xrightarrow{b_{n-1}} t_n \xrightarrow{b_n} t_{n+1} \xrightarrow{b_{n+1}} \dots$ with $b_n \in \{0, 1, 2\}^*$ and $\text{ad}(b_n) \rightarrow \infty$.
- There is a derivation $P \triangleright C \vdash t : T$ and $A = \text{supp}(P)$.

By performing step by step the s.c.r.s. b_0, b_1, \dots , we get a sequence of derivations $P_n \triangleright C \vdash t_n : T$ of support A_n (and we write $\mathbf{C}_n, \mathbf{T}_n$ for \mathbf{C}^{P_n} and \mathbf{T}^{P_n}). When performing $t_n \xrightarrow{b_n} t_{n+1}$, notice that $\mathbf{C}_n(a)$ and $\mathbf{T}_n(a)$ are not modified for any a such that $b_n \not\leq \bar{a}$ i.e. $\mathbf{C}_n(a) = \mathbf{C}_{n+1}(a)$ and $\mathbf{T}_n(a) = \mathbf{T}_{n+1}(a)$.

Let $a \in \mathbb{N}^*$ and $N \in \mathbb{N}$ be such that, for all $n \geq N$, $\text{ad}(b_n) > \text{ad}(a)$. There are two cases:

- $a \in A_n$ for all $n \geq N$. Moreover, $\mathbf{C}_n(a) = \mathbf{C}_N(a)$, $\mathbf{T}_n(a) = \mathbf{T}_N(a)$ for all $n \geq N$, and $t'(a) = t_n(a) = t_N(a)$.

- $a \notin A_n$ for all $n \geq N$.

We set $A' = \{a \in \mathbb{N}^* \mid \exists N, \forall n \geq N, a \in A_n\}$. We define a labelled tree P' whose support is A' by $P'(a) = \mathbf{C}_n(a) \vdash t'|_a : \mathbf{T}_n(a)$ and we set $\mathbf{C}'(a) = \mathbf{C}_n(a), \mathbf{T}'(a) = \mathbf{T}_n(a)$ for any $n \geq N(\text{ad}(a))$ (where $N(\ell)$ is the smallest rank N such that $\forall n \geq N, \text{ad}(a_n) > \ell$).

Lemma 10.9. The labelled tree P' is a derivation.

Proof. Let $a \in A'$ and $n \geq N(|a| + 1)$. Thus, $t'(a) = t_n(a)$ and the types and contexts involved at node a and its premises are the same in P' and P_n . So the node a of P' is correct, because it is correct for P_n . \square

Lemma 10.10. If P is approximable, so is P' .

Proof. Let ${}^0B \subseteq \text{bisupp}(P')$. We set $\ell = \max\{\text{ad}(p) \mid p \in {}^0B\}$ ($\text{ad}(p)$ is the applicative depth of the underlying $a \in \text{supp}(P)$).

By strong convergence, there is N s.t., $\forall n \geq N, \text{ad}(b_n) \geq \ell + 1$, and thus, $t_N(a) = t_n(a) = t'(a)$, $\mathbf{C}_N(a) = \mathbf{C}_n(a) = \mathbf{C}'(a)$ and $\mathbf{T}_N(a) = \mathbf{T}_n(a) = \mathbf{T}'(a)$ for all a s.t. $\text{ad}(a) \leq \ell$ and $n \geq N$. In particular, ${}^0B \subseteq \text{bisupp}(P_N)$.

Since P is approximable, by Lemma 10.5, P_N is approximable. So, there is ${}^fP_N \leq P$ s.t. ${}^0B \subseteq \text{bisupp}({}^fP_N)$.

We have ${}^0B \subseteq \text{bisupp}({}^fP_N) = \text{bisupp}({}^fP')$. Thus, P' is approximable. \square

Proposition 10.5 (Infinitary Subject Reduction). Assume $t \rightarrow^\infty t'$ and $P \triangleright C \vdash t : T$. Then there exists a derivation P' s.t. $P' \triangleright C \vdash t' : T$.

Moreover, if P is approximable, P' may be chosen to be approximable.

Proof. Consequence of Lemmas 10.9 and 10.10 \square

10.5 Typing Normal Forms and Subject Expansion

As hinted at in Sec. 10.1.1, a proof of *Weakly Normalizable* \Rightarrow *Typable* proceed by giving first an (unforgetful) typing of NF, and then, using a subject expansion property (exactly as in the proof of Proposition 5.4).

In this section, we want to prove that NF are typable in \mathbf{S} and that all the quantitative derivations typing a NF are approximable. We will actually describe all the quantitative derivations typing a NF (and prove them to be approximable). Then, we will prove an infinitary subject expansion property, what is enough to show the above implication.

Normal forms are coinductive assemblages of HNF (Sec. 9.3.2). It is then important to understand how a HNF $t = \lambda x_1 \dots \lambda x_p. x t_1 \dots t_q$ may be typed. Let us have a look at figure 10.6 and ignore for the moment the **rdeg** and positions (between angle brackets) annotations (note that this figure is the counterpart of Fig. 3.4 for system \mathbf{S} with some additional technical annotations): the head variable x has been assigned an arrow type $(S_k^1)_{kk \in K(1)} \rightarrow \dots \rightarrow (S_k^q)_{kk \in K(q)} \rightarrow T$ whereas the first argument t_1 is typed with types S_k^1 (k ranging over $K(1)$), \dots , the q -th argument t_q is typed with types S_k^q (k ranging over $K(q)$).

The subterms $x, x t_1, x t_1 t_2, \dots, x t_1 \dots t_{q-1}$ are typed with arrow types, as well as the subterms $\lambda x_p. x t_1 \dots t_q, \lambda x_{p-1} x_p. x t_1 \dots t_q, \dots, \lambda x_1 \dots x_p. x t_1 \dots t_q$ starting with abstractions. By contrast, notice that subterm $x t_1 \dots t_q$ has type T and that this type T may be *any* type (as in Lemma 3.5). So, we say that the type of subterm $x t_1 \dots t_q$ is **unconstrained**, whereas for instance:

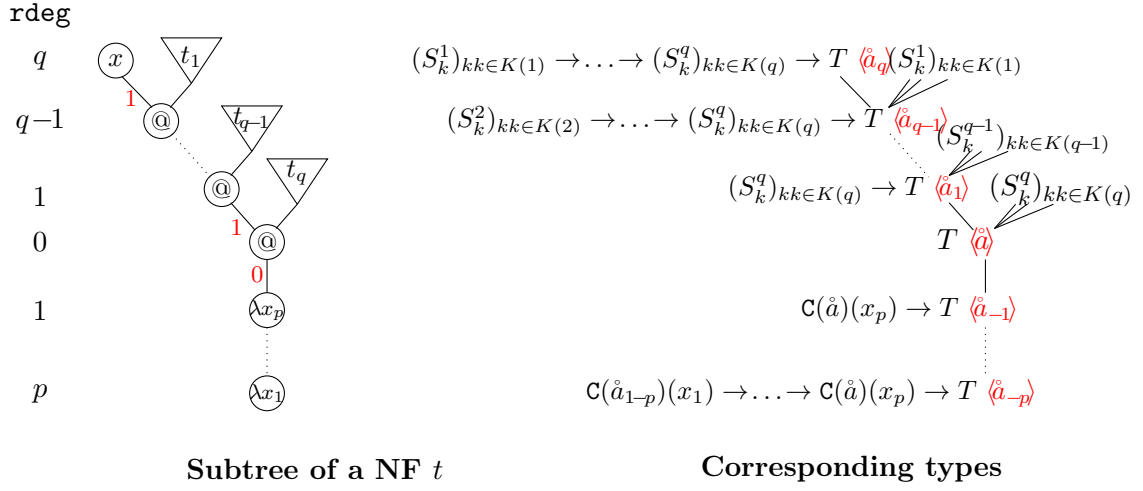


Figure 10.6: Typing Normal Forms in System S

- $x t_1 \dots t_{q-1}$ has type $(S_k^q)_{kk \in K(q)} \rightarrow T$: this type depends on the types given to the subterm t_q . We say informally that it *calls* for the types of t_q .
- $\lambda x_p. x t_1 \dots t_q$ has type $\mathbf{C}(0^p)(x_p) \rightarrow T$: this type depends on the types assigned to x_p deeper in the term. We say informally that it *calls* for the types of x_p .

10.5.1 Support Candidates

Before using the considerations above, we must devote our attention to the possible forms of the support $A := \mathbf{supp}(P)$ of a derivation P typing a NF t . This is done by the *preorder* \prec defined below, extending the prefix order.

We write $a \prec a'$ when there exists a_0 such that $a_0 \leq a$, $a_0 \leq a'$, $\mathbf{ad}(a) = \mathbf{ad}(a_0)$ and we observe that if P is a derivation typing t , then $\mathbf{supp}(P)$ is downward closed for \prec over $\mathbf{supp}(t)$, meaning by that, for all a_1 s.t. $\bar{a}_1 \in \mathbf{supp}(t)$ and $a_2 \in \mathbf{supp}(P)$, then $a_1 \prec a_2$ implies $a_1 \in \mathbf{supp}(P)$.

For instance, $021031 \prec 021037$ since $02103 \leq 021031$, 021037 and $\mathbf{ad}(02103) = \mathbf{ad}(021031) = 2$. If $021037 \in \mathbf{supp}(P)$, since 7 is an argument track, $t(02103) = @$ (*i.e.* 02103 points to an **app**-rule in P). Last, 02031, which is this **app**-rule left-hand side, should also be in $\mathbf{supp}(P)$, as well as every prefix of 021037. And we have thus $021031 \in \mathbf{supp}(P)$ as expected.

Conversely, a non-empty set A downward closed for \prec over $\mathbf{supp}(t)$ s.t. $\bar{A} \subseteq \mathbf{supp}(t)$. will be called a **support candidate** for a derivation typing t and we prove that, for all support candidates A associated with a NF t , there is actually a derivation P typing t s.t. $A = \mathbf{supp}(P)$. This will be Lemma 10.11.

Notation \hat{a} For now, let us consider P a derivation typing the NF t and $a \in A := \mathbf{supp}(P)$. We have $t|_a = \lambda x_1 \dots x_n. u$, where u is not an abstraction. The integer n is the **order** (as in [15] or Definition 2.8) of $t|_a$. We say then that a is an order n position.

- If $n \geq 1$, we say that a is a **non-zero position** and we set $\hat{a} = a \cdot 0^n$ (so that $t|_{\hat{a}} = u$) and $\mathbf{rdeg}(a) = n$.
- If $n = 0$, we distinguish two subcases, by first defining \hat{a} as the shortest prefix $a_0 \leq a$ s.t. $a = a_0 \cdot 1^\ell$ (for some ℓ) and setting $\mathbf{rdeg}(a) = \ell$:

- If $\ell = 0$, then we set $\mathring{a} = a$ and we say that a is an **unconstrained position**.
- If $\ell \geq 1$, we say that a is a **partial position**.

We then extend the function $a \mapsto \mathring{a}$ to the set $\{a \in \mathbb{N}^* \mid \bar{a} \in \text{supp}(t)\}$ (note that this function depends on t).

As it has been observed above, a is unconstrained when, intuitively, the type of the underlying subterm does not depend on deeper parts of the derivation. If $i \geq 0$, we write \mathring{a}_i for $\mathring{a} \cdot 1^i$ and \mathring{a}_i for the rank i prefix of \mathring{a} (e.g., $\mathring{a} = \mathring{a}_{-2} \cdot 0^2$ if $t|_{\mathring{a}}$ is of order ≥ 2). More generally, from the beginning of Sec.10.5, we observe that if $\text{rdeg}(a) = d$, then $\mathbb{T}(a)$ is an arrow type $F_1 \rightarrow \dots \rightarrow F_d \rightarrow \mathbb{T}(\mathring{a})$, where $\mathbb{T}(\mathring{a})$ is an unconstrained type. More precisely, using Fig. 10.6 and Notation 10.1:

- When a is a non-zero position i.e. $\mathring{a} = a \cdot 0^d$ and $t|_a$ is of the form $\lambda x_{q-d+1} \dots \lambda x_p. x t_1 \dots t_q$ with $t|_{\mathring{a}} = x t_1 \dots t_q$, then $F_1 = \mathbb{C}(a \cdot 0)(x_{q-d+1})$, $F_2 = \mathbb{C}(a \cdot 0^2)(x_{q-d+2})$, \dots , $F_d = \mathbb{C}(a \cdot 0^d)(x_p) = \mathbb{C}(\mathring{a})(x_p)$.
- When a is partial i.e. $a = \mathring{a} \cdot 1^d$ and $t|_{\mathring{a}}$ is of the form $x t_1 \dots t_q$ with $t|_a = x t_1 \dots t_{p-d}$, then $t|_{\mathring{a}} = t|_a t_{p-d+1} \dots t_q$ and $F_1 := (k \cdot \mathbb{T}(\mathring{a} \cdot 1^{d-1} \cdot k))_{k \in \text{ArgTr}^1(a)}$, \dots , $F_d := (k \cdot \mathbb{T}(\mathring{a} \cdot k))_{k \in \text{ArgTr}^k(a)}$ (see end of Sec. 10.2.3 for notation ArgTr^i): thus, F_i is the sequence of types given to the i -th argument t_{p-d+i} of $t|_a$ w.r.t. position a .

10.5.2 Natural Extensions

Let A be a support candidate for t and \mathring{T} a function from $\mathring{A} := \{\mathring{a} \mid a \in A\}$ to the set of types. We want to extend \mathring{T} on A (into a function \mathbb{T}) so that we get a correct derivation P typing t .

As we have seen above, we must capture the way *calls* are made in a derivation by a type to others located deeper. For that, to each $a \in \mathbb{N}^*$, we attribute an *indeterminate* X_a . Intuitively, X_a calls for $\mathbb{T}(a)$, the type given to the subterm at position a . For all $a \in A$, $x \in \mathcal{V}$, we set $A_a(x) = \{a_0 \in A \mid a \leq a_0, t(a) = x, \nexists a'_0, a \leq a'_0 \leq a_0, t(a'_0) = \lambda x\}$ so that we intend to have $\mathbf{Ax}_a^P(x) = A_a(x)$ (as well as $\text{tr}^P(a_0) = \lfloor a_0 \rfloor$) when P is built.

Combining all the above intuitions, we set, for all $a \in A$, $x \in \mathcal{V}$, $\mathbf{E}(a)(x) = (\lfloor a_0 \rfloor \cdot X_{a_0})_{a_0 \in A_a(x)}$ (thus, $\mathbf{E}(a)(x)$ calls for the types given to a in \mathbf{ax} -rules). If $a \in A$ is partial or unconstrained, $d = \text{rdeg}(a)$ (i.e. $a = \mathring{a} \cdot 1^d$) and $1 \leq i \leq d$, we define the sequence $\mathbf{R}_i(a)$ by $\mathbf{R}_i(a) = (k \cdot X_{\mathring{a}_{d-i} \cdot k})_{k \in \text{ArgTr}_A^i(a)}$ with ArgTr_A^i defined as for derivations (thus $\mathbf{R}_i(a)$ calls for the types given to the argument of the i -th application below a).

- If $a \in A$ is a non-zero position e.g., $t|_a$ is of the form $\lambda x_1 \dots x_d. t|_{\mathring{a}}$. We then set $\mathbf{Cal}(a) = \mathbf{E}(a \cdot 0)(x_1) \rightarrow \dots \rightarrow \mathbf{E}(\mathring{a})(x_d) \rightarrow \mathbb{T}(\mathring{a})$.
- If $a \in A$ is partial, we set $\mathbf{Cal}(a) = \mathbf{R}_1(a) \rightarrow \dots \rightarrow \mathbf{R}_n(a) \rightarrow \mathbb{T}(\mathring{a})$.
- If $a \in A$ is unconstrained, we set $\mathbb{T}(a) = \mathbf{Cal}(a)$.

We then extend \mathring{T} (defined on unconstrained positions) to A by the following coinductive definition: for all $a \in A$, $\mathbb{T}(a) = \mathbf{Cal}(a)[\mathbb{T}(a')/X_{a'}]_{a' \in \mathbb{N}^*}$. For all $a \in A$, we define the contexts $\mathbb{C}(a)$ by $\mathbb{C}(a)(x) = \mathbf{E}(a)(x)[\mathbb{T}(a')/X_{a'}]_{a' \in A_a(x)}$.

Those definitions are well-founded, because whether a is non-zero position or a partial one, every occurrence of an X_k is at depth ≥ 1 and the coinduction is *productive*. Eventually, let P be the labelled tree whose support is A and such that, for $a \in A$, $P(a)$ is $\mathbb{C}(a) \vdash t|_a : \mathbb{T}(a)$.

Lemma 10.11. The labelled tree P defined above is a derivation.

Proof sketch. Let $a \in A$. Whether $t(a)$ is x , λx or $@$, we check the associated rule has been correctly applied. Roughly, this comes from the fact that the variable $X_{a'}$ is "on the good track" (*i.e.* $[a']$) in $\mathbf{E}(a)(x)$, as well as in $\mathbf{R}_i(a)$, thus allowing to retrieve correct typing rules. \square

We call then the derivation P built above the **natural extension** of the pair (A, \hat{T}) . Natural extension give all the possible quantitative derivations typing a NF. For our purpose, they also give:

Lemma 10.12. A normal form t is unforgetfully typable.

Proof sketch. We set $A = \mathbf{supp}(t)$ and $\hat{T}(a) = o$ for each unconstrained position (where o is a type variable). Then, the extension P of (A, \hat{T}) is an unforgetful derivation typing t . \square

10.5.3 Approximability

It is enough for a derivation typing a NF to be quantitative in order to be valid.

Lemma 10.13. If P is a quantitative derivation typing a NF t , then P is approximable.

We explain here why every quantitative derivation P typing a normal form is approximable. This means that we can build, for any finite part 0B of $\mathbf{bisupp}(P)$, a finite derivation ${}^fP \leq P$ containing 0B . We will proceed by:

- Choosing a finite support candidate ${}^fA \subseteq A$ of t *i.e.* we will discard all positions in A but finitely many.
- Then, choosing, for each $\hat{a} \in {}^fA$, a finite part of ${}^f\mathbf{T}(\hat{a})$ of $\mathbf{T}(\hat{a})$.

The natural extension of $({}^fA, {}^fT)$ will be a derivation ${}^fP \leq P$ typing t .

Namely, we define P_n , the depth n truncation of P as follows:

- We define A_n by discarding every position $a \in A$ s.t. $\mathbf{ad}(a) > n$ or a contains a track $> n$ (*i.e.* $A_n = \{a \in A \mid \mathbf{ad}(a) \leq n \text{ and } \max(a) \leq n\}$). Since $t \in \Lambda^{001}$, A does not have infinite branch of finite applicative depth and thus, A_n is a *finite* set of positions.
- For each $\hat{a} \in \hat{A}_n$, we define $\hat{T}_n(\hat{a})$ by discarding every $c \in \mathbf{supp}(\mathbf{T}(\hat{a}))$ s.t. $\mathbf{ad}(c) > n$ or c has a track $> n$ (*i.e.* $\mathbf{supp}(\hat{T}_n(\hat{a})) = \{c \in \mathbf{supp}(\mathbf{T}_n(\hat{a})) \mid \mathbf{ad}(c) \leq n \text{ and } \max(c) \leq n\}$). Since $\mathbf{T}_n(\hat{a}) \in \mathbf{Typ}$ (and not in $\mathbf{Typ}^{111} - \mathbf{Typ}^{001}$), $\hat{T}_n(\hat{a})$ is a *finite* type.

We define now P_n as the natural extension of (A_n, \hat{T}_n) . Using the quantitativity of P , we may prove then that, for all ${}^0B \subseteq \mathbf{bisupp}(P)$, there exists a large enough n s.t. ${}^0B \subseteq \mathbf{bisupp}(P_n)$. The idea is the following: as we have seen, each biposition may \mathbf{p} "call" a chain of deeper bipositions, but the set of bipositions called by \mathbf{p} is finite and we may define the **called applicative depth** $\mathbf{cad}(\mathbf{p})$ of \mathbf{p} as the maximal applicative depth of a biposition called by \mathbf{p} . Then, since 0B is finite, we define n as $\max(\{\mathbf{cad}(\mathbf{p}) \mid \mathbf{p} \in {}^0B\})$ (modulo the maximum of tracks contained in the called bipositions) and we check that ${}^0B \subseteq P_n$ (see Appendix A.4 for a complete proof). By Lemmas 10.12 and 10.13, we may now assert:

Proposition 10.6. Every NF is approximably and unforgetfully typable in system \mathbf{S} .

10.5.4 The Infinitary Subject Expansion Property

In Section 10.4.4, we defined the derivation P' resulting from a s.c.r.s., starting at an (approximable or not) derivation P . Things do not work so smoothly for subject expansion when we try to define a good derivation P which results from a derivation P' typing the limit of a s.c.r.s.. Indeed, approximability play a central role w.r.t. expansion. Assume that:

- $t \rightarrow^\infty t'$. Say by means of the s.c.r.s. $t = t_0 \xrightarrow{b_0} t_1 \xrightarrow{b_1} \dots t_n \xrightarrow{b_n} t_{n+1} \rightarrow \dots$ with $b_n \in \{0, 1, 2\}^*$ and $\text{ad}(b_n) \rightarrow \infty$.
- P' is an approximable derivation of $C' \vdash t' : T'$.
- We dispose of an arbitrary injection $a \mapsto [a]$ from \mathbb{N}^* to $\mathbb{N} \setminus \{0, 1\}$.

We want to show that there exists a derivation P concluding with $C' \vdash t : T'$. A complete proof is given in Appendix A.3.3.

The main point is to understand how subject expansion works with a finite derivation ${}^fP' \leq P'$. The techniques of Sec. 10.1.3 can now be formally performed. We give a sketch of the proof.

Since ${}^fP'$ is finite, for a large enough n , t' can be replaced by t_n inside ${}^fP'$, according to Lemma 10.8: we set ${}^fP_n = {}^fP'[t_n/t']$, which is a finite derivation typing t_n . But when t_n is typed instead of t' , we can perform n steps of $[\cdot]$ -expansion (starting from fP_n) to obtain a finite derivation fP typing t .

By monotonicity of expansion (Lemma 10.4), the set D containing all the fP is a directed set. Then, by Theorem 10.3, we define P as the join of the fP when ${}^fP'$ ranges over $\text{Approx}(P')$. This yields a derivation satisfying the desired properties.

Proposition 10.7. Assume $t \rightarrow^\infty t'$ and $P' \triangleright C' \vdash t' : T'$.

If P' is approximable, then there exists an approximable derivation P s.t. $P \triangleright C' \vdash t : T'$.

Since infinitary subject reduction and expansion (in s.c.r.s) preserve unforgetful derivations, we can now prove our main characterization theorem :

Theorem 10.4. A term t is weakly-normalizing in Λ^{001} if and only if t is typable by means of an approximable unforgetful derivation and if and only if it is hereditary head normalizing.

Proof. We use the proof scheme of 3.3.1, already used in the case of finite weak normalization (*cf.* proof of Proposition 5.1).

- The implication «If t is typable, then t is HHN» is given by Proposition 10.4.
- The implication «If t is HHN, then t is WN» is obvious.
- The implication «If t is WN, then t is typable» is proved like this: assume t to be WN and consider a s.c.r.s. converging to the NF t' of t . Let P' be an unforgetful and approximable derivation typing t' (such a derivation exists by Proposition 10.6). Then, the derivation P obtained by Proposition 10.7 above is approximable, unforgetful and types t .

□

10.6 Conclusion

We have provided an intersection type system characterizing weak normalization in the infinitary calculus Λ^{001} . The use of functions from the set of integers to the set of types to represent intersection – instead of multisets or conjunctions – allows us to express a validity condition that could only be suggested in Gardner/de Carvalho's type assignment system. Our type system is relatively simple and offers many ways to describe proofs (*e.g.*, tracking, residuals).

Interestingly, system **S** can be used to give a *positive* answer to TLCA Problem # 20: can the the so called **Hereditary Permutations (HP)** be characterized by means of types? Tatsuta [102] proved that it is not possible in the inductive case. By lack of time, this problem and its solution could not be presented in this thesis, but they are sketched in Appendix A.7, along with the useful definitions.

It is natural to seek out whether this kind of framework could be adapted to other infinitary calculi and if we could also characterize strong normalization in Λ^{001} , using for instance a memory operator [19]. Moreover, sequential intersection may be connected to Grellois and Melliès infinitary exponential modality [51], as well as Bucciarelli and Ehrhard indexed linear logic [16]. It is to be noticed that derivation approximations provide *affine* approximations that behave *linearly* in Mazza's polyadic calculus [80].

This chapter implicitly raises two questions:

- **Question 1:** We know from Sec. 10.1.3 (and Appendix A.1) that some mute terms are typable in system \mathcal{R} and in system **S** (when the latter is not endowed with the approximability criterion). What is the set of typable terms in system **S** when approximability is put aside?
- **Question 2:** The derivation of **S**-are very low-level objects. Indeed, typing an application is very constraining since it relies on syntactic equality, whereas the multiset constructions of system \mathcal{R} makes it more flexible. Is system **S** (without approximability) less expressive than system \mathcal{R} is?

Part IV is dedicated to answering those two questions:

- We prove in Chapter 12 that, actually, every term is typable in system **S**. From the semantic point of view, we explain how this result has interesting consequences on a relational model of the pure λ -calculus, whose points are the derivable judgments of system \mathcal{R} .
- We prove in Chapter 13 that every \mathcal{R} -derivation can be “simulated” by a **S**-derivation. In particular, this implies that system **S** provides a complete description of the model based on system \mathcal{R} .

Part IV

Understanding Unproductive
Reduction
through Logical Methods

Presentation

In the last part of this thesis, we present two independent contributions:

- **Contribution 1:** In Chapter 12, we prove that, when the approximability condition used to ensure soundness with coinductive type grammar in Chapter 10 is dropped, then:
 - Every term is (non-trivially) typable in a relevant way, both in system \mathbf{S} or in system \mathcal{R} (the coinductive version of system \mathcal{R}_0). We then said that system \mathbf{S} and system \mathcal{R} are **completely unsound**.
 - The order of λ -terms can be type-theoretically characterized.

This contribution can be seen as a **linearization of λ -terms**, because system \mathbf{S} is linear (it does not feature contraction and weakening), and since it satisfies subject reduction and confluence, it can be understood as a deterministic calculus.

- **Contribution 2:** In Chapter 13, we explicit the natural collapse of system \mathbf{S} (coinductive grammar with sequential intersection) onto system \mathcal{R} (coinductive grammar with multiset intersection) by showing that:
 - Every \mathcal{R} -derivation is the collapse of an \mathbf{S} -derivation (**surjectivity of the collapse**).
 - Every sequence of reduction choices (defined in Sec. 4.1.2) in \mathcal{R} can be implemented in system \mathbf{S} .

As a consequence, the rigid syntax directed type system \mathbf{S} does not bring any limitation compared to system \mathcal{R} (syntax directed, but non-rigid/non-deterministic) or to (the coinductive versions of) Gardner original system (rigid but not syntax directed, since it processes permutation explicitly).

Chapter 11 gives a user-friendly presentation of some useful techniques and notions associated to this part.

In sharp contrast with the previous chapters of this thesis, in Part IV we must study types that do not ensure any kind of productivity/normalization. The two mentioned contributions share a same method, which is one of the main technical contribution of this PhD. It is referred to as the **collapsing strategy**. Before giving a high-level account of those new techniques, let us first understand why the standard methods of intersection types (and even their extensions to coinductive frameworks, as in Chapter 10) cannot be used.

The Productivity of Most Type Systems A pivotal element to prove that some terms (*e.g.*, the head normalizing, weakly normalizing, strongly normalizing terms in the case (or 001-weakly normalizing terms) are typable, along with subject expansion, consists in typing *blocks* of the form $x t_1 \dots t_q$ (*i.e.* a zero head normal form). This is well illustrated by Figure 3.4, p. 100 and 10.6, p. 228. More precisely, this figure shows that intersection type systems *work* because they are designed (among other things) so that the blocks of the form $x t_1 \dots t_q$ are easily typable (including the case where some t_k are untyped): to type $x t_1 \dots t_q$ in a standard intersection type system, we just have to assign to x a type of the form $I_1 \rightarrow \dots \rightarrow I_p \rightarrow B$, where I_1, \dots, I_k are the (possibly empty in some cases) intersection of the types given to t_1, \dots, t_q .

Conversely, another key feature of intersection type systems – at least, of reasonable systems *e.g.*, \mathcal{D}_0 , \mathcal{R}_0 , \mathcal{S} , $\mathcal{H}_{\lambda_\mu}$, $\mathcal{S}_{\lambda_\mu}$ or \mathbf{S} endowed with approximability – is that they ensure that any typed subterm will output a block of the form $x t_1 \dots t_q$ – up to a series of abstractions (or naming, in the case of λ_μ) – after a finite number of steps. Those systems are **productive** in that aspect. It is only natural: they aim at characterizing (some form of) normalization.

On the other hand, with a coinductive type grammar, it is not difficult to type *e.g.*, $\Omega = \Delta \Delta$, which⁴ is a mute term (Sec. 2.3.2). No part of a mute term can stabilize. In particular, the reduction of a mute term will never output a stable block of the form $x t_1 \dots t_q$. So, if we want to study the derivations of system \mathbf{S} or system \mathcal{R} , our study cannot reduce to typed blocks of the form $x t_1 \dots t_q$ and then proceed by expansion. In particular, this method would not work to describe the set of typable terms in system \mathcal{R} .

The two problems addressed by Contributions 1 and 2 above are easy to solve for (terms that reduce to) blocks of the form $x t_1 \dots t_q$. But the example of Ω shows that this does not encapsulate their whole generality. Contributions 1 and 2 also concern terms that do not productively reduce.

The Collapsing Strategy and Logical Methods So, how do we handle type-theoretic questions involving **unproductive terms**? The method that we describe relies upon some of the following ideas:

1. We reduce the two problems of each chapter to a first order theory \mathcal{T} (containing only constants). Those constants correspond to (sets of) bipositions of system \mathbf{S} (Sec. 10.3.1). This is possible only because system \mathbf{S} is rigid, contrary to system \mathcal{R} (Sec. 4.1.1). In all cases, the transformation of the two problems above into such a theory demands a lot of work and cannot be described quickly, but this does not matter for this discussion.
2. We show that, when the theory \mathcal{T} is not contradictory, then the problems can be positively answered: from a non-contradictory theory, we can build some derivations of a desired form. This point is reminiscent of some methods of Zermelo-Fraenkel's Set Theory *e.g.*, the proof of the Theorem of Completeness (in a far simpler case).
3. We prove that \mathcal{T} is coherent, and this is the most difficult part of our contribution. For that, we assume *ad absurdum* that \mathcal{T} is contradictory and we consider a (first order) proof of the contradiction of \mathcal{T} . We call such a proof a **chain** (because this consists in a series of equalities/relations). Then:

⁴Such a typing of Ω can be found in Sec. A.1.

- Chains interact with redexes in an undecidable (and therefore, unhandleable) way.
- One idea would be to eliminate all the redexes. But since we must also handle some mute terms, this is not possible in general.
- This problem is escaped by defining a finite reduction strategy, that given a chain \mathcal{C} , eliminates only the redexes that interact wrongly with \mathcal{C} and outputs a new chain \mathcal{C}' that is said to be **normal**.
- Normal chains have a form that is handleable: we then prove that they do not exist *i.e.* there is no *normal* proof of contradiction of \mathcal{T} .
- Since any proof of contradiction of \mathcal{T} could be normalized, we conclude that such a proof does not exist. Thus, \mathcal{T} is coherent.

The proof of Theorem 12.2 features a variant of this method, in which no *ad absurdum* argument is needed, but some chains must still be normalized.

Threads, Syntactic Polarity and Redex Towers Let us now say a few words about the collapsing strategy, and gives the name of the concepts and methods to be informally presented in Chapter 11.

- As mentioned above, we will need to consider relations on sets of bipoositions. Those sets are called **threads** and correspond to the tracking of symbols inside a derivation. The symbols of a thread occur either positively or negatively. Threads and syntactic polarity are informally presented in Sec. 11.
- Threads can be **consumed** in **app**-rules, meaning that they are destroyed. It is actually not the interaction between threads and redex that is problematic/undecidable, but the consumption of a *negative* occurrence of the *left* premise of an **app**-rule. But we show that this can only occur in a *redex tower i.e.* a series of redexes, which can be collapsed. This is informally explained in Sec. 11.2.

This last point implies that terms are reduced to fulfill the method. Thus, the effect of reduction on threads must be precisely described. For that, residuation (presented in Sec. 2.1.5 and defined in Sec. 10.3.5 for system \mathbf{S}) plays a key role.

We claim that the method presented in this part is modular and has some canonicity: not only it is applied thrice in this document (once for Theorem 12.1, one for Theorem 12.2 and once for Theorem 13.2), but many definitions varies from Chapter 12 to Chapter 13. Intuitively, Chapter 12 consider threads of (bi)positions whereas Chapter 13 consider threads of *edges* (that are abusively denoted by bipoositions or by positions) *i.e.* the sets of the constants considered in the theory \mathcal{T} differ. As a consequence, residuation of threads, which plays an important part in the collapsing technique, is not defined in the same way in the two chapters. Many binary relations between threads, needed in the proof of complete unsoundness, are unnecessary in Chapter 13.

In other words, the two contributions do not involve the same first order theories (different sets of constants and different relations), which suggests that the technique is indeed modular. In both cases, the technique relies upon threads, syntactic polarity, consumption, and the collapsing of redex towers to destroy the cases of negative consumption on the left-hand side.

The Correspondences between Chapter 12 and 13 Although the two chapters do not have exactly the same structure, they share some common elements, summarized with the following table:

	Chapter 12	Chapter 13
If some theory \mathcal{T} is consistent, then the problem is solved	Corol. 12.1	Prop. 13.2
Describing the theory \mathcal{T}	Sec. 12.2	Sec. 13.3, 13.4.1, 13.4.3
Defining problematic chains	Def. 12.2 (brother ch.)	Def. 13.13 (nihilating ch.)
Defining normal chains	Def. 12.4	Def. 13.17
Defining threads	Def. 12.1	Def. 13.10
Specific Lemma(s)	Lem. 12.6, 12.9, 12.8, 12.11	Lem. 13.11
The collapsing strategy	Sec. 12.4	Sec. 13.5
Normal chain do not exist	Sec. 12.3.3	Sec.13.5.3

In each case, the specific lemmas are the properties used to prove that the *normal* chains do not exist. The threads are the fundamental concepts on which the theories \mathcal{T} are based. Their definitions and descriptions also follow the same structure:

	Chapter 12	Chapter 13
Ascendance, polar inversion	Sec. 12.2.3	Sec. 13.4.1
Consumption	Sec. 12.2.4	Sec. 13.4.1
Syntactic polarity	Def. 12.3	Def. 13.15
Form of ascendant threads	Lem. 12.2	Lem. 13.5
Form of threads	Lem. 12.3	Lem. 13.6
Uniqueness of consumption	Lem. 12.4	Lem. 13.7

Chapter 11

An Informal Presentation of Threads

In this chapter, we give important intuitions to be extensively used in Chapters 12 and 13, and formalized for system \mathbf{S} . We informally present the notions of *ascendance*, *polar inversion*, *threads*, *syntactic polarity* and *consumption*. Those notions are based on tracking type symbol in derivation. We know from Remark 4.1 that it is actually not possible to do that in system \mathcal{R}_0 . However, it is far more convenient to present the intuitions in this system (this chapter is concluded with a short formalization in system \mathbf{S} , Sec. 11.3), and ascendance, polar inversion etc can very well be understood with figures representing \mathcal{R}_0 -derivations. In Sec. 11.2, we present redex towers, which generalize redexes: a redex can be collapsed in one step of reduction. A redex tower can be collapsed in a finite number of reduction steps. We hint at why those objects are necessary to study and handle threads.

Throughout this chapter, we use the informal expression **type symbol**. In system \mathbf{S} , type symbols are pointed to by **bipositions** (Sec. 10.3.1). Thus, most identifications and concepts that are informally presented here are formalized with bipositions in system \mathbf{S} .

11.1 Threads, Syntactic Polarity and Consumption

In this section, we are interested in the types of the subjects of the judgments nested in a \mathcal{R}_0 -derivation Π (not in the types in the contexts of those judgments).

11.1.1 Ascendance, Polar Inversion and Threads

Throughout this section, we consider 4 pairwise distinct type variables o , o_1 , o_2 and o_3 and terms t , u , v such that $t = (\lambda zxy.x(yz)y)(fu)v$ and a \mathcal{R}_0 -derivation Π typing t represented in Fig. 11.1. Very informally, we confound (the subterms of) t and (the subderivations of) Π in our discourse.

Let us remind rules **abs** and **app** of system \mathcal{R}_0 :

$$\frac{\Gamma; x : [\sigma_i]_{i \in I} \vdash t : \tau}{\Gamma \vdash \lambda x.t : [\sigma_i]_{i \in I} \rightarrow \tau} \text{ abs} \qquad \frac{\Gamma \vdash t : [\sigma_i]_{i \in I} \rightarrow \tau \quad (\Delta_i \vdash u : \sigma_i)_{i \in I}}{\Gamma +_{i \in I} \Delta_i \vdash tu : \tau} \text{ app}$$

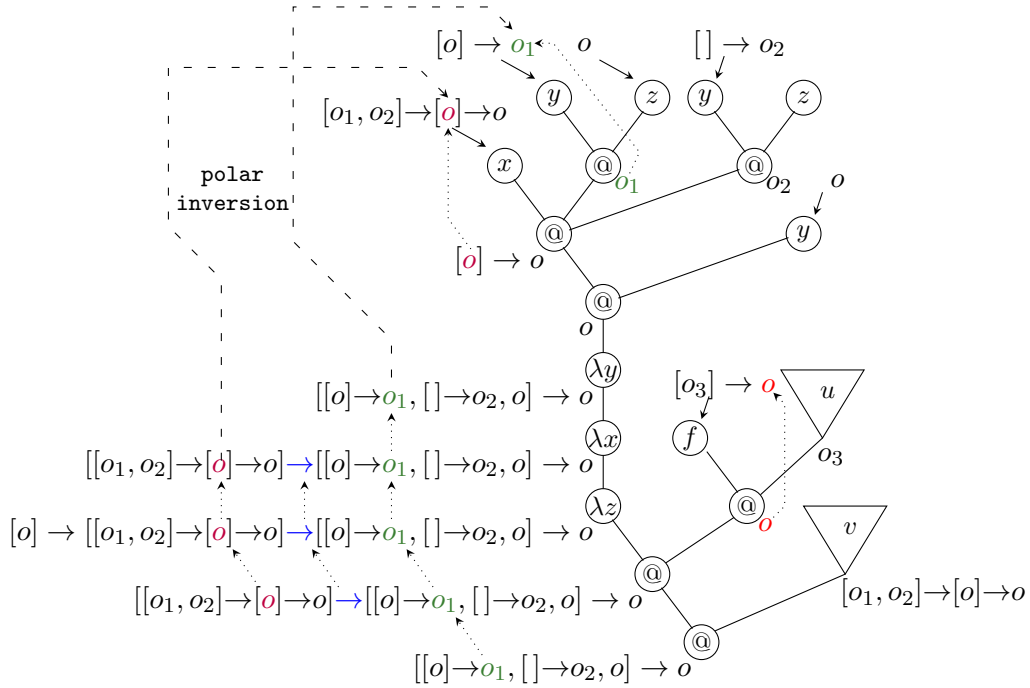


Figure 11.1: Ascendance, Polar Inversion and Threads (Informal)

- In the **abs**-rule, the occurrence of τ in the premise and that (as a subpart of $[\sigma_i]_{i \in I} \rightarrow \tau$) in the conclusion can be identified. We say the former is the **ascendant** of the latter (note that the σ_i do not have ascendants, but they will have *polar inverses*).
- Likewise, in the **app**-rule, the occurrence of τ (as a subpart of $[\sigma_i]_{i \in I} \rightarrow \tau$) in the premise and that in the conclusion can be identified. We also say that the former is the *ascendant* of the latter.

Following this idea, some types symbols can be ascendants of others: in Fig. 11.1, we have represented the ascendance relation with dotted arrows (the two long *dashed* arrows annotated with polar inversion, which we will describe later). We call a set of ascendants an **ascendant thread**. For instance, in Fig. 11.1, 6 ascendant threads are outlined: one is blue, one is red, two are green and two are purple.

- The red ascendant thread contains 2 occurrences of the type variable o (ascendance through the **app**-rule typing $f u$).
- The blue one goes from the root of $(\lambda z x y. x(yz)y)(f u)$ (position 1) to the root of $\lambda x. x(yz)y$ and contains 3 occurrences of the arrow \rightarrow (one ascendance through an **app**-rule and two through **abs**-rules).
- The leftmost green ascendant thread goes from the root of t to the root of $\lambda y. x(yz)y$ (below an arrow of polar inversion). It contains 5 occurrences of the type variable o_1 .
- The rightmost green ascendant thread contains 2 occurrences of the type variable o_1 (ascendance through an **app**-rule typing yz).

- The leftmost purple ascendant thread goes from the root of $(\lambda zxy.x(yz)y)u$ (pos. 1) to the root of $\lambda xy.x(yz)y$ (pos. $1^2 \cdot 0$, below an arrow of polar inversion). It contains 3 occurrences of the type variable o .
- The rightmost purple ascendant thread contains 2 occurrences of the type variable o (ascendance through the **app**-rule typing $x(yz)$).

Observation 11.1.

- The top occurrence of an ascendant thread is found either in an **ax**-rule (*e.g.*, the red one or the rightmost green one) or in an **abs**-rule (*e.g.*, the blue one or the leftmost green one).
- The bottom occurrence of a thread is found either at the root of the typed term (*e.g.*, the leftmost green one) or in a (left or right) premise of an **app**-rule (*e.g.*, the five other ascendant threads). In that case, we say that the thread is *consumed* in this **app**-rule. Consumption is the object of the next section.

Intuitively, an **abs**-rule typing an abstraction $\lambda x.t$ “calls” the types of x in t , and puts them as the source of the arrow type typing $\lambda x.t$ *i.e.* if $\lambda x.t$ is typed $[\sigma_i]_{i \in I} \rightarrow \tau$, then the types σ_i of $[\sigma_i]_{i \in I}$ correspond to the types assigned to (the free occurrences of) x . We say that each occurrence of σ_i (in $\lambda x.t : [\sigma_i]_{i \in I} \rightarrow \tau$) is the **polar inverse** of that in $x : \sigma_i$). Following this idea, some type symbols can be the polar inverses of others. In Fig. 11.1:

- One dashed arrow represents polar inversion from an occurrence of the type variable o_1 to another. The target of this arrow is the occurrence of o_2 in one of the **ax**-rules typing y . The source of this arrow is the occurrence of o_2 when the types of y are “called” by λy : it is located at the root of $\lambda y.x(yz)y$.
- Likewise, another arrow of polar inversion relates two occurrences of the type variable o , from a node labelled with λx (corresponding to an **abs**-rule) to a node labelled x (corresponding to an **ax**-rule).

A **thread** is a set of occurrences of a same type symbol, that are identified through ascendance or polar inversion. In Fig. 11.1, 4 threads are outlined: the red one, the blue one, the green one and the purple one. The green thread, as well as the purple one, is the union of two ascendant threads and features a polar inversion. The red and the blue threads are just ascendant threads (without polar inversion): the red one originates from f , that is not abstracted (f occurs free in t) and the blue one stops at the arrow created by λx (it is neither in the target nor in the source of the type of $\lambda x.x(yz)y$, so that it does not have an ascendant nor a polar inverse).

11.1.2 Syntactic Polarity and Consumption

In this section, we present the notion of syntactic polarity and consumption. We will often refer to Fig. 11.2, representing a \mathcal{R}_0 derivation typing the term $t = (\lambda xy.x(yz)y)vu$ with $x, y \notin \text{fv}(u)$.

Consumption is a phenomenon that occurs in **app**-rules. Let us recall:

$$\frac{\Gamma \vdash t : [\sigma_i]_{i \in I} \rightarrow \tau \quad (\Delta_i \vdash u : \sigma_i)_{i \in I}}{\Gamma +_{i \in I} \Delta_i \vdash tu : \tau} \text{app}$$

Let $i_0 \in I$. We notice that the type σ_{i_0} occurs both in the left premise of the rule and in a right premise (as a part of the type $[\sigma_i]_{i \in I} \rightarrow \tau$ of t and as a type of the argument u), but it does not occur in the conclusion typing tu . In other words, σ_{i_0} does have a **descendant** (the inverse notion of **ascendant**). We then say that σ_{i_0} (and more generally, $[\sigma_i]_{i \in I}$) has been **consumed** in the **app**-rule. Moreover, the occurrence of σ_{i_0} that is in the leftmost premise can be identified with the one located in an argument premise. We say that the former one is **left-consumed** and the latter one is **right-consumed**. For instance, in Fig. 11.2:

- The multiset type $[\sigma_1, \sigma_2]$ is consumed at the root of Π .
- The multiset type $[[\sigma_1, \sigma_2] \rightarrow [\sigma]\tau]$ is consumed at the root of (the subderivation typing) $(\lambda xy.x y u)v$.
- The multiset type $[\sigma_1, \sigma_2]$ is consumed at the root of xy .

Equivalently, it will be convenient (Sec. 13.1.2) that $[\sigma_i]_{i \in I}$ (occurring as the source of the arrow type $[\sigma_i]_{i \in I} \rightarrow \tau$) is the **left key** of the **app**-rule above, whereas the occurrences σ_i typing u (on the right-hand side) constitute the **right key** of this **app**-rule. Thus, the left key corresponds to the type symbols that are left-consumed and the right-key to the type symbols that are right-consumed.

Observation 11.2. In Observation 11.1, we remarked that the bottom end of an ascendant thread was located either in the root of the derivation or in the premise of an **app**-rule: equivalently, in an ascendant thread, at most one occurrence can be consumed. When this occurs, we also say that the *ascendant thread* has been *consumed* in the **app**-rule consuming its bottom occurrence.

For instance, coming back to Fig. 11.1:

- The red ascendant thread is right-consumed in the **app**-rule typing the root of t (pos. ε).
- The blue ascendant thread and the left purple ascendant threads are left-consumed at the root of t .
- The left green ascendant thread is not consumed (its bottom occurrence is at the root of t).
- The right green ascendant thread is right-consumed at position $1^2 \cdot O^3 \cdot 1 \cdot 2$ (corresponding to an **app**-rule typing yz).
- The right purple ascendant thread is left-consumed in the **app**-rule typing $x(yz)y$ (pos. $1^2 \cdot 0^3$).

Let us call the **top ascendant** of a type symbol the top occurrence of its ascendant thread. The **syntactic polarity** of a type symbol \mathbf{s} depends on the position of its top ascendant: if the top ascendant of \mathbf{s} is in an **abs**-rule (*i.e.* the type symbol is created by an λx), the syntactic polarity of \mathbf{s} is negative and if its top ascendant is in an **ax**-rule (*i.e.* the type symbol directly ascends to a leaf of the derivation), then the syntactic polarity of \mathbf{s} is positive. Thus, all the occurrences of an *ascendant* thread have the same syntactic polarity.

In Fig. 11.2, we have colored the parts of the types whose polarity is negative (resp. positive, resp. unknown) in blue (resp. red, resp. purple). For instance, there is not

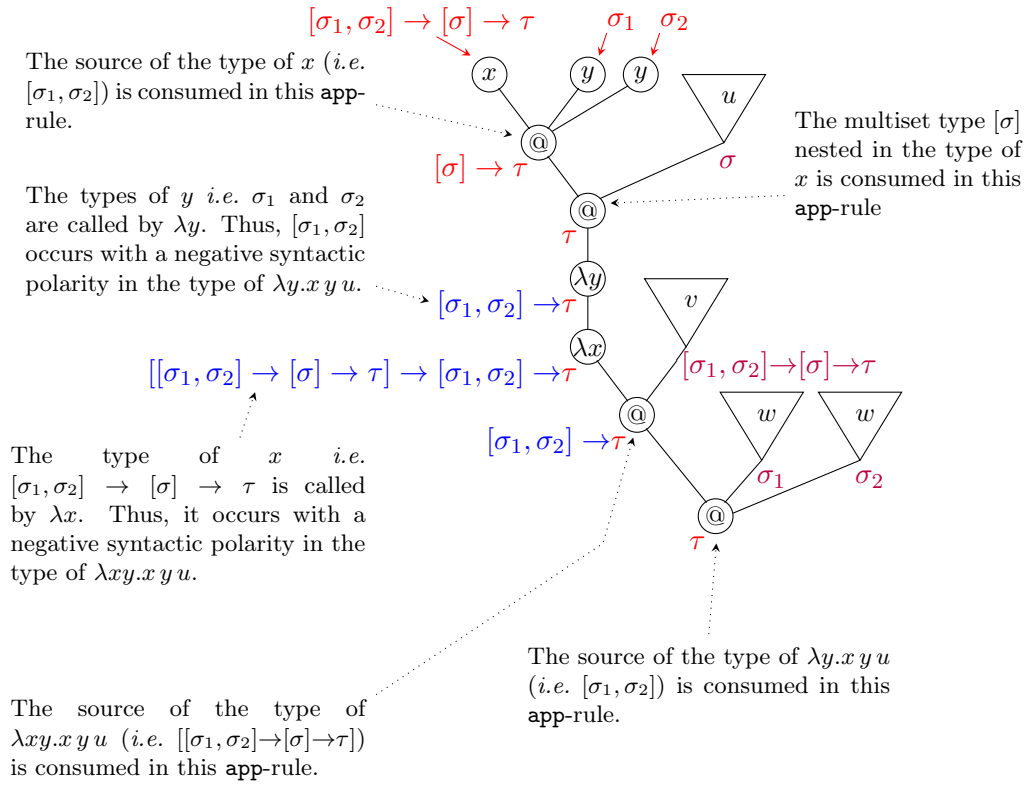


Figure 11.2: Consumption and Syntactic Polarity (Informal)

enough information to know how to precisely color the types of the subterms u , v and w (it depends on their parsing trees).

Remark 11.1. The syntactic polarity of a type symbol in a given type τ does not depend on its position in τ , but on the position of τ inside the derivation. It actually only depends on the parsing tree of the subject (hence the attribute *syntactic*).

Observation 11.3.

- Note that only the top occurrence of an ascendant thread can have a polar inverse. Thus, a thread contains either one or two ascendant threads. For instance, in Fig. 11.2, the red and the blue ascendant threads are actually full threads: the red one is a positive ascendant thread whereas the blue one is a negative ascendant thread. In contrast, the green thread and the purple one each contains two ascendant threads: one is negative and the other is positive.
- By Observation 11.2, an ascendant thread is consumed at most once. Since a full thread contains at most two ascendant threads, a full thread can be at most consumed twice (once positively and once negatively).

For instance, in Fig. 11.1, the purple thread is consumed twice:

- It is left-consumed positively in the **app**-rule typing the root of $x(y z)y$ (pos. $1^2 \cdot 0^3$).
- It is left-consumed negatively in the **app**-rule typing the root of t .

11.1.3 Referents of Threads, Applicative Depth and Brotherhood

In Chapter 13, we will need to define the referent of a thread: when a thread θ has a positive¹ occurrence, then the **referent** of θ is the top positive occurrence of θ .

Remember that applicative depth (defined in Sec. 2.3.5) is the number of nestings inside arguments of applications. For a type symbol \mathbf{s} in a derivation Π , we define $\text{ad}(\mathbf{s})$ as the applicative depth of the rule in which $\text{ad}(\mathbf{s})$ is located. Then:

Observation 11.4. We then note that all the occurrences of an *ascendant* thread have the same applicative depth (since ascendance passes through abstractions or left-hand sides of applications).

For instance, in Fig. 11.1, the applicative depth of all the occurrences of the purple, the blue and the left green ascendant threads is equal to 0. The applicative depth of all the occurrences of the red and the right green ascendant threads is equal to 1.

Observation 11.5. Moreover, if type symbol \mathbf{s}_2 is the polar inverse of symbol \mathbf{s}_1 , then symbol \mathbf{s}_2 is above symbol \mathbf{s}_1 in the derivation tree: indeed, \mathbf{s}_2 is located in some **ax**-rule typing x and \mathbf{s}_1 in an **abs**-rule binding this occurrence of x . In particular, the applicative depth of \mathbf{s}_2 is greater or equal than that of \mathbf{s}_1 *i.e.* if a thread has both positive and negative occurrences, then the positive occurrences have greater applicative depth.

We then define the **applicative depth of a thread** θ as the maximal applicative depth of an occurrence of θ . In particular, the applicative depth of θ is equal to the applicative depth of the referent of θ . In Fig. 11.1, the applicative depth of the purple thread is equal to 0 and those of the green and the red threads are equal to 1.

Observation 11.6. Assume that θ_L is left-consumed positively and θ_R is right-consumed in the same **app**-rule. Then $\text{ad}(\theta_L) < \text{ad}(\theta_R)$. Indeed, let us say that θ_L occurs at type symbol \mathbf{s}_L in the left-premise of the **app**-rule and that θ_R occurs at type symbol \mathbf{s}_R in some argument premise. Since, by hypothesis, \mathbf{s}_L is positive, $\text{ad}(\theta_L) = \text{ad}(\mathbf{s}_L)$ thanks to Observations 11.4 and 11.5. Moreover, $\text{ad}(\mathbf{s}_R) \leq \text{ad}(\theta_R)$. Since $\text{ad}(\mathbf{s}_R) = \text{ad}(\mathbf{s}_L) + 1$, we obtain $\text{ad}(\theta_L) \leq \text{ad}(\theta_R)$.

Two type symbols are said to be **brothers** if

- they are nested in the same multiset types. . .
- . . . or they are the *roots*² of some types located in two argument premises of a same **app**-rule.

In Fig. 11.3 (we have removed some matterless parts of Fig. 11.1), the bottom purple occurrence of o_1 and the bottom green occurrence of o_2 are brother symbols because they are in the same multiset types. Likewise, the occurrences of o_1 and o_2 below v are brother symbols. In the upper-right corner of the figure, two occurrences of o_1 and o_2 are brother symbols, because (1) they occur at the root of the type (2) they are located in the two argument premises of the **app**-rule below x .

Let θ_1 and θ_2 be two threads. We then say that θ_1 and θ_2 are **brother threads** if θ_1 (resp. θ_2) has an occurrence \mathbf{s}_1 (resp. \mathbf{s}_2) such that \mathbf{s}_1 and \mathbf{s}_2 are brother symbols. Then, it is not difficult to see that::

¹ Defining referents for threads that only have negative occurrences is a bit more complicated and it will only be done in Chapter 13. We ignore this detail for now.

² Informally, the root of type o is o and the root of type $[\sigma_i]_{i \in I} \rightarrow \tau$ is this apparent occurrence of \rightarrow .

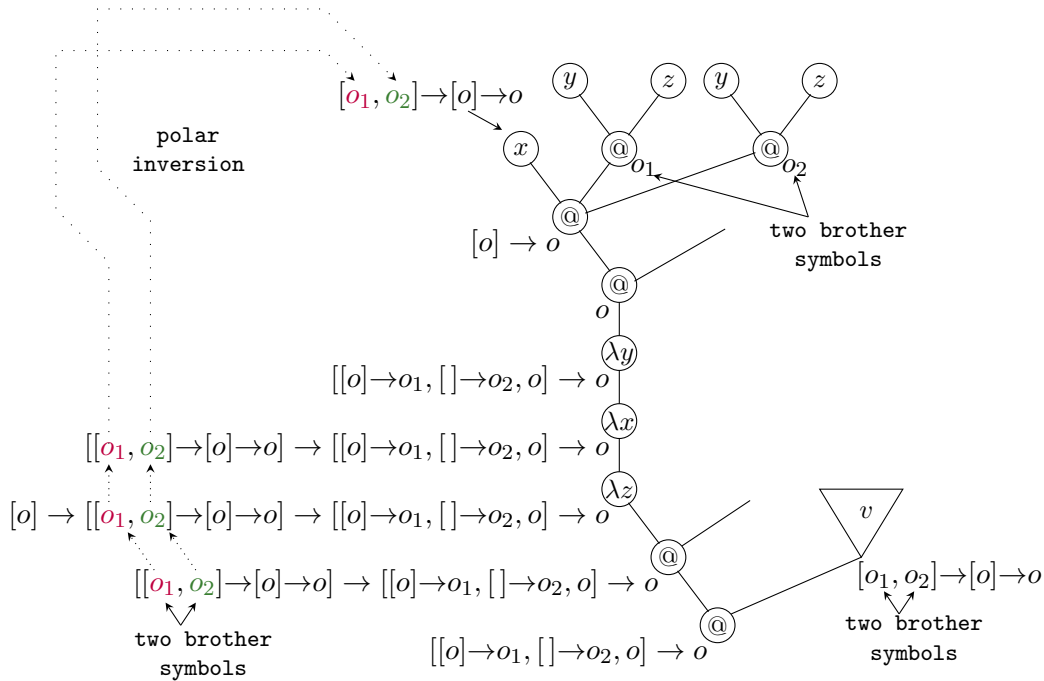


Figure 11.3: Brother Threads

Observation 11.7.

- If θ_1 and θ_2 are brother threads, then actually, every occurrence of θ_1 is brother with an occurrence of θ_2 , because brotherhood between type symbols is stable under ascendance and polar inversion.
- Thus, two brother threads have the same applicative depth.
- Moreover, two brother threads are consumed in the same places in the same way (on the left/right-hand side, positively/negatively).

Observation 11.7 is illustrated with Fig. 11.3 in which two threads (a green one and a purple one) are outlined: for the first point, each purple symbol is brother with a green symbol. Moreover, the applicative depths of the two threads are equal to 1 (2nd point). For the 3rd point:

- The two threads are left-consumed positively at the **app**-rule just below x .
- The two threads are both left-consumed negatively in the **app**-rule at the root of the tree.

A last observation, on the “compatibility” of brotherhood with consumption:

Observation 11.8. If two brother threads are consumed in some **app**-rule, then there are the counter-parts of two brother threads on the other side of the application.

In Fig. 11.3:

- W.r.t. the **app**-rule below x , the purple and the green threads are the left counterparts of the two brother threads starting at the root yz .

- W.r.t. the **app**-rule at the root, the purple and the green threads are the counterparts of the two brother threads whose bottom occurrences are indicated below v .

11.2 Collapsing Redex Towers

For reasons that are difficult to understand³ now, we shall need to find a method to eliminate any given *negative* thread that is *left*-consumed. This method, however, is easy to present informally.

11.2.1 Negative Left-Consumption and Redex Towers

In this section, we discuss the relation between negative left-consumption and some kind of terms to be called *redex towers*. Then, we explain how negative left-consumption can be destroyed by collapsing a redex tower.

Let us call the **stack** $\mathbf{st}(\tau)$ of a \mathcal{R}_0 -type τ the list of the multiset types that occur at top level in τ from the right to the left *e.g.*, $\mathbf{st}([o] \rightarrow o) = [o]$, $\mathbf{st}([\sigma] \rightarrow [\tau_1, \tau_2] \rightarrow [o] \rightarrow o) = [\sigma] \cdot [\tau_1, \tau_2] \cdot [o] \cdot o$ and $\mathbf{st}(o) = \varepsilon$. Inductively, $\mathbf{st}(o) = \varepsilon$ and $\mathbf{st}([\sigma_i]_{i \in I} \rightarrow \tau) = [\sigma_i]_{i \in I} \cdot \mathbf{st}(\tau)$.

An **abs**-rule stacks a new multiset type in the subject ($t : \tau$ gives $\lambda x.t : [\sigma_i]_{i \in I} \rightarrow \tau$) whereas an **app**-rule unstacks a multiset type ($t : [\sigma_i]_{i \in I} \rightarrow \tau$ gives $t u : \tau$).

- A thread θ is negative when its top occurrence is in an **abs**-rule and more precisely, in the source $[\sigma_i]_{i \in I}$ of the arrow type $[\sigma_i]_{i \in I} \rightarrow \tau$ of the subject $\lambda x.u$ of this **app**-rule. Let us say that $\lambda x.u$ occurs as position $b_{\lambda x}$.
- This thread θ can be consumed only when $[\sigma_i]_{i \in I}$ is the first element of the stack *e.g.*, if $t_1 : [\sigma_i]_{i \in I} \rightarrow \tau$, an **app**-rule will give $t_1 t_2 : \tau$, but if $t_1 : [\sigma'_1, \sigma'_2] \rightarrow [\sigma_i]_{i \in I} \rightarrow \tau$ ($[\sigma_i]_{i \in I}$ is not the first element of the stack), then an **app**-rule will give $t_1 t_2 : [\sigma_i]_{i \in I} \rightarrow \tau$ and $[\sigma_i]_{i \in I}$ has not been consumed.
- Thus, θ is consumed in an **app**-rule corresponding to position $b_{\textcircled{a}}$ only if
 - (c1) From $b_{\textcircled{a}}$ to $b_{\lambda x}$ included, there is exactly the same number of abstractions and of applications. . .
 - (c2) . . . and from b_{λ} to any position $b \geq b_{\textcircled{a}}$ below b_{λ} , there are more abstractions than applications (if not, $[\sigma_i]_{i \in I}$ would be consumed *above* $b_{\textcircled{a}}$).

Those remarks are exemplified by Fig. 11.4, that represents a derivation Π typing a term $t = (\lambda x_1 x_2. (\lambda x_3 x. r) u_1) u_2 u_3 s$. The variable x is typed twice in r (with types σ_1 and σ_2), so that the **abs**-rule typing $\lambda x.r$ “calls” for σ_1 and σ_2 . We have colored the *negative* threads called by λx in blue. Thus, their top occurrences are located at position $b_{\lambda} = 1^3 \cdot 0^2 \cdot 1 \cdot 0^2$ rooting $\lambda x.u$. Those *negative* threads are *left*-consumed in the **app**-rule at position $b_{\textcircled{a}} = \varepsilon$. Some multiset types without importance are denoted $[*]$. Conditions (c1) and (c2) are illustrated by the left-hand side of this figure.

Very interestingly for us, conditions (c1) and (c2) imply that the “spine” of applications and abstractions from $b_{\textcircled{a}}$ to b_{λ} can be collapsed step by step (indeed, by (c2), there is always an application below an abstraction *i.e.* there is at least one redex in the

³See for instance the interaction lemmas in Sec. 12.3.2 and the conclusion of this section, or Sec. 13.4.4.

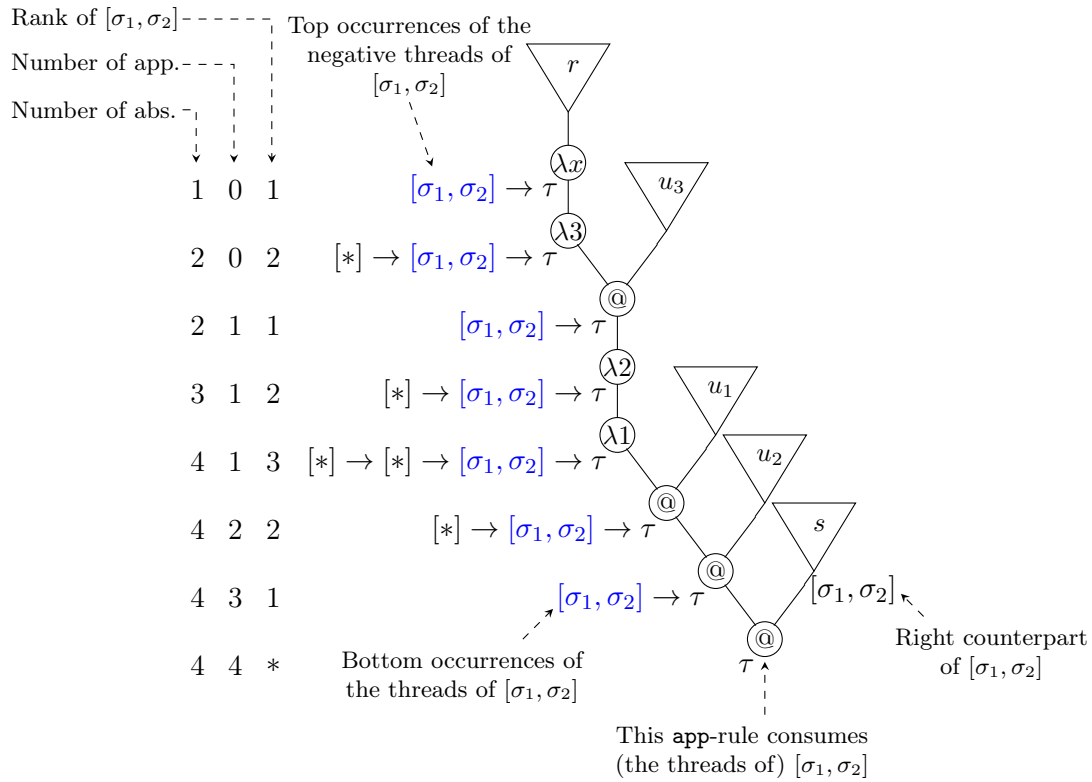


Figure 11.4: Negative Left-Consumption and Redex Towers (Informal)

“spine”). Thus, we say that there is a **redex tower** rooted at position $b_{@}$, whose **top abstraction** is at position b_{λ} (i.e. it is λx). The **function part** of the redex tower is the subterm rooted at position $b_{\lambda} \cdot 0$. The **last argument** is the argument of $b_{@}$ i.e. the subterm rooted at position $b_{@} \cdot 2$. The **height** of the redex tower is $|b_{\lambda}| - |b_{@}|$. According to the discussion above:

Observation 11.9.

- If the top occurrence of a *negative* thread is a position b_{λ} and its bottom occurrence is a position $b_{@}$, then there is a redex tower rooted at $b_{@}$ whose top abstraction is at b_{λ} .
- This negative thread can be destroyed by head reducing finitely many times the redex tower, so that no application or abstraction of the “spine” of the tower redex remains. We say then that the redex tower is collapsed.

The **collapsing strategy** on a redex tower consists in head reducing it until its spine is destroyed. For instance, in Fig. 11.4, the term t is a height 7 redex tower whose top abstraction is λx , whose function part is r and whose last argument is s . Fig. 11.5 presents the collapsing of the redex tower. The sizes of the negative threads (of $[\sigma_1, \sigma_2]$) decrease step by step till these threads are destroyed. We have set $r^1 = r[u_1/x_1]$, $u_3^1 = u_3[u_1/x_2]$, $r^2 = r^2[u_2/x_2]$, $u_3^2 = u_3^1[u_2/x_2]$, $r^3 = r^2[u_3^2/x_3]$ and $r^4 = r^3[s/x]$.

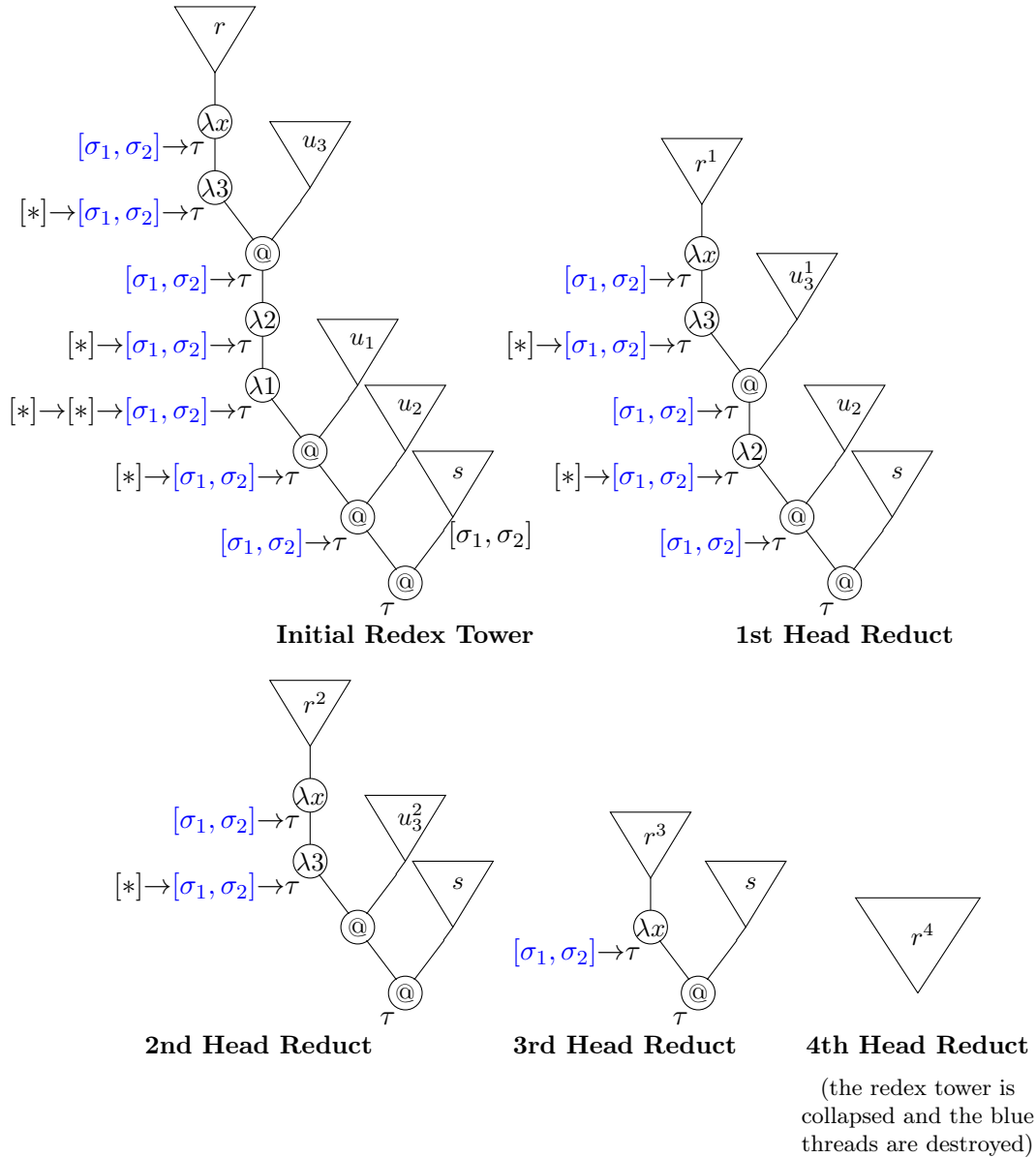


Figure 11.5: Collapsing a Redex Tower

11.2.2 Collapsing a Redex Tower Sequence

In this section, we make the following observation: if a negative thread occurs at the root of a \mathcal{R}_0 -typed term t , then this term is of order ≥ 1 (i.e. $t \rightarrow_h^* \lambda x.t'_0$ for some t'_0). Why should we care about that? This argument using syntactic polarity is the key point (embodied by Lemma 12.15) to prove Theorem 12.2 in Chapter 12, which states that if a term t is typable with a type variable in system \mathbf{S} (without approximability condition), then t is a zero term. Indeed, syntactic polarity is an handleable feature inside \mathbf{S} -derivations whereas, once again, we cannot rely upon the possibility of stabilizing \mathbf{S} -typable terms.

Let $t = (\lambda x_1 x_2. (\lambda x_3 x_4 z. u) v_3 v_4) v_1 v_2$ be a typed term, that is represented⁴ in Fig. 11.6.

⁴We also write $\lambda 1, \lambda 2, \lambda 3, \lambda 4$ instead of $\lambda x_1, \lambda x_2, \lambda x_3, \lambda x_4$.

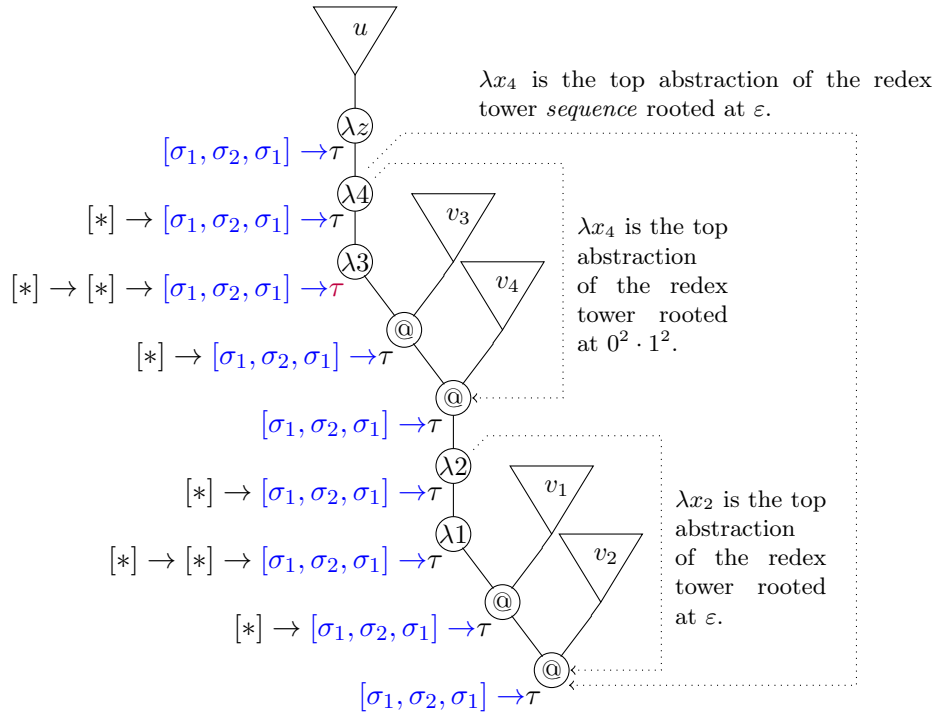


Figure 11.6: A Redex Tower Sequence whose Function Part is an Abstraction

Then t is a redex tower of height 3 and the function part of this redex tower (*i.e.* $(\lambda x_3 x_4 z.u)v_3 v_4$) is also a redex tower of height 3: indeed, $t|_{1^2.0^2} = (\lambda x_3 x_4 z.u)v_3 v_4$ and the function part of this latter redex tower is $\lambda z.u$. We say that t is a **redex tower sequence** whose *top abstraction* is λx_4 , whose *height* is 7 (because the top abstraction λx_4 of the t.r.s. occurs at depth 7) and whose *function part* is $\lambda z.u$

Following Obs. 11.3, such a sequence can be collapsed by collapsing the redex towers one after another (this corresponds to a finite segment of the head reduction strategy). Indeed, $t \rightarrow_{\mathbf{h}}^4 \lambda x.u'$ (with $u' = u[v_4/x_4][v_3/x_3][v_2/x_2][v_1/x_1]$: this meta-expression corresponds to a reduction of the “spine” of t from top to bottom instead of bottom to top).

Observation 11.10.

- In a derivation, a redex tower sequence and its function part have the same type, with the same syntactic polarity *e.g.*, the function part of the redex tower sequence t of Fig. 11.6 and its function part $\lambda x.u$ have the same type $[\sigma_1, \sigma_2, \sigma_3] \rightarrow \tau$: the syntactic polarity of τ is unknown but the sources of the arrow type both occur with a negative polarity in t and $\lambda x.u$.
- If a typed term t has an arrow type whose source has a negative syntactic polarity, then t is either an abstraction or a redex tower whose function part is an abstraction (if not, the type of t would be only positive). In both cases, the order of t is ≥ 1 .

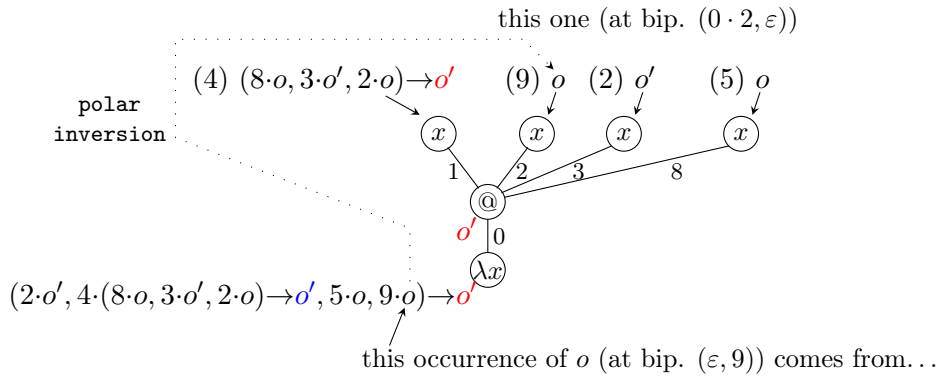


Figure 11.7: Threads and Tracking in P_{ex}

11.3 Formalizing Ascendance and Polar Inversion in System \mathbf{S}

As observed in Sec. 4.1.1, and in particular in Remark 4.1, tracking is not possible in system \mathcal{R}_0 since it could not be done in a deterministic way. This was made clear with Fig. 4.1: it is impossible to know which occurrence of o is the polar inverse of which one in the root of the typed term. Thus, the definitions of thread, ascendance and all the associated notions in this chapter are informal and can work only when two equal types do not occur in undistinguishable places. This is one of the drawbacks on non-rigid labelled tree (see Example 2.1.1 in Sec. 2.1.1).

Of course, in system \mathbf{S} , tracks give rise to bipoitions: they ensure that there is no ambiguity in pointing. We can thus formally define ascendance and polar inversion in a derivation of system \mathbf{S} .

Before formally defining ascendance and polar inversion for bipoitions in system \mathbf{S} , we need to recall that, given a type T and a sequence type $(S_k)_{k \in K}$, a position $c \in \text{supp}(T)$ corresponds to the position $1 \cdot c \in \text{supp}((S_k)_{k \in K} \rightarrow T)$, since T occurs in the target of this arrow. And if $k \in K$, the position c in S_k corresponds to position $k \cdot c$ in $(S_k)_{k \in K}$.

Example 11.1 (A Thread in P_{ex}). In Fig. 11.7 (representing derivation P_{ex} from Example 10.2, p. 214), we have represented a thread in red and blue: the three occurrences in red are positive (corresponding to bipoitions $(\varepsilon, 1)$, $(0, 1)$ and $(0 \cdot 1, 1)$ from bottom to top) and the one is negative (bipoition $(\varepsilon, 4 \cdot 1)$). Remember that a thread corresponds to the moves of a same type symbol inside a derivation (here, o') even if all the occurrences of this type symbol are not necessarily in the thread.

11.3.1 Applications and Tracking in System \mathbf{S}

Assume that, in a \mathbf{S} -derivation P typing a term t :

- The judgment $C \vdash u : (S_k)_{k \in K} \rightarrow T$ is the left premise of the judgment $C \uplus (\uplus_{k \in K} D_k) \vdash uv : T$.
- $C \uplus (\uplus_{k \in K} D_k) \vdash uv : T$ occurs at position a in P (thus, $t(a) = @$).
- c is a position in T (i.e. $c \in \text{supp}(T)$).

Then:

- The judgment $C \vdash u : (S_k)_{k \in K} \rightarrow T$ occurs at position $a \cdot 1$.
- Position $c \in \text{supp}(T)$ corresponds to position $1 \cdot c$ in $\text{supp}((S_k)_{k \in K} \rightarrow T)$.

Thus:

- T is the type of the judgment at position a and c is a position in T .
- $(S_k)_{k \in K} \rightarrow T$ is the type of judgment at position $a \cdot 1$ and $1 \cdot c$ is the corresponding position in $(S_k)_{k \in K} \rightarrow T$.

So, the ascendant of biposition (a, c) is biposition $(a \cdot 1, 1 \cdot c)$ *i.e.* we set, for all $(a, c) \in \text{bisupp}(P)$:

$$(a, c) \rightarrow_{\text{asc}} (a \cdot 1, 1 \cdot c) \text{ when } t(a) = @$$

For instance, in Example 11.1, $\Delta(0) = @$ and $(0, \varepsilon) \rightarrow_{\text{asc}} (0 \cdot 2, 1)$.

11.3.2 Abstractions and Tracking in System \mathfrak{S}

Assume that, in a \mathfrak{S} -derivation P typing a term t :

- The judgment $C \vdash \lambda x.u : (S_k)_{k \in K} \rightarrow T$ occurs at position a .
- c is a position in T .

Then:

- The judgment $C; x : (S_k)_{k \in K} \vdash u : T$ occurs at positions $a \cdot 0$.
- Position c in T corresponds to position $1 \cdot c$ in $(S_k)_{k \in K} \rightarrow T$.

Thus:

- T is the type of the judgment at position $a \cdot 0$ and c is a position in T .
- $(S_k)_{k \in K} \rightarrow T$ is the type of judgment at position a and $1 \cdot c$ is the corresponding position in $(S_k)_{k \in K} \rightarrow T$.

So, the ascendant of biposition $(a, 1 \cdot c)$ is biposition $(a \cdot 0, 1 \cdot c)$ *i.e.* we set, for all $(a, c) \in \mathbb{N}^* \times \mathbb{N}^*$ such that $(a, 1 \cdot c) \in \text{bisupp}(P)$:

$$(a, 1 \cdot c) \rightarrow_{\text{asc}} (a \cdot 0, 1 \cdot c) \text{ when } t(a) = \lambda x$$

For instance, in Example 11.1, $\Delta(\varepsilon) = \lambda x$ and $(\varepsilon, \varepsilon) \rightarrow_{\text{asc}} (0, 1)$.

Polar Inversion (Formal) Considering the same judgment $C \vdash \lambda x.u : (S_k)_{k \in K} \rightarrow T$ at position a , we now assume that $k_0 \in K$ and $c \in \text{supp}(S_{k_0})$.

By typing constraints, there is an **ax**-rule at some position⁵ a_0 concluding with $x : (k_0 \cdot S_{k_0} \vdash x : S_{k_0})$ (with the notation **pos** of Sec. 10.3.2, $a_0 = \text{pos}(a \cdot 0, x, k_0)$). Then position $c \in S_{k_0}$ corresponds to position $k_0 \cdot c$ in $(S_k)_{k \in K} \rightarrow T$. Thus:

- S_{k_0} is the type of the judgment at position a_0 and c is a position in S_{k_0} .
- $(S_k)_{k \in K} \rightarrow T$ is the type of judgment at position a and $k_0 \cdot c$ is the corresponding position in $(S_k)_{k \in K} \rightarrow T$.

So, the **polar inverse** of biposition $(a, k_0 \cdot c)$ is biposition (a_0, c) *i.e.* we set, for all $(a, c) \in \mathbb{N}^*$ and $k \geq 2$ such that $(a, k \cdot c) \in \text{bissupp}(P)$:

$$(a, k \cdot c) \rightarrow_{\text{pi}} (\text{pos}(a \cdot 0, x, k), c) \text{ when } t(a) = \lambda x$$

For instance, in Example 11.1, $\Delta(\varepsilon) = \lambda x$, $\text{pos}(0, x, 4) = 0 \cdot 1$, so that $(\varepsilon, 4 \cdot 1) \rightarrow_{\text{pi}} (0 \cdot 1, 1)$. Moreover, in Fig. 11.7, we have $(\varepsilon, 9) \rightarrow_{\text{pi}} (0 \cdot 2, \varepsilon)$ (this is represented by the dashed arrow) since $\text{pos}(0, x, 9) = 0 \cdot 2$ (the **ax**-rule typing x and using track 9 is at position $0 \cdot 2$). Note that, contrary to Fig. 4.1, the occurrences of o can be traced back to **ax**-rules.

Remark 11.2. Note that the ascendance relation \rightarrow_{asc} depends on the structure of t and the polar inversion relation on the structure of P (because polar inversion uses axiom tracks).

⁵ In Example 11.1, the blue occurrence of o' is at position $4 \cdot 1$ in $(2 \cdot o', 4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o', 5 \cdot o, 9 \cdot o) \rightarrow o'$ *i.e.* $k_0 = 4$ and $c = 1$. Then $a_0 = 0 \cdot 1$ since the **ax**-rule typing x using axiom track $k_0 = 4$ is at position $0 \cdot 1$.

Chapter 12

Complete Unsoundness: a Linearization of the λ -Calculus

Let us remind (Sec. 2.2) that a term t is usually regarded as *normalizing*, when it can be reduced to a *normal form (NF)* (*i.e.* a term that does not contain some kind of redexes), meaning that the execution of t terminates. Usually, in a simple type system (*e.g.*, Sec. 3.1.3), typability *ensures* strong normalization whereas in an ITS, it *characterizes* normalization (Chapters 3 and 5).

There are many different sets of normalizing terms (weak-n, head-n, weak head-n), but none of them contain any **mute term** (Sec. 2.3.2): we recall (Definition 2.7) that a mute term is a “persisting” redex *i.e.* t is mute if any reduct of t can be reduced to a redex. The reason why mute terms cannot be defined as normalizing is that no reduct of such a term has a stable position 2.3.1. An example of mute term is $\Omega = \Delta\Delta$ where $\Delta = \lambda x.xx$. Indeed, $\Omega \rightarrow_{\beta} \Omega \rightarrow_{\beta} \dots$ (root reduction). Mute terms are regarded as the “*most undefined λ -terms*” [12]. Therefore, if a type system is able to type a mute term, we say here that it is (**semantically**) **unsound**.

Now, what happens if we use a *coinductive* grammar¹ to generate types (roughly meaning that types may be infinite)? It is not difficult to see why this yields an unsound type system: we can build a type R_o satisfying $R_o = R_o \rightarrow o$ (where o is a type variable). Thus, tu can be typed with o when t and u have been typed R_o . We can then type Ω with o : if x is assigned R_o , then xx is typed o , so that Δ is typed $R_o \rightarrow o$ *i.e.* R_o . Thus, we can type $\Omega = \Delta\Delta$ with o . To avoid that, coinductive/recursive type or proof systems are usually endowed with a validity criterion [7, 96] or a guard condition [84], and we use such a criterion in Chapter 10 (see Sec. 10.1.3 and Definition 10.4)

Actually, coinduction allows us to build a **reflexive type** R *i.e.* the type R satisfies $R = R \rightarrow R$. With that type, in the coinductive versions of standard STS and irrelevant ITS, we can easily type every term (**complete unsoundness**). An ITS is relevant (Sec. 3.2.1 and 3.3.5) when it forbids weakening: in a relevant ITS, if $\Gamma \vdash t : A$ is derivable, then Γ only assigns types to variables that occur freely in t *e.g.*, $\lambda x.y$ will usually have a type of the form $\{ \} \rightarrow A$, where $\{ \}$ is an empty type, because x does not occur free in y and is thus untyped (see Example 3.2). The question of characterizing the set of typable terms in a relevant coinductive intersection type systems (**RICTS**) turns out to be far more difficult: typing rules constrain the empty type to occur in unforeseeable places if we do not consider a NF. But here, we already know that typability does not entail normalization or productivity (discussion p. IV). Thus, there may be a chance

¹See Sec. 9.2.3, p. 192 for a short presentation of coinductive grammar.

that some very erratic λ -terms could not be typable in an RCITS. In that case, RCITS would be able to characterize a class of regular λ -terms, bigger than the known ones. The contribution of this chapter is to prove that every term is typable in the standard RCITS \mathcal{R} that we will consider.

Types as Denotations

Let us discuss another use of typing with coinductive types. When an Intersection type system both satisfy **subject reduction** and **subject expansion** (meaning that typing is preserved under (anti)reduction, types may be seen as **invariants of execution** and thus, as suitable **denotations** for λ -terms.

As invariants of execution, types may help us to discriminate between λ -terms: if t_1 and t_2 cannot be typed with the same types, then they are not β -equivalent. Let us give a simple example with **zero terms**. A zero term is a λ -term that is *not* β -equivalent to an abstraction (a term of the form $\lambda x.u$). For instance, Ω is a zero term (it is equal to its unique reduct). In the case of finite types, by typing constraints and subject reduction, a typable *non-zero* term will necessarily be typed with an arrow type. Thus, if a term is typable with a type variable (not an arrow), we can assert that it is a zero term.

Completely unsound type systems raise the question of whether they can discriminate *pure* terms according to their **order** (the order of t is the supremal $n \in \mathbb{N} \cup \{\infty\}$ s.t. $t \rightarrow_{\beta}^* \lambda x_1 \dots \lambda x_n.t'$, Definition 2.8, p. 66). We hinted above at the fact that the zero term Ω is typable with a type variable o when using coinductive types (see also the end of Appendix A.1). More generally, it is a question of interest to know whether some completely unsound ITS is able to type *any* zero term with a type variable. Such ITS would be thus *order-discriminating*, as system \mathcal{R}_0 is, meaning that the set of types inhabited by two terms of distinct order are distinct (Sec. 3.4.5).

Contributions

The goal of this chapter is to prove that system \mathcal{R} , the coinductive version² of system \mathcal{R}_0 (Sec. 3.2.4), although seemingly more restrictive than other type systems, is also able to type any λ -term t , and that it is also order-discriminating. We present a proof for the set of *finite* λ -terms, but this can be adapted for the infinitary λ -calculus Λ^∞ ([57] and Sec. 9) when derivations are coinductively defined.

Let us discuss informally a few difficulties of this problem: naively, when xu occurs in t , we would like to assign to x a type of the form $A \rightarrow B$, where A is the type of u , and proceed by induction. However, x may be substituted in the course of a reduction sequence, and so, typing constraints on x are not easily readable. For instance, if $t = (\lambda x.xu)K$ with $K = \lambda y.x$ (so that $t \rightarrow Ku = t'$ *i.e.* x is replaced by K), then not only the type of x must be of the form $A \rightarrow B$ where A is the type of u , but also of the form $A' \rightarrow B' \rightarrow C'$ because K starts with two abstractions (since x can be replaced with K).

In the finite case, in the purpose of proving that the terms of a given set (*e.g.*, the set of HN terms) are typable, we escape this problem by typing normal forms (*e.g.*, HNF) and then proceeding by expansion. This relies upon the productivity of the involved type system. For instance, see Corollary 3.1 and the proof of Proposition 3.10 in the

²See Sec. 13.1.3 for a formal definition of system \mathcal{R} , although this is not needed to understand the system.

case of HN in system \mathcal{R}_0 , and the proofs of Proposition 10.6 and Theorem 10.4 in the case of *infinitary* weak normalization.

This method also works for order discrimination: in the purpose of proving that a typable zero term is typable with a type variable, we first type the zero head normal forms with a type variable o (see *e.g.*, Lemma 3.5 and Fig. 3.4, p. 98), then we proceed by expansion. This was also discussed in Sec. 3.4.5.

But, as noted above and in the discussion p. 236), with a coinductive type grammar, no form of normalization/productivity is ensured by typability. We must then proceed differently. We then implement the method suggested on p. 236:

- Instead of proving the complete unsoundness of system \mathcal{R} , we prove that of system \mathbf{S} (Sec. 10.2) because every \mathbf{S} -derivation collapses on a derivation. Thus, the complete unsoundness of \mathbf{S} entails that of \mathcal{D} , by collapsing sequences (Proposition 12.2). Why proceed with system \mathbf{S} instead of \mathcal{R} ? Because the rigid system \mathbf{S} features pointers called *bipositions* (Sec. 10.3.1) and the non-rigid system \mathcal{R} does not (Sec. 10.3.4).
- In system \mathbf{S} , thanks to tracks and rigidity, we can characterize (Sec. 12.2) the possible forms of \mathbf{S} -derivations (notion of **bisupport candidate**). Roughly speaking, those forms are sets of (bi)positions that must be stable under some relations. This intuition is first explained in a simpler case in Sec. 12.2.1. It is easy to see that the methodology, deeply relying upon pointers, would be impossible to apply in a non-rigid type system.
- Due to relevance, some (bi)positions must be empty. We ensure that if a *root* position cannot be reached by a constant representing emptiness under the stability relations above, then every term is typable (Corollary 12.1).
- To prove the complete unsoundness of \mathbf{S} and thus that of \mathcal{R} , we must reason about the potential *proofs* of emptiness of the root (such a proof is called a **nihilating chain**). Assuming *ad absurdum* that such a proof exists (Sec. 12.3), we reach a contradiction (see below), allowing us to conclude (Theorem 12.1).
- We explain why this result provides us with a new *non-sensible* relational model for pure λ -calculus such that two terms with different orders will have different denotations (Theorem 12.2).

The main technical difficulties lie in the fourth point: as suggested in the presentation p. 235, emptiness propagates in a non-controllable/describable way through redexes (Sec. 12.3.2). We resort then to a finite reduction strategy (the **collapsing strategy**, Sec. 12.4.2) *normalizing* potential *proofs* of emptiness.

Normal nihilating chains (proofs of emptiness) are proved *not* to exist (chains become handleable under this assumption *i.e.* when they do not interact with redexes). Since the collapsing strategy could normalize any nihilating chain (if one existed), we conclude that chains do not exist and thus, that every term is typable.

Remark 12.1.

- The second theorem has also an interesting consequence on system \mathcal{D}_w (whose complete unsoundness is almost obvious, Sec. 12.1.2), which is the coinductive version of the *irrelevant* and idempotent intersection system $\mathcal{D}_{0,w}$ (Sec. 3.3.5). It is easy to prove that every term is \mathcal{D}_w -typable, but *not* that \mathcal{D}_w is order discriminating.

- All the techniques and results in this chapter hold for the infinitary λ -calculus Λ^∞ (Sec. 9.3.1), but to lighten the proofs and statements, we only consider *finite* λ -terms.

12.1 Coinductive Type Systems

In this chapter, we explain why complete unsoundness is pretty straightforward to prove in type systems that feature a coinductive grammar, provided they allow weakening *i.e.* they are irrelevant.

12.1.1 A Coinductive Simple Type System

We give here a first proof of complete unsoundness for **Curry**, the version of Curry's system **Curry**₀ (Sec. 3.1.3) with a coinductive type grammar, that we present now. We consider the set of *simple types* generated by the following *coinductive* grammar:

$$A, B ::= o \in \mathcal{O} \parallel A \rightarrow B$$

As in the inductive case, the *context* (metavariables Γ, Δ) is a partial function from the set of variables \mathcal{V} to the set of simple types. If for all $x \in \text{dom}(\Gamma) \cap \text{dom}(\Delta)$, $\Gamma(x) = \Delta(x)$, we write $\Gamma :: \Delta$ for the context of domain $\text{dom}(\Gamma) \cup \text{dom}(\Delta)$ extending Γ and Δ . If $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \{\}$, we may write $\Gamma; \Delta$ instead of $\Gamma :: \Delta$.

The set of typing derivations of **Curry** is defined *inductively* by the following rules:

$$\frac{}{\Gamma; x : A \vdash x : A} \text{ax} \quad \frac{\Gamma; x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \rightarrow B} \text{abs}$$

$$\frac{\Gamma \vdash t : A \rightarrow B \quad \Delta \vdash u : A}{\Gamma :: \Delta \vdash tu : B} \text{app}$$

Notice that the **app**-rule can also be applied only when $\Gamma :: \Delta$ is defined. The regular, *inductive* version of this system (*i.e.* system **Curry**₀) is known to ensure strong normalization (Theorem 3.1). In sharp contrast:

Proposition 12.1. System **Curry** is **completely unsound**: if t is a term, then there exists a **Curry**-derivation typing t .

Proof. Let R be the simple type *coinductively* defined by $R = R \rightarrow R$. Then a straightforward induction on the structure of t shows that $\Gamma \vdash t : R$ is derivable for any context Γ s.t. $\text{fv}(t) \subseteq \text{dom}(\Gamma)$ and $\Gamma(x) = R$ for all $x \in \text{dom}(\Gamma)$ (see below).

$$\frac{}{x : R \vdash x : R} \quad \frac{\Gamma; x : R \vdash t : R}{\Gamma \vdash \lambda x.t : R} \quad \frac{\Gamma \vdash t : R \quad \Delta \vdash u : R}{\Gamma :: \Delta \vdash tu : R}$$

□

12.1.2 Complete Unsoundness and Relevance

Let us now understand the difference between relevant and irrelevant intersection type systems makes it very easy to prove complete unsoundness in the latter case whereas there is no obvious method in the former one.

We present the coinductive versions of system \mathcal{D}_0 and $\mathcal{D}_{0,w}$ (Sec. 3.2.3 and 3.3.5). The types of system \mathcal{D} and system \mathcal{D}_w are defined *coinductively*:

$$B, A_i ::= o \parallel \{A_i\}_{i \in I} \rightarrow B$$

We also call $\{A_i\}_{i \in I}$ a *set type* and it is assumed that I is countable. As in Sec. 3.2.3, the set types represent intersection and the countable intersection operator \wedge is the set-theoretic union: $\wedge_{j \in J} \{A_i^j\}_{i \in I(j)} := \cup_{j \in J} \{A_i^j\}_{i \in I(j)}$. Note that, in contrast, to systems \mathcal{D}_0 and $\mathcal{D}_{0,w}$, set types can have an infinite cardinality in system \mathcal{R} .

The definitions of contexts and associated notation etc are the same as in the inductive case (we remind that context are *total* functions). The typing derivations of \mathcal{D} and \mathcal{D}_w are defined *inductively* by the same rules as systems \mathcal{D}_0 and $\mathcal{D}_{0,w}$ except that they involve coinductive types. Typing derivations of \mathcal{D} and \mathcal{D}_w may be infinite because an **app**-rule may have an infinite number of arguments, although they are of a finite height. Implicitly, in the **abs**-rule, if x is not in the context domain, then $\lambda x.t$ is typed with $\{\} \rightarrow B$, as in system \mathcal{D}_0 (Example 3.2, p. 85). Likewise, we define \mathcal{R} , the version of \mathcal{R}_0 (Sec. 3.2.4) featuring a coinductive type grammar (also allowing multisets of countable cardinality). Note that systems \mathcal{R} and $\mathcal{R}_{0,w}$ satisfy subject reduction and expansion.

As systems \mathcal{D}_0 and \mathcal{R}_0 , systems \mathcal{D} and \mathcal{R} are relevant because no weakening is allowed: roughly speaking, if $\Gamma \vdash t : B$ is derivable and $x \in \text{fv}(t)$, then $\Gamma(x)$ is the intersection of types of the free occurrences of x seen as a *subterm* of t *i.e.*, as already observed (Sec. 3.2.1), relevance ensures some *resource-awareness*. Recall from Examples 3.2 and 3.3, p. 85 and 87, that the type of $\lambda x.x$ (resp. $\lambda y.x$) *must* be of the form $\{B\} \rightarrow B$ in \mathcal{D}_0 and $[\tau] \rightarrow \tau$ in \mathcal{R}_0 (resp. $\{\} \rightarrow B$ in \mathcal{D}_0 and $[\] \rightarrow [\tau]$ in \mathcal{R}_0): there are no other ways to type them in systems \mathcal{D} and \mathcal{R} , whereas, in system \mathcal{D}_w , $\lambda x.x$ can be typed with *e.g.*, $\{A, B, C\} \rightarrow B$ and $\lambda y.x$ with *e.g.*, $\{B\} \rightarrow B$ (Example 3.4, p. 95).

System \mathcal{D}_w is irrelevant, so that its complete unsoundness is easy to prove: the proof for the simple type system above can be straightforwardly adapted for \mathcal{D}_w , so that every term is easily typable in \mathcal{D}_w , this time with R coinductively defined by $R = \{R\} \rightarrow R$ in \mathcal{D} .

But the induction fails in the **abs**-case with \mathcal{D} or \mathcal{R} : by relevance, when $x \notin \text{fv}(t)$, we have $\lambda x.t : \{\} \rightarrow R$ (resp. $\lambda x.t : [\] \rightarrow R$) and not $\lambda x.t : \{R\} \rightarrow R$ (*i.e.* R) (resp. $\lambda x.t : [R]_\omega \rightarrow R$). Likewise, in \mathcal{R}_w , the *irrelevant* version of \mathcal{R} , the reflexive type R would be defined by $R = [R]_\omega \rightarrow R$. But the induction would also fail in the **abs**-case *e.g.*, when $x \notin \text{fv}(t)$, the type of $\lambda x.t$ would be $[\] \rightarrow R$, which is distinct from R .

Thus, there is not easy argument to ensuring complete unsoundness in \mathcal{D} or \mathcal{D} . By lack of productivity (discussion p. 236), since Ω is typable both in \mathcal{D} and \mathcal{R} , the standard methods of intersection types will not work. But we will only study the complete unsoundness of system \mathcal{R} , since, as its finite counterpart \mathcal{D}_0 , system \mathcal{D} does not satisfy subject expansion (Sec. 3.3.4).

Remark 12.2. Systems \mathcal{D} , \mathcal{D}_w and \mathcal{R} should be more rigorously defined (although this does not much matter in this chapter): this is done only for system \mathcal{R} in Sec. 13.1.3.

From now on, we prove that every term is typable in the *relevant* intersection type system \mathcal{R} , satisfying both subject reduction and subject expansion.

12.1.3 Typing some Notable Terms in System \mathcal{R}

We use system \mathcal{R} to type a few terms from Sec. 2.1.4 satisfying fixpoint equations. Some of them are not head normalizing. Remember that $\Delta_f = \lambda x.f(xx)$, $Y = \lambda f.\Delta_f \Delta_f$ (Y

is *Curry fixpoint combinator*). Moreover, if $I = \lambda x.x$ and $K = \lambda xy.x$, then $YI \rightarrow \Omega$ (satisfying $\Omega \rightarrow_{\beta} \Omega$), $Yf \rightarrow Y_f := \Delta_f \Delta_f$ (satisfying $Y_f \rightarrow_{\beta} f(Y_f)$) and $YK \rightarrow_{\beta} Y_{\lambda} := (\lambda x.\lambda y.xx)\lambda x.\lambda y.xx$ (satisfying $Y_{\lambda} \rightarrow_{\beta} \lambda y.Y_{\lambda}$). Let us recall Definition 2.8 from Sec. 2.3.2:

Definition. Let t be a λ -term. The **order** of t is $\sup\{n \in \mathbb{N} \mid \exists x_1, \dots, x_n, u \text{ s.t. } t \rightarrow_{\beta}^* \lambda x_1 \dots \lambda x_n.u\}$.

Iterating reduction on Y_f and Y_{λ} infinitely many times, we see that Y_f (resp. Y_{λ}) *strongly converges* to the infinitary term $f^{\omega} := f(f(\dots))$ (resp. $\lambda y.\lambda y.\dots$) w.r.t. Λ^{∞} (Sec. 9.3.1). Thus, Ω and Y_f are both *zero terms* (terms of order 0) and Y_{λ} a term of infinite order, as noted in Sec. 2.3.2. The term Ω is mute and Y_f is a hereditary head normalizing term (Definition 9.5, p. 200).

By Definition 3.3, the *order of a type* τ is the number of top-level arrows in τ . When R is the reflexive type satisfying $R = [R]_{\omega} \rightarrow R$, by unfolding the right occurrence of R n times, we have $R = \underbrace{[R]_{\omega} \rightarrow \dots [R]_{\omega}}_n \rightarrow R$ for any n , so R has an infinite order.

Because of rule **abs** and subject reduction (that is satisfied in \mathcal{R}), a term of order n may only be typed with a type of order $\geq n$, as explained in Sec. 3.4.5. Since the reflexive type is of infinite order, it does not give any information about the order of the term it types. However, some \mathcal{R} -derivations can capture more precisely the order of terms. For all \mathcal{R} -type τ , we define coinductively R_{τ} by $R_{\tau} = [R_{\tau}]_{\omega} \rightarrow \tau$. For instance, we consider the following typing of Y (omitting left-hand sides of **ax**-rules):

$$\Pi_{\Delta_f} = \frac{\frac{\frac{f : [\tau] \rightarrow \tau}{f : [[\tau] \rightarrow \tau]; x : [R_{\tau}]_{\omega} \vdash f(xx) : \tau} \quad \frac{x : R_{\tau} \quad (x : R_{\tau})_{\omega}}{x : [R_{\tau}]_{\omega} \vdash xx : \tau}}{f : [[\tau] \rightarrow \tau] \vdash \Delta_f : R_{\tau} (= [R_{\tau}]_{\omega} \rightarrow \tau)}}{\Pi_{\Delta_f} \triangleright f : [[\tau] \rightarrow \tau] \vdash \Delta_f : [R_{\tau}]_{\omega} \rightarrow \tau \quad (\Pi_{\Delta_f} \triangleright f : [[\tau] \rightarrow \tau] \vdash \Delta_f : R_{\tau})_{\omega}}$$

$$\Pi_Y = \frac{f : [[\tau] \rightarrow \tau]_{\omega} \vdash \Delta_f \Delta_f : \tau}{\vdash Y : [[\tau] \rightarrow \tau]_{\omega} \rightarrow \tau}$$

Thus, in system \mathcal{R} , Y is typable with $[[\tau] \rightarrow \tau]_{\omega} \rightarrow \tau$ for *any* type τ . Notice that we also have derived $f : [[\tau] \rightarrow \tau]_{\omega} \vdash Y_f : \tau$ for any \mathcal{R} -type τ .

Using suitable instances or variants of Π_Y , we can build $\Pi_{\Omega} \triangleright \vdash \Omega : \tau$ (for any τ) and $\Pi_{\lambda} \triangleright \vdash Y_{\lambda} : [] \rightarrow [] \rightarrow \dots$

By instantiating τ with a type variable o , we get $\vdash \Omega : o$ and $\vdash Y_f : o$. Thus, the zero terms Ω and Y_f are typed³ with types of order 0 whereas Y_{λ} (whose order is infinite) is typed with a type of infinite order, as it was constrained to be.

We will generalize this result (not only for terms built from fixpoint combinators like Ω or $\lambda x.\Omega$) and show that, for all *pure* terms t of order n , there is a \mathcal{R} -derivation (or a \mathcal{D}_w -derivation) typing t with a type of order n (Theorem 12.2).

12.1.4 Type System S (Sequential Intersection)

In this section, we explain why the complete unsoundness of system \mathcal{R} can be proved by using system **S** and why it is actually a better idea to proceed with this latter system.

³ Note that $Y_f \rightarrow f(Y_f)$ (Y_f is HN) and Y_f is typable with o in the finite system \mathcal{R}_0 .

Indeed, we want to have a good control on where types are created and how argument branches occur in an **app**-rule. For that, we use system **S** (Sec. 10.2), in which intersection is represented by means of *sequences* and *bipositions* provide pointers inside derivations. See Sec. 10.3.1 and 10.3.2 for examples and useful associated notions, including **axiom tracks** and the notation $\mathbf{tr}^P(a)$

In this chapter, we only need to consider *right* bipositions (not the left ones): by relevance, every context in a **S**-derivation P is determined by $\mathbf{supp}(P)$, the types given in **ax**-rules and the **ax**-tracks that those **ax**-rules use. Indeed, since t is finite, we have $\mathcal{C}^P(a)(x) = \uplus_{a_0 \in \mathbf{Ax}_a^P(x)} (\mathbf{tr}^P(a_0) \cdot \mathbf{T}^P(a_0))$. This indicates that in a **S**-derivation, contexts and types can be computed from the support $\mathbf{supp}(P)$ and the types created in axiom rules. We then change the denotation of notation $\mathbf{bisupp}(P)$:

Notation 12.1. Let P be a **S**-derivation. In this chapter, we denote by $\mathbf{bisupp}(P)$ the set of right bipositions in P .

In this chapter, we just say “biposition” instead of “right biposition”. and some new notations are useful to handle **S**-derivations: assume that P types t . We set $A = \mathbf{supp}(P)$ and $B = \mathbf{bisupp}(P)$. If $x \in \mathcal{V}$, $a \in A$, we set $\mathbf{Ax}_a^P(x) = \{a_0 \in A \mid a \leq a_0, t(a) = x, \nexists a'_0, a \leq a'_0 \leq a_0, t(a'_0) = \lambda x\}$ (occurrences of x in P above a , that are not bound w.r.t. a). To adapt the results of this chapter to Λ^∞ , we just need to consider *quantitative* derivations (Sec. 10.3.2) (so that left bipositions are not needed either) and we prove that every term $t \in \Lambda^\infty$ is typable by means of a quantitative derivation.

We define coinductively a *collapse* π from the set of types of **S** to the set of types of \mathcal{R} by $\pi(o) = o$ and $\pi((S_k)_{k \in K} \rightarrow T) = [\pi(S_k)]_{k \in K} \rightarrow \pi(T)$. This collapse can be straightforwardly extended to a collapse from the set of derivations of **S** to the set of derivations of \mathcal{R} , noticing that $(S_k)_{k \in K} = (S'_k)_{k \in K'}$ implies $\pi((S_k)_{k \in K}) = \pi((S'_k)_{k \in K'})$. For instance, the **S**-derivation $P_{\mathbf{ex}}$ (Example 10.2, p. 214) collapses on the \mathcal{R} -derivation $\Pi_{\mathbf{ex}}$ p. 105, as already observed, where:

$$P_{\mathbf{ex}} = \frac{\frac{x : (4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o')}{x : (2 \cdot o', 4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o', 5 \cdot o, 9 \cdot o) \vdash xx : o'} \quad \frac{x : (9 \cdot o) \text{ [2]} \quad x : (2 \cdot o') \text{ [3]} \quad x : (5 \cdot o) \text{ [8]}}{x : (2 \cdot o', 4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o', 5 \cdot o, 9 \cdot o) \vdash xx : o'}}{\vdash \lambda x.xx : (2 \cdot o', 4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o', 5 \cdot o, 9 \cdot o) \rightarrow o'}$$

$$\Pi_{\mathbf{ex}} = \frac{\frac{x : [o, o', o] \rightarrow o' \quad x : [o] \quad x : [o'] \quad x : [o]}{x : [o', [o, o', o] \rightarrow o', o, o] \vdash xx : o'}}{\vdash \lambda x.xx : [o', [o, o', o] \rightarrow o', o, o] \rightarrow o'}$$

As a consequence of the collapse:

Proposition 12.2. If a term t is **S**-typable, then it is \mathcal{R} -typable.

Thus, if every term is typable in **S** (the proof of which takes the remainder of this chapter), then every term is typable in \mathcal{R} .

⁴ To adapt the results of this chapter to Λ^∞ , we just need to consider *quantitative* derivations (Sec. 10.3.2) (so that left bipositions are not needed either) and we prove that every term $t \in \Lambda^\infty$ is typable by means of a quantitative derivation.

In Chapter 10, we restricted the set of \mathbf{S} -types (*i.e.* we replaced \mathbf{Typ}^{111} by \mathbf{Typ}^{001} in Sec. 10.2.2) by allowing only finite sequences of arrows, since the normal forms of Λ^{001} feature finite series of abstractions. In this chapter, we consider the set \mathbf{Typ}^{111} with no restriction, since we need to type terms whose order is infinite (*e.g.*, Y_λ in Sec. 12.1.3).

12.2 Bisupport Candidates

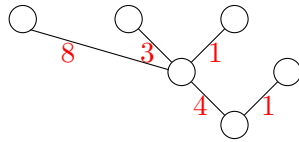
In this section, we characterize, for a given term t , the *bisupport candidate* *i.e.* the (potential) forms of a derivation typing t . By “form”, we intuitively mean a set of unlabelled positions (that must be stable under some suitable relations). We make explicit that idea by studying first the possible forms of a \mathbf{S} -type in Sec. 12.2.1. The notion of unlabelled position has a meaning only because tracks of \mathbf{S} allow us to define suitable pointers. This would be impossible in system \mathcal{R} .

12.2.1 A Toy Example: Support Candidates for Types

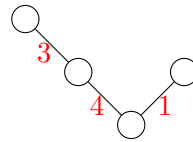
In this section, we explain how the notion of “form” of a support can be formalized by giving a characterization of the supports of \mathbf{S} -types in terms of stability conditions.

The definition of a particular \mathbf{S} -type T can be understood as a two-step process: first, we choose the support $C := \mathbf{supp}(T)$, next, we choose the type labels $T(c)$ (in the signature $\mathcal{O} \cup \{\rightarrow\}$) given to the positions $c \in C$. However, not all the subsets $C \subseteq \mathbb{N}^*$ are fit to be the support of a type, and not all the possible decorations of a suitable set C yield a correct type.

For instance, let us consider the two sets of positions C_1 and C_2 below. Do they define the supports of some types T_1 and T_2 ?

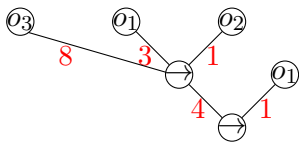


$$C_1 = \{\varepsilon, 1, 4, 4 \cdot 1, 4 \cdot 3, 4 \cdot 8\}$$

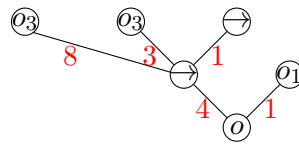


$$C_2 = \{\varepsilon, 1, 4, 4 \cdot 3\}$$

As it turns out, C_1 is the support of a type *e.g.*, $(4 \cdot (8 \cdot o_3, 3 \cdot o_1) \rightarrow o_2) \rightarrow o_1$ (figure below). By contrast, no type T may satisfy $\mathbf{supp}(T) = C_2$, because a non-terminal node of a type (necessarily an arrow) should have a child on track 1 (by convention, its right-hand side), but $4 \in C_2$ and $4 \cdot 1 \notin C_2$.



Type $(4 \cdot (8 \cdot o_3, 3 \cdot o_1) \rightarrow o_2) \rightarrow o_1$



Wrong decoration

This motivates the following notion: a **support candidate (s-candidate)** of type is a subset $C \subseteq \mathbb{N}^*$ such that there exists a type T satisfying $C = \mathbf{supp}(T)$. Given an s-candidate C , it is easy to define a correct type whose support is C :

- The non-terminal nodes of C should be decorated with arrows and ...
- ... the leaves of C should be decorated with type variables.

So was done for the decoration on the left-hand side, representing the type $(4 \cdot (8 \cdot o_3, 3 \cdot o_1) \rightarrow o_2) \rightarrow o_1$. As a counterexample, the decoration on the right-hand side is incorrect: ε (non-terminal) is labelled with $o \in \mathcal{O}$ and $4 \cdot 1$ (leaf) with \rightarrow .

The observations about C_1 and C_2 above suggest considering two relations \rightarrow_{t_1} and \rightarrow_{t_2} defined by:

- For all $c \in \mathbb{N}^*$, $k \in \mathbb{N}$, $c \cdot k \rightarrow_{t_1} c$.
- For all $c \in \mathbb{N}^*$, $k \geq 2$, $c \cdot k \rightarrow_{t_2} c \cdot 1$.

A set of positions C is closed under \rightarrow_{t_1} (*i.e.* $c_1 \in C$ and $c_1 \rightarrow_{t_1} c_2$ implies $c_2 \in C$) iff it is a tree. Stability under condition \rightarrow_{t_2} means that if a node c is not terminal, then it has a child on track 1. We have:

Lemma 12.1. Let $C \subseteq \mathbb{N}^*$. Then C is a type support candidate (*i.e.* there exists a type T s.t. $C = \text{supp}(T)$) iff C is non-empty and is closed under \rightarrow_{t_1} and \rightarrow_{t_2} .

Thus, relations \rightarrow_{t_1} and \rightarrow_{t_2} are enough to characterize s-candidates. We call them **stability relations** *e.g.*, C_1 is stable under \rightarrow_{t_1} and \rightarrow_{t_2} , whereas $4 \cdot 3 \in C_2$, $4 \cdot 3 \rightarrow_{t_2} 4 \cdot 1$ but $4 \cdot 1 \notin C_2$, so that C_2 is not stable under \rightarrow_{t_2} (this example means that 4 has no child on track 1 whereas it is not terminal and thus cannot be decorated by an arrow nor by a type variable).

When $c_1 \rightarrow_{t_1} c_2$ or $c_1 \rightarrow_{t_2} c_2$, we say that c_1 **subjugates** c_2 , because c_1 demands c_2 to ensure a correct formation of the support.

12.2.2 Toward the Characterization of Bisupport Candidates

In Sec. 12.2.2, we give some basic concepts and notations in the purpose of characterizing the “form” of S-derivations, as suggested in the introduction of Sec. 12.2,

The form of a derivation typing a term t depends on the structure of t , but also on a suitable choice of axiom tracks (Sec. 10.2) in the **ax**-rules. We explain here how it is done, as well as how we intend to capture relevance-related emptiness.

We want to prove that every term t is S-typable *i.e.* typable by a derivation P of system **S**. By analogy with the notion of candidate supports for types (previous section), the idea is to characterize the **bisupport candidate (b-candidate)** for the derivations typing a given term t *i.e.* sets $B \subseteq \mathbb{N}^* \times \mathbb{N}^*$ s.t. there exists a derivation P typing t satisfying $B = \text{bisupp}(P)$ (such a characterization is eventually given by Proposition 12.3).

This characterization requires that one captures first the way emptiness is constrained to occur in relevant derivations. Then, we will have to ensure that emptiness does not compromise typability *i.e.* emptiness must not propagate everywhere in the derivations typing a given term t . If it did, a derivation typing t would be empty (*i.e.* t would not be typable) and we want to show that this does not happen, in the purpose of proving that every term is typable in **S**.

Before going on with the characterization of b-candidates, we recall that, in the purpose of defining a derivation typing a term t , every time we use an axiom rule, we must choose an *axiom track* such that no conflict occurs (axiom tracks may be “called” by **abs**-rule). For that, it is enough to *arbitrarily* fix an injective function $[\cdot] : \mathbb{N}^* \rightarrow \mathbb{N} \setminus \{0, 1\}$, whose inverse function is written **pos** (injectivity ensures that there is no track conflict). Thus, let t be a term: we want to prove that there exists an S-derivation P typing t s.t., if

$a \in \text{supp}(P)$ points to an axiom rule, then $\text{tr}^P(a) = [a]$ (thus, the value of $[a]$ matters only for axioms). Actually, the notion of bisupport candidate (for a derivation typing t) depends on $[\cdot]$: we define a $[\cdot]$ -bisupport candidate as a $\emptyset \neq B \subseteq \mathbb{N}^* \times \mathbb{N}^*$ such that there exists a $[\cdot]$ -derivation P typing t s.t. $B = \text{bisupp}(P)$ (a $[\cdot]$ -derivation is a derivation P s.t. $\text{tr}^P(a) = [a]$ for all axiom rule a of P). No matter $[\cdot]$, we will show that there exists a $[\cdot]$ -bisupport candidate and thus, a $[\cdot]$ -derivation typing t (implying that t is typable).

We notice now that not every position $a \in \mathbb{N}^*$ (or biposition $(a, c) \in \mathbb{N}^* \times \mathbb{N}^*$) may be in a derivation typing a given term t . For instance, we have $\text{supp}(\lambda x.xx) = \{\varepsilon, 0, 0 \cdot 1, 0 \cdot 2\}$, so, if P types $\lambda x.xx$, then $a \in \text{supp}(P)$ implies $\bar{a} = \varepsilon, 0, 0 \cdot 1$ or $0 \cdot 2$ i.e. $\text{supp}(P) \subseteq \{\varepsilon, 0, 0 \cdot 1, 0 \cdot 2\}$. For instance, $\text{supp}(P_{\text{ex}}) = \{\varepsilon, 0, 0 \cdot 1, 0 \cdot 2, 0 \cdot 3, 0 \cdot 8\}$ (Sec. 10.2) .

Let t be a term. More generally, we set $\mathbb{A}^t = \{a \in \mathbb{N}^* \mid \bar{a} \in \text{supp}(t)\}$ and $\mathbb{B}^t = (\mathbb{A}^t \times \mathbb{N}^*) \cup \{\mathbf{p}_\perp\}$ (where \mathbf{p}_\perp is an ‘‘empty biposition’’ constant), so that, if P is a derivation typing t , then a position (resp. a biposition) of P must be in \mathbb{A}^t (resp. in $\mathbb{B}^t \setminus \{\mathbf{p}_\perp\}$) i.e. $\text{supp}(t) \subseteq \mathbb{A}^t$ and $\text{bisupp}(P) \subset \mathbb{B}^t \setminus \{\mathbf{p}_\perp\}$. We will later define a relation \rightarrow_\bullet , using the ‘‘constant of emptiness’’ \mathbf{p}_\perp , such that $\mathbf{p} \rightarrow_\bullet \mathbf{p}_\perp$ indicates that \mathbf{p} cannot be in $\text{supp}(P)$. Thus, we will describe how emptiness propagates inside the bisupport candidates.

Not every $a \in \mathbb{A}^t$ may be in a derivation P typing t e.g., $2 \in \text{supp}(u)$ with $u = (\lambda x.y)z$, but if P types u , then subterm z is left untyped – $\lambda x.y$ must be typed with $() \rightarrow T$ (relevance) – and 2 cannot be in $\text{supp}(P)$.

We drop P and t from most notations, in which they are implicit now. We set $\mathbb{A}_a(x) = \{a_0 \in \mathbb{A} \mid a \leq a_0, t(a_0) = x, \nexists a'_0, a \leq a'_0 < a_0, t(a'_0) = \lambda x\}$. Thus, if P is a $[\cdot]$ -derivation, then, with the notation Ax_a^P from Sec. 12.1.4, $\text{Ax}_a^P(x) \subset \mathbb{A}_a(x)$ for all $a \in \text{supp}(P)$, $x \in \mathcal{V}$ and $\mathbb{A}_a(x)$ may be considered as the set of position candidates for ax-rules typing the free occurrences of x above a .

If $t(a) = \lambda x$, we set $\text{Tr}_\lambda(a) = \{[a_0] \mid a_0 \in \mathbb{A}_{a \cdot 0}(x)\}$ (see previous section for the choice of $[\cdot]$): $\text{Tr}_\lambda(a)$ is the set of axiom tracks dedicated to x above the **abs**-rule at position a .

Notice the following fact while assuming:

- $t|_b = \lambda x.u$ (with $b \in \text{supp}(t)$) and $t|_{b'} \neq x$ with $b' \in \text{supp}(t), b' \geq b$.
- P is a $[\cdot]$ -derivation typing t .

For any $a \in \text{supp}(P)$ s.t. $\bar{a} = b$ (and thus, $t|_a = \lambda x.u$), $t|_a$ will be typed with a type of the form $(S_k)_{k \in K} \rightarrow T$, but K may not contain any $[a_0]$ when $\bar{a}_0 \geq b'$ since no such a_0 (whose dedicated axiom track is $[a_0]$) is the position of an **ax**-rule typing x . In short, when a variable x is not at some places in t , we already now that emptiness should ‘‘occur’’ at some particular tracks if we perform an abstraction λx . This give us more fine-grained information about occurrences of emptiness in a derivation typing t than the case where $\lambda x.u : () \rightarrow T$ because x does not occur free in u : system **S** should be able to provide us information about emptiness *track by track*.

12.2.3 Tracking a Type in a Derivation

Let us now try to express the stability conditions (as in Sec. 12.2.1) that a $[\cdot]$ -bisupport candidate for a derivation typing t should satisfy. We will need to ensure six points:

- Identification of the components (*i.e.* the bipositions) of a same type T in a derivation from bottom to top (see Fig. 12.1): relation of **ascendance** \rightarrow_{asc} .
- Identification of the components of type given in an **ax**-rule to a variable x (S_5 in Fig. 12.1) and its occurrence called by the abstraction λx : relation of **polar inversion** \rightarrow_{pi} .
- Identification of the matching components of the types of u and v in the **app**-rule typing uv (types S_k in the **app**-rule of Fig. 12.1): relation of **consumption** \rightarrow .
- Correct type formation, as in Sec. 12.2.1: extensions of relations \rightarrow_{t1} and \rightarrow_{t2} .
- The type of a subterm of the form $\lambda x.u$ is an arrow type (and not a type variable): relation \rightarrow_{abs} .
- For technical reasons, we also need a "big-step" stability condition, meaning that the support of a derivation is a tree: relation $\rightarrow_{\text{down}}$.

The heuristics of ascendance, consumption, polar inversion (and threads) are informally presented in Sec. 11.1.

We consider a fixed term t and an injection $[\cdot] : \mathbb{N}^* \rightarrow \mathbb{N} \setminus \{0, 1\}$. We formalize these ideas in the following sections. For the discussion below, let us recall again that, given a type T and a sequence type $(S_k)_{k \in K}$, a position $c \in \text{supp}(T)$ corresponds to the position $1 \cdot c \in \text{supp}((S_k)_{k \in K} \rightarrow T)$, since T occurs in this arrow type right-hand side. And if $k \in K$, position c in S_k corresponds to position $k \cdot c$ in $(S_k)_{k \in K}$. In Fig. 12.1, we indicate once more the position of a judgment between angle brackets *e.g.*, $C; x : (S_k)_{k \in K} \vdash t : T \langle a \cdot 0 \rangle$ means that judgment $C; x : (S_k)_{k \in K} \vdash t : T$ is at position $a \cdot 0$.

Abstraction rule

$$\frac{x : (5 \cdot S_5) \vdash x : S_5 \langle \text{pos}(5) \rangle^{\text{ax}}}{C; x : (S_k)_{k \in K} \vdash t : T \langle a \cdot 0 \rangle} \text{abs} \quad (\text{with } 5 \in K)$$

$$C \vdash \lambda x.t : (S_k)_{k \in K} \rightarrow T \langle a \rangle$$

Application rule

$$\frac{C \vdash t : (S_k)_{k \in K} \rightarrow T \langle a \cdot 1 \rangle \quad (D_k \vdash u : S_k \langle a \cdot k \rangle)_{k \in K}}{C \uplus (\uplus_{k \in K} D_k) \vdash tu : T \langle a \rangle} \text{app}$$

Figure 12.1: Ascendance, Polar Inversion and Consumption

- Assume that, in a $[\cdot]$ -derivation P , we find an **abs**-rule at position a as in the figure above: the judgment $C; x : (S_k)_{k \in K} \vdash t : T$ (pos. $a \cdot 0$) yields $C \vdash \lambda x.t : (S_k)_{k \in K} \rightarrow T$ below (pos. a). The occurrence of T in the conclusion of the rule is intuitively the same as that in its premise: we say the former is the **ascendant** of the latter, since it occurs above in the typing derivation. Likewise, in the **app**-rule, the occurrence of T in

$C \uplus_{k \in K} D_k \vdash t u : T$ stems from that of premise $C \vdash t : (S_k)_{k \in K} \rightarrow T$: the first occurrence of T is also the ascendant of T in the conclusion of the rule.

Using the correspondence between c and $1 \cdot c$ above, we defined in Sec. 11.3 the relation of ascendance for all $(a, c) \in \mathbb{B}^t$ by:

- $(a, c) \rightarrow_{\text{asc}} (a \cdot 1, 1 \cdot c)$ if $t(a) = @$.
- $(a, 1 \cdot c) \rightarrow_{\text{asc}} (a \cdot 0, c)$ if $t(a) = \lambda x$.

Relation $\mathbf{p}_1 \rightarrow_{\text{asc}} \mathbf{p}_2$ means that \mathbf{p}_2 is the ascendant of \mathbf{p}_1 *i.e.* \mathbf{p}_1 and \mathbf{p}_2 are corresponding pointers to the same type symbol in the conclusion and the (left) premise of the rule at some position a . For instance (*cf.* p. 259), in P_{ex} , $(\varepsilon, \varepsilon) \rightarrow_{\text{asc}} (0, 1) \rightarrow_{\text{asc}} (0 \cdot 1, \varepsilon)$: those 3 bipoositions point to type symbol o' , from the judgment concluding P_{ex} to the axiom rule where it was created (position $0 \cdot 1$). The three occurrences of this ascendant thread are represented in red in Fig. 12.2 (note that two axiom rules of P_{ex} could not fit in the figure).

$$\frac{\frac{x : (4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o') \vdash x : (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o' \quad x : (9 \cdot o) \vdash x : o \quad [2] \quad \dots\dots\dots}{x : (2 \cdot o', 4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o', 5 \cdot o, 9 \cdot o) \vdash x x : o'}}{\vdash \lambda x. x x : (2 \cdot o', 4 \cdot (8 \cdot o, 3 \cdot o', 2 \cdot o) \rightarrow o', 5 \cdot o, 9 \cdot o) \rightarrow o'}$$

Figure 12.2: A Thread in P_{ex}

- Let us have another look at the **abs**-rule at position a (see Fig. 12.1). Assume $5 \in K$. Then the occurrence of S_5 in $(S_k)_{k \in K} \rightarrow T$ stems from an axiom rule concluding with $x : (5 \cdot S_5) \vdash x : S_5$: we say that the occurrence S_5 (in $(S_k)_{k \in K} \rightarrow T$) is the **polar inverse** of the occurrence of S_5 in the axiom rule. Assume on the contrary that $8 \notin K$. So S_8 does not exist and there is no axiom rule typing x and using axiom track 8 above a . Morally, S_8 is empty.

We recall that, if $t(a) = \lambda x$, then $K \subset \text{Tr}_\lambda(a)$ and if $k \in K$ (*e.g.*, $k = 5$ above), the axiom rule typing x above a and using axiom track k will be at position $\text{pos}(k)$. Thus, the relation of polar inversion for bipoositions is defined for all $(a, c) \in \mathbb{B}^t$ and $x \in \mathcal{V}$ s.t. $t(a) = \lambda x$ by:

- $(a, k \cdot c) \rightarrow_{\text{pi}} (\text{pos}(k), c)$ if $k \in \text{Tr}_\lambda(a)$ (first case)

If $k \notin \text{Tr}_\lambda(a)$ (*e.g.*, $k = 8$ above), then $k \notin K$ and S_k is morally empty (it does not exist), for that we set, for all $(a, c) \in \mathbb{B}^t$, $k \geq$ and $x \in \mathcal{V}$ s.t. $t(a) = \lambda x$:

- $(a, k \cdot c) \rightarrow_{\text{pi}} \mathbf{p}_\perp$ if $k \notin \text{Tr}_\lambda(a)$ (second case)

When $\mathbf{p}_2 \neq \mathbf{p}_\perp$, relation $\mathbf{p}_1 \rightarrow_{\text{pi}} \mathbf{p}_2$ means that \mathbf{p}_2 is the (positive) polar inverse of \mathbf{p}_1 . For the first case, position $k \cdot c \in (S_k)_{k \in K} \rightarrow T$ corresponds indeed to position $c \in S_k$. For the second case (when $k \notin \text{Tr}_\lambda(a)$), we have seen that S_k must be empty, so $(a, k \cdot c)$ *cannot* be in a $[\cdot]$ -derivation, so we relate $(a, k \cdot c)$ to the constant \mathbf{p}_\perp , meaning “empty bipoosition”. The relation \rightarrow_{pi} depends both on t and on the choice of $[\cdot]$.

For instance, in P_{ex} (with a suitable choice of $[\cdot]$), we have $(\varepsilon, 9) \rightarrow_{\text{pi}} (0 \cdot 2, \varepsilon)$ (because the **ax**-rule typing x and using track 9 is at position $0 \cdot 2$): indeed, $(\varepsilon, 9)$ and $(0 \cdot 2, \varepsilon)$

both point to the type symbol o . Likewise, $(\varepsilon, 4 \cdot 1) \rightarrow_{\mathbf{p}_1} (0 \cdot 1, 1)$ and in Fig. 12.2, the occurrence of o' at bipoosition $(\varepsilon, 9)$ is colored in blue whereas that at $(0 \cdot 1, 1)$ is the top occurrence of o colored in red. Likewise, $(\varepsilon, 4) \rightarrow_{\mathbf{p}_1} (0 \cdot 1, \varepsilon)$ (they both point to the type symbol \rightarrow) and $(\varepsilon, 4 \cdot 3) \rightarrow_{\mathbf{p}_1} (0 \cdot 1, 3)$ (they both point to the type symbol o'). Since 3 and 8 are not used as axiom tracks for x , we should have $(\varepsilon, 3) \rightarrow_{\mathbf{p}_1} \mathbf{p}_\perp$ and $(\varepsilon, 8 \cdot 3 \cdot 2 \cdot 1 \cdot 0) \rightarrow_{\mathbf{p}_1} \mathbf{p}_\perp$ (for a good choice of $[\cdot]$).

Remark 12.3 (Polar Inversion in this Chapter). The definition of polar inversion differs from that of Sec. 11.3.2 for three reasons.

- First, in Chapter 11, polar inversion is defined in any derivation of system \mathbf{S} (polar inversion in P depends on P , by Remark 11.2, p. 252) where here, polar inversion is defined on $[\cdot]$ -candidate bisupports, not on a particular derivation since, for now, nothing ensures that one exists.
- Second, the function \mathbf{pos} uses only one argument (whereas it needs three in Sec. 11.3.2) thanks to the function $[\cdot]$, whose injectivity ensures that x and a are not necessary.
- Third, the bipoosition \mathbf{p}_\perp was not needed in a particular \mathbf{S} -derivation. This bipoosition is a technicality that is only useful in the purpose of describing candidate bisupports (see Proposition 12.3 below), which is needed to obtain the main theorem of this chapter.

12.2.4 Type Formation, Type Destruction

In this section, we conclude the definitions of the stability relations that characterize the form of \mathbf{S} -derivations, yielding the notion of **subjagation**, as in Sec. 12.2.1.

The notion of **consumption**, motivated in Sec. 11.1, is associated with rule **app**. Assume $t(a) = @$, $t|_a = uv$ with $u : (S_k)_{k \in K} \rightarrow T$ and $v : S_k$ for all $k \in K$ as in Fig. 12.1 so that uv can be typed with T . Each type S_k occurs in $(S_k)_{k \in K} \rightarrow T$ and $v : S_k$. However, it is absent in the type of uv : we say it has been **consumed**. Formally, we set, for all $(a, c) \in \mathbb{B}^t$, $k \geq 2$ s.t. $t(a) = @$:

- $(a \cdot 1, k \cdot c) \xrightarrow{a} (a \cdot k, c)$

Indeed, the premise concluding with $u : (S_k)_{k \in K} \rightarrow T$ (resp. with $v : S_k$) is at position $a \cdot 1$ (resp. $a \cdot k$). Position $c \in \mathbf{supp}(S_k)$ corresponds to position $k \cdot c$ in $\mathbf{supp}((S_k)_{k \in K} \rightarrow T)$. For instance, in $P_{\mathbf{ex}}$, there is an **app**-rule at position 0 and $(0 \cdot 1, 8) \xrightarrow{0} (0 \cdot 8, \varepsilon)$ (pointing to type symbol o) and $(0 \cdot 1, 3) \xrightarrow{0} (0 \cdot 3, \varepsilon)$ (pointing to o').

We set $\rightarrow = \cup \{ \xrightarrow{a} \mid a \in \mathbb{A}, t(a) = @ \}$ and write \leftarrow for the symmetric relation.

Let P be a $[\cdot]$ -derivation typing a term t . If $\mathbf{p}_1 \rightarrow_{\mathbf{asc}} \mathbf{p}_2$ or $\mathbf{p}_1 \rightarrow_{\mathbf{p}_1} \mathbf{p}_2$ or $\mathbf{p}_1 \rightarrow \mathbf{p}_2$, then $\mathbf{p}_1 \in \mathbf{bisupp}(P)$ iff $\mathbf{p}_2 \in \mathbf{bisupp}(P)$ (by construction of those relations).

However, relations $\rightarrow_{\mathbf{asc}}$, $\rightarrow_{\mathbf{p}_1}$ and \rightarrow are not enough to express the stability conditions characterizing a bisupport candidate for a derivation typing t (as we have for types with Lem. 12.1). We need to ensure that types are correctly formed and that an abstraction is typed with an arrow type (and not with a type variable). For that, we define additional relations $\rightarrow_{\mathbf{t}_1}$, $\rightarrow_{\mathbf{t}_2}$, $\rightarrow_{\mathbf{abs}}$ and $\rightarrow_{\mathbf{down}}$ below.

- Relations $\rightarrow_{\mathbf{t}_1}$ and $\rightarrow_{\mathbf{t}_2}$ ensure that the types are correctly defined and are natural extensions of those of Sec 12.2.1:

- For all $(a, c) \in \mathbb{B}^t$ and $k \in \mathbb{N}$, $(a, c \cdot k) \rightarrow_{t1} (a, c)$.
- For all $(a, c) \in \mathbb{B}^t$ and $k \geq 2$, $(a, c \cdot 1) \rightarrow_{t2} (a, c \cdot k)$.

• Note that, for instance, if $t = I = \lambda x.x$ and $B = \{p_\varepsilon\} = \{(\varepsilon, \varepsilon)\}$, then B is stable under \rightarrow_{asc} , $asc \leftarrow$, \rightarrow , \leftarrow , \rightarrow_{t1} , \rightarrow_{t2} but obviously, there is no derivation P typing t such that $\mathbf{bisupp}(P) = B$ (indeed, t is necessarily typed with an arrow type *i.e.* $1 \in \mathbf{supp}(T^P(\varepsilon))$, which means $(\varepsilon, 1) \in \mathbf{bisupp}(P)$). Thus, those relations are not enough to describe a candidate bisupport.

The relation \rightarrow_{abs} ensures that, if $\lambda x.u$ is a typed subterm of t , then its type T is an arrow and not a type variable. When T is an arrow, then $\mathbf{supp}(T)$ must at least contain 1 (besides ε), the position of the root of right-hand side of the arrow. We set then, for all $a \in \mathbb{A}^t$ s.t. $t(a) = \lambda x$:

- $(a, \varepsilon) \rightarrow_{abs} (a, 1)$

• Relation \rightarrow_{down} is included in the reflexive transitive closure of $asc \leftarrow \cup \leftarrow \cup \rightarrow_{t1} \cup \rightarrow_{abs}$, but turns out to be useful (to obtain Lemma 12.13). When the biposition (a', c) is in a derivation P , then every position below a' must be in P (must have a non-empty type). We set then, for all $a, a' \in \mathbb{A}$ and $c \in \mathbb{N}^*$ s.t. $a \leq a'$:

- $(a', c) \rightarrow_{down} (a, \varepsilon)$
- $p_\perp \rightarrow_{down} (a, \varepsilon)$

The 2nd case is also useful to ensure Lemma 12.13.

• We set $\rightarrow_\bullet = \rightarrow \cup \leftarrow \cup \rightarrow_{t1} \cup \rightarrow_{t2} \cup \rightarrow_{abs} \cup \rightarrow_{down}$. If $p_1 \rightarrow_\bullet p_2$, notice that, by construction, $p_1 \in \mathbf{bisupp}(P)$ implies $p_2 \in \mathbf{bisupp}(P)$. We say then that p_1 **subjugates** p_2 , generalizing Sec. 12.2.1.

12.2.5 Threads and Minimal Bisupport Candidate

We prove now that the relations above are indeed enough to express a sufficient condition of typability (Corollary 12.1). For that, we will formally define the notion of *thread* (from Sec. 11.1).

As we have seen, if P is a $[\cdot]$ -derivation, then $\mathbf{bisupp}(P)$ is closed under \rightarrow_{asc} , $asc \leftarrow$, \rightarrow_{pi} , $pi \leftarrow$, \rightarrow , \leftarrow , \rightarrow , \rightarrow_{t1} , \rightarrow_{t2} , \rightarrow_{abs} and \rightarrow_{down} . Of course, p_\perp , the empty biposition, cannot be in P . It turns that it is enough to characterize candidate bisupports (Proposition 12.3). In this statement, \equiv is the reflexive, transitive, symmetric closure of $\rightarrow_{asc} \cup \rightarrow_{pi}$. We have:

Proposition 12.3. Let $B \subseteq \mathbb{B}^t$. Then B is a $[\cdot]$ -candidate bisupport for a derivation typing t (*i.e.* there exists a $[\cdot]$ -derivation s.t. $B = \mathbf{bisupp}(P)$) iff (1) B is non-empty, (2) B is closed under \equiv and \rightarrow_\bullet , and (2) B does not contain p_\perp .

If the closure of a set B contains p_\perp , then intuitively, B needs to use a slot that is constrained (by relevance) to be empty: thus, no derivation can contain B .

Proof. The necessity of these conditions has been discussed in the previous subsections.

Conversely, assume that $\emptyset \neq B \subset \mathbb{B}^t \setminus \{\mathbf{p}_\perp\}$ is closed under \equiv and \rightarrow_\bullet . We want a derivation P s.t. $\mathbf{bissupp}(P) = B$. For that, we need to suitably decorate the $\mathbf{p} \in B$. Mainly, a non-terminal biposition must be labelled with \rightarrow and a terminal one with a fixed type variable o , in order to get correct types (as in Sec. 12.2.1). Thus, we set $\mathbf{Lves}(B) = \{(a, c) \in B \mid (a, c \cdot 1) \notin B\}$ and we define P on B by $P(\mathbf{p}) = o$ if $\mathbf{p} \in \mathbf{Lves}(B)$ and $P(\mathbf{p}) = \rightarrow$ if not. We now verify that P is a correct \mathbf{S} -derivation using the definition of \equiv and \rightarrow_\bullet .

Let $A = \{a \in \mathbb{A}^t \mid \exists c \in \mathbb{N}^*, (a, c) \in B\}$. Thus, $A \subseteq \mathbb{A}$. For all $a \in A$, we set $\mathbf{T}(a)(c) = P(a, c)$ whenever $(a, c) \in B$. For all $a \in A$ and $x \in \mathcal{V}$, we set $A_a(x) = A \cap \mathbb{A}_a(x)$ and $\mathbf{C}(a)(x) = \uplus_{a_0 \in \mathbb{A}_a(x)} (\lfloor a_0 \rfloor \cdot \mathbf{T}(a_0))$. Thus, $\mathbf{T}(a)$ and $\mathbf{C}(a)(x)$ are functions from \mathbb{N}^* to $\mathcal{O} \cup \{\rightarrow\}$. By hypothesis, if $\mathbf{thr}(\mathbf{p}) = \theta_\perp$, then $\mathbf{p} \notin B$.

For all $a \in \mathbb{A}$, if $a \in A$, $\mathbf{dom}(\mathbf{T}(a))$ is a tree and $\mathbf{T}(a)$, as a labelled tree, is a correct type and if $a \notin A$, $\mathbf{dom}(\mathbf{T}(a)) = \emptyset$.

Indeed, if $a \in A$, then $\mathbf{dom}(\mathbf{T}(a))$ is non-empty. The definitions of $\mathbf{Lves}(B)$, $\mathbf{T}(a)(c)$ and Lemma 12.1 grant then that $\mathbf{T}(a)$ is a correct type.

We consider now $a \in A$ and check that the typing rules are respected.

- Assume $t(a) = x$. Then, by definition of \mathbf{C} , $\mathbf{C}(a)(y) = ()$ if $y \neq x$ and $\mathbf{C}(a)(x) = (\lfloor a \rfloor \cdot \mathbf{T}(a))$, so that a is a correct axiom rule.
- Assume $t(a) = @$.
 - Since $(a, c) \rightarrow_{\text{asc}} (a \cdot 1, 1 \cdot c)$, then $(a, c) \in B$ iff $(a \cdot 1, 1 \cdot c) \in B$ and even $(a, c) \in \mathbf{Lves}(B)$ iff $(a \cdot 1, 1 \cdot c) \in \mathbf{Lves}(B)$, so that functions $c \mapsto \mathbf{T}(a)(c)$ and $c \mapsto \mathbf{T}(a \cdot 1)(1 \cdot c)$ are equal.
 - Moreover, by \rightarrow and \leftarrow , $(a \cdot 1, k \cdot c) \in B \iff (a \cdot k, c) \in B$, and even $(a \cdot 1, k \cdot c) \in \mathbf{Lves}(B) \iff (a \cdot k, c) \in \mathbf{Lves}(B)$. Thus, for all $k \geq 2, c \in \mathbb{N}^*$, $\mathbf{T}(a \cdot 1, k \cdot c) = \mathbf{T}(a \cdot k, c)$.
 - Since $a \in A$, $(a, \varepsilon) \in B$, so $(a \cdot 1, 1) \in B$, so $(a \cdot 1, \varepsilon) \notin \mathbf{Lves}(B)$, so $\mathbf{T}(a \cdot 1)(\varepsilon) = \rightarrow$, by definition of \mathbf{T} .

Thus, $\mathbf{T}(a \cdot 1) = (\mathbf{T}(a \cdot k))_{k \geq 2} \rightarrow \mathbf{T}(a)$.

By definition of \mathbf{C} and of $A_a(x)$, we easily obtain $\mathbf{C}(a)(x) = \cup_{k \geq 1, a \cdot k \in A} \mathbf{C}(a \cdot k)(x)$ as expected.

Thus, a is a correct **app**-rule.

- Assume $t(a) = \lambda x$. Then, by \rightarrow_{abs} , $(a, 1) \in B$ and $(a, \varepsilon) \notin \mathbf{Lves}(B)$, so that $\mathbf{T}(a)(\varepsilon) = \rightarrow$. Since $(a, 1 \cdot c) \rightarrow_{\text{asc}} (a \cdot 0, c)$, then $(a, 1 \cdot c) \in B$ iff $(a \cdot 0, c) \in B$ and even $(a, 1 \cdot c) \in \mathbf{Lves}(B)$ iff $(a \cdot 0, c) \in \mathbf{Lves}(B)$, so that functions $c \mapsto \mathbf{T}(a, 1 \cdot c)$ and $(c \mapsto \mathbf{T}(a \cdot 0, c))$ are equal. Moreover, let $k \geq 2$:

- If $k \notin \mathbf{Tr}_\lambda(a)$, then $(a, k \cdot c) \rightarrow_{\text{pi}} \mathbf{p}_\perp$, so $\mathbf{thr}(a, k \cdot c) = \theta_\perp$, so, by hypothesis, $(a, k \cdot c) \notin B$, so $k \cdot c \notin \mathbf{supp}(\mathbf{T}(a))$.

- If $k \in \text{Tr}_\lambda(a)$, then $(a, k \cdot c) \rightarrow_{\text{pi}} (\text{pos}(k), c)$, so $(a, k \cdot c) \equiv (\text{pos}(k), c)$. So $(a, k \cdot c) \in B$ iff $(\text{pos}(k), c) \in B$ and even $(a, k \cdot c) \in \text{Lves}(B)$ iff $(\text{pos}(k), c) \in L(B)$.

This shows that, for all $k \geq 2$, functions $c \mapsto \text{T}(a, k \cdot c)$ and $c \mapsto \text{C}(a \cdot 0)(x)(c)$ are equal, by definition of \mathbf{C} .

Thus, we have $\text{T}(a) = \text{C}(a \cdot 0)(x) \rightarrow \text{T}(a \cdot 0)$ and a is a correct **abs**-rule.

□

Remark 12.4. We only use one type variable o in the above proof. A general method (yielding every $[\cdot]$ -derivation whose bisupport is B) is to label $\mathbf{p}_1, \mathbf{p}_2 \in \text{Lves}(B)$ with the same type variable whenever $\mathbf{p}_1 \equiv_{\text{@}} \mathbf{p}_2$, where $\equiv_{\text{@}}$ is the reflexive, transitive, symmetric closure of $\rightarrow_{\text{asc}} \cup \rightarrow_{\text{pi}} \cup \rightarrow$.

From now on, it will be better to reason modulo \equiv (it may already be guessed that \equiv *should* commute with $\rightarrow, \rightarrow_{\text{t1}}, \dots$, which is made explicit in Sec. 12.3.2) and to focus on subjugation.

Definition 12.1. Let t be a term and $[\cdot] : \mathbb{N}^* \rightarrow \mathbb{N} \setminus \{0, 1\}$ an injection, and $\rightarrow_{\text{asc}}, \rightarrow_{\text{pi}}$ the relations of ascendance and polar inversion in \mathbb{B}^t defined w.r.t. $[\cdot]$.

- An **ascendant thread** is an equivalence class of relation \equiv_{asc} , the reflexive, transitive, symmetric closure of \rightarrow_{asc} .
- A **thread** (metavariable θ) is an equivalence class of relation \equiv (see Fig. 12.3).
- The **quotient set** \mathbb{B}^t / \equiv is denoted **Thr**.

In Fig. 12.2, the red occurrences of o' correspond to the ascendant thread $\{(\varepsilon, 1), (1, \varepsilon), (0, 1, 1)\}$ and the blue occurrence of o' to another ascendant thread (with only one element). The four colored occurrences of o' correspond to a thread.

The notation **Thr** implicitly depends on t and $[\cdot]$. The thread of $(a, c) \in \mathbb{B}$ is written $\text{thr}(a, c)$ and we set

$$\theta_\varepsilon = \text{thr}(\varepsilon, \varepsilon) \qquad \theta_\perp = \text{thr}(\mathbf{p}_\perp)$$

Threads and ascendant threads are informally discussed in Sec. 11.1.1 (see in particular Fig. 11.1). If $\text{thr}(\mathbf{p}) = \theta$, we say that θ **occurs at bposition** \mathbf{p} , also written $\theta : \mathbf{p}$ or $\mathbf{p} : \theta$.

We consider now the extension of every other relation modulo \equiv . Namely, we write $\theta_1 \xrightarrow{a} \theta_2$ if $\exists \mathbf{p}_1, \mathbf{p}_2, \theta_1 = \text{thr}(\mathbf{p}_1), \theta_2 = \text{thr}(\mathbf{p}_2), \mathbf{p}_1 \xrightarrow{a} \mathbf{p}_2$. Thus, $\theta_1 \xrightarrow{a} \theta_2$ iff $\theta_1 : \mathbf{p}_1 \xrightarrow{a} \mathbf{p}_2 : \theta_2$ for some $\mathbf{p}_1, \mathbf{p}_2$. In that case, we say that θ_1 (resp. θ_2) has been **left-consumed** (resp. **right-consumed**) at bposition \mathbf{p}_1 (resp. \mathbf{p}_2).

We proceed likewise for $\rightarrow_{\text{t1}}, \rightarrow_{\text{t2}}, \rightarrow_{\text{abs}}, \rightarrow_{\text{down}}, \rightarrow_\bullet$, thus defining $\tilde{\rightarrow}_{\text{t1}}, \tilde{\rightarrow}_{\text{t2}}, \tilde{\rightarrow}_{\text{abs}}, \tilde{\rightarrow}_{\text{down}}, \tilde{\rightarrow}_\bullet$. Notation $\tilde{\rightarrow}_\bullet^*$ denotes the reflexive transitive closure of relation $\tilde{\rightarrow}_\bullet$.

Corollary 12.1. If θ_\perp is not in the transitive closure of $\{\theta_\varepsilon\}$ by $\tilde{\rightarrow}_\bullet$, then t is typable in \mathbf{S} (by means of a $[\cdot]$ -derivation).

Proof. Let $\mathbb{B}_{\min} = \{\mathbf{p} \in \mathbb{B} \mid \theta_\varepsilon \tilde{\rightarrow}_\bullet^* \text{thr}(\mathbf{p})\}$ i.e. \mathbb{B}_{\min} is the union of the reflexive transitive closure of $\text{thr}(\varepsilon)$ under $\tilde{\rightarrow}_\bullet$. If $\theta_\varepsilon \rightarrow_\bullet^* \theta_\perp$ does not hold, then \mathbb{B}_{\min} satisfies the hypotheses of Proposition 12.3. So there exists a derivation P s.t. $\text{bisupp}(P) = \mathbb{B}_{\min}$ and thus, t is typable. □

Since a \mathbf{S} -derivation contains \mathbf{p}_ε , by Proposition 12.3, any $[\cdot]$ -derivation typing t will satisfy $\mathbb{B}_{\min} \subseteq \mathbf{bisupp}(P)$ and thus, \mathbb{B}_{\min} is the **minimal bisupport candidate** for a $[\cdot]$ -derivation typing t .

Given $t \in \Lambda$ and $[\cdot]$, let \mathcal{T} be the first order theory whose set of constants is $\mathbf{Thr}(P)$ and whose axioms are $\theta_\varepsilon \neq \theta_\perp$ and $(\theta_1 = \theta_2)$ (for all pairs (θ_1, θ_2) such that $\theta_1 \xrightarrow{\bullet} \theta_2$). Then Corollary 12.1 states that there exists a $[\cdot]$ -derivation P typing t iff \mathcal{T} is not contradictory. This corresponds to the presentation p. 236.

12.3 Nihilating Chains

We begin Sec. 12.3 with a global description of the key steps leading to the final result (complete unsoundness of \mathbf{S}) and a presentation of the central notion of nihilating chain.

In the purpose of proving that every term is typable, we want to prove that, for all term t and injection $[\cdot] : \mathbb{N}^* \rightarrow \mathbb{N} \setminus \{0, 1\}$, there is a $[\cdot]$ -derivation typing t . According to Corollary 12.1, we must show that θ_\perp is not in the reflexive transitive closure of θ_ε by $\xrightarrow{\bullet}$. A proof of $\theta_\varepsilon \xrightarrow{\bullet}^* \theta_\perp$ would involve a nihilating chain:

Definition 12.2. A (**nihilating**) **chain** is a *finite* sequence of the form $\theta_0 \xrightarrow{\bullet} \theta_1 \xrightarrow{\bullet} \dots \xrightarrow{\bullet} \theta_m$ with $\theta_0 = \theta_\varepsilon$ and $\theta_m = \theta_\perp$.

In order to apply Corollary 12.1, we must then prove that there is no nihilating chain. In other words, this corollary implies:

Proposition 12.4. If the nihilating chains do not exist, then system \mathbf{S} is completely unsound.

We proceed *ad absurdum* and consider $\theta_0 \xrightarrow{\bullet} \theta_1 \xrightarrow{\bullet} \dots \xrightarrow{\bullet} \theta_m$ with $\theta_0 = \theta_\varepsilon$ and $\theta_m = \theta_\perp$. However, $\xrightarrow{\bullet}$ can be $\xrightarrow{\rightarrow}$, $\xrightarrow{\leftarrow}$, $\xrightarrow{\tau_1}$, $\xrightarrow{\tau_2}$, $\xrightarrow{\text{abs}}$ or $\xrightarrow{\text{down}}$. The structure of the proof is the following:

- We define (Definition 12.3) the notion of *polarity* for bipoositions: a bipoosition is negative when it is created by an **abs**-rule (modulo \rightarrow_{asc}) and positive if not.
- The termination of a finite collapsing strategy (Sec. 12.4.2) guarantees that positivity can be assumed to only occur at suitable places in the chain without loss of generality. In that case, we say that the nihilating chain is *normal* (Definition 12.4).
- In normal chains, the different cases of subjugation interact well (Sec. 12.3.2), so that, from any normal chain, we may build another that begins with $\theta_\varepsilon \xrightarrow{\bullet} \theta_1$ (Sec. 12.3.3). This is easily shown to be impossible, allowing us to conclude that nihilating chains do not exist and that every term is \mathbf{S} -typable.

12.3.1 Polarity and Threads

In this section, we describe the form of threads and ascendant threads (Definition 12.1).

Notice that \rightarrow_{asc} is functional: if $\mathbf{p}_1 \rightarrow_{\text{asc}} \mathbf{p}_2$, we write $\mathbf{p}_2 = \mathbf{asc}(\mathbf{p}_1)$. Notice also that **asc** is injective. Thus, $\mathbf{p}_1 \equiv_{\text{asc}} \mathbf{p}_2$ iff $\exists i \geq 0$, $\mathbf{p}_2 = \mathbf{asc}^i(\mathbf{p}_1)$ or $\mathbf{p}_1 = \mathbf{asc}^i(\mathbf{p}_2)$.

Given a bipoosition $\mathbf{p} = (a, c) \in \mathbb{B}$, we call a the **outer** and c the **inner position** of \mathbf{p} . Since **asc** may only add the prefix 0 or 1 to a and add/remove the prefix 1 to c , by induction:

Lemma 12.2. If $(a_1, c_1) \equiv_{\text{asc}} (a_2, c_2)$ then $\exists a_3 \in \{0, 1\}^*$, $(a_2 = a_1 \cdot a_3$ or $a_1 = a_2 \cdot a_3)$ and $\exists i \geq 0$, $(c_2 = 1^i \cdot c_1$ or $c_1 = 1^i \cdot c_2)$.

We set, for all $\mathbf{p} \in \mathbb{B}$, $\text{Asc}(\mathbf{p}) = \text{asc}^i(\mathbf{p})$, where i is maximal (*i.e.* $\text{asc}^i(\mathbf{p})$ is defined, but not $\text{asc}^{i+1}(\mathbf{p})$). Thus, $\text{Asc}(\mathbf{p})$ is the **top ascendant** of \mathbf{p} *e.g.*, in P_{ex} , $\text{Asc}(\varepsilon, 1) = (0 \cdot 1, 1)$. As noted in Observation 11.1, p. 241, a top ascendant is either located in an **ax**-node or an **abs**-node (**asc** is total on **app**-nodes), motivating the notion of (syntactic) polarity (see Sec. 11.1.2 for more details) for bipositions:

Definition 12.3.

- Let $\mathbf{p} \in \mathbb{B}^t \setminus \{\mathbf{p}_\perp\}$ and $(a_0, c_0) = \text{Asc}(\mathbf{p})$. We define the **polarity** of \mathbf{p} as follows: if $t(a_0) = x$ for some $x \in \mathcal{V}$, then we set $\text{Pol}(\mathbf{p}) = \oplus$ and if $t(a_0) = \lambda x$, then we set $\text{Pol}(\mathbf{p}) = \ominus$. We also set $\text{Pol}(\mathbf{p}_\perp) = \ominus$.
- If $\text{thr}(\mathbf{p}) = \theta$ and $\text{Pol}(\mathbf{p}) = \oplus/\ominus$, we say that θ occurs positively/negatively at biposition \mathbf{p} .
- If θ is left/right-consumed at \mathbf{p} and $\text{Pol}(\mathbf{p}) = \oplus$ (resp. $\text{Pol}(\mathbf{p}) = \ominus$), we say that θ is left/right-consumed positively (resp. negatively) at biposition \mathbf{p} .

Then, we write for instance $\theta_1 \oplus \xrightarrow{a} \ominus \theta_2$ to mean that θ_1 is left-consumed positively and θ_2 is right-consumed negatively in the **app**-rule at position a . In Fig. 12.2, the blue occurrence of σ' is negative and the red ones are positive.

Since $\rightarrow_{\mathbf{p}_i}$ also defines an injective function (out of θ_\perp) and $\mathbf{p}_1 \rightarrow_{\mathbf{p}_i} \mathbf{p}_2$ implies that \mathbf{p}_1 (in a λx) and \mathbf{p}_2 (in an axiom or \mathbf{p}_\perp) do not have ascendants:

Lemma 12.3.

- For all $\mathbf{p}_1, \mathbf{p}_2 \in \mathbb{B}^t$, $\mathbf{p}_1 \equiv \mathbf{p}_2$, $\text{Pol}(a_1, c_1) = \oplus$ and $\text{Pol}(a_2, c_2) = \ominus$ iff $\text{Asc}(a_1, c_1) \rightarrow_{\mathbf{p}_i} \text{Asc}(a_2, c_2)$.
- For all $\mathbf{p} \in \mathbb{B}^t$, $\text{thr}(\mathbf{p}) = \theta_\perp$ iff $\text{Asc}(\mathbf{p}) = (a_0, k \cdot c_0)$ with $t(a_0) = \lambda x$ and $k \notin \text{Tr}_\lambda(a_0)$.

Lemmas 12.2 and 12.3 may be illustrated by Fig. 12.3 (see also the more detailed⁵ Fig. 11.1, p. 240), in which we only represent (with thick lines) the outer positions in the occurrences of the threads.

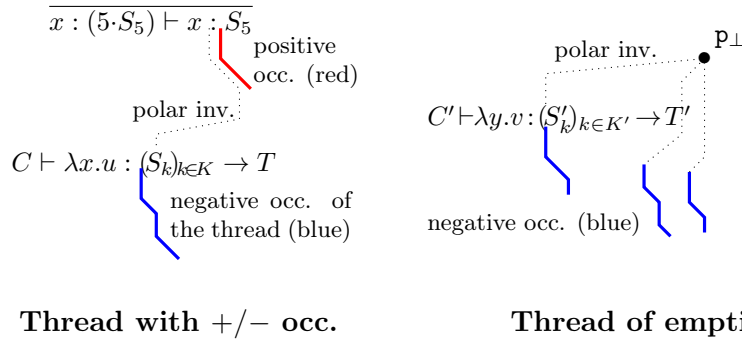


Figure 12.3: Threads

⁵ Note that the thread of emptiness is absent from Fig. 11.1, since it is necessary only in this chapter.

Thus, except the empty thread θ_{\perp} , Lemma 12.3 means a thread can have *at most* two “connex components” (a series of positive ascendants and/or a series of negative ascendants respectively) *e.g.*, the green or the purple threads in Fig. 11.1. All bipositions of θ_{\perp} are negative.

Remark 12.5.

- If the top occurrence of an ascendant thread is in an **ax**-rule typing a variable x s.t. x is not bound in t , then the thread has one (positive) “connex component” *e.g.*, the red thread in Fig. 11.1.
- If a series of ascendants ends at the *root* of an **abs**-rule introduction λx (*i.e.* $\text{Asc}(a_0, c_0) = (a, \varepsilon)$ with $t(a) = \lambda x$), then the thread has one (negative) “connex component” *e.g.*, the blue thread in Fig. 11.1. Notice that such a thread does *not* contain \mathbf{p}_{\perp} .

Lemma 12.3 will be refined into Lemmas 12.7 and 12.10.

Since a consumed biposition does not have a descendant, Lemma 12.2 and 12.3 imply the following lemma (which is the formal version of Observation 11.3, p. 11.3):

Lemma 12.4 (Uniqueness of Consumption). Let $\otimes \in \{\oplus, \ominus\}$ and $\theta \in \text{Thr}$, $\theta \neq \theta_{\perp}$. Then, there is a most one θ' s.t. $(\theta^{\otimes} \dot{\rightarrow} \theta'$ or $\theta^{\otimes} \dot{\leftarrow} \theta')$.

In the figure above, consumption may occur at the bottom of each “connex component”.

12.3.2 Interactions in Normal Chains

In Sec. 12.3.2, we present the notion of normal chain and explicit some interesting interaction properties that allow us to simplify/rewrite them.

As it has been discussed in the Introduction p. 254, the possibility for a variable x (of a redex or of a redex to be created later) to be substituted in a reduction sequence is problematic. We recall that a biposition is negative when it was “created” in an abstraction λx and that left-consumption is associated to left-hand sides of application. Thus, a negative left-consumption hints at the presence of redex (this intuition will be made more explicit in Sec. 12.4.2). More precisely, as it was informally explained in Sec. 11.2.1, it indicates the presence of a *redex tower*. This suggests the following notion (that was alluded to p. 236):

Definition 12.4. A nihilating chain is **normal** if no thread is left-consumed negatively in it (the chain does not contain a link $\theta_i^{\ominus} \dot{\rightarrow} \theta_{i+1}$ or $\theta_i \dot{\leftarrow}^{\ominus} \theta_{i+1}$).

Some relations of subjugation may roughly be related to the negative hand side of an arrow (*e.g.*, left-consumption destroys $(S_k)_{k \in K}$ in $t : (S_k)_{k \in K} \rightarrow T$) or positive hand side (*e.g.*, ascendance) or the root of a type (*e.g.*, \rightarrow_{abs} from inner position ε to 1). On the other hand, in a type, the track values give a concise way to indicate nesting inside arrows *i.e.* an argument track means a nesting inside the source of an arrow and track 1 means nesting in its target. This will help us now to better understand the possible forms of normal chains, yielding the series of **Interaction Lemmas** below.

Let $\mathbf{p} = (a, c) \in \mathbb{B}^t$. According to Lemma 12.2, ascendance/descendance can only add or remove the prefix 1 to c . From that, we deduce that the only way to remove an argument track while visiting a thread is to pass from the negative hand side to

the positive hand side of $\rightarrow_{\mathbf{p}_1}$ (indeed, track k is absent from the inner position of \mathbf{p}_2 in $\mathbf{p}_1 = (a, k \cdot c) \rightarrow_{\mathbf{p}_1} (\mathbf{pos}(k), c) = \mathbf{p}_2$). Moreover, a biposition that is left-consumed must start with an argument track (a track ≥ 2) by definition of \rightarrow . Thus, if a thread is left-consumed *positively*, the inner position of all its occurrences will contain an argument track (this is the meaning of Lemma 12.7).

By definition of $\rightarrow_{\mathbf{abs}}$ and $\rightarrow_{\mathbf{down}}$, the target of $\rightarrow_{\mathbf{abs}}$ (resp. of $\rightarrow_{\mathbf{down}}$) must have an inner position equal to 1 (resp. ε). This, with the previous observation (related to Lemma 12.7), shows that a *thread* that is consumed positively cannot be the target of $\tilde{\rightarrow}_{\mathbf{abs}}$ or $\tilde{\rightarrow}_{\mathbf{down}}$ (Lemma 12.8 below).

It is easy to see that $\tilde{\rightarrow}_{\mathbf{t}_1}$ and $\tilde{\rightarrow}$ commute (Lemma 12.6 below). Using a similar reasoning as the one above (presence of an argument track in the inner position), we may also prove that $\rightarrow_{\mathbf{t}_2}$ and $\tilde{\rightarrow}$ commute when the involved thread is left-consumed positively in $\tilde{\rightarrow}$ (Lemma 12.9).

Formal Proofs of the Intersection Lemmas The end of Sec. 12.3.2 is technical and is dedicated to the proof of the Interaction Lemmas described above. A summary of these lemmas can be found at the beginning of Sec. 12.3.3.

Lemma 12.5. If $(a_1, c_1) \equiv (a_2, c_2)$, then $(a_1, c_1 \cdot k) \equiv (a_2, c_2 \cdot k)$ for all k .

Proof. This holds when we replace \equiv by $\rightarrow_{\mathbf{asc}}$ and $\rightarrow_{\mathbf{p}_i}$. The lemma follows by induction. \square

Lemma 12.5 is useful to prove Lemmas 12.6, 12.7 and 12.10.

Lemma 12.6 (Exchange of $\tilde{\rightarrow}$ and $\tilde{\rightarrow}_{\mathbf{t}_1}$). If $\theta_1 \tilde{\rightarrow}_{\mathbf{t}_1} \theta_2$ and $\theta_2 \tilde{\rightarrow} \theta_4$, then, $\exists \theta_3, \theta_1 \rightarrow \theta_3$ and $\theta_3 \tilde{\rightarrow}_{\mathbf{t}_1} \theta_4$.

Proof. Say $\theta_1 : (a, c \cdot \ell) \rightarrow_{\mathbf{t}_1} (a, c) : \theta_2$ and $\theta_2 : (a' \cdot 1, k \cdot c') \rightarrow (a' \cdot k, c') : \theta_4$. By Lemma 12.5, $(a, c \cdot \ell) \equiv (a \cdot 1, k \cdot c' \cdot \ell)$. So we set $\theta_3 = \mathbf{thr}(a' \cdot 1, k \cdot c' \cdot \ell)$, so that $\theta_1 : (a' \cdot 1, k \cdot c' \cdot \ell) \rightarrow (a' \cdot 1, k \cdot c' \cdot \ell) : \theta_3$ and $\theta_3 : (a' \cdot 1, k \cdot c' \cdot \ell) \rightarrow_{\mathbf{t}_1} (a' \cdot 1, k \cdot c') : \theta_4$ as expected. \square

Lemma 12.7. Assume $\mathbf{Pol}(a \cdot 1, k) = \oplus$ and $(a \cdot 1, k) \equiv (a_2, c_2)$.

- If $\mathbf{Pol}(a_2, c_2) = \oplus$, then $\exists i, c_2 = 1^i \cdot k$.
 Moreover, for all $c \in \mathbb{N}^*$, $(a \cdot 1, k \cdot c) \equiv (a_2, c_2 \cdot c)$.
- If $\mathbf{Pol}(a_2, c_2) = \ominus$, let $(a_0, 1^j \cdot k \cdot c)$ and $\ell = \lfloor a_0 \rfloor$. Then $\exists j, c_2 = 1^j \cdot \ell \cdot 1^i k$.
 Moreover, for all $c \in \mathbb{N}^*$, $(a \cdot 1, k \cdot c) \equiv (a_2, c_2 \cdot c)$.

If $\mathbf{Pol}(a \cdot 1, k \cdot c) = \oplus$ for some c , then $\mathbf{Pol}(a \cdot 1, k) = \oplus$ and the lemma can be applied.

Proof. The two first points come from Lemmas 12.2, 12.3 and 12.5.

Regarding the end of the statement: by induction on i , we also prove that, for all $k \geq 2, a, c \in \mathbb{N}$, $(a^i, c^i) := \mathbf{asc}^i(a, k)$ is defined iff $\mathbf{asc}^i(a, k \cdot c)$ is and in that case, $\mathbf{asc}^i(a, k \cdot c) = (a^i, c^i \cdot c)$. Thus, if $(a_0, c_0) = \mathbf{Asc}(a, k)$, then $\mathbf{Asc}(a, k \cdot c) = (a_0, c_0 \cdot c)$. This implies that $\mathbf{Pol}(a, k) = \mathbf{Pol}(a, k \cdot c)$. \square

Lemma 12.7 is useful to prove Lemmas 12.8 and 12.9:

Lemma 12.8 (Elimination of $\tilde{\rightarrow}_{\mathbf{abs}}$ and $\tilde{\rightarrow}_{\mathbf{down}}$). If $\theta^{\oplus} \tilde{\rightarrow} \theta'$, there is no θ_0 s.t. $\theta_0 \tilde{\rightarrow}_{\mathbf{abs}} \theta$ or $\theta_0 \tilde{\rightarrow}_{\mathbf{down}} \theta$.

Proof. Say $\theta : (a \cdot 1, k \cdot c) \rightarrow (a \cdot k, c) : \theta'$ (with necessarily $k \geq 2$). We assume *ad absurdum* that $\theta_0 \xrightarrow{\text{abs}} \theta$ or $\theta_0 \xrightarrow{\text{down}} \theta$.

The first case implies that $\theta = \mathbf{thr}(a', \varepsilon)$ and the second one that $\theta = \mathbf{thr}(a', 1)$ for some a' . But this is impossible according to Lemma 12.7. \square

Lemma 12.9 (Exchange of $\oplus \xrightarrow{\text{t}}$ and $\xrightarrow{\text{t2}}$). If $\theta_1 \xrightarrow{\text{t2}} \theta_2$ and $\theta_2 \oplus \xrightarrow{\text{t}} \theta_4$, then, $\exists \theta_3, \theta_1 \oplus \xrightarrow{\text{t}} \theta_2$ and $\theta_3 \xrightarrow{\text{t2}} \theta_4$.

Proof. Say $\theta_1 : (a, c \cdot \ell) \rightarrow_{\text{t2}} (a, c \cdot 1) : \theta_2$ and $\theta_2 : (a' \cdot 1, k \cdot c') \oplus \rightarrow (a' \cdot k, c') : \theta_4$.

By Lemma 12.7, $c \cdot 1 = 1^i \cdot k \cdot c'$ or $c \cdot 1 = 1^j \cdot \ell' \cdot 1^i \cdot k \cdot c'$ for some i, j, ℓ' . Thus, $c' = c'_0 \cdot 1$ for some c'_0 .

Also by Lemma 12.7 (last statement), we have $(a, c \cdot \ell) \equiv (a' \cdot 1, k \cdot c'_0 \cdot \ell)$.

We set then $\theta_3 = \mathbf{thr}(a' \cdot k, c'_0 \cdot \ell)$, so that $\theta_1 : (a' \cdot 1, k \cdot c'_0 \cdot \ell) \rightarrow (a' \cdot k, c'_0 \cdot \ell) : \theta_3$ and $\theta_3 : (a' \cdot k, c'_0 \cdot \ell) \rightarrow_{\text{t2}} (a' \cdot k, c'_0 \cdot 1) : \theta_4$, as expected. \square

We recall from Lemma 12.3 that, if the thread of a biposition $\mathbf{p} = (a, c)$ is θ_\perp , then \mathbf{p} is negative and was "created" in an *abs*-rule (thus, c must have an argument track). In that case, it is easy to see, that when we postfix anything to the inner position c (*i.e.* we use relation \rightarrow_{t1}), we get a biposition $\mathbf{p}' = (a, c \cdot c')$ whose thread is also θ_\perp . Using the presence of an argument track k (as in the positive case discussed above), we can also prove that θ_\perp is stable under $\xrightarrow{\text{t2}}$ and cannot be the target of $\xrightarrow{\text{abs}}$ and $\xrightarrow{\text{down}}$. All this is captured by Lemma 12.11.

Lemma 12.10. Assume $(a, 1^i \cdot k) \equiv \mathbf{p}_\perp$ with $k \geq 2$. Then, for all $c \in \mathbb{N}^*$, $(a, 1^i \cdot k \cdot c) \equiv \mathbf{p}_\perp$. Moreover, if $(a, 1^i \cdot k \cdot c) \equiv \mathbf{p}_\perp$ with $k \geq 2$, then $(a, 1^i \cdot k) \equiv \mathbf{p}_\perp$, and we can apply the lemma.

Proof. Let $(a_0, 1^{i_0} \cdot k) = \mathbf{Asc}(a, 1^i \cdot k)$ and for all j such that it is defined, $(a^i, 1^{i(j)} \cdot k) = \mathbf{asc}^j(a, 1^i \cdot k)$.

By induction on j and Lemma 12.2, for all $c \in \mathbb{N}^*$, $\mathbf{asc}^j(a, 1^i \cdot k \cdot c)$ is defined iff $\mathbf{asc}^j(a, 1^i \cdot k \cdot c)$ is, and in that case, $\mathbf{asc}^j(a, 1^i \cdot k \cdot c) = (a^j, 1^{i(j)} \cdot k \cdot c)$.

Thus, $\mathbf{Asc}(a, 1^i \cdot k \cdot c) = (a_0, 1^{i_0} \cdot k \cdot c)$.

Assume moreover that $(a, 1^i \cdot k) \equiv \mathbf{p}_\perp$. Then $\mathbf{Asc}(a, 1^i \cdot k) \rightarrow_{\mathbf{p}_i} \mathbf{p}_\perp$. This implies that $t(a_0) = \lambda x, i_0 = 0$ and $k \notin \text{Tr}_\lambda(a_0)$. Thus, $\mathbf{Asc}(a, 1^i \cdot k \cdot c) = (a_0, k \cdot c) \rightarrow_{\mathbf{p}_i} \mathbf{p}_\perp$. So $(a, 1^i \cdot k \cdot c) \equiv \mathbf{p}_\perp$.

Now, assume instead that $(a, 1^i \cdot k \cdot c) \equiv \mathbf{p}_\perp$. Then $\mathbf{Asc}(a, 1^i \cdot k \cdot c) \rightarrow_{\mathbf{p}_i} \mathbf{p}_\perp$. This implies that $t(a_0) = \lambda x, i_0 = 0$ and $k \notin \text{Tr}_\lambda(a_0)$. By the above induction, $\mathbf{Asc}(a, 1^i \cdot k) = (a_0, k) \rightarrow_{\mathbf{p}_i} \mathbf{p}_\perp$. So $(a, 1^i \cdot k) \equiv \mathbf{p}_\perp$. \square

Lemma 12.10 is useful to prove Lemma 12.11:

Lemma 12.11 (The Thread of Emptiness in Action).

- If $\mathbf{thr}(\mathbf{p}) = \theta_\perp$, then $\text{Pol}(\mathbf{p}) = \ominus$.
- If $\theta \xrightarrow{\text{t1}} \theta_\perp$ or $\theta \rightarrow_{\text{t2}} \theta_\perp$, then $\theta = \theta_\perp$.
- We cannot have $\theta \xrightarrow{\text{abs}} \theta_\perp$ or $\theta \xrightarrow{\text{down}} \theta_\perp$.

Proof.

- The implication $\mathbf{thr}(\mathbf{p}) = \theta_\perp \Rightarrow \text{Pol}(\mathbf{p}) = \ominus$ comes from Lemma 12.3.

- Assume $\theta \xrightarrow{\tau_1} \theta_\perp$ e.g., say $\theta : (a, c \cdot \ell) \rightarrow_{\tau_1} (a, c) : \theta$. By Lemma 12.3, we have $(a, c) \equiv (a_0, k \cdot c_0)$ with $t(a_0) = \lambda x, k \notin \text{Tr}_\lambda(a_0), k \geq 2$. By Lemma 12.5, $(a, c \cdot \ell) \equiv (a_0, k \cdot c_0 \cdot \ell)$. But $(a_0, k \cdot c_0 \cdot \ell) \rightarrow_{\text{pi}} \mathbf{p}_\perp$. So $\text{thr}(a, c \cdot \ell) = \mathbf{p}_\perp$ i.e. $\theta = \theta_\perp$.
- Assume $\theta \xrightarrow{\tau_2} \theta_\perp$ e.g., say $\theta : (a, c \cdot \ell) \rightarrow_{\tau_2} (a, c \cdot 1) : \theta_\perp$.
 By Lemma 12.3, we have $(a, c \cdot 1) \equiv_{\text{asc}} (a_0, k \cdot c_0)$ with $t(a_0) = \lambda x, k \notin \text{Tr}_\lambda(a_0), k \geq 2$.
 By Lemma 12.2, $c \cdot 1 = 1^i \cdot k \cdot c'_0 \cdot 1$ for some i, c'_0 .
 By Lemma 12.10, $(a, 1^i \cdot k) \equiv \mathbf{p}_\perp$ and then $(a, c \cdot \ell) = (a, 1^i \cdot k \cdot c'_0 \cdot \ell) \equiv \mathbf{p}_\perp$. Thus, $\theta = \theta_\perp$.
- If $\mathbf{p} : \theta_\perp$, by Lemmas 12.2 and 12.3, $\mathbf{p} = (a, 1^i \cdot k \cdot c)$. Thus, $\mathbf{p} \rightarrow_{\text{abs}} \mathbf{p}' = (a', 1)$ or $\mathbf{p} \rightarrow_{\text{down}} \mathbf{p}' = (a', \varepsilon)$ is impossible.

□

When the considered nihilating chain is not normal, the arguments involving the presence of argument tracks fail and it is not difficult to find counter-examples to the commutation of $\xrightarrow{\tau}$ and $\xrightarrow{\tau_2}$ or to Lemma 12.8.

12.3.3 Complete Unsoundness (almost) at Hand

Now, using the Interaction Lemmas 12.6, 12.8, 12.9, 12.11, we may build (by the Claim below), from any *normal* chain, a nihilating chain of the form $\theta_\varepsilon = \theta_0^\oplus \xrightarrow{\tau} \theta_1^\oplus \xrightarrow{\tau} \dots \xrightarrow{\tau} \theta_\ell$ θ_\perp . Then we prove that a chain of this latter form cannot exist. More concretely, this *almost* proves that system \mathbf{S} is completely unsound (by Proposition 12.4). Almost, because only the non-existence of normal nihilating chains will be proved at the end of this Sec. 12.3.3. The only point that will remain to be verified is that normal nihilating chains can be considered without loss of generality (which is the object of Sec. 12.4).

The Interaction Lemmas can be summarized as follows:

- $\theta_1 \xrightarrow{\tau_1} \theta_2 \xrightarrow{\tau} \theta_3$ can be replaced by $\theta_1 \xrightarrow{\bullet} \theta'_2 \xrightarrow{\tau_1} \theta_3$ (Lemma 12.6).
- $\theta_1 \xrightarrow{\tau_1} \theta_2^\oplus \xrightarrow{\tau} \theta_3$ can be replaced by $\theta_1^\oplus \xrightarrow{\tau} \theta_2 \xrightarrow{\tau} \theta_3$ (Lemma 12.9).
- $\theta_1 \xrightarrow{\text{down}} \theta_2^\oplus \xrightarrow{\tau} \theta_3$ and $\theta_1 \xrightarrow{\text{abs}} \theta_2^\oplus \xrightarrow{\tau} \theta_3$ is impossible (Lemma 12.8).
- $\theta \xrightarrow{\tau_{1/2}} \theta_\perp$ imply $\theta = \theta_\perp$ and $\theta \xrightarrow{\text{abs/down}} \theta_\perp$ is impossible (Lemma 12.11).

Moreover, the Consumption Lemma (Lemma 12.4) can also be seen as an interaction lemma: it entails that $\theta_1 \xleftarrow{\oplus} \theta_2^\oplus \xrightarrow{\tau} \theta_3$ implies $\theta_1 = \theta_3$ i.e. nothing happens when $\xleftarrow{\oplus}$ is followed by $\xrightarrow{\tau}$. These observations are enough to prove that normal chains do not exist.

In particular, $\theta_\varepsilon : (a \cdot 1, k \cdot c) \rightarrow (a \cdot k, c) : \theta_1$ for some $(a, c) \in \mathbb{B}, k \in \mathbb{N}$. We prove now that $\theta_\varepsilon \rightarrow \dots$ is impossible: by Lemma 12.2, $\text{asc}^i(\varepsilon, \varepsilon) = (a_0, 1^{i_0})$ for some a_0, i_0 .

- Assume $\text{Pol}(\varepsilon, \varepsilon) = \oplus$: since $\text{Pol}(a \cdot 1, k \cdot c) = \oplus$, by Lemma 12.2, $(\varepsilon, \varepsilon) \equiv (a \cdot 1, k \cdot c)$ is impossible.
- Assume $\text{Pol}(\varepsilon, \varepsilon) = \ominus$: $\exists \mathbf{p}, (a_0, 1^{i_0}) \rightarrow_{\text{pi}} \mathbf{p}$ is impossible by definition of \rightarrow_{pi} . Thus, $(\varepsilon, \varepsilon) \equiv \mathbf{p}$ and $\text{Pol}(\mathbf{p}) = \oplus$ is impossible.

So, there are no nihilating chains containing only $\oplus \xrightarrow{\tau}$. And thus, there are no normal nihilating chains:

Proposition 12.5. There is no *normal* nihilating chain.

In the next section, we describe a method to build, from any given nihilating chain (if such a chain existed), a normal chain. By the above proposition, it will *ad absurdum* entail that there are no nihilating chains. But before that, in order to be valid, the reasoning and the Proposition above need the following Claim to be proved.

Claim: From any *normal* chain, we can build a nihilating chain of the form $\theta_\varepsilon = \theta_0^\oplus \rightsquigarrow \theta_1^\oplus \rightsquigarrow \dots \rightsquigarrow \theta_\ell = \theta_\perp$.

Proof. Let \mathcal{C} be a *normal* nihilating chain. Thus, \mathcal{C} is of the form $\theta_0 \rightsquigarrow_\bullet \theta_1 \rightsquigarrow_\bullet \dots \rightsquigarrow_\bullet \theta_m$ with $\theta_0 = \theta_\varepsilon$ and $\theta_m = \theta_\perp$. We are going to apply the algorithm below to \mathcal{C} .

Description of the algorithm: We first apply commutations to \mathcal{C} . During this process, we implicitly use Lemma 12.4 every time that some relation $\theta_{i-1} \overset{\oplus}{\leftarrow} \theta_i \rightsquigarrow \theta_{i+1}$ pops up somewhere in the chain: since this implies $\theta_{i-1} = \theta_{i+1}$, we just remove θ_i from the chain.

The first stage of the algorithm consists in using Lemmas 12.6 and 12.9 as many times as possible, so that $\theta_{i-1} \rightarrow_{\mathbf{t1}} \theta_i \rightsquigarrow \theta_{i+1}$ or $\theta_{i-1} \rightarrow_{\mathbf{t2}} \theta_i \rightsquigarrow \theta_{i+1}$ does not occur anymore in the chain.

Then, we are interested in the last relation $\theta_{m-1} \rightsquigarrow_\bullet \theta_m = \theta_\perp$ of the chain.

By Lemma 12.11, if it is $\rightsquigarrow_{\mathbf{t1}}$ or $\rightsquigarrow_{\mathbf{t2}}$, then $\theta_{m-1} = \theta_\perp$ and we discard θ_m (second stage of the algorithm).

Also by Lemma 12.11, the last relation can neither be $\rightsquigarrow_{\mathbf{abs}}$ nor $\rightsquigarrow_{\mathbf{down}}$.

Since θ_\perp only occurs negatively (Lemma 12.11) and the chain is normal, then the last relation cannot be $\overset{\leftarrow}{\rightsquigarrow}$.

So the last relation of the chain after the two stages of the algorithm is $\theta_{\ell-1}^\oplus \rightsquigarrow \theta_\ell = \theta_\perp$ (with $\ell \geq m$).

Assume now *ad absurdum* that the chain contains a relation that is not \rightsquigarrow .

We investigate the maximal k such that relation $\theta_{k-1} \rightsquigarrow_\bullet \theta_k$ is *not* \rightsquigarrow .

We have just proved that $k < \ell$, so that we have $\theta_{k-1} \rightsquigarrow_\bullet \theta_k \overset{\oplus}{\rightsquigarrow} \theta_{k+1}$. By Lemma 12.8, this relation cannot be $\rightsquigarrow_{\mathbf{abs}}$ or $\rightsquigarrow_{\mathbf{down}}$.

By the first stage of the algorithm, it can neither be $\rightsquigarrow_{\mathbf{t1}}$ nor $\rightsquigarrow_{\mathbf{t2}}$. So we must have $\theta_{k-1} \overset{\oplus}{\leftarrow} \theta_k \rightsquigarrow \theta_{k+1}$.

So we must have $\theta_{k-1} \overset{\leftarrow}{\rightsquigarrow} \theta_k$. Since the chain is normal, we have $\theta_{k-1} \overset{\oplus}{\leftarrow} \theta_k \overset{\oplus}{\rightsquigarrow} \theta_{k+1}$.

By the procedure described above, it is impossible (we could remove θ_k from \mathcal{C}), so when the algorithm is completed, we have a chain of the form $\theta_0 \rightsquigarrow_\bullet \theta_1 \rightsquigarrow_\bullet \dots \rightsquigarrow_\bullet \theta_\ell = \theta_\perp$. \square

12.4 Normalizing Nihilating Chains

In this section, we prove that negative left-consumption in a nihilating chain can be avoided (without loss of generality). By Proposition 12.4, this will allow us to prove the theorem of complete unsoundness. For that, the notions of quasi-residual of a biposition and of residuals of threads are fundamental and we present them. The main technique to be implemented here was presented from a perspective in Sec. 11.2.

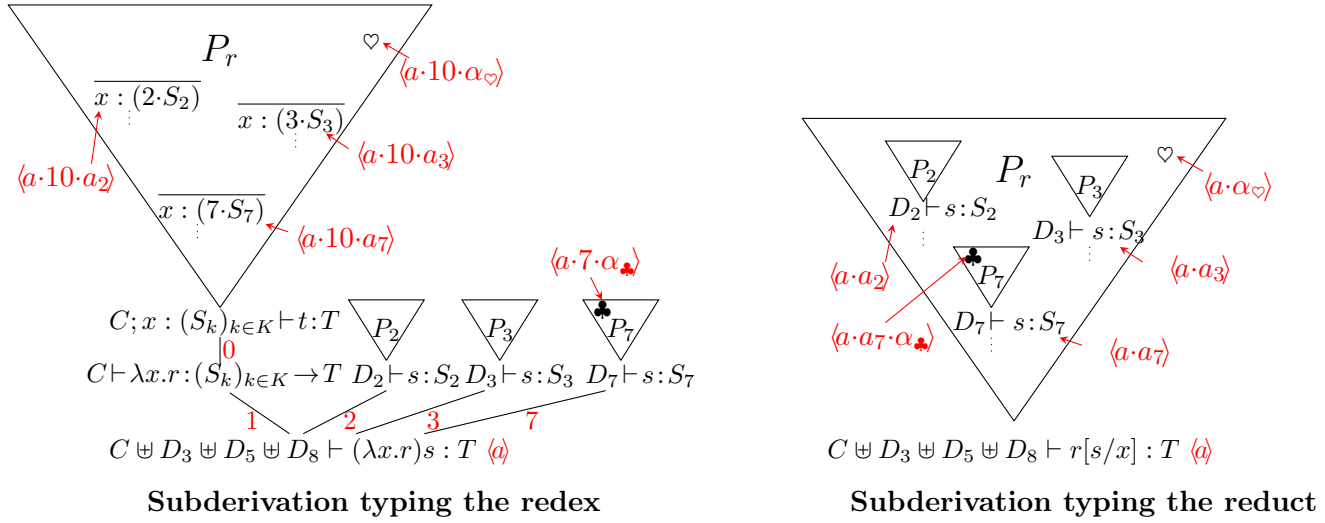


Figure 12.4: Residuals and Quasi-Residuals (copy of Fig. 10.5, p. 218)

Remember from Sec. 10.3.3 that system \mathbf{S} enjoys both subject reduction and expansion, meaning typing is invariant under (anti)reduction in system \mathbf{S} , and that moreover, in system \mathbf{S} , subject reduction is processed in a deterministic way thanks to the tracks (contrary to system \mathcal{R}_0 , Sec. 4.1.2).

The determinism of reduction allowed us to define in Sec. 10.3.5 residuation for some positions and bipoitions. Residuation (Sec. 2.1.5) is a way to describe how positions move or are destroyed during reduction, and the residuation relation is functional for system \mathbf{S} (in contrast with the λ -calculus) because system \mathbf{S} is linear and forbids duplication (Remark 10.4).

We extend now residuation to quasi-residuation for system \mathbf{S} , as it was done for the λ -calculus in Sec. 2.1.5. Quasi-residuation for positions does not preserve the labelling. For that, we reuse Fig. 10.5 which is copied into Fig. 12.4. We also refer to Sec. 10.3.5 for a description of the moves of the symbols.

12.4.1 Quasi-Residuals

In this section, we define residuation for threads and state some of its most important properties.

From Sec. 12.3.3, we know that we only have to escape the case of negative left consumption, to grant that nihilating chains do not exist and thus ensuring that every term is typable (discussion beginning Sec. 12.3). To achieve this purpose, the main tool is a normalizing reduction strategy, called the **collapsing strategy**, which destroys threads that are left-consumed negatively and allows us to build a normal chain from any nihilating chain. This reduction strategy is finite, despite the fact that no form of normalization is not ensured in system \mathbf{S} . But to be formulated, it needs the notion of residual for threads, which can be ensured only if it is well-behaved for bipoitions.

Residuals were defined for most positions, but not for all of them *e.g.*, $a \cdot 1$, that corresponds to the abstraction of the redex, is destroyed during reduction and does not have a residual. When $(a, c) \in \text{bisupp}(P)$ and $a' = \text{Res}_b(a)$ is defined, we set $\text{Res}_b(a, c) = (a', c)$, thus defining the residuals of bipoitions.

The partial function Res_b on bipoitions may be quite naturally extended to a total function QRes_b (standing for **quasi-residual**) from \mathbb{B}^t to $\mathbb{B}' := \mathbb{B}^{t'}$. For instance, the abstraction of the redex, typed with a type of the form $(S_k)_{k \in K} \rightarrow T$, is destroyed, but

the types S_k are still present in the typing of $r[s/x]$ (they occur as types of subterm s).

Formal Definitions of Quasi-Residuals As in Sec. 10.3.5, or what concerns (quasi-)residuals, metavariable a will now denote only positions in \mathbb{A} s.t. $\bar{a} = b$. Metavariables α and c range over \mathbb{N}^* . For instance, $\alpha \neq a$ means that $\bar{\alpha} \neq b$. If $k \in \text{Tr}_\lambda(a)$, a_k is the unique position s.t. $\text{pos}(k) = a \cdot 10 \cdot a_k$ (see a_2, a_3, a_3 in Fig. 12.4). Let us have a look at the positions in t : positions a and $a \cdot 1$ point to the root and the abstraction of the redex, position $a \cdot 10$ points to the root of r , position $a \cdot k$ (with $k \geq 2$) points to the root of s , position $a \cdot 10 \cdot a_k$ (with $k \in \text{Tr}_\lambda(a)$) points to an occurrence of x . Then, those positions a , $a \cdot 1$ and $a \cdot 10 \cdot a_k$, that respectively point to the root, the abstraction or the variable of the redex are considered to be **destroyed** when the redex is fired. The associated bipoositions will not have a proper residual, but we define below their *quasi-residuals*. This is an adaptation of the notion of quasi-reduction for the positions of λ -terms (Sec. 2.1.5).

First, let us remember from Sec. 10.3.5 how the proper residual for positions and bipoositions are defined:

- *Out of the redex:* If $\alpha \not\neq a$, then α is not in the redex. We set $\text{Res}_b(\alpha) = \alpha$.
- *Inside r :* Position $a \cdot 10 \cdot \alpha \in \mathbb{B}$ (paradigm \heartsuit) has a residual (except when $\alpha = a_k$ for some k) and should become $a \cdot \alpha$ after reduction: we set $\text{Res}_b(a \cdot 10 \cdot \alpha) = a \cdot \alpha$ for $\alpha \neq a_k$.
- *Inside some argument derivations:* Assume $k \in \text{Tr}_\lambda(a)$. Argument derivation at $a \cdot k$ will replace **ax**-rule typing at position $a \cdot 10 \cdot a_k$ (which is destroyed). So its position after reduction will be $a \cdot a_k$. More generally, the $a \cdot k \cdot \alpha \in \mathbb{B}$ (paradigm \clubsuit) will be found at $a \cdot a_k \cdot \alpha$ after reduction. We set then $\text{Res}_b(a \cdot k \cdot \alpha) = a \cdot a_k \cdot \alpha$ when $k \in \text{Tr}_\lambda(a)$.
- *Some bipoositions:* We set $\text{Res}_b(\alpha, c) = (\text{Res}_b(\alpha), c)$ when $\text{Res}_b(\alpha)$ is defined.

Now, we define the quasi-residuals for all bipoositions of \mathbb{B} and some positions of \mathbb{A} .

- *Extension of Res_b :* $\text{QRes}_b(a) = \text{Res}_b(a)$ and $\text{QRes}_b(a, c) = (\text{Res}_b(a), c)$ whenever $\text{Res}_b(a)$ is defined.
- *Root of the Redex:* We have $\text{T}(a) = \text{T}(a \cdot 10)$ and, for all $c \in \mathbb{N}^*$, $(a, c) \equiv_{\text{asc}} (a \cdot 10, c)$, so we set $\text{QRes}_b(a) = \text{QRes}_b(a \cdot 10) = a$ and $\text{QRes}_b(a, c) = \text{QRes}_b(a \cdot 10, c) = (a, c)$.
- *Variable of the redex:* Let $k \in \text{Tr}_\lambda(a)$. We have $\text{T}(a \cdot 10 \cdot a_k) = \text{T}(a \cdot k)$ (e.g., $\text{T}(a \cdot 10 \cdot a_2) = \text{T}(a \cdot 2) = S_2$ in Fig. 12.4). So we set $\text{QRes}_b(a \cdot 10 \cdot a_k) = \text{Res}_b(a \cdot k) = a \cdot a_k$.
- *Abstraction of the redex:*
 - *Target of the Arrow Type:* We have $\text{T}(a \cdot 1)|_1 = \text{T}(a \cdot 10)$ and actually, $(a \cdot 1, c) \rightarrow_{\text{asc}} (a \cdot 10, c)$ for all $c \in \mathbb{N}^*$, so we set $\text{QRes}_b(a \cdot 1, c) = \text{QRes}_b(a \cdot 10, c)$.
 - *Source of the Arrow type (1):* If $k \in \text{Tr}_\lambda(a)$, then $\text{T}(a \cdot 1)|_k = \text{T}(a \cdot k)$ (e.g., $\text{T}(a \cdot 1)|_7 = \text{T}(a \cdot 7) = S_7$ in Fig. 12.4) and actually, $(a \cdot 1, k \cdot c) \xrightarrow{a} (a \cdot k, c)$ for all $c \in \mathbb{N}^*$. So we set $\text{QRes}_b(a \cdot 1, k \cdot c) = \text{Res}_b(a \cdot k, c) = (a \cdot a_k, c)$.

- *Source of the Arrow type (2)*: If $k \geq 2$ and $k \notin \text{Tr}_\lambda(a)$, then $(a \cdot 1, k \cdot c) \rightarrow_{\text{pi}} \mathbf{p}_\perp$ and there is no **ax**-rule $\geq a$ typing x using track k . So we say that $(a \cdot 1, k \cdot c)$ is **nihilated** after reduction. We set $\text{QRes}_b(a \cdot 1, k \cdot c) = \mathbf{p}_\perp$.
- *Root of the type*: To ensure Lemma 12.13, it is convenient to set $\text{QRes}_b(a \cdot 1, \varepsilon) = \text{QRes}_b(a \cdot 1, 1) = (a, \varepsilon)$.
- *Nihilated argument derivations*: Assume $k \notin \text{Tr}_\lambda(a)$, $k \geq 2$, then there is no **ax**-rule typing x using axiom track k . So argument P_k is not moved inside r but nihilated after reduction. We then set $\text{QRes}_b(a \cdot k \cdot \alpha, c) = \mathbf{p}_\perp$ and also say that $(a \cdot k \cdot \alpha, c)$ has been **nihilated** after reduction.

Remark 12.6.

- We refer the quasi-residual of $a \cdot 10$ as $\text{QRes}_b(a \cdot 10)$ and not as $\text{Res}_b(a \cdot 10)$ because, if $t(a \cdot 10) = x$, then $a \cdot 10$ has no proper residual.
- Thus, QRes_b is a total function on bipositions and Res_b is a *partial injective* function. Moreover, $t'(\text{QRes}_b(\alpha)) = t(\alpha)$ is not true in general *i.e.* quasi-residuation does not preserve labelling (contrary to Res_b , Lemma 10.3), as in the case of the pure λ -calculus (Sec. 2.1.5).

We also consider the relations $\rightarrow_{\text{asc}}, \rightarrow_{\text{pi}}, \rightarrow_\bullet, \dots$ defined w.r.t. \mathbb{B}' (we do not distinguish them graphically from those of \mathbb{B}) and $[\cdot]'$, an injective function from the set of leaves of \mathbb{A}' to $\mathbb{N} \setminus \{0, 1\}$ naturally defined from $[\cdot]$: we set $[\alpha']' = k$ if there is $\alpha \in \mathbb{A}$ s.t. $t(\alpha) = y \neq x$ and $\text{Res}_b(\alpha) = \alpha'$. We check that $\alpha' \mapsto [\alpha']'$ is still an injective function. We set $\text{pos}'(k) = \alpha'$ if there is α' s.t. $[\alpha']' = k$. Thus, $\text{pos}'(k) = \text{Res}_b(\text{pos}(k))$. Notice that $\text{Res}_b(\alpha) \in \mathbb{A}'$ and $\text{QRes}_b(\mathbf{p}) \in \mathbb{B}'$. We also define asc' , Thr' and $\text{thr}'(\cdot)$.

Relations \rightarrow_{asc} and \rightarrow_{pi} and thus \equiv are compatible with reduction. By case analysis (still guided by Fig. 12.4):

- Assume $\mathbf{p}_1 \rightarrow_{\text{asc}} \mathbf{p}_2$ or $\mathbf{p}_1 \rightarrow_{\text{pi}} \mathbf{p}_2$ and \mathbf{p}_1 is nihilated. Then $\text{QRes}_b(\mathbf{p}_1) = \text{QRes}_b(\mathbf{p}_2) = \mathbf{p}_\perp$. We assume below that \mathbf{p}_1 is not nihilated.
- Assume $\mathbf{p}_1 = (\alpha, c) \rightarrow_{\text{asc}} \mathbf{p}_2$. If $\alpha \neq a, a \cdot 1$, then $\text{QRes}_b(\mathbf{p}_1) \rightarrow_{\text{asc}} \text{QRes}_b(\mathbf{p}_2)$. If $\alpha = a, a \cdot 1$, then $\text{QRes}_b(\mathbf{p}_1) = \text{QRes}_b(\mathbf{p}_2)$.
- If $\mathbf{p}_1 = (\alpha, k \cdot c) \rightarrow_{\text{pi}} \mathbf{p}_2$ with $\mathbf{p}_2 \neq \mathbf{p}_\perp$. If $\alpha \neq a \cdot 1$, then $\text{Res}_b(\mathbf{p}_1) \rightarrow_{\text{pi}} \text{Res}_b(\mathbf{p}_2)$. If $\alpha = a \cdot 1$, then $\text{QRes}_b(\mathbf{p}_1) = \text{QRes}_b(\mathbf{p}_2)$.
- Assume $\mathbf{p} = (\alpha, k \cdot c) \rightarrow_{\text{pi}} \mathbf{p}_\perp$. If $t(\alpha) = \lambda x$, then $\text{QRes}_b(\mathbf{p}) = \mathbf{p}_\perp$. If $t(\alpha) = \lambda y \neq \lambda x$, then $\text{QRes}_b(\mathbf{p}) \rightarrow_{\text{pi}} \mathbf{p}_\perp$.

This entails, by induction on \equiv :

Lemma 12.12. If $\mathbf{p}_1 \equiv \mathbf{p}_2$, then $\text{QRes}_b(\mathbf{p}_1) \equiv \text{QRes}_b(\mathbf{p}_2)$.

This Lemma allows us to define (quasi-)residuals for *threads*. We set $\text{QRes}_b(\theta) = \text{thr}'(\text{QRes}_b(\mathbf{p}))$ for any $\mathbf{p} : \theta$. By case analysis, we have:

- If $\mathbf{p}_1 \rightarrow_\bullet \mathbf{p}_2$ and \mathbf{p}_2 is nihilated, then $\text{QRes}_b(\mathbf{p}_1) = \text{QRes}_b(\mathbf{p}_2) = \mathbf{p}_\perp$. We assume below that \mathbf{p}_1 is not nihilated.
- Assume $\mathbf{p}_1 = (\alpha \cdot 1, k \cdot c) \rightarrow (\alpha \cdot k, c) = \mathbf{p}_2$. If $\alpha \neq a$, then $\text{QRes}_b(\mathbf{p}_1) \rightarrow \text{QRes}_b(\mathbf{p}_2)$ and if $\alpha = a$, then $\text{QRes}_b(\mathbf{p}_1) = \text{QRes}_b(\mathbf{p}_2)$.

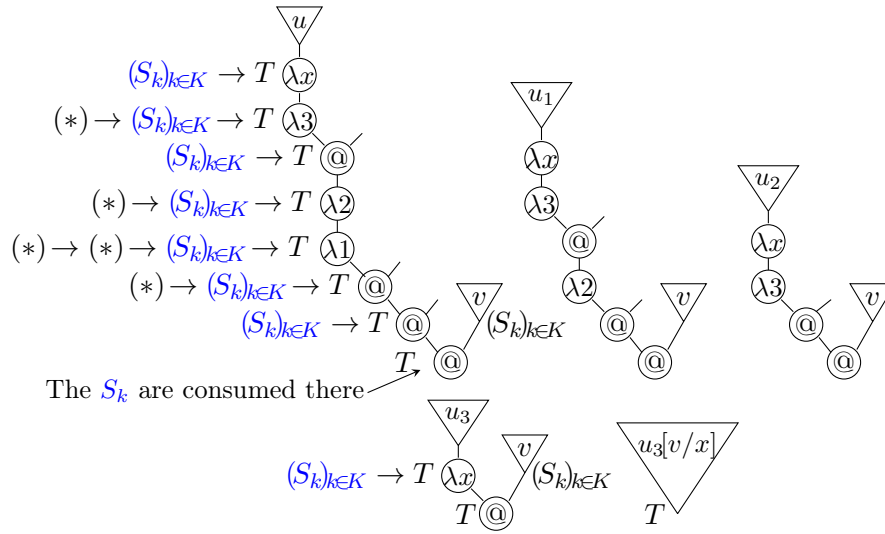


Figure 12.5: Collapsing a Redex Tower

- Assume $p_1 = (\alpha \cdot k, c) \leftarrow (\alpha \cdot 1, k \cdot c) = p_2$. If $\alpha \neq a$, then $\text{QRes}_b(p_1) \leftarrow \text{QRes}_b(p_2)$ and if $\alpha = a$, then $\text{QRes}_b(p_1) = \text{QRes}_b(p_2)$.
- Assume $p_1 \rightarrow_{t_1} p_2 = (\alpha, c)$. If $p_2 \neq (a \cdot 1, \varepsilon)$, $\text{QRes}_b(p_1) \rightarrow_{t_1} \text{QRes}_b(p_2)$. If $p_2 = (a \cdot 1, \varepsilon)$, $\text{QRes}_b(p_1) = \text{QRes}_b(p_2) = (a, \varepsilon)$.
- Assume $p_1 \rightarrow_{t_2} p_2 = (\alpha, c \cdot 1)$. If $p_2 \neq (a \cdot 1, 1)$, then $\text{QRes}_b(p_1) \rightarrow_{t_2} \text{QRes}_b(p_2)$. If $p_2 = (a \cdot 1, 1)$, then $\text{QRes}_b(p_1) \rightarrow_{\text{down}} \text{QRes}_b(p_2) = (a, \varepsilon)$.
- Assume $p_1 = (\alpha, \varepsilon) \rightarrow_{\text{abs}} (\alpha, 1) = p_2$. If $\alpha \neq a \cdot 1$, then $\text{QRes}_b(p_1) \rightarrow_{\text{abs}} \text{QRes}_b(p_2)$. If $\alpha = a \cdot 1$, then $\text{QRes}_b(p_1) = \text{QRes}_b(p_2) = (a, \varepsilon)$.
- Assume $p_1 \rightarrow_{\text{down}} p_2$. Then $\text{QRes}_b(p_1) \rightarrow_{\text{down}} \text{QRes}_b(p_2)$.

This yields:

Lemma 12.13. Let $\theta_1, \theta_2 \in \text{Thr}$. We set $\theta'_i = \text{QRes}_b(\theta_i)$.

- If $\theta_1 \tilde{\rightarrow} \theta_2$, then $\theta'_1 \tilde{\rightarrow} \theta'_2$ or $\theta'_1 = \theta'_2$.
- If $\theta_1 \tilde{\rightarrow}_{t_1} \theta_2$, then $\theta'_1 \tilde{\rightarrow}_{t_1} \theta'_2$ or $\theta'_1 = \theta'_2$.
- If $\theta_1 \tilde{\rightarrow}_{t_2} \theta_2$, then $\theta'_1 \tilde{\rightarrow}_{t_2} \theta'_2$, $\theta'_1 \tilde{\rightarrow}_{\text{down}} \theta'_2$ or $\theta'_1 = \theta'_2$.
- If $\theta_1 \tilde{\rightarrow}_{\text{abs}} \theta_2$, then $\theta'_1 \tilde{\rightarrow}_{\text{abs}} \theta'_2$ or $\theta'_1 = \theta'_2$.
- If $\theta_1 \tilde{\rightarrow}_{\text{down}} \theta_2$, then $\theta'_1 \tilde{\rightarrow}_{\text{down}} \theta'_2$ or $\theta'_1 = \theta'_2$.

Besides, $\text{Res}_b(\theta_\varepsilon) = \theta_\varepsilon$ and $\text{Res}_b(\theta_\perp) = \theta_\perp$ as expected, so the above lemma implies that, if there is a nihilating chain for t of length m , then there is one for t' of length $\leq m$ (with $t \rightarrow^* t'$).

12.4.2 The Collapsing Strategy

We explain now how to *normalize* a chain *i.e.* discard negative left-consumption (which was more informally explained in Sec. 11.2.1). This will allow us to use Proposition 12.5 to finally conclude that nihilating chains do not exist.

The idea is that if $\theta_1 : \mathbf{p}_1^\ominus \xrightarrow{a} \theta_2$, then either $t|_a$ is a redex and, by definition of Res_b , we have $\text{Res}_b(\theta_1) = \text{Res}_b(\theta_2)$ (*i.e.* θ_1 and θ_2 are **collapsed** by the reduction step) or there is a redex between \mathbf{p}_1 and a . When we reduce it, the relative height of \mathbf{p}_1 will decrease. More precisely, the 2nd case is associated to the notion of **redex tower** (see also Sec. 11.2.1 for a high-level presentation). In Fig. 12.5, we have represented a redex tower of *height* 7, meaning that the sequence type $(S_k)_{k \in K}$ called by the abstraction λx is left-consumed on its prefix of rank 7. It is very similar to that of Fig. 11.4 p. 247. The types of some subterms are indicated on their left (except for v) *e.g.*, subterm $\lambda x.u$ has type $(S_k)_{k \in K} \rightarrow T$. We write $\lambda 1, \lambda 2, \lambda 3$ for $\lambda x_1, \lambda x_2, \lambda x_3$ and $(*)$ for sequence types of no matter.

At each step of reduction, we reduce the height of consumption by 2. Thus, in a finite number of steps, we get a proper redex and we collapse θ_1 on θ_2 . This is illustrated by Fig 12.5 or Fig. 11.5. We now formalize this argument.

Formalizing the Collapsing Strategy Assume then $\theta_1 : \mathbf{p}_1 = (\alpha \cdot 1, k \cdot c)^\ominus \xrightarrow{\alpha} \theta_2$. Then $\text{Asc}(\theta_1) = (\alpha_*, k \cdot c)$ and $t(a_*) = \lambda x$ for some α_*, x . We set $h = |\alpha_*| - |\alpha|$ and we call h the **height** of the consumption. By Lemma 12.2, for $1 \leq i \leq h$, we may write $\mathbf{p}_i = (\alpha_i, 1^{j(i-1)} \cdot k \cdot c)$ for $\text{asc}^{i-1}(\theta_1)$ where $\alpha_{i+1} = \alpha_i \cdot k_i$ for a $k_i \in \{0, 1\}$.

If $h = 1$, then $\alpha_* = \alpha \cdot 1$ and we set $b = \bar{\alpha}$ so that $t|_b$ is a redex and $\text{QRes}_b(\mathbf{p}_1) = \text{QRes}_b(\mathbf{p}_2)$. Thus, $\text{Res}_b(\theta_1) = \text{Res}_b(\theta_2)$.

Assume now $h > 1$. Then let $1 \leq i_0 \leq h - 1$ be maximal s.t. $t(a_{i_0}) = @$. Actually, $i_0 \leq h - 2$ (if $i_0 = h - 1$, $t|_{\alpha_{h-1}}$ is a redex, so $\alpha_{h-1} = \alpha$ *i.e.* $h = 1$).

We set $b = \bar{\alpha^{i_0}}$ so that $t|_b$ is a redex. We set $\mathbf{p}'_i = (\alpha'_i, c'_i) = \text{QRes}_b(\mathbf{p}_i)$ (so that $\mathbf{p}'_{i_0} = \mathbf{p}'_{i_0+1} = \mathbf{p}'_{i_0+2}$). By induction on i , if $1 \leq i \leq i_0$, then $(\text{asc}')^{i-1}(\mathbf{p}_1) = \mathbf{p}'_i$ and $|\alpha'_i| - |\alpha| = i$ and if $i_0 < i \leq h - 2$, then $(\text{asc}')^{i-1}(\mathbf{p}_1) = \mathbf{p}'_{i+2}$ and $|\alpha'_{i+2}| - |\alpha| = i$. Thus, $(\text{asc}')^{h-2}(\mathbf{p}_1) = (\alpha'_h, k \cdot c)$ and $|\alpha'_h| - |\alpha| = h - 2$. Since $t'(\alpha'_{h-2}) = t(\alpha_h)$ (proper residual), $\text{Asc}'(\mathbf{p}'_1) = \mathbf{p}'_h$ and $\text{Pol}'(\mathbf{p}'_1) = \ominus$. Thus, $\theta'_1 = \text{Res}_b(\theta_1)^\ominus \rightarrow \theta'_2 = \text{Res}_b(\theta_2)$, but the height has decreased by 2. In a finite number of steps, we equate then θ_1 and θ_2 .

We extend the notation Res_b for finite reduction sequences, so that we get this lemma (whose informal version is Observation 11.9, p. 247):

Lemma 12.14. If $\theta_1^\ominus \xrightarrow{\alpha} \theta_2$, then there is a reduction strategy rs such that $\text{Res}_{\text{rs}}(\theta_1) = \text{Res}_{\text{rs}}(\theta_2)$.

This Lemma, along with the observation concluding Sec. 12.4.1, yields:

Proposition 12.6. There is a reduction strategy, that we call the **collapsing strategy**, producing a normal chain from any nihilating chain.

12.4.3 Redex Towers

The notion of redex tower, illustrated by Fig. 12.5 and 11.4, can be formalized. We will use it again in the course of proving Theorem 12.2.

First, we can easily extend the notion of ascendance to positions of \mathbb{A} . So we set, for all $a \in \mathbb{A}$, $\mathbf{asc}(a) = a \cdot 1$ if $t(a) = @$, $\mathbf{asc}(a) = a \cdot 0$ if $t(a) = \lambda x$ ($\mathbf{asc}(a)$ is undefined when $t(a) = x$). We also define $\mathbf{Asc}(a)$ as $\mathbf{asc}^i(a)$ where i is maximal such that this expression is defined.

Now, let $a \in \mathbb{A}$. We define first the **consumption degree** $\mathbf{cdeg}_a(a')$ of some extensions a' of a . Intuitively, to compute $\mathbf{cdeg}_a(a')$, we visit the ascendants of a , one after another. The postfix 1 means that we visit an application left-hand side and we increment $\mathbf{cdeg}_a(a')$, and the postfix 0 means that we visit an abstraction and we decrement $\mathbf{cdeg}_a(a')$.

For instance, in Fig. 12.5, if a is the address of the root of the redex tower, we have $\mathbf{cdeg}_a(a \cdot 1) = 1$, $\mathbf{cdeg}_a(a \cdot 1^2) = 2$, $\mathbf{cdeg}_a(a \cdot 1^3) = 3$, $\mathbf{cdeg}_a(a \cdot 1^3 \cdot 0) = 2$, $\mathbf{cdeg}_a(a \cdot 1^3 \cdot 0^2) = 1$, $\mathbf{cdeg}_a(a \cdot 1^3 \cdot 0^2 \cdot 1) = 2$, $\mathbf{cdeg}_a(a \cdot 1^3 \cdot 0^2 \cdot 1 \cdot 0) = 1$, $\mathbf{cdeg}_a(a \cdot 1^3 \cdot 0^2 \cdot 1 \cdot 0^2) = 0$ where $t(a \cdot 1^3 \cdot 0^2 \cdot 1 \cdot 0^2) = \lambda x$. Thus, the consumption degree reaches 0 for the first time when the abstraction of the redex tower is reached.

So, we define by induction:

- $\mathbf{cdeg}_a(a) = 0$.
- If $\mathbf{cdeg}_a(a') \geq 0$, then $\mathbf{cdeg}_a(a' \cdot 1) = \mathbf{cdeg}_a(a') + 1$ (resp. $\mathbf{cdeg}_a(a' \cdot 0) = \mathbf{cdeg}_a(a') - 1$) when $t(a) = @$ (resp. $t(a) = \lambda x$).
- If $\mathbf{cdeg}_a(a') = -1$, then $\mathbf{cdeg}_a(a' \cdot 1)$ (resp. $\mathbf{cdeg}_a(a' \cdot 0)$) is undefined when $t(a') = @$ (resp. $t(a') = \lambda x$).

As it can also be seen with Fig. 12.5, if $d = \mathbf{cdeg}_a(a') \geq a$, then $\mathbf{T}(a')$ is an arrow type ending with $\mathbf{T}(a)$ and, more precisely, $\mathbf{T}(a)$ is preceded in $\mathbf{T}(a')$ by exactly d arrows.

Definition 12.5. Let t a term and $b \in \mathbf{supp}(t)$ such that $t(b) = @$. We say there is a **redex tower** of height $h \geq 1$ if

- $\mathbf{cdeg}_b(\mathbf{asc}^h(b)) = 0$
- For all $1 \leq i \leq h - 1$, $\mathbf{cdeg}_b(\mathbf{asc}^i(b)) > 0$.

The extension below will also be useful to prove Theorem 12.2. Inductively, a redex tower sequence is a redex tower whose left-hand side (the part over the abstraction) is itself a redex tower sequence. We can then collapse one redex tower after the other. See Sec. 11.3 for a high-level presentation and in particular, Fig. 11.6.

Definition 12.6. Let t a term and $b \in \mathbf{supp}(t)$ such that $t(b) = @$. We say there is a **redex tower sequence** of height $h \geq 1$ at position if $\mathbf{cdeg}_b(\mathbf{asc}^h(b)) = 0$.

For the proof of Theorem 12.2 (order discrimination), we need to describe the thread θ_ε . So, for $n \geq 0$, we write $\varepsilon(n) = \mathbf{asc}^n(\varepsilon)$, $\mathbf{p}_\varepsilon(n) = \mathbf{asc}^n(\varepsilon, \varepsilon)$ and $\mathbf{i}(n)$ the natural number such that $\mathbf{p}_\varepsilon(n) = (a(n), 1^{\mathbf{i}(n)})$ when they are defined (if $\mathbf{p}_\varepsilon(n)$ is defined, then $a(n)$ is defined; the form $1^{\mathbf{i}(n)}$ is justified by Lemma 12.2).

The following lemma is just the formalization of Observation 11.10, p. 249:

Lemma 12.15. Let t be a term and $[\cdot] : \mathbb{N}^* \rightarrow \mathbb{N} \setminus \{0, 1\}$ an injection and all the associated notations (\mathbf{asc} , \mathbf{Pol} , $\vec{\rightarrow}_\bullet$, etc).

If $\mathbf{Pol}(\varepsilon, \varepsilon) = \ominus$, then there exist $a \in \mathbb{A}$ and a finite segment \mathbf{rs} of the head reduction strategy such that $\mathbf{asc}(\varepsilon, \varepsilon) = (a, \varepsilon)$, $t \xrightarrow{\mathbf{rs}} \lambda x.t'$ and $\mathbf{Res}_{\mathbf{rs}}(a, \varepsilon) = (\varepsilon, \varepsilon)$.

Proof. Assume $\text{Pol}(\varepsilon, \varepsilon) = \ominus$. By Lemma 12.2, $\text{Pol}(\varepsilon, \varepsilon) = \ominus$ means that θ_ε only occurs negatively (an ascendant of $(\varepsilon, \varepsilon)$ is of the form $(a', 1^i)$ so cannot be on the left-hand side of \rightarrow_{pi}). In that case, if $(a, c) = \text{Asc}(\varepsilon, \varepsilon)$, we then have $t(a) = \lambda x$ (negativity) and $c = \varepsilon$ (if we had $c = 1^i$ with $i \geq 1$, then (a, c) would have an ascendant).

The claim about the head reduction strategy is proved using exactly the same technique as in Sec. 12.4.2: more precisely, we destroy the highest redex tower sequence (Definition 12.6) rooted at ε , which yields a reduct of t that is an abstraction $\lambda x.t'$. \square

12.5 Applications

We can now prove the complete unsoundness of systems \mathbf{S} and \mathcal{R} , using the residuation of threads, the collapsing strategy and the non-existence of normal threads, which is ensured by the Interaction Lemmas. But the collapsing strategy can also be applied to prove that systems \mathbf{S} and \mathcal{R} are order discriminating.

Theorem 12.1. The relevant intersection type system \mathcal{R} , featuring coinductive types, is completely unsound: every λ -term is typable in \mathcal{R} .

Proof. By Propositions 12.2, 12.4, 12.5 and 12.6.

- By Proposition 12.5, there is no normal nihilating chain.
- By using the collapsing strategy (Proposition 12.2), if nihilating chains existed, so would the normal nihilating chains. Thus, nihilating chains do not exist.
- By Proposition 12.6, the non-existence of nihilating chains entails the complete unsoundness of system \mathbf{S} .
- By Proposition 12.2, the complete unsoundness of system \mathbf{S} entails that of \mathcal{R} .

\square

Now that we know that every term may be typed, we can study some semantical aspects of system \mathcal{R} (and \mathcal{D}_w), namely, how it may be order-sensitive.

Order-Discrimination

The goal of this section is to prove:

Proposition 12.7. Let t be a zero term and o a type variable, then there is context C such that $C \vdash t : o$ is derivable in system \mathbf{S} .

Exactly as in Sec. 3.4.5, this proposition easily entails:

Theorem 12.2. Let t be a term of order n . Then there is a context Γ and a type τ of order n (see Sec. 12.1.3) such that $\Gamma \vdash t : \tau$ is derivable in system \mathcal{R} (and there is a context Γ and type B of order n s.t. $\Gamma \vdash t : B$ is derivable in system \mathcal{D}_w).

Proof. Let t be a term of order $\geq n$ and (by Theorem 12.1) Π a \mathcal{R} -derivation concluding with $\Gamma \vdash t : \tau$.

There is a term $t' = \lambda x_1 \dots \lambda x_n.t'_0$ such that $t \rightarrow_\beta^* t'$. By subject reduction, there is a derivation Π' concluding with $\Gamma \vdash \lambda x_1 \dots \lambda x_n.t'_0 : \tau$. By the **abs**-rule, τ must be of

order $\geq n$. This implies that the statement is true for terms of infinite order.

Now, assume that t is of order $n < \infty$. There is a term $t' = \lambda x_1 \dots \lambda x_n. t'_0$ such that $t \rightarrow_{\beta}^* t'$ and t'_0 is of order 0.

Let $o \in \mathcal{O}$. By Propositions 12.7 and 12.2, there is a \mathcal{R} -derivation Π'_0 concluding with $\Gamma_0 \vdash t'_0 : o$. Using n **abs**-rules, we get a derivation Π' concluding with a judgment of the form $\Gamma \vdash \lambda x_1 \dots \lambda x_n. t'_0 : B$, with B of order n .

By subject expansion, there is a derivation Π concluding with $\Gamma \vdash t : B$. The statement is thus proved. \square

Now, let us prove Proposition 12.7. For that, we consider t be a term such that $\theta_{\varepsilon} \xrightarrow{\bullet}^* \mathbf{thr}(\varepsilon, 1)$ *i.e.* s.t. $(\varepsilon, 1) \in \mathbb{B}_{\min}$ (see Corollary 12.1), which implies that the type of t cannot be a type variable by the proof of this same corollary. We prove that t is of order ≥ 1 , which is enough to conclude.

We then consider a λ -chain *i.e.* a chain of the form $\theta_{\varepsilon} = \theta_0 \xrightarrow{\bullet} \dots \xrightarrow{\bullet} \theta_m = \mathbf{thr}(\varepsilon, 1)$, of minimal length. The notion of normal chains extends to λ -chains and by the collapsing strategy, we can replace t by a reduct t' such that the considered chain is normal.

Assume $\mathbf{Pol}(\varepsilon, \varepsilon) = \ominus$. Then, by Lemma 12.15, there is a reduct of t' of the form $\lambda x. t''$. So $t \rightarrow_{\beta}^* \lambda x. t''$ and we may conclude.

We prove now *ad absurdum* that $\mathbf{Pol}(\varepsilon, \varepsilon) = \oplus$ is impossible. So we assume $\mathbf{Pol}(\varepsilon, \varepsilon) = \oplus$ for the remainder of the proof.

We need to consider $B_0 = \{(a, c) \in \mathbb{A} \mid \theta_{\varepsilon} \xrightarrow{\bullet}^* \mathbf{thr}(a, c)\}$. To obtain a contradiction, we prove that B_0 is closed under normal chains but that $(\varepsilon, 1)$ is not in B_0 .

We have to consider two different subcases: when θ_{ε} also has some negative occurrences and when it does not. In both cases, we should prove that no relation allows exiting B_0 (except $\ominus \rightarrow$), reaching thus a contradiction.

However, the second case may be seen as a particular case of the first one and may be skipped. So we assume now that there is $\mathbf{p} \equiv (\varepsilon, \varepsilon)$ such that $\mathbf{Pol}(\mathbf{p}) = \ominus$.

This implies, with h such that $\mathbf{Asc}(\varepsilon, \varepsilon) = \mathbf{asc}^h(\varepsilon, \varepsilon)$, that $t(\varepsilon(h)) = x$ and there is $0 \leq h_{\lambda} < h$ such that $t(\varepsilon(h_{\lambda})) = \lambda x$.

Thus, $(\varepsilon(h_{\lambda}), k \cdot 1^{i(h)}) \rightarrow_{\mathbf{p}_i} \mathbf{p}_h = \mathbf{Asc}(\varepsilon, \varepsilon)$ with $k = \lfloor \varepsilon(h) \rfloor$. For all $0 \leq h_{\lambda}$ such that it is defined, we write $\mathbf{p}'_{\varepsilon}(n) = (\varepsilon(n), 1^{j(n) \cdot k \cdot i(h)})$ for the unique $\mathbf{p}'_{\varepsilon}(n) \in \mathbb{B}$ such that $\mathbf{asc}^{h_{\lambda} - n}(\mathbf{p}'_{\varepsilon}(n)) = (\varepsilon(h_{\lambda}), k \cdot 1^{i(h)})$. The form $(\varepsilon(n), 1^{j(n) \cdot k \cdot i(h)})$ is justified by Lemma 12.2. We write h_0 for the minimal integer such that $\mathbf{p}'_{\varepsilon}(h_0)$ is defined.

Notice that $j(h_{\lambda}) = 0 < i(h_{\lambda})$: if we had $i(h_{\lambda}) = 0$, we could prove that $\mathbf{Pol}(\varepsilon, \varepsilon) = \ominus$, since this implies $\mathbf{asc}^{h_{\lambda}}(\varepsilon, \varepsilon) = (\varepsilon(h_{\lambda}), \varepsilon)$. Moreover, $t(\varepsilon(h_{\lambda})) = \lambda x$, so that $\mathbf{asc}^{n+1}(\varepsilon, \varepsilon)$ is not defined *i.e.* $\mathbf{Asc}(\varepsilon, \varepsilon) = (\varepsilon(h_{\lambda}), \varepsilon)$ and $\mathbf{Pol}(\varepsilon, \varepsilon) = \ominus$, which contradicts our assumption.

From $j(h_{\lambda}) < i(h_{\lambda})$, by an easy induction, we deduce that, for all $h_0 \leq n \leq h_{\lambda}$, $j(n) < i(n)$.

Lemma 12.16. Assume that $\mathbf{Pol}(\varepsilon, \varepsilon) = \oplus$. Then $B_0 = \{(\varepsilon(n), 1^i) \mid n \leq h, i \leq i(n)\} \cup \{(\varepsilon(n), 1^{j(n) \cdot k \cdot i} \mid h_{\lambda} \leq n \leq h, i \leq i(h)\}$.

Moreover, B_0 is closed under $\rightarrow_{\mathbf{down}}$, $\rightarrow_{\mathbf{abs}}$, $\rightarrow_{\mathbf{t}_2}$ and B_0 has an empty intersection with $\mathbf{dom}(\oplus \rightarrow) \cup \mathbf{codom}(\rightarrow)$

Proof. Let $B'_0 = \{(\varepsilon(n), 1^i) \mid n \leq h, i \leq i\}$ and $\{(\varepsilon(n), 1^{j(n) \cdot k \cdot i} \mid h_{\lambda} \leq n \leq h, i \leq i(h)\}$

Clearly, $B'_0 \cup B''_0 \subseteq B_0$. We study $B'_0 \cup B''_0$ and prove that $B'_0 \cup B''_0 = B_0$.

We notice first that if $\mathbf{p} = (a, c) \in B'_0 \cup B''_0$ is such that c contains an argument track, then $\mathbf{p} \in B''_0$ and $\text{Pol}(\mathbf{p}) = \ominus$.

- $B'_0 \cup B''_0$ is clearly closed under \equiv and $\rightarrow_{\text{down}}$.
- Closure under $\rightarrow_{\mathbf{t1}}$: the only problematic case is $\mathbf{p}_0 = (\varepsilon(n), 1^{j(n) \cdot j} \cdot k) \rightarrow_{\mathbf{t1}} (\varepsilon(n), 1^{j(n) \cdot j} \cdot k)$. Since $\mathbf{p}_0 \in B''_0$, we must show that $\mathbf{p} \in B'_0$. This is guaranteed by $j(n) < i(n)$.
- $\mathbf{p}_0 = (a_0, c_0)^\oplus \rightarrow \mathbf{p}$ with $\mathbf{p}_0 \in B'_0 \cup B''_0$, because $\mathbf{p}_0 \rightarrow \mathbf{p}$ implies that c_0 holds an argument track, and thus, as notice above, that $\text{Pol}(\mathbf{p}_0) = \ominus$.
- $\mathbf{p} \rightarrow (\varepsilon(n), c_0) \in B'_0 \cup B''_0$ is impossible, because $\varepsilon(n) \in \{0, 1\}^*$ (no argument track in $\varepsilon(n)$).
- Closure under $\rightarrow_{\mathbf{t2}}$: assume that $\mathbf{p}_0 = (\varepsilon(n), 1^{j(n)} \cdot k) \rightarrow_{\mathbf{t2}} (\varepsilon(n), 1^{j(n)+1})$. Since $\mathbf{p}_0 \in B''_0$, we must show that $\mathbf{p} \in B'_0$. This is guaranteed by $j(n) < i(n)$, so that $j(n) + 1 \leq i(n)$.
- Assume $\mathbf{p}_0 = (\varepsilon(n), \varepsilon) \rightarrow_{\text{abs}} (\varepsilon(n), 1) = \mathbf{p}$ so that \mathbf{p}_0 in B'_0 . In order to have $\mathbf{p} \in B'_0$, it is enough to prove that $i(n) \geq 1$. If we had $i(n) = 0$, then we could prove that $\text{Pol}(\varepsilon, \varepsilon) = \ominus$ as is the proof of $i(h_\lambda) > 0$ above.

By point 1, $B'_0 \cup B''_0$ is stable under \equiv . Moreover, $B'_0 \cup B''_0$ contain θ_ε and is stable under $\rightarrow_{\mathbf{t1}}$ by point 2. So $B'_0 \cup B''_0 \supseteq B_0$. Thus, $B_0 = B'_0 \cup B''_0$ and the Lemma is proven. \square

By the Lemma, $(\varepsilon, 1) \notin B_0$, so let k be minimal such that $\theta_k \notin B_0$. We are now interested in the relation $\theta_k \xrightarrow{\bullet} \theta_{k+1}$ in the chain. According to the Lemma, this cannot be $\xrightarrow{\mathbf{t2}}$, $\xrightarrow{\text{down}}$, $^\oplus \xrightarrow{}$, $\xleftarrow{}$ or $\xrightarrow{\text{abs}}$. Contradiction since the chain is normal.

Thus, $\text{Pol}(\varepsilon, \varepsilon) = \ominus$, and we have proven above that this implied that t was of order ≥ 1 . This concludes the proof of Proposition 12.7.

Conclusion We proved that every term is typable in a reasonable relevant intersection type system (Theorem 12.1). If we take the typing rules of \mathbf{S} *coinductively*, we can also type every infinitary λ -term [34, 57].

Derivations of system \mathbf{S} collapse on system \mathcal{R} (Sec. 12.1.4), the coinductive version of Gardner and de Carvalho's system \mathcal{R}_0 [22, 43]. Thanks to subject reduction and expansion, this yields a **relational model of pure λ -calculus** [17] (finite or infinite) in which, by Theorem 12.1, no term has a trivial denotation, including the mute terms. This model is thus **non-sensible** [11] since it does not equate all the non-head normalizing terms (*e.g.*, Ω and $\lambda x.\Omega$ of respective order 0 and 1) by Theorem 12.2.

We presented a first semantical result about this model (Theorem 12.2), but its equational theory has yet to be studied. According to the same theorem, this model equates all the closed zero terms. It then differs both from the non-sensible model of Berarducci trees and that of Lévy-Longo trees, respectively related to Λ^{111} and Λ^{001} in [57] (see Sec. 9.3). This work may suggest a new notion of tree, that could shed some light on Open Problem # 18 of TLCA (the problem of finding trees related to various contextual equivalences).

Moreover, the implications of the complete unsoundness of system \mathcal{R} on the semantical level remain to be understood: the tools could be of some use to compare coinductive or recursive type systems *before* they are endowed with some validity or guard condition,

or maybe to build other models of pure λ -calculus, for instance, to get some semantical proof of the *easiness* [56] of sets of mute terms, as in [15].

One may also wonder whether system **S** satisfies infinitary subject expansion (subject reduction is easy *cf.* Sec. 10.4.4). We know that it is the case with the approximability condition (Proposition 10.7), but the proof of this property deeply relies on approximations, so that there is no obvious method to answer this question yet.

Chapter 13

The Surjectivity of the Collapse of Sequential Intersection Types

In this chapter, we present the last contribution of this thesis: every (coinductive) derivation based on multiset intersection can be represented by a derivation based on sequential intersection. In other words, we prove that every derivation of system \mathcal{R} (the coinductive counterpart of system \mathcal{R}_0 from Sec. 3.2.4) is the collapse of a derivation system \mathbf{S} (Sec. 10.2). Moreover, we show that we can endow inside system \mathbf{S} any sequence of reduction choices (Sec. 4.1.2) of system \mathcal{R} . In particular, this proves that working with a syntax directed and deterministic framework does not cause¹ any loss of generality. This also allows studying the model suggested by Chapter 12 only through system \mathbf{S} .

In the λ -calculus, termination is called normalization and has many definitions (Sec. 2.2). As we saw in Chapter 3, intersection types provide nice characterizations of normalization, along with operational properties related for instance to some reduction strategy (*e.g.*, a term t is head normalizing iff the head reduction strategy terminates on t). See in particular Sec. 3.3.1.

Remembering Sec. 3.1.2, the first intersection type systems (introduced by Coppo and Dezani) featured idempotent intersection operators, but Gardner and de Carvalho [22, 43] provided a new characterization of the set of HN terms by means of a type system \mathcal{R}_0 , which resorted to *non-idempotent* intersection types. This framework allows us to replace Tait’s Realizability Argument (presented in Sec. 4.3) – used to prove the implication “Typable \Rightarrow Normalizing” – by a considerably simpler, arithmetical one (see *e.g.*, the proof of Proposition 3.8). The initial version of system \mathcal{R}_0 , denoted \mathcal{G} , represents intersection types with lists of the form $A_1 \wedge \dots \wedge A_n$ and features an explicit permutation rule. However, this permutation rule, which burdens the derivations, can be discarded by representing intersection types with *multisets*. We thus obtain a syntax directed presentation of the non-idempotent system \mathcal{R}_0 (Sec. 3.2.1 and 3.2.4) *e.g.*, an intersection type is now represented by a multiset type $[\sigma, \tau, \sigma]$ (with $[\sigma, \tau, \sigma] = [\sigma, \sigma, \tau] \neq [\sigma, \tau]$).

In Chapter 10, we have investigated a type-theoretical characterization of weak normalization in Λ^{001} , an infinitary λ -calculus (presented in Sec. 9.3.2) which was introduced in [57]. The finitary type system \mathcal{R}_0 can be given an infinitary variant \mathcal{R} by taking its rules *coinductively* (instead of *inductively*) and allowing multisets to be infinite. We may notice \mathcal{R} allows unsound infinite derivation *e.g.*, some non-head normalizing terms are typable in \mathcal{R} (*e.g.*, $\Omega = \Delta \Delta$ can be easily typed in \mathcal{R} , see Appendix A.1). This ob-

¹See the table of Sec. 4.1.3 summarizing the principal features of various intersection type systems.

ervation suggested using a validity criterion relying on the notion of *approximability* to discard unsound proofs. However, we observe in Sec. 10.3.4 that this notion of approximability could not be formulated in \mathcal{R} , roughly because it is not possible to distinguish two occurrences of the same type in a multiset (see next section). This led us to resort to **rigid** constructions.

For that, we introduced system \mathbf{S} , which is also syntax directed but in which *multisets* of types are coinductively replaced by *families* of types indexed by (non necessarily consecutive) integers. Those families are called **sequences** and those integers **tracks** (Sec. 10.2.1). Indeed, tracks allow tracking (recall p. 38).

To be equal, two \mathbf{S} -types need to be *syntactically* equal – let us informally say that the equality is **tight** in \mathbf{S} e.g., $(2 \cdot S, 3 \cdot T, 8 \cdot S) \neq (2 \cdot S, 3 \cdot T, 9 \cdot S)$.

In contrast, the order of enumeration of the elements of a multiset does not matter ($[\sigma, \sigma, \tau] = [\sigma, \tau, \sigma]$): let us say the equality between multisets is **loose**. Thus, in system \mathbf{S} , types and contexts are very low-level and, as it turns out, the application typing rule can be used only in case of tight equality. Indeed, the **app**-rule of system \mathbf{S} , given in Sec. 10.2.3, can be restated as follows:

$$\frac{C \vdash t : (S_k)_{k \in K} \rightarrow T \quad (D_k \vdash u : S'_k)_{k \in K'} \quad (S_k)_{k \in K} = (S'_k)_{k \in K'}}{C \uplus_{k \in K} D_k \vdash t u : T} \text{app}$$

On the other hand, the **app**-rule of system \mathcal{R}_0 (Sec. 3.2.4) corresponds to:

$$\frac{\Gamma \vdash t : [\sigma_i]_{i \in I} \rightarrow \tau \quad (\Delta_i \vdash u : \sigma'_i)_{i \in I'} \quad [\sigma_i]_{i \in I} = [\sigma'_i]_{i \in I'}}{\Gamma + (+_{i \in I'} \Delta_i) \vdash t u : \tau} \text{app}$$

Thus, the **app**-rule in system \mathcal{R}_0 is based on multiset equality.

The second use that we made of system \mathbf{S} is proving that *every* term was typable in the infinite system \mathcal{R} (Chapter 12). This is not an obvious statement as for other infinitary type systems, because system \mathcal{R} is *relevant* (weakening is forbidden) and there is no trivial method to type a non-head normalizing term. We could not reason directly on system \mathcal{R} because we needed to describe the support of a prospective derivation of a given term t and those of the types nested in the potential derivation *before* we decorated them with labels (e.g., type variables or arrows). But with multiset constructions, it is impossible to see the support of a derivation as a set of *pointers*, whereas it is very natural with sequential constructions. This work provides a new relational model [17] for pure λ -calculus (that we also refer to as \mathcal{R}) that is able to capture semantical information about any *non-head* normalizing term, as their *order* by Theorem 12.2 (Definition 2.8: the order of a term t is the *supremal* n such that $t \rightarrow^* \lambda x_1 \dots x_n. t'$ for some t').

Reduction Choices

A type system enjoys subject reduction when typing is stable under reduction (if a term is typable, any of its reduct will be typable with the same type and context). Syntax directed intersection type systems are usually not *deterministic* regarding subject reduction: if $t \rightarrow t'$ and Π is a derivation typing $t = (\lambda x.r)s$, then, the proof of the subject reduction property can yield several derivations Π' typing $t' = r[s/x]$. For this reason, we say there are different *reduction choices*.

For instance, the type system \mathcal{R} is syntax directed but not deterministic (Sec. 4.1.2: when we reduce t to t' , there are several natural ways to produce a derivation Π' . This is possible as soon as the variable x has been assigned several times a same type σ . In

sharp contrast, the use of *tight* equalities in system **S** ensures that there is only one built-in way – under the same hypotheses – to produce a derivation typing t' (Sec. 10.3.3). Thus, system **S** is (dynamically) deterministic. We even say that the unique reduction choice is *trivial*, because, as it will turn out (Sec. 13.2.1), it is based upon an identity isomorphism: roughly speaking, reduction is based on the track equality *e.g.*, if there is an axiom leaf typing x using track 8, then it will be substituted by an argument derivation located on track 8 and so on, even when x has been assigned several times the type S (with $S = S_8$).

The Question of Representability

Rigid types, sequence types and derivations of system **S** can be naturally collapsed into regular types, multisets types and derivations of system \mathcal{R} . Actually, \mathcal{R} -types and multisets types are easily identifiable to equivalence classes of rigid (sequence) types. We (coinductively) **collapse** families indexed by integers into multisets. For instance, if we forget about tracks, the distinct sequences $(2 \cdot S, 3 \cdot T, 8 \cdot S)$, $(2 \cdot S, 3 \cdot T, 9 \cdot S)$ and $(3 \cdot T, 9 \cdot S, 15 \cdot S)$ all collapse on $[S, S, T]$.

The application rule of system \mathcal{R} is based upon a *loose* equality: if, inside a rigid derivation P of system **S** that uses *tight equality*, we collapse every sequence type into a multiset type, we obtain an \mathcal{R} -derivation Π . However, it is not clear that, starting from an \mathcal{R} -derivation Π , we can find a rigid derivation P that collapses into Π . For instance, it would demand that we can choose a good rigid representative for every type introduced in an axiom rule, so that we have a (tight) equality in *all* the applications rules. Since Π can be infinite in depth or in width and the typing constraints propagate in complicated ways inside the derivation, the possibility of such a good choice is not easily ensured. The presence of redexes is very problematic (Sec. 13.1.2) and since \mathcal{R} -typability does not ensure any form of productivity/normalization (any term is typable in \mathcal{R} by Theorem 12.1), they cannot be avoided.

Note that a positive answer to this question (the representability of \mathcal{R} -derivation by means of **S**-derivations) would mean that every object of the interpretation of a term t in \mathcal{R} (seen as a model) is obtained from **S**. Thus, the model \mathcal{R} could be studied through **S** without loss of generality.

Moreover, another feature of **S** may seem limited: in contrast to system \mathcal{R} , the substitutions inside an **S**-derivation are performed *deterministically*, while we reduce the subject. This absence of reduction choices can be seen as restrictive compared to system \mathcal{R} , because, even when there are several occurrences of the same type, substitution can be processed only in one way in system **S**, which may be considered as a restriction compared to system \mathcal{R} . This raises the following question: can we build a rigid representative P of an \mathcal{R} -derivation Π w.r.t. any reduction choice we would have done “by-hand”? If we perform a reduction choice at each step of a reduction sequence, we speak of **reduction choice sequence**.

Contributions

The three contributions of this article are the following:

- We prove (Theorem 13.1) that any derivation Π , approximable or not, has a rigid representative P ...

- ...and that any reduction choice sequence of length $\leq \omega$ can be built-in inside such a representative P , without assuming this reduction sequence to be sound (*i.e.* *strongly converging*, see Sec. 9.3).
- For the first point, we implement the technique that is described in the presentation p. 235, which allows us to work when no form of productive reduction/normalization is ensured.

To obtain these results, we represent every quantitative \mathcal{R} -derivation Π by means of an *hybrid* derivation P_h (in a new type system \mathbf{S}_h) in which the tight equality (in the **app**-rule) is loosened and replaced by a congruence. Next, we endow those hybrid derivations with deterministic reduction choices (to be called *interfaces*), yielding *operable* derivations (in another type system \mathbf{S}_{op}). We then show that every “by-hand” reduction sequence of (possible) infinite length can be encoded in an interface. The *trivial* derivations are the operable derivation (system \mathbf{S}) in which the interface uses only identity interfaces. Finally, we prove that every operable derivation is *isomorphic* to a trivial derivation. This result concludes the proof of the Representation Theorem (Theorem 13.1) whose meaning is that \mathbf{S} has a “full expressive power” over \mathcal{R} .

The most difficult point is the last one *i.e.* establishing that every \mathcal{R} -derivation has a *trivial* \mathbf{S} -representative. In a finitary/productive framework (for the notion of productivity, see *e.g.*, [42] or the discussion p. 236), this could be possible by studying first the derivations typing a (partial) normal form (for which representation is usually easily ensured), and then proceeding by subject expansion. However, as already noted, typability in system \mathcal{R} does not imply any kind of normalization (every term can be \mathcal{R} -typed). To prove that, for any \mathcal{R} -derivation Π , we consider an *ad hoc* first order theory \mathcal{T} such that Π has a trivial representative iff \mathcal{T} is consistent. Then, we prove that, for all Π , \mathcal{T} is consistent. For that, we reason *ad absurdum* by considering a proof of inconsistency of \mathcal{T} (such a proof will be called a **brother chain**) and with the help of a finite reduction strategy called the **collapsing strategy**, we prove that such a chain could be in some sense *normalized*. The proof is concluded when we show that the existence of a normal brother chain would bring a contradiction.

13.1 From Representing Types in System \mathbf{S} to Representing Derivations

13.1.1 Multiset Types as Collapses of Sequential Types

In this section, we formally define the set of \mathcal{R} -types as a collapse (quotient set) the set of \mathbf{S} -types.

Let us first remember that, in system \mathbf{S} (Sec. 10.2.2), tracks 0 and 1 are special, compared to tracks ≥ 2 (called **argument tracks**):

- For \mathbf{S} -types, track 1 is dedicated to the targets of arrows (whereas tracks ≥ 2 are used for the types in their sources)
- For \mathbf{S} -derivations, track 0 is dedicated to the **abs**-rule and track 1 to the premise typing the left-hands sides of the **app**-rules (whereas tracks ≥ 2 are used for their argument premises).

For \mathcal{R}_0 -types, the order of the types in the source of arrows does not matter (*e.g.*, $[\sigma, [o, o'] \rightarrow o, \sigma] \rightarrow o' = [\sigma, \sigma, [o', o] \rightarrow o] \rightarrow o'$) and in an \mathcal{R}_0 -derivation, the

order of the argument premises of an **app**-rule does not either *i.e.*, intuitively, the tracks ≥ 2 can be freely permuted. This suggests the following notion of (labelled or not) tree isomorphism, that can change the value of any argument track, but preserves tracks 0 and 1:

Definition 13.1. Let U_1 and U_2 be two (labelled) trees or forests.

A **01-isomorphism** ϕ from U_1 to U_2 is a bijection from $\text{supp}(U_1)$ to $\text{supp}(U_2)$ such that:

- ϕ is monotonic for the prefix order and preserves length.
- If $a \cdot k \in \text{supp}(U_1)$ with $a \in \mathbb{N}^*$ and $k = 0, 1$, then $\phi(a \cdot k) = \phi(a) \cdot k$.
- In the labelled case: for all $a \in \text{supp}(U_1)$, $U_2(\phi(a)) = U_1(a)$.

We write $U_1 \equiv U_2$ when U_1 and U_2 are 01-isomorphic. Since rigid types are labelled tree, Definition 13.1 instantiates into:

Definition 13.2. Let U_1 and U_2 be two (sequence) types. A **(sequence) type isomorphism** from U_1 to U_2 is a 01-isomorphism from U_1 to U_2

Example 13.1. In Fig. 13.1, we see two isomorphic labelled trees T_1 and T_2 w.r.t. some isomorphism ϕ . We have $\text{supp}(T_1) = \{\varepsilon, 1, 4, 4 \cdot 1, 4 \cdot 3, 4 \cdot 8, 8\}$, $\text{supp}(T_2) = \{\varepsilon, 1, 3, 5, 5 \cdot 1, 5 \cdot 2, 5 \cdot 7\}$ and ϕ is defined by $\phi(\varepsilon) = \varepsilon$, $\phi(1) = 1$, $\phi(4) = 5$, $\phi(4 \cdot 1) = 5 \cdot 1$, $\phi(4 \cdot 3) = 5 \cdot 7$, $\phi(4 \cdot 8) = 5 \cdot 2$, $\phi(8) = 3$.

Remark 13.1. If $\phi : T_1 \rightarrow T_2$ is a 01-labelled tree isomorphism, $a_1 \in \text{supp}(T_1)$ and $a_2 = \phi(a_1) \in \text{supp}(T_2)$, then $\overline{a_1} = \overline{a_2}$ (see Sec. 10.2.1 for this notation).

If U is a (labelled or not) tree or forest and ϕ is a monotonic, length-preserving injection from $\text{supp}(U)$ to \mathbb{N}^* s.t., in the labelled case, $\phi(a \cdot k) = \phi(a) \cdot k$ whenever $k = 0, 1$, we write $\phi(U)$ for the *unique* (labelled) tree or forest s.t. $\text{supp}(\phi(U)) = \{\phi(a) \mid a \in \text{supp}(U)\}$ and $\phi(U)(a') = U(\phi^{-1}(a'))$. In that case, $\phi(U) \equiv U$. Such a function ϕ then is called a **01-resetting of U** .

Remark 13.2. Alternatively, we can define $U_1 \equiv U_2$ for types and sequence types by coinduction, without reference to 01-stable isomorphisms:

- $o \equiv o$
- $(S_k)_{k \in K} \equiv (S'_k)_{k \in K'}$ if there is a bijection $\rho : K \rightarrow K'$ such that, for all $k \in K$, $S_k \equiv S'_{\rho(k)}$.
- $(S_k)_{k \in K} \rightarrow T \equiv (S'_k)_{k \in K'} \rightarrow T'$ if $(S_k)_{k \in K} \equiv (S'_k)_{k \in K'}$ and $T \equiv T'$.

Mutable supports and relabellings Actually, 01-resetting a labelled (or not) tree or forest U consists in assigning new track values to the edges labelled with tracks $k \geq 2$. In that case, we may (a bit abusively) use the position $a \cdot k$ of U to stand for the edge from a to $a \cdot k$ and we set $\text{lab}(a \cdot k) = k$: the number k is indeed the track that labels the edge from a to $a \cdot k$. In the forest case, if $k \in \text{Rt}(F)$, then we also use k to denote not an edge but the root of the tree $F|_k$. In particular, if $x : (k \cdot S) \vdash x : S$, is the conclusion

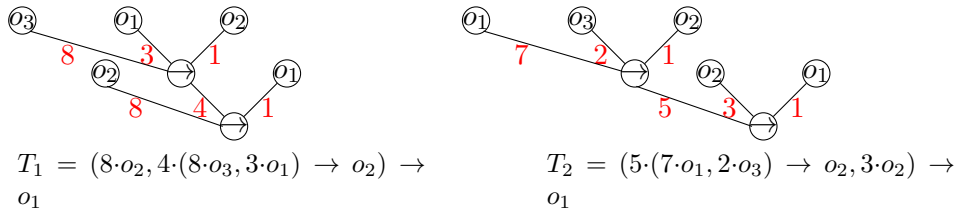


Figure 13.1: 01-Isomorphic Labelled Trees (S-Types)

of an **ax**-rule we say that k is the **axiom root** of this **ax**-rule *i.e.* k denotes both the integer that labels the root of the singleton sequence type ($k \cdot S$) and this root itself.

This motivates the notion of *mutable support* and of *01-relabelling*:

Definition 13.3. Let U be a labelled (or not) tree or forest.

- The **mutable support** $\text{supp}_{\text{mut}}(U) = \{a \cdot k \in \text{supp}(U) \mid a \in \mathbb{N}^*, k \in \mathbb{N} \setminus \{0, 1\}\}$ (in the unlabelled case, we understand $\text{supp}(U)$ as U).
- A **01-relabelling** of U is a function θ from $\text{supp}_{\text{mut}}(U)$ to $\mathbb{N} \setminus \{0, 1\}$ such that, for all $a \in \mathbb{N}$, $k_1, k_2 \in \mathbb{N} \setminus \{0, 1\}$ such that $a \cdot k_1, a \cdot k_2 \in \text{supp}_{\text{mut}}(U)$, $k_1 \neq k_2$ implies $\theta(a \cdot k_1) \neq \theta(a \cdot k_2)$.

Intuitively, $\text{supp}_{\text{mut}}(U)$ is the set of edges of U whose track value may be changed and $\theta(a \cdot k)$ is the new track (given by θ) of the edge between position a and the mutable position $a \cdot k$. Indeed, from a 01-relabelling θ of U , we can define a 01-resetting of U denoted ϕ^θ by induction:

- $\phi^\theta(\varepsilon) = \varepsilon$.
- $\phi^\theta(a \cdot k) = \phi^\theta(a) \cdot k$ if $k = 0, 1$
- $\phi^\theta(a \cdot k) = \phi^\theta(a) \cdot \theta(a \cdot k)$ if $k \in \mathbb{N} \setminus \{0, 1\}$.

Thanks to Definition 13.3, the function ϕ^θ is indeed a 01-resetting of U . We call it the **resetting induced by θ** .

Definition 13.3 can also be instantiated for rigid types:

Definition 13.4. A 01-resetting (resp. 01-relabelling) whose domain is a rigid (sequence) type is called a **(sequence) type resetting** (resp. a **(sequence) type relabelling**).

Example 13.2. The transformation from T_1 to T_2 in Fig. 13.1 can be seen as the relabelling θ defined on $\text{supp}_{\text{mut}}(T_1) = \{4, 4 \cdot 3, 4 \cdot 8, 8\}$ by $\theta(4) = 5$ (the edge labelled 4 receives the new label 5), $\theta(4 \cdot 3) = 7$, $\theta(4 \cdot 8) = 2$, $\theta(8) = 3$.

Let F and F' two 01-isomorphic (labelled) forests. A **root isomorphism** is a function ρ from $K = \text{Rt}(F)$ to $K' = \text{Rt}(F')$ such that, for all $k \in K$, $F|_k \equiv F'|_{\rho(k)}$. Thus, a root isomorphism is a function ρ from $\text{Rt}(F)$ to $\text{Rt}(F')$ that can be extended to a 01-isomorphism from F to F' . Conversely, every isomorphism ϕ from F to F' induces a root isomorphism from F to F' , that we denote $\text{Rt}(\phi)$.

Example 13.3. Let T_1 and T_2 be the types of Fig. 13.1. Let us set $F_1 = (3 \cdot T_1, 4 \cdot o, 7 \cdot (2 \cdot o, 5 \cdot o') \rightarrow o')$ and $F_2 = (2 \cdot o, 7 \cdot (3 \cdot o', 5 \cdot o) \rightarrow o', 9 \cdot T_2)$. Then $\text{Rt}(F_1) = \{3, 4, 7\}$, $\text{Rt}(F_2) = \{2, 7, 9\}$ and $\rho : \text{Rt}(F_1) \rightarrow \text{Rt}(F_2)$ defined by $\rho(3) = 9$, $\rho(4) = 2$, $\rho(7) = 5$ is a root isomorphism from F_1 to F_2 .

Let us try now to build the (possibly) infinite versions of \mathcal{R}_0 -types as collapses of rigid types. As already remarked several times, in the finite case, when we forget about tracks, a finite rigid type collapses on an \mathcal{R}_0 -type *e.g.*, $T = (7 \cdot o_1, 3 \cdot o_2, 2 \cdot o_1) \rightarrow o$, $T' := (9 \cdot o_2, 7 \cdot o_1, 6 \cdot o_1) \rightarrow o$ or $T'' = (7 \cdot o_2, 3 \cdot o_1, 2 \cdot o_1) \rightarrow o$ all three collapse on $\tau = [o_1, o_2, o_1] \rightarrow o$. But note that T , T' and T'' collapse on τ precisely because they are isomorphic types.

Thus, we can define the set of \mathcal{R} -types, that correspond to the infinitary version of \mathcal{R}_0 -types, as the quotient set Typ/\equiv . For instance, in Fig. 13.1, both T_1 and T_2 collapse on $[o_2, [o_1, o_3] \rightarrow o_2] \rightarrow o_1$. Multiset types of system \mathcal{R} are the \equiv -equivalence classes of sequence types. Note that \mathcal{R} allows infinite multiset types and infinite nesting of types inside multisets.

System \mathcal{R} is defined as \mathcal{R}_0 (Sec. 3.2.4), except that we use the coinductive types of \mathcal{R} instead of just finite types of \mathcal{R}_0 and we allow coinductive derivations to type the infinite terms of Λ^{001} . A countable version of the binary operator $+$ can be easily defined (more details will be given in Sec. 13.1.3).

The following observations and notations will be useful to define isomorphisms by induction:

Notation 13.1.

- If $S = F \rightarrow T$, where F is a sequence type and T a type, we write $\text{Sc}(S)$ for the sequence type F (the **source** of S) and $\text{Tg}(S)$ for the type T (the **target** of S).
- If $\psi : F_1 \rightarrow F_2$ is a sequence type isomorphism and $\phi : T_1 \rightarrow T_2$ is a type isomorphism, then $\psi \rightarrow \phi : (F_1 \rightarrow T_1) \rightarrow (F_2 \rightarrow T_2)$ is the type isomorphism defined by: $(\psi \rightarrow \phi)(k \cdot c) = \phi(k \cdot c)$ when $k \geq 2$ and $(\psi \rightarrow \phi)(1 \cdot c) = 1 \cdot \phi(c)$. If F is a sequence type, then $F \rightarrow \phi$ denotes $\text{id}_F \rightarrow \phi$.
- Conversely, if ϕ is a type isomorphism from $F_1 \rightarrow T_1$ to $F_2 \rightarrow T_2$, then $\text{Tg}(\phi)$ and $\text{Sc}(\phi)$ are respectively the type isomorphism from T_1 to T_2 and the sequence type isomorphism from F_1 to F_2 induced by ϕ . Thus, $\phi = \text{Sc}(\phi) \rightarrow \text{Tg}(\phi)$ and $\text{Tg}(\psi \rightarrow \phi) = \phi$ and $\text{Sc}(\psi \rightarrow \phi) = \phi$.

13.1.2 The Representation Theorem and Hybrid Derivations

Now that we have *formally* defined \mathcal{R} -types (types based on multiset constructions) as collapses of \mathbf{S} -types, we can better understand some difficulties related to the representation of the \mathcal{R} -derivations by means of \mathbf{S} -derivations.

As observed before, \mathcal{R} -derivations collapse² on \mathbf{S} -derivations *e.g.*, in Sec. 10.2.3, P_{ex} collapses on Π_{ex} , but the surjectivity of this collapse is not ensured at this stage. From the introduction, we recall that we will give a positive answer to this question:

Question 1: Is every \mathcal{R} -derivation the collapse of an \mathbf{S} -derivation?

²This collapse will be also formalized (Sec. 13.1.3).

From Theorem 12.1, we know that every term is typable in \mathcal{R} . If we try to proceed by induction on the structure of an \mathcal{R} -derivation Π , we may be easily stuck. For instance, assume that:

- $\Pi = \frac{\Pi_{\lambda x.r} \triangleright \Gamma \vdash \lambda x.r : [\sigma_i]_{i \in I} \rightarrow \tau \quad (\Pi_i \triangleright \Delta_i \vdash s : \sigma_i)_{i \in I}}{\Gamma + (+_{i \in I} \Delta_i) \vdash (\lambda x.r)s : \tau}$.
- $\Pi_{\lambda x.r}$ and the Π_i are all \mathbf{S} -representable.

The second assumption means that there are \mathbf{S} -derivations $P_{\lambda x.r} \triangleright C \vdash \lambda x.r : (S_k)_{k \in K} \rightarrow T$ and $(P_i \triangleright D_i \vdash s : S'_i)_{i \in I'}$ that respectively collapse on $\Pi_{\lambda x.r}$ and the Π_i . But $P_{\lambda x.r}$ and the P_i can be used to represent Π only if (1) we can ensure that the S_k and the S'_i are syntactically equal (modulo some permutations) and (2) that we can avoid track conflict.

For point (1), notice that, given a term t typable with type τ in \mathcal{R} , there may be some \mathbf{S} -types T that collapse on τ such that t is *not* \mathbf{S} -typable with T *e.g.*, if $\bar{S} = \sigma = \bar{S}'$, but $S \neq S'$, then $T := (2 \cdot S) \rightarrow S'$ collapses on $\tau := [\sigma] \rightarrow \sigma$ and $I := \lambda x.x$ can be \mathcal{R} -typed with τ and \mathbf{S} -typed with $(2 \cdot S) \rightarrow S$ or $(5 \cdot S') \rightarrow S'$, but *not* with T .

Thus, some typing constraints inside r or s could *a priori* forbid that we can equalize the S_k and the S'_i since it is difficult to describe the \mathbf{S} -types of r and s that collapse on τ or σ_i . Since \mathcal{R} -types may be infinite and there is no productive reduction, we cannot resort to Tait's realizability argument or even step-indexed logical relations. We then use the method described in the Presentation of p. 253 that is also used in Chapter 12. The remainder of this chapter is dedicated to proving that the answer to the above question is positive, that is:

Theorem 13.1 (Representation). Every \mathcal{R} -derivation is the collapse of an \mathbf{S} -derivation.

First, the above discussion suggests that system \mathbf{S} is too constraining for the question of representability to be addressed directly and that we should relax the **app**-rule. This motivates to define the set **Deriv_h** of **hybrid derivations** by replacing, in **Deriv**, the rule **app** by:

$$\frac{C \vdash t : (S_k)_{k \in K} \rightarrow T \quad (D_k \vdash u : S'_k)_{k \in K'} \quad (S_k)_{k \in K} \equiv (S'_k)_{k \in K'}}{C \uplus (\uplus_{k \in K} D_k) \vdash tu : T} \mathbf{app}_h$$

Thus, the **app_h**-rule demands that, for an application tu to be typed, $(S_k)_{k \in K}$, the source of the arrow type given to t must be *01-isomorphic* to the sequence type $(S'_k)_{k \in K'}$ given to u *i.e.* **app_h** demands that $(S_k)_{k \in K} =: \mathbf{L}^P(a) \equiv \mathbf{R}^P(a) := (S'_k)_{k \in K'}$. We call the sequence types $\mathbf{L}^P(a)$ (resp. $\mathbf{R}^P(a)$) the **left key** (resp. the **right key**) at position a in P . We often write simply $\mathbf{R}(a)$ and $\mathbf{L}(a)$ for those sequence types. Note that a hybrid derivation P is trivial when for all $a \in \mathbf{supp}_@ (P)$, $\mathbf{L}^P(a) = \mathbf{R}^P(a)$ *i.e.* when the left and the right keys are equal.

More formally, let P be a hybrid derivation. We set $\mathbf{supp}_@ (P) = \{a \in \mathbf{supp}(P) \mid t(\bar{a}) = @\}$. For all $a \in \mathbf{supp}_@ (P)$, we set $\mathbf{L}^P(a) = \mathbf{Sc}(\mathbf{T}^P(a \cdot 1))$ and $\mathbf{R}^P(a) = (\mathbf{T}^P(a \cdot k))_{k \in \mathbf{ArgTr}^P(a)}$, where $\mathbf{ArgTr}^P(a) = \{k \geq 2 \mid a \cdot k \in \mathbf{supp}(P)\}$ (this is a particular case of Notation 10.1).

The notion of **quantitative** derivation (Sec. 10.3.2) straightforwardly extends to hybrid derivations, as well as the notations \mathbf{pos}^P , \mathbf{Ax}^P etc.

13.1.3 System \mathcal{R} and the Hybrid Construction

In this section, we prove that representability is easy when considering *hybrid* derivation instead of *trivial* derivations. We also make more precise the definition of system \mathcal{R} (an infinitary version of Gardner/de Carvalho's system \mathcal{R}_0) that has already been informally used in several occasions (*e.g.*, Sec. 10.1.3 or 12.1). The idea is to define system \mathcal{R} with sequential constructions and then specifying that the tracks do not matter (for that, we resort to equivalence relations using 01-isomorphisms).

We first recall that we have defined the \mathcal{R} -types as the equivalence classes of rigid types under \equiv .

If U is an \mathbf{S} -type or a sequence type, its equivalence class is written \overline{U} . We may now define coinductively the notation of the collapses of \mathbf{S} -types:

- The equivalent class of a sequence type $F = (S_k)_{k \in K}$ is the multiset type written $[\overline{S_k}]_{k \in K}$
- We write $\overline{F} \rightarrow \overline{T}$ for $\overline{F \rightarrow T}$.
- If o is a type variable, \bar{o} is the singleton $\{o\}$ is written simply o (instead of $\{o\}$).

If $\overline{T} = \tau$ (resp. $\overline{F} = [\sigma_i]_{i \in I}$), we say that T (resp. F) is a **parser** of τ (resp. $[\sigma_i]_{i \in I}$).

Countable Multiset Sum We may define countable sum operator $+$ on multiset types. We then adapt Remark 3.5 and prove that the multiset sum is infinitarily associative and commutative as expected. For instance, let $F = (S_k)_{k \in K}$ be a sequence type. We define the function m_F from the set of \mathcal{R} -types to $\mathbb{N} \cup \{\infty\}$ by: for all \mathcal{R} -type σ , $m_F(\sigma) = \#\{k \in K \mid \overline{S_k} = \sigma\}$ *i.e.* $m_F(\sigma)$ is the (possibly infinite) number of occurrences of σ in m . Since $F \equiv F'$ obviously implies $m_F = m_{F'}$. From that, the verifications are easy.

Semi-Rigid Derivations An \mathcal{R} -context is a function from the set of term variables \mathcal{V} to the set of (infinite) multiset types. An \mathcal{R} -judgment is a triple of the form $\Gamma \vdash t : \tau$ where Γ is an \mathcal{R} -context, t a 001-term and τ an \mathcal{R} -types. The set of *semi-rigid* derivations is the set of trees (labelled with \mathcal{R} -judgments) defined *coinductively* by the following rules

$$\frac{}{x : [\tau] \vdash x : \tau} \text{ax} \qquad \frac{\Gamma; x : [\sigma_i]_{i \in I} \vdash t : \tau}{\Gamma \vdash \lambda x. t : [\sigma_i]_{i \in I} \rightarrow \tau} \text{abs}$$

$$\frac{\Gamma \vdash t : [\sigma_i]_{i \in I} \rightarrow \tau \quad (\Delta_k \vdash u : \sigma'_k)_{k \in K} \quad [\sigma_i]_{i \in I} = [\sigma'_k]_{k \in K}}{\Gamma +_{k \in K} \Delta_k \vdash t u : \tau} \text{app}$$

Why *semi-rigid* derivations? Because the argument subderivations are still placed on argument tracks $k \in K$, while obviously, they should not matter when we work with multisets. Let P_1 and P_2 be two semi-rigid derivations. We can write $P_1 \equiv P_2$ to mean that there is a 01-labelled isomorphism from P_1 to P_2 . We define the set of \mathcal{R} -derivation as the quotient set of that of semi-rigid derivations by the relation \equiv . Then, we obtain exactly the rules of system \mathcal{R}_0 , except that the types are taken in $\mathbf{Typ}_{\mathcal{R}}$, as expected in Sec. 13.1.1. Notice that the derivations Π and Π' of Subsection 10.1.3 are \mathcal{R} -derivations w.r.t. this formal definition. Quantitativity can be defined in system \mathcal{R}

(see Appendix A.6.1).

Let Π be a *quantitative* \mathcal{R} -derivation. We show now that Π has a hybrid *quantitative* representative P . Let then \tilde{P} be a quantitative semi-rigid derivation representing \tilde{P} and $A = \text{supp}(\tilde{P})$. For each $a \in \mathbf{Ax}^P$, we choose an integer $\text{tr}(a)$ greater ≥ 2 s.t. no conflict arises (that is, for all $x \in \mathcal{V}$, there are no $a', a'' \in \mathbf{Ax}_a^P(x)$ s.t. $a' \neq a''$ and $\text{tr}(a') = \text{tr}(a'')$).

For each axiom leaf $a \in A$ of Π , we choose $\mathbf{T}_{\text{ax}}(a)$, a type representing $\tau(a)$.

We write $\mathbf{C}(a)(x)$ for the sequence type $(\text{tr}(a_0) \cdot \mathbf{T}_{\text{ax}}(a_0))_{a_0 \in \mathbf{Ax}_a(x)}$. and we choose for each $a \in A$ a representative $T(a)$ of $\tau(a)$, by a induction.

- If a is an axiom leaf, we set $T(a) = \mathbf{T}_{\text{ax}}(a)$.
- If $a \cdot 0 \in A$, then $t(a) = \lambda x$ and we set $T(a) = \mathbf{C}(a \cdot 0)(x) \rightarrow T(a \cdot 0)$.
- If $a \cdot 1 \in A$, then we set $T(a) = \mathbf{Tg}(T(a \cdot 1))$.

The above induction shows that the definition is sound ($\overline{T(a)} = \tau(a)$ for all $a \in A$) and that we have $\overline{P^*} = \Pi$. We call this process the **hybrid construction**.

13.2 Subject Reduction

Systems \mathcal{R} and \mathbf{S} satisfy subject reduction and subject expansion, and \mathbf{S}_h almost does:

Property 13.1.

- Subject Reduction: if $t \rightarrow t'$ and $\triangleright_{\mathbf{S}_h} C \vdash t : T$, then $\triangleright_{\mathbf{S}_h} C \vdash t' : T'$ for some $T' \equiv T$.
- Subject Expansion: if $t \rightarrow t'$ and $\triangleright_{\mathcal{R}} \Gamma \vdash t' : \tau$, $\triangleright_{\mathbf{S}_h} C \vdash t' : T'$, then $\triangleright_{\mathbf{S}_h} C \vdash t : T$ for some $T \equiv T'$.

In this section, we explain how subject reduction is handled with trivial and hybrid derivations and why T may be replaced by an isomorphic type T' in \mathbf{S}_h (and *vice versa*). In system \mathbf{S}_h , we retrieve some determinism by considering **(root) interfaces** *i.e.* sequence type isomorphisms that constrain how reduction is processed in a derivation and how axiom leaves typing x , the variable or a redex, should be replaced by argument derivations typing while respecting the rules of \mathbf{S}_h . This is explained in Sec. 13.2.1 from a high-level perspective. In Sec. 13.2.2, we give a few intuitions on the means to capture sequences of reduction choices.

13.2.1 Encoding Reduction Choices with Interfaces

In Sec. 13.2.1, we explain how reduction choices (Sec. 4.1.2) in system \mathbf{S}_h can be encoded by using sequence type isomorphisms that we call *interfaces*.

For the remainder of the section, we assume that $t|_b = (\lambda x.r)s$, $t \xrightarrow{b} t'$ (so that $t'|_b = r[s/x]$), each axiom rule concluding with $x : (k \cdot S_k) \vdash x : S_k$ will be replaced by a subderivation $P_{k'} \triangleright D_{k'} \vdash s : S'_{k'}$ satisfying $S'_{k'} \equiv S_k$. So that there may be many ways to produce P' typing t' if $(S_k)_{k \in K}$ (and so, $(S'_k)_{k \in K'}$ as well) contains some 01-isomorphic

Assumptions: $\rho_a(2) = 8, \rho_a(7) = 5$
 (so that $S_2 \equiv S'_8, S_7 \equiv S'_5$)

Comment: since $\rho_a(2) = 8, \rho_a(7) = 5$, the argument subderivation P_8 (resp. P_5) on track 8 (resp. 5) will replace the axiom rule using track 2 (resp. 7).

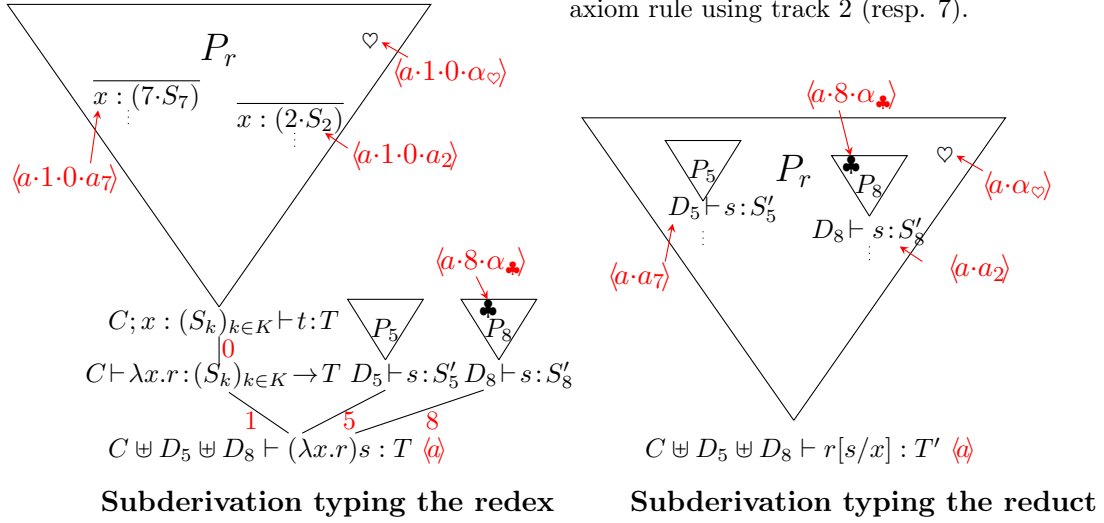


Figure 13.2: Subject Reduction and Residuals

types. We say then that there are **reduction choices** in system \mathbf{S}_h . This also holds for system \mathcal{R} .

All this is illustrated by the left part of Fig. 13.2, which generalizes Fig. 10.5, p. 218, since system \mathbf{S}_h extends system \mathbf{S} : under the same hypotheses, we assume that $a \in \text{supp}(P)$ is such that $\bar{a} = b$ (thus, a is the position of a judgment typing the redex to be fired) and that there are exactly 2 **ax**-rules typing x above a , using axiom tracks 2 and 7. Notice that the **ax**-rule typing x on track 7 (assigning S_7) must be above $a \cdot 1 \cdot 0$, so that its position is of the form $a \cdot 1 \cdot 0 \cdot a_7$. Likewise, for the other **ax**-rule assigning S_2 on track 2. We omit **ax**-rules right-hand sides. We indicate again the position of a judgment between angle brackets *e.g.*, $\langle a \cdot 1 \cdot 0 \cdot a_7 \rangle$ means that judgment $x : (7 \cdot S_7) \vdash x : S_7$ is at position $a \cdot 1 \cdot 0 \cdot a_7$.

By typing constraints, there must be two argument derivations typing s with types isomorphic to S_2 and S_7 . We assume that those two argument derivations are on track 5 and 8 and conclude with $D_k \vdash s : S'_k$ ($k = 5, 8$) where *e.g.*, $S_2 \equiv S'_8$ and $S_7 \equiv S'_5$. If moreover, $S_2 \equiv S_7$, then the **ax**-rule #2 can be replaced by P_5 as well as by P_8 : there is a reduction choice. In all cases, the type of $r[s/x]$ may change *e.g.*, if $x : S_7$ corresponds to the head variable of r , then S_7 is replaced by an isomorphic type S'_5 (or S'_8), so T may also be replaced by an isomorphic T' .

We use now 01-root isomorphisms (Sec. 13.1.1) to represent particular reduction choices in system \mathbf{S}_h . Let P be a hybrid derivation:

- Let $a \in \text{supp}_@ (P)$. A **root interface** (resp. an **interface**) at position a is a root isomorphism (resp. a sequence type isomorphism) from $L^P(a)$ to $R^P(a)$.
- Let $b \in \text{supp}(t)$ such that $t(b) = @$. A **total (root) interface** at position b is a family of (root) interfaces for all $a \in \text{supp}(P)$ s.t. $\bar{a} = b$.
- A **total interface** is the datum for an interface ϕ_a for all $a \in \text{supp}_@ (P)$.

For all $a \in \text{supp}_{\textcircled{a}}(P)$, we write $\text{Inter}^P(a)$ for the set of interfaces at position a in P . Note that a root interface is a function (between root tracks) that can be extended to an interface isomorphism and conversely, every interface isomorphism ϕ .

An **operable derivation** is a hybrid derivation endowed with a total interface. If P is an operable derivation whose interface is $(\phi_a)_{a \in \text{supp}_{\textcircled{a}}(P)}$, we usually only write ρ_a for $\text{Rt}(\phi_a)$ (so that ρ_a is a root interface) and we set $\mathbf{L}^P = \{(a \cdot 1, k \cdot c) \in \text{bisupp}(P) \mid k \in \mathbb{N} \setminus \{0, 1\}\}$ and $\mathbf{R}^P = \{(a \cdot k, c) \in \text{bisupp}(P) \mid k \in \mathbb{N} \setminus \{0, 1\}\}$. For all $\mathbf{p} = (a \cdot 1, k \cdot c) \in \mathbf{L}^P$, we just write $\phi(\mathbf{p})$ for $(a \cdot k', c')$ with $k' \in \mathbb{N} \setminus \{0, 1\}$, $c' \in \mathbb{N}^*$ and $k' \cdot c' = \phi_a(k \cdot c)$.

Assuming that $S_2 \equiv S_7$ in Fig. 13.2, we have seen above that the two **ax**-rules typing x could be indifferently replaced by P_5 or P_8 . But if a is endowed with the root interface ρ_a s.t. $\rho_a(2) = 8$ and $\rho_a(7) = 5$, then the **ax**-rule typing x on track 2 *must* be replaced by P_8 and the other one by P_5 , as on the right part of the figure.

Remark 13.3. Still assuming that $S_2 \equiv S_7$, the root interface ρ_a defined by $\rho_a(2) = 5$ and $\rho_a(7) = 8$ is licit, but it would produce another derivation typing $r[s/x]$, obtained from the right part of Fig. 13.2 by swapping the two inner triangles.

In contrast, if we assume now that P is a trivial derivation (system **S**), then $(S_k)_{k \in K} = (S'_k)_{k \in K'}$ and there is always one straightforward canonical way to produce a derivation P' typing t (remember Sec. 10.3.3): for all $k \in K$, we replace the axiom rule concluding with $x : (k \cdot S_k) \vdash x : S_k$ by P_k i.e. $P|_{a \cdot k}$ (a process that can seem somewhat limited, as discussed on p. 289): implicitly, a trivial derivation is endowed with a trivial interface i.e. we consider only the identity isomorphism from $\mathbf{L}^P(a)$ to $\mathbf{R}^P(a)$ for all $a \in \text{supp}_{\textcircled{a}}(P)$.

13.2.2 Residuation and Encoding Reduction Choices

In Sec. 13.2.2, we give a high level input on how interfaces can actually capture *sequences* of reduction choices.

As seen in Sec. 13.2.1, a total *root* interface $(\rho_a)_{\bar{a}=b}$ (we write $\bar{a} = b$ for $a \in \text{supp}(P)$, $\bar{a} = b$) at position b is enough to formally capture the notion of *reduction choice* used implicitly to define a derivation P' typing t' from a derivation P typing t when $t \xrightarrow{b} t'$. This allows us to define a suitable notion of **residuals** of positions (the residual of $\alpha \in \text{supp}(P)$, if it exists, is a $\alpha' \in \text{supp}(P')$ that may be denoted $\text{Res}_b^\rho(\alpha')$ since it depends both on b and $(\rho)_{\bar{a}=b}$).

Now, if instead of endowing P with a total *root* interface $(\rho_a)_{\bar{a}=b}$ at position b , we endow it with a total interface $(\phi_a)_{\bar{a}=b}$ at position b , we can define a notion of **residuals** (and **quasi-residuals**) for right bipoitions (Sec. 10.3.1) as we did for system **S** (trivial derivation) in Sec. 10.3.3 and 12.4.1. The definitions are not so straightforward because they involve type isomorphisms. See Appendix B.1 for the details.

The residual (resp. quasi-residual) of (α, c) may be then denoted $\text{Res}_b^\phi(\alpha, c)$ (resp. $\text{QRes}(\alpha, c)$). Interestingly, if we can define the residual of interfaces: more precisely, if P is endowed with a total interface at position b and $\alpha \in \text{supp}_{\textcircled{a}}(P)$ is such that $\bar{\alpha} \neq b$, then α has a residual $\alpha' := \text{Res}_b^\phi(\alpha)$ w.r.t. ϕ and there is a bijection $\text{ResI}_b^\phi(\alpha)$ from $\text{Inter}^P(\alpha)$ to $\text{Inter}^{P'}(\alpha')$

Thus, every interface $\psi'_{\alpha'}$ at position α' in the derivation P' typing the reduct t' may be seen as the residual $\text{ResI}_b^\phi(a)(\psi_\alpha)$ of some interface ψ_α at position $\alpha := (\text{Res}_b^\phi)^{-1}(\alpha')$ in the derivation P . Since (1) reduction choices in P' can be implemented with some

interfaces of P' and (2) interfaces of P' are residuals³ of interfaces of P , we can directly implement reduction choices in P' with interfaces in P (instead of P'). By induction, this allows us to endow directly inside some interfaces of P a sequence of reduction choices along a reduction sequence $t = t_0 \xrightarrow{b_0} \dots \xrightarrow{b_{n-1}} t_n$. This explains why:

Lemma 13.1. Every sequence of reduction choices of length $\leq \omega$ in a quantitative derivation Π can be built-in in an operable derivation representing Π .

Proof. A complete proof of this statement can be found in Appendix B.1.4. \square

13.3 Representation Theorem and Isomorphisms of Derivations

In this section, we explain why the proof of the Representation Theorem may be related to the notion of isomorphism of derivations and then, we will describe those isomorphisms in different suitable ways, till we introduce the notion of relabelling of derivation in Sec. 13.3.3.

As we will see in Sec. 13.4, some edges inside a derivations may be identified (because they correspond to a same type moving through the judgments). This motivates to consider the notion of **thread** (from Sec. 11.1.1), that is formally defined in the setting of this chapter in Sec. 13.4.1.

Since a hybrid derivation P is a tree of \mathbb{N}^* that is labelled with rigid judgments, and for all $a \in \text{supp}(P)$, $\mathbb{T}^P(a)$ is also a labelled tree of \mathbb{N}^* , it is easy to define the notion of isomorphism (Definition 13.5 below) from one hybrid derivation P_1 to another P_2 , using suitably the notion of 01-isomorphism Sec. 13.1.1, by just specifying how (2) the support of P_1 is mapped onto that of P_2 (2) how the types assigned in **ax**-rules in P_1 are mapped onto those of P_2 (since a quantitative hybrid derivation P can be computed from $\text{supp}(P)$ and the datum of the types assigned in **ax**-rules).

Actually, two *hybrid* derivations P_1 and P_2 are isomorphic iff they collapse on the same \mathcal{R} -derivation. From Sec. 13.2.1, we recall that an operable derivation is a hybrid derivation P that is endowed with a total interface $(\phi_a)_{a \in \text{supp}_{\mathbb{Q}}(P)}$. This motivates the notion of **isomorphism of operable derivations**. Informally, if P_1 and P_2 are operable derivations, then an isomorphism Ψ of hybrid derivations from P_1 to P_2 is actually an isomorphism of operable derivations if Ψ commutes with the interfaces of P_1 and P_2 .

Now, since we remember that a trivial derivation is endowed with identity interfaces, the Representation Theorem is a consequence of this one:

Theorem 13.2. Every operable derivation is isomorphic to a trivial derivation.

With Lemma 13.1, this theorem means that any \mathcal{R} -derivation Π and any sequence of reduction choices w.r.t. Π can be encoded by an **S**-derivation P , as expected from the Introduction. Let us now define isomorphisms (Sec. 13.3.1) and resettings (Sec. 13.3.2). In Sec. 13.3.3, we extend the notion of 01-relabelling to define that of *relabelling of derivations*, that allow describing more lightly isomorphisms of derivations and, prospectively, to prove Theorem 13.2 above.

Remark 13.4. Intuitively, system \mathbf{S}_h is *statically* rigid, because hybrid derivations have a bisupport and any symbol in an \mathbf{S}_h -derivation can be pointed to. But it is not *dynamically* rigid: indeed, there is no canonical way to perform subject reduction on

³via the interfaces of P corresponding to position b , that define residuation when firing the redex.

\mathbf{S}_h -derivation since the premises of the application rule are of the form $t : (S_k)_{k \in K} \rightarrow T$, $(u : S'_{k'})_{k' \in K'}$ and $(S_k)_{k \in K} \equiv (S'_k)_{k \in K'}$ but nothing indicates how reduction should be processed when there are several isomorphic S_k (*i.e.* there are reduction choices).

System \mathbf{S}_{op} is rigid, but it is not syntax directed: the application rule in system \mathbf{S}_{op} is intuitively indexed by the set of sequence type isomorphisms, since a sequence type must be specified in each use of **app**-rule.

Thus, the only fully rigid syntax directed system is system \mathbf{S} . The twin theorems 13.1 and 13.2 entail that there is no loss of generality to consider the rigid and syntax directed system \mathbf{S} instead of the non-rigid system \mathcal{R} or the non-syntax direct system \mathbf{S}_{op} . In particular, Theorem 13.1 means that there is no need to burden the typing system with a complicated permutation rule. In the finite case, this means that Gardner/de Carvalho's original system \mathcal{G} (with the **perm**-rule) and system \mathcal{R}_0 are both subsumed by system \mathbf{S}_0 , the finite version of system \mathbf{S} .

13.3.1 Isomorphisms of Operable Derivations

In Sec. 13.3.1, we define a natural notion of isomorphism between hybrid or operable derivations.

Let P_1 and P_2 be two hybrid derivations collapsing one the same \mathcal{R} -derivation Π (thus, intuitively, P_1 and P_2 should be isomorphic as hybrid derivations). We set $A_i = \text{supp}(P_i)$ and we write \mathbf{C}_i , \mathbf{T}_i , tr_i , pos_i for \mathbf{C}^{P_i} , \mathbf{T}^{P_i} , tr^{P_i} , pos^{P_i} and so on. We write \mathbf{Ax}_i for the set of leaves of A_i ($i = 1, 2$).

Definition 13.5. A **hybrid derivation isomorphism** Ψ from P_1 to P_2 is the datum of:

- Ψ_{supp} a 01-isomorphism from A_1 to A_2 . We often write Ψ instead of Ψ_{supp} .
- For each $a_1 \in \mathbf{Ax}_1$, a type isomorphism Ψ_{a_1} from $\mathbf{T}_1(a_1)$ to $\mathbf{T}_2(a_2)$, where $a_2 = \Psi(a_1)$.

We often just write $\Psi(a_1)$ instead of $\Psi_{\text{supp}}(a_1)$, for $a_1 \in \text{supp}(P_1)$. We can check that isomorphisms of hybrid derivations behave as expected (*e.g.*, a judgment of P_1 is mapped onto a judgment of P_2 that represents the same \mathcal{R} -judgment) and allow defining many useful isomorphisms, as the isomorphisms from the keys of P_1 to the keys of P_2 . The following claims are detailed in Appendix B.2.1 and are proved by downward induction, using the typing rules of \mathbf{S}_h :

- If $\Psi(a_1) = \Psi(a_2)$, then $t(a_1) = t(a_2)$ (since $\overline{a_1} = \overline{a_2}$), and Ψ induces a bijection from \mathbf{Ax}_1 to \mathbf{Ax}_2 and from $\text{supp}_{\text{ax}}(P_1)$ to $\text{supp}_{\text{ax}}(P_2)$. In particular, Ψ maps the **ax**-rules (resp. the **app**-rules) of P_1 onto the axioms (the **app**-rules) of P_2 .
- Let $a_1 \in A_1$ and $a_2 = \Psi(a_1) \in A_2$, $C_1 = \mathbf{C}_1(a_1)$, $C_2 = \mathbf{C}_2(a_2)$, $T_1 = \mathbf{T}_1(a_1)$ and $T_2 = \mathbf{T}_2(a_2)$, so that $P_1(a_1) = C_1 \vdash t|_b : T_1$ and $P_2(a_2) = C_2 \vdash t|_b : T_2$ with $b = \overline{a_1} = \overline{a_2}$. Then, from Ψ , we can canonically define:
 - A type isomorphism, denoted⁴ Ψ_{a_1} , from T_1 to T_2 .
 - A sequence type isomorphism, denoted $\Psi_{a_1, x}$, from $C_1(x)$ to $C_2(x)$, for all $x \in \mathcal{V}$.

⁴This thus extends the notation Ψ_{a_1} for all $a_1 \in \text{supp}(P)$, and not only for the positions of **ax**-rules.

This indeed means that $P(a_1)$ and $P(a_2)$ represent the same \mathcal{R} -judgments, as hinted at above.

- We can canonically define from Ψ a bijection from $\mathbf{bissupp}(P_1)$ to $\mathbf{bissupp}(P_2)$. The image of a biposition $\mathbf{p}_1 \in \mathbf{bissupp}(P_1)$ by this bijection is simply noted $\Psi(\mathbf{p}_1)$. The bijection Ψ maps right (resp. left) bipositions of P_1 onto right (resp. left) bipositions of P_2 .

An important point to formulate properly Theorem 13.2 is that Ψ also induce isomorphisms from the left keys (resp. the right keys) of the **app**-rules of P_1 to those of P_2 *i.e.* from Ψ , we can canonically define, for all $a_1 \in \mathbf{supp}_\circlearrowleft(P_1)$ and $a_2 = \Psi(a_1) \in \mathbf{supp}_\circlearrowleft(P_2)$:

- A sequence type isomorphism $\Psi_{a_1}^L$ from $L_1(a_1)$ to $L_2(a_2)$.
- A sequence type isomorphism $\Psi_{a_1}^R$ from $R_1(a_1)$ to $R_2(a_2)$.

Definition 13.6. Let P_1 and P_2 be two operable derivation typing the same term t . Their interface isomorphisms are written $(\phi_{i,a})_{a \in \mathbf{supp}_\circlearrowleft(P_i)}$ ($i = 1, 2$). An **operable derivation isomorphism** is a hybrid derivation isomorphism Ψ from P_1 to P_2 such that for all $a_1 \in \mathbf{supp}_\circlearrowleft(P_1)$ and $a_2 = \Psi(a_1)$, the following diagram is commuting:

$$\begin{array}{ccc}
 L_1(a_1) & \xrightarrow{\phi_{1,a_1}} & R_1(a_1) \\
 \downarrow \Psi_{a_1}^L & & \downarrow \Psi_{a_1}^R \\
 L_2(a_2) & \xrightarrow{\phi_{2,a_2}} & R_2(a_2)
 \end{array}$$

13.3.2 Resetting an Operable Derivation

In Sec. 13.3.2, we explain how to *reset* a hybrid or operable derivation into a new one that is isomorphic to the former. This is a first step before characterizing the resetting that define a trivial derivation later on (Sec. 13.4.3).

Let P an operable derivation. We reuse the notations A , \mathbf{C} , \mathbf{T} , \mathbf{Ax} , \mathbf{tr} , \mathbf{pos} and $(\phi_a)_{a \in \mathbf{supp}_\circlearrowleft(P)}$. In the last section, we have defined, the notion of isomorphism from one operable derivation P_1 to another P_2 . Now, using 01-resettings (Sec. 13.1.1) and in particular, of type resetting, we want to *build*, given only one operable derivation P , a derivation P_0 that is isomorphic to P . This will serve the purpose of proving that every operable derivation is isomorphic to a trivial derivation.

Definition 13.7. A **resetting** of P is given by the data of:

- $\Psi_{\mathbf{supp}}$ a 01-resetting of A .
- For each $a \in \mathbf{Ax}$, Ψ_a a type resetting of $\mathbf{T}(a)$.
- $\Psi_{\mathbf{tr}}$ an *injection* from \mathbf{Ax} to $\mathbb{N} \setminus \{0, 1\}$.

The function $\Psi_{\mathbf{supp}}$ describes the new support of the derivation and the function Ψ_a the new supports of the types assigned in the axiom rules. The function $\Psi_{\mathbf{tr}}$ is used to define new track values. Theoretically, we only need to avoid track conflict, but the injectivity of $\Psi_{\mathbf{tr}}$ will actually serve to produce a derivation P_0 isomorphic to P such

that two distinct axiom rules of P_0 do not use the same axiom track (which of course ensures the absence of track conflict). We state the following proposition in an informal (but intuitive way). See Appendix B.2.2 for a complete statement and its proof.

Proposition 13.1. Let P an operable derivation. From a resetting Ψ of P , we can canonically define another operable derivation, denoted $\Psi(P)$, such that Ψ induces an isomorphism of operable derivation from P to $\Psi(P)$.

13.3.3 Relabelling a derivation

We recall that tracks are numbers that label edges (of types or of derivations). Let P be an operable derivation. We want to find a trivial derivation P_0 that is isomorphic to P (as an operable derivation). But roughly speaking, defining an isomorphism of operable derivation whose domain is P is a matter of giving new values to the tracks that are present in P . These new values must be chosen appropriately to:

- (1) respect the typing rules of \mathbf{S}_h
- (2) respect the interface of P and yield a trivial derivation.

In system \mathbf{S}_h , tracks 0 and 1 are special (they are dedicated to the premise of the **abs**-rule or the left-premise of the **app_h**-rule and also to the target of \rightarrow) and their value is fixed by 01-isomorphisms. But the value of tracks ≥ 2 may be changed: we say that they are *mutable*. We write informally $\mathbf{E}(P)$ for the set of edges nested in P whose tracks are mutable. An element $\mathbf{e} \in \mathbf{E}(P)$ is called a **mutable edge**, that may be of 3 natures:

- The edges of the source of arrows nested in types (**inner mutable edges**).
- The edges leading to an argument derivation in some **app_h**-rule (**argument edges**).
- Not edges but the **axiom roots** (p. 292) which are labelled with axiom tracks in contexts.

The set of argument edges corresponds to $\mathbf{supp}_{\text{mut}}(P)$ (Sec.13.1.1). By analogy with $\mathbf{supp}_{\text{mut}}$, we define the **mutable bisupport** of P by $\mathbf{bisupp}_{\text{mut}}(P) = \{(a, c) \in \mathbf{bisupp}(P) \mid c \in \mathbf{supp}_{\text{mut}}(\mathbf{T}^P(a))\} \cup \{(a, x, \ell \cdot c) \in \mathbf{bisupp}(P) \mid c \in \mathbf{supp}_{\text{mut}}(\mathbf{C}^P(a)(x))\}$. Implicitly:

- A right biposition $\mathbf{e} = (a, c \cdot k) \in \mathbf{bisupp}_{\text{mut}}(P)$ stands for the edge from c to $c \cdot k$ in the type $\mathbf{T}^P(a)$. Its label $\mathbf{lab}(\mathbf{e})$ is then k .
- A left biposition $\mathbf{e} = (a, x, \ell \cdot c \cdot k) \in \mathbf{bisupp}_{\text{mut}}(P)$ stands for the inner edge from c to $c \cdot k$ in the sequence type $\mathbf{C}^P(a)(x)$. We set then $\mathbf{lab}(\mathbf{e}) = k$.
- A left biposition $\mathbf{e} = (a, x, \ell)$ does not stand for a proper edge but for the “axiom root” of $\mathbf{C}^P(a)(x)|_\ell$ (we recall that $\varepsilon \notin \mathbf{supp}(\mathbf{C}^P(a)(x))$ since $\mathbf{C}^P(a)(x)$ is a forest). We set then $\mathbf{lab}(\mathbf{e}) = \ell$.

However, contexts are inactive in the typing rules and moreover, in a hybrid derivation P , every context is determined by the types and axiom tracks given in the axiom leaves (notwithstanding outer argument tracks), so that we do not need to consider mutable *left* bipositions/edges except for the **ax**-tracks in **ax**-rules. Metavariable \mathbf{e} is used to denote mutable edges. Thus, we set $\mathbf{E}(P) = \mathbf{supp}_{\text{mut}}(P) \cup \{(a, c) \in \mathbf{bisupp}_{\text{mut}}(P) \mid a \in$

$\mathbb{N}^*, c \in \mathbb{N}^*\} \cup \mathbf{Ax}^P \times \{\varepsilon\}$. A $(a, \varepsilon) \in \mathbf{Ax}^P \times \{\varepsilon\}$ can be seen as a placeholder for the new axiom track value that will be assigned to the axiom at position a .

Remark 13.5 (Axiom Roots). Note that the set of axiom roots in **ax**-rules (cf. p. 13.1.1) cannot be safely identified with \mathbf{Ax}^P because it is possible that $\text{supp}_{\text{mut}}(P) \cap \mathbf{Ax}^P \neq \emptyset$: indeed, if $a \cdot k \in \mathbf{Ax}^P$, we also have $a \cdot k \in \text{supp}_{\text{mut}}(P)$. In the latter case, $a \cdot k$ is not taken as an **ax**-rule but as the argument edge from a to $a \cdot k$.

Actually, as it has been noticed before (the hybrid construction of Sec. 13.1.3 makes use of that), *everything* in a hybrid derivation P can be computed from the **ax**-rules *via* the support of P . If we remember the notion of relabelling from Sec. 13.1.1, resetting an operable derivation P (hopefully, into a *trivial* derivation) is mainly a matter of giving new good track values to:

- the mutable tracks of the types assigned in **ax**-rules.
- the axiom tracks given in **ax**-rules
- the argument tracks of argument rules.

This suggests the notion of *referent* as well as that of *relabelling* of a derivation: the referents of P are the mutable $\mathbf{e} \in \mathbf{E}(P)$ that generate the derivation P and a *relabelling* of P is the datum of new values to assign to the referents.

Definition 13.8. Let $\mathbf{e} \in \mathbf{E}(P)$.

- \mathbf{e} is an **inner referent** (written $\mathbf{e} \in \text{ref}_{\text{in}}(P)$) if $\mathbf{e} = (a, c)$ for some $a \in \mathbf{Ax}^P$ and $c \in \text{supp}_{\text{mut}}(\mathbb{T}^P(a))$.
- \mathbf{e} is an **axiom referent** if $\mathbf{e} = (a, \varepsilon)$ for some $a \in \mathbf{Ax}^P$ (i.e. $\mathbf{e} \in \mathbf{Ax}^P \times \{\varepsilon\}$)
- \mathbf{e} is an **argument referent** if $\mathbf{e} = a$ for some $a \in \text{supp}_{\text{mut}}(P)$ (i.e. $\mathbf{e} \in \text{supp}_{\text{mut}}(P)$).

A **referent** of P is an element of the set $\text{ref}(P) = \text{ref}_{\text{in}}(P) \cup \mathbf{Ax}^P \times \{\varepsilon\} \cup \text{supp}_{\text{mut}}(P)$

Definition 13.9. A **relabelling** Θ of an operable derivation P is given by:

- A relabelling Θ_{arg} of $\text{supp}_{\text{mut}}(P)$.
- For all $a \in \mathbf{Ax}$, a relabelling Θ_a of $\mathbb{T}^P(a)$.
- An *injection* Θ_{tr} from $\mathbf{Ax} \times \{\mathbf{ax}\}$ to $\mathbb{N} \setminus \{0, 1\}$.

When such a Θ is given, we reuse the construction of the resetting induced by a relabelling of Sec. 13.1.1.

- We define a function $\Psi_{\text{supp}}^\Theta$ as the 01-resetting of $\text{supp}(P)$ induced by Θ_{arg} .
- For all $a \in \mathbf{Ax}$, we define Ψ_a^Θ as the 01-resetting of $\mathbb{T}^P(a)$ induced by Θ_a .
- We set $\Psi_{\text{tr}}^\Theta(a) = \Theta_{\text{tr}}(a, \varepsilon)$ for all $a \in \mathbf{Ax}$.

Thus, Ψ^Θ is a resetting of P (Definition 13.7). We then set $P^\Theta = \Psi^\Theta(P)$ (Proposition 13.1). Thus, P^Θ is a derivation that is isomorphic to P .

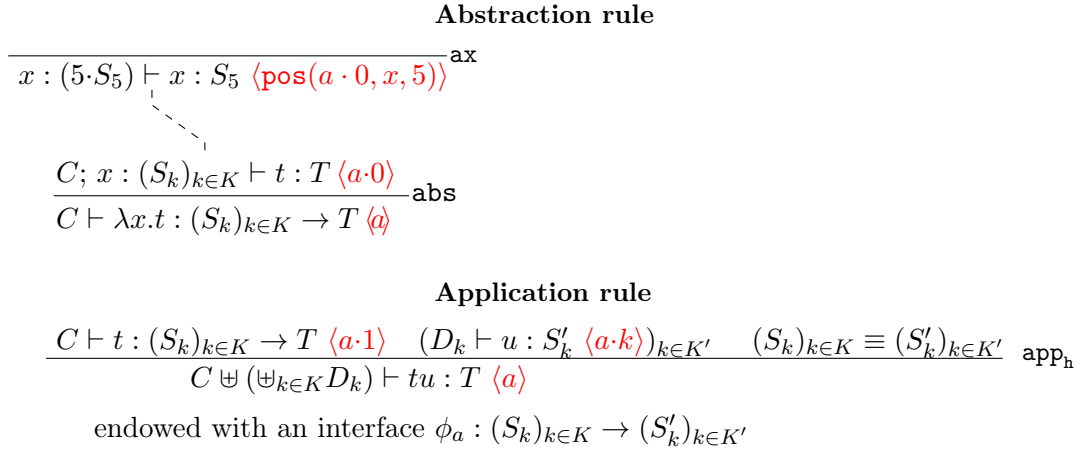


Figure 13.3: Ascendance, Polar Inversion and Consumption (Hybrid Setting)

13.4 Edge Threads

In this section, we start to use the ideas and techniques presented on p. 235 (that are also implemented in Chapter 12). For that, we will consider the notions of ascendance, polar inversion, syntactic polarity and threads.

Let us first shortly discuss the moves of a type inside a hybrid derivation by looking at Fig. 13.3 (we refer to Sec. 11.1.1, 11.3.1 and 11.3.2 for more detailed descriptions of those moves). As it was noticed in Chapter 11 and Sec. 12.2.3, types move inside a derivation (and they occur at several places). In particular, we refer to Sec. 11.1.1, 11.3.1 and 11.3.2 for more detailed descriptions of those moves and the notions of **ascendance** and **polar inversion**. In system \mathbf{S}_{op} , this can be illustrated with Fig. 13.3. For instance, in the **abs**-rule, type T “moves” from the premise (as the type of t) to the conclusion, where it is the target of the arrow type $(S_k)_{k \in K} \rightarrow T$ given to $\lambda x.t$. We say the former occurrence of T is the ascendant of the latter. Likewise, in the **app_h**-rule, the type T occurs as the target of the arrow type given to t and as the type of tu . Moreover, in the **abs**-rule, the occurrence of S_5 corresponds to that in the right-hand side of the **ax**-rule at position $\text{pos}(a, k, x)$. We also say that the occurrence of S_5 in the **ax**-rule is the ascendant of the one in the **abs**-rule. Intuitively, this means that there is a congruence, that we also⁵ denote \equiv , on $\mathbf{E}(P)$, that identifies labelled edges according to the moves of the types inside P . Concretely, two labelled edges \mathbf{e}_1 and \mathbf{e}_2 are congruent iff the typing rules of \mathbf{S}_h constrain them to be labelled with the same track, as expected. We denote by $\text{Thr}_{\mathbf{E}}(P)$ the quotient set of $\mathbf{E}(P)$ by \equiv and an element $\theta \in \text{Thr}_{\mathbf{E}}(P)$ is called a **mutable edge thread** or simply a **thread**. Of course, an isomorphism Ψ of hybrid derivations must assign the same track value to two edges \mathbf{e}_1 and \mathbf{e}_2 of the same thread θ .

This discussion explains how to capture point (1) of the beginning of Sec.13.3.3 *i.e.* respecting the typing rules of \mathbf{S}_h . For point (2), we also want to respect the interface $(\phi_a)_{a \in \text{supp}_{\text{a}}(P)}$ of P while obtaining a trivial interface (*i.e.* using only identity isomorphisms). In the **app_h**-rule of Fig. 13.3, the interface ϕ_a will map the sequence type $(S_k)_{k \in K}$ onto $(S'_k)_{k \in K'}$. Thus, ϕ_a will identify every mutable edge of $(S_k)_{k \in K}$ with a mutable edge of $(S'_k)_{k \in K'}$. We write $\theta_1 : \mathbf{e}_1 \xrightarrow{a} \mathbf{e}_2 : \theta_2$ or simply $\theta_1 \xrightarrow{a} \theta_2$ if the interface

⁵Not to be confounded with isomorphisms between labelled trees.

ϕ_a maps some $\mathbf{e}_1 \in \theta_1$ onto some $\mathbf{e}_2 \in \theta_2$ and we say then that θ_1 and θ_2 are **consumed** at position a . In that case, θ_1 and θ_2 may be labelled with different track values (since ϕ_a is just a 01-isomorphism and not simply the identity), but in order to get a trivial derivation, they must be assigned the same value by Ψ . We set $\overset{a}{\rightsquigarrow} = \cup_{a \in \text{supp}_{\text{mut}}(P)} \overset{a}{\rightsquigarrow}$. Thus, $\overset{a}{\rightsquigarrow}$ is the union of the $\overset{a}{\rightsquigarrow}$.

To prove Theorem 13.2, we must then prove that we can assign a value $\text{Val}(\theta)$ to each edge thread $\theta \in \text{Thr}_{\mathbb{E}}(P)$, such that, if $\theta_1 \xrightarrow{\phi_a} \theta_2$ for some $a \in \text{supp}_{\text{at}}(P)$, then $\text{Val}(\theta_1) = \text{Val}(\theta_2)$. This assignation Val must be *consistent i.e.* satisfy the two following points:

- No track conflict should arise from Val (in the contexts of $\text{app}_{\mathbb{h}}$ -rules).
- Two brother threads should not be mapped on the same value. Two edge threads θ_1 and θ_2 are **brother threads** if they respectively hold two distinct mutable edges \mathbf{e}_1 and \mathbf{e}_2 coming from the same node (*e.g.*, edges 4·3 and 4·8 in T_1 in Fig. 13.1 cannot *both* be mapped *e.g.*, on $7 \cdot 5$ *i.e.* *both* be relabelled with track 5).

Actually, the first condition (no track conflict) is easy and we may prove that such a good assignation Val exists iff there is no proof (based upon the interface $(\phi_a)_{a \in \text{supp}_{\text{at}}(P)}$) showing that there are two brother threads that should be given an equal track value. Such a proof would be called a **brother chain** and would have the form $\theta_0 \overset{a_0}{\overset{\leftarrow}{\rightsquigarrow}} \theta_1 \overset{a_1}{\overset{\leftarrow}{\rightsquigarrow}} \dots \overset{a_{n-1}}{\overset{\leftarrow}{\rightsquigarrow}} \theta_n$, where θ_0 and θ_n are two brother threads and $\overset{a}{\overset{\leftarrow}{\rightsquigarrow}}$ is the symmetric closure of $\overset{a}{\rightsquigarrow}$. This corresponds to a proof of equality in an *ad hoc* first order theory \mathcal{T}_P whose set of constants is $\text{Thr}_{\mathbb{E}}(P)$ and whose axioms are $\text{Val}(\theta_1) = \text{Val}(\theta_2)$ for all θ_1, θ_2 s.t. $\theta_1 \overset{a}{\rightsquigarrow} \theta_2$ for some $a \in \text{supp}_{\text{at}}(P)$.

13.4.1 Threads and Consumption of Mutable Edges

In Sec. 13.4.1, we formally define edge threads and consumption, as suggested in the introduction of Sec. 13.4 and consumption, which is a notion coming from Sec. 11.1.2.

We have abusively denoted the edges of a labelled tree or of a derivation by their deepest extremities *e.g.*, the position $3 \cdot 2 \cdot 5$ also denotes the edge from position $3 \cdot 2$ to position $3 \cdot 2 \cdot 5$. Thus, it is not a surprise that **ascendance** and **polar inversion** are defined as in Sec. 11.3.2 or in Sec. 12.2.3.

The relation of **ascendance** \rightarrow_{asc} is defined by:

- For all $a \in \text{supp}(P)$ such that $t(a) = \text{@}$, for all $c \in \text{supp}_{\text{mut}}(\mathbb{T}(a))$, $(a, c) \rightarrow_{\text{asc}} (a \cdot 1, 1 \cdot c)$.
- For all $a \in \text{supp}(P)$ such that $t(a) = \lambda x$ for some x , for all $c \in \mathbb{N}^*$ such that $1 \cdot c \in \text{supp}_{\text{mut}}(\mathbb{T}(a))$, $(a, 1 \cdot c) \rightarrow_{\text{asc}} (a \cdot 0, c)$.

Relation $\mathbf{e}_1 \rightarrow_{\text{asc}} \mathbf{e}_2$ means that \mathbf{e}_2 is the ascendant of \mathbf{e}_1 *i.e.* \mathbf{e}_1 and \mathbf{e}_2 are corresponding pointers to the same edge in the conclusion and the (left) premise of the rule at some position a . Note that an argument or axiom thread has no ascendant or descendant.

The relation of **polar inversion** \rightarrow_{pi} is defined by:

- For all $a \in \text{supp}(P)$ such that $t(a) = \lambda x$ for some x , for all $k \cdot c \in \text{supp}_{\text{mut}}(\mathbb{T}^P(a))$ (with $k \in \mathbb{N} \setminus \{0, 1\}$), $(a, k \cdot c) \rightarrow_{\text{pi}} (\text{pos}(a \cdot 0, x, k), c)$.

Relation $\mathbf{e}_1 \rightarrow_{\text{pi}} \mathbf{e}_2$ means that \mathbf{e}_E is the *polar inverse* of \mathbf{e}_1 . If $\mathbf{e}_1 = (a, c) \rightarrow_{\text{pi}} (\text{pos}(a \cdot 0, x, k), c) = \mathbf{e}_2$, then observe that when $c = \varepsilon$, then \mathbf{e}_2 is an axiom referent whereas it is an inner referent when $c \neq \varepsilon$ (and in that case, $c \in \text{supp}_{\text{mut}}(T^P(\text{pos}(a \cdot 0, x, k)))$).

Remark 13.6 (Polar Inversion with Edges). This definition of polar inversion matches that in Sec. 11.3.2, but not that in Sec. 12.2.3 since a special “biposition of emptiness” must be considered in Chapter 12. Note that Chapter 11 and Chapter 12 feature threads of *bipositions* and not threads of *edges*.

The lemma below states that two edges of the same thread are labelled with the same track, as expected from Sec. 13.4:

Lemma 13.2. Let $\mathbf{e}_1, \mathbf{e}_2 \in \text{Thr}_{\mathbf{E}}(P)$ such that $\mathbf{e}_1 \equiv \mathbf{e}_2$. Then $\text{lab}(\mathbf{e}_1) = \text{lab}(\mathbf{e}_2)$.

Proof. By induction, since the lemma is true when \equiv is replaced with \rightarrow_{asc} and \rightarrow_{pi} . \square

And more generally:

Lemma 13.3. Let Θ be a relabelling of P . Then, for all $\mathbf{e}_1, \mathbf{e}_2 \in \mathbf{E}(P)$ such that $\mathbf{e}_1 \equiv \mathbf{e}_2$. Then $\Theta(\mathbf{e}_1) = \Theta(\mathbf{e}_2)$.

Proof. By induction on \equiv . Note that the lemma is true when \equiv with \rightarrow_{asc} and \rightarrow_{pi} because of the inductive definitions that are associated to the resetting of derivation given (see Appendix B.2). \square

Consumption and Interfaces The relation **consumption** is associated with rule app_h . This notion was informally presented in Sec. 11.1.2 and then, formally defined for system \mathbf{S} in Sec. 12.2.4. However, it is more complex in system \mathbf{S}_{op} due to the presence of non-trivial interfaces. Assume $t(a) = @, t|_a = uv$ with $u : (S_k)_{k \in K} \rightarrow T$ for all $k \in K$ and $v : S'_k$ for all $k \in K'$ for all $k \in K$ with $\phi_a : (S_k)_{k \in K} \xrightarrow{\sim} (S'_k)_{k \in K'}$ as in Fig. 13.3 so that uv can be typed with T . The types S_k ($k \in K$) and S'_k ($k \in K'$) occur in the premises of the judgment typing tu , however, they are absent in this judgment. We say that they have been **consumed**. Intuitively, for all $k \in K$, the sequence type isomorphism ϕ_a given by the interface of P will identify every edge \mathbf{e} of S_k with some edge \mathbf{e}' of some $S'_{k'}$. Formally, we set, for all $a, c, c' \in \mathbb{N}^*, k, k' \in \mathbb{N} \setminus \{0, 1\}$ such that $(a \cdot 1, k \cdot c) \in \text{bisupp}_{\text{mut}}(P)$, $\phi_a(k \cdot c) = k' \cdot c'$:

- If $c \neq \varepsilon$, $(a \cdot 1, k \cdot c) \xrightarrow{a} (a \cdot k', c')$
- $(a \cdot 1, k) \xrightarrow{a} a \cdot k'$.

Indeed, the premise concluding with $u : (S_k)_{k \in K} \rightarrow T$ is at position $a \cdot 1$. The interface ϕ_a will map S_k onto the type $S'_{k'}$ (with $\rho_a = \text{Rt}(\phi_a)$ and $k' = \rho_a(k)$, so that $S'_{k'} \equiv S_k$), that occurs in the judgment $D_{k'} \vdash v : S'_{k'}$ at position $a \cdot k'$.

- A $c \in \text{supp}_{\text{mut}}(S_k)$ corresponds to the edge $k \cdot c$ in $\text{supp}((S_k)_{k \in K} \rightarrow T)$ (see the first observation in Sec. 12.2.3). But the edge of S_k that ends at some position c will be mapped (by the interface) ϕ_a on the edge of $S'_{k'}$ that ends at $c' := \phi_a|_k(c)$.
- Moreover, biposition $(a \cdot 1, k)$, that stands for the *inner* edge joining $(a \cdot 1, \varepsilon)$ with $(a \cdot 1, k)$, will be mapped on the *argument* edge joining a with $a \cdot k'$, denoted simply by $a \cdot k'$.

We set $\rightarrow = \cup\{\overset{a}{\rightarrow} \mid a \in \text{supp}_{\text{@}}(P)\}$ and write \leftarrow for the symmetric relation. Implicitly, the relations $\overset{a}{\rightarrow}$ and \rightarrow both depend on the interface ϕ of P .

Note that consumption depends on the interface ϕ of the operable derivation P , whereas ascendance and polar inversion do not.

13.4.2 Edge Threads and Syntactic Polarity

As announced in Sec. 13.4, let us define now **mutable edge threads** (see Sec. 11.1.1 and Sec. 11.1.2 for more intuitions):

Definition 13.10. Let P a hybrid derivation.

- An **ascendant (edge) thread** is an equivalence class of relation \equiv_{asc} , the reflexive, transitive, symmetric closure of \rightarrow_{asc} .
- An **edge thread** (metavariable θ) is an equivalence class of relation \equiv (see Fig. 12.3). Relation $\mathbf{e} \in \theta$ is also written $\theta : \mathbf{e}$, $\mathbf{e} : \theta$ and we say that θ **occurs** at \mathbf{e} . The thread of an edge \mathbf{e} is denoted $\text{thr}_{\mathbf{E}}^P(\mathbf{e})$ or just $\text{thr}(\mathbf{e})$.
- The **quotient set** $\mathbf{E}(P)/\equiv$ is denoted $\text{Thr}_{\mathbf{E}}(P)$.

This definition is very close to Definition 12.1. We also consider the extension of relation \rightarrow up to \equiv . We write $\theta_1 \overset{a}{\rightarrow} \theta_2$ if $\exists \mathbf{e}_1, \mathbf{e}_2$, $\theta_1 = \text{thr}(\mathbf{e}_1)$, $\theta_2 = \text{thr}(\mathbf{e}_2)$, $\mathbf{e}_1 \overset{a}{\rightarrow} \mathbf{e}_2$. Thus, $\theta_1 \overset{a}{\rightarrow} \theta_2$ iff $\theta_1 : \mathbf{e}_1 \overset{a}{\rightarrow} \mathbf{e}_2 : \theta_2$ for some $\mathbf{e}_1, \mathbf{e}_2$. In that case, we say that θ_1 (resp. θ_2) has been **left-consumed** (resp. **right-consumed**) at biposition \mathbf{e}_1 (resp. \mathbf{e}_2).

13.4.3 Brother Chains and Representation

Before studying threads and ascendant threads more precisely and defining syntactic polarity (as in Sec. 11.1), we explain why Theorem 13.2 is equivalent to a first order theory involving threads as constants, and in particular, brother threads, that will correspond to elements that must not be reassigned a same label. The concept of brother threads is informally presented in Sec. 11.1.3.

Let Θ be a relabelling of P and assume that $\phi(\mathbf{e}_L) = \mathbf{e}_R$. Thus, the edges \mathbf{e}_L and \mathbf{e}_R are identified by the interface (we have $\mathbf{e}_L \rightarrow \mathbf{e}_R$). We recall that $\text{lab}(\mathbf{e}_L)$ is the label of the edge \mathbf{e}_L , $\text{lab}(\mathbf{e}_R)$ that of \mathbf{e}_R and that $\Theta(\mathbf{e}_L)$ and $\Theta(\mathbf{e}_R)$ (that is $\Theta(\phi_a(\mathbf{e}_L))$) will be their new labels assigned by Θ . According to the discussion of 13.4, if we want Θ to produce a trivial derivation P^Θ (we say then that Θ **trivializes** P), we need \mathbf{e}_L and \mathbf{e}_R to be assigned the same label. Indeed, using the notation L^P from Sec. 13.2.1):

Lemma 13.4. Let P be an operable derivation P and Θ a relabelling of P . If, for all $p \in L^P$, $\Theta(p) = \Theta(\phi(p))$, then P^Θ is a trivial derivation.

Proof. See the proof of Lemma B.3 in Appendix B.3 □

Given an operable derivation P with the usual notations, we consider now the following first order theory \mathcal{T}_P , whose sets of constants is $\text{Thr}_{\mathbf{E}}(P)$, whose unique function symbol is Val and that holds the axioms $\text{Val}(\theta_1) = \text{Val}(\theta_2)$ for all $\theta_1, \theta_2 \in \text{Thr}_{\mathbf{E}}(P)$ such $\exists a \in \text{supp}_{\text{@}}(P)$, $\theta_1 \overset{a}{\rightarrow} \theta_2$. Intuitively, $\text{Val}(\theta_1)$ stands for $\Theta(\theta)$ (*i.e.* $\Theta(\mathbf{e})$ for any $\mathbf{e} \in \theta$)

where Θ is a relabelling that trivializes P .

Intuitively, two edges $\mathbf{e}_1, \mathbf{e}_2 \in \mathbf{E}(P)$ are *brother edges* if they correspond to two edges in the source of a same arrow, so that \mathbf{e}_1 and \mathbf{e}_2 cannot be relabelled with the same track value (see Definition 13.9 and Sec. 13.1.1). Formally, the definition of brotherhood must be a little more general (and handle axiom and argument edges).

Definition 13.11. Let P be a hybrid derivation and $\mathbf{e}_1, \mathbf{e}_2 \in \mathbf{E}(P)$.

- \mathbf{e}_1 and \mathbf{e}_2 are **strict brother edges** if they have a node in common or type *i.e.*:
 - either $\mathbf{e}_1 = a \cdot k_1, \mathbf{e}_2 = a \cdot k_2$ (with $a \in \text{supp}_{\text{mut}}(t)$) (*i.e.* argument edges of a same **app**-rule)
 - or $\mathbf{e}_1 = (a, c \cdot k_1), \mathbf{e}_2 = (a, c \cdot k_2)$ (with $a \in \text{Ax}^P$) (*i.e.* two edges that are in the source of a same arrow (whose pos. is c) in a type assigned in an axiom rule).
 - or $\mathbf{e}_1 = (\text{pos}(a, x, k_1), \varepsilon), \mathbf{e}_2 = (\text{pos}(a, x, k_2), \varepsilon)$ (with $a \in \text{supp}(P), x \in \mathcal{V}$) (*i.e.* two axiom roots of a free occurrence of some variable).

for some $k_1 \neq k_2 \geq 2$.

- \mathbf{e}_1 and \mathbf{e}_2 are **(non-strict) brother edges** if they have a node in common or they are both axiom edges *i.e.* if they are strict brother edges or $(\mathbf{e}_1, \mathbf{e}_2 \in \text{Ax}^P \times \{\varepsilon\})$ and $\mathbf{e}_1 \neq \mathbf{e}_2$).

As expected, \mathbf{e}_1 and \mathbf{e}_2 are brothers if one cannot assign them the same track *i.e.* they may be the cause of a track conflict in a relabelling. In particular, the last condition in the definition of strict brother states that axiom tracks must be chosen so that no track conflict occurs in contexts. The definition of non-strict brother edges is written to match Definitions 13.7 and 13.9. If we assign pairwise different track value to every axiom roots, then we will obtain from P a (trivial) derivation P_0 such that no axiom track is used *twice* in P_0 , which is stronger than the statement of Theorem 13.2. We will prove in Sec. 13.5.3 that this is possible.

As in Sec. 11.1.3, we extend the notion of brotherhood to threads:

Definition 13.12. $\theta_1, \theta_2 \in \text{Thr}_{\mathbf{E}}(P)$ are **brother threads** if $\theta_1 : \mathbf{e}_1$ and $\theta_2 : \mathbf{e}_2$ for some brother edges $\mathbf{e}_1, \mathbf{e}_2$.

Actually, the only thing that could go wrong is that the interface ϕ of P may syntactically constrain two brother threads to be relabelled with the same edge in a prospective relabelling that would trivialize P . Indeed, if this does not happen, we can prove that such a relabelling exists:

Proposition 13.2. Let P be an operable derivation. If \mathcal{T}_P cannot prove an equality of the form $\text{Val}(\theta_1) = \text{Val}(\theta_2)$ for some (non-strict) brother threads θ_1 and θ_2 , then P is isomorphic to a trivial derivation.

Proof. Under the assumption of the statement, we define an equivalence relation \sim on $\text{Thr}_{\mathbf{E}}(P)$ by $\theta_1 \sim \theta_2$ iff \mathcal{T}_P proves that $\text{Val}(\theta_1) = \text{Val}(\theta_2)$ *i.e.* $\mathcal{T}_P \vdash \text{Val}(\theta_1) = \text{Val}(\theta_2)$.

Since $\text{Thr}_{\mathbf{E}}(P)$ is a countable set, let \mathbf{i} be an injection from $\text{Thr}_{\mathbf{E}}(P)/\sim$ to $\mathbb{N} \setminus \{0, 1\}$. We define then a relabelling Θ of P by:

- $\Theta_{\text{arg}}(a) = \mathbf{i}(\text{thr}(a))$ for all $a \in \text{supp}_{\text{mut}}(P)$.
- $\Theta_a(c) = \mathbf{i}(\text{thr}(a, c))$ for all $a \in \text{Ax}$ and $c \in \text{supp}_{\text{mut}}(\mathbf{T}(a))$.
- $\Theta_{\text{tr}}(a) = \mathbf{i}(\text{thr}((a, t(a), \text{tr}^P(a))))$ for all $a \in \text{Ax}$.

Since no brother edges are assigned the same value by hypothesis, this definition matches the clauses of definition 13.9 and Θ is indeed a relabelling of P . By Lemma 13.4, P^Θ is a trivial derivation, that is isomorphic to P , thanks to Sec. 13.3.3. \square

According to Proposition 13.2, in order to prove Theorem 13.2, we must now check that, for any operable derivation P , there is not proof in \mathcal{T}_P that two brother threads θ_1 and θ_2 should be assigned the same value. Such a proof would involve a brother chain:

Definition 13.13. A **brother chain** is a *finite* sequence of the form $\theta_0 \xleftrightarrow{a_0} \theta_1 \xleftrightarrow{a_1} \theta_2 \dots \theta_{n-1} \xleftrightarrow{a_{n-1}} \theta_n$ where θ_0 and θ_n are two brother threads and $\xleftrightarrow{a_i}$ is the symmetric closure of $\xrightarrow{a_i}$.

With this vocable, Proposition 13.2 can be restated:

Proposition 13.3. Let P be an operable derivation. If there is no brother chain in P , then P is isomorphic to a trivial derivation.

We must then prove that brother chains *do not* exist.

13.4.4 Towards the Final Stages of the Proof

In Sec. 13.4.4, we give a global input on the final steps of the proof of Theorem 13.2 (quoted below) before implementing them in the later sections.

Theorem. Every operable derivation is isomorphic to a trivial derivation.

Since Lemma 13.1 ensures that every way of performing finite or not sequences of subject-reduction can be built-in inside an operable, it establishes that the “rigid” framework \mathbf{S} does not cause any loss of expressivity compared to type system \mathcal{R} resorting to multiset constructions.

By Proposition 13.2 and the discussion that follows, it is sufficient to prove that brother chain do not exist. We start by giving an outline of the final stages of the proof.

First, notice that, in a derivation P , some types are “called” by an **abs**-rule (*i.e.* correspond to a part of the context that becomes the source of an arrow type). For instance, in Fig. 13.3, the source $(S_k)_{k \in K}$ of the type of $\lambda x.t$ was just “called” by the **abs**-rule. We say then that the types S_k occur with a **negative syntactic polarity** in the judgment typing $\lambda x.t$ at position a . Parts of a type that cannot be traced back to an **abs**-rule are said to occur with a **positive** syntactic polarity. In P_{ex} , the only part of the type of $\lambda x.x x$ that occurs positively is the target o' . This notion extends to edges $\mathbf{e} \in \mathbf{E}(P)$: an edge \mathbf{e} occurs negatively if it can be traced back to an **abs**-rule and positive if not. If $\theta_1 : \mathbf{e}_1 \xrightarrow{\phi_a} \mathbf{e}_2 : \theta_2$ and \mathbf{e}_1 is negative (resp. positive), we say that the thread θ_1 is **left-consumed negatively** (resp. **positively**) at position a .

According to the end of Sec. 13.4.3, in order to prove Theorem 13.2, we must prove that brother chains *do not* exist. For that, we assume *ad absurdum* that there is some brother chain $\theta_0 \xleftrightarrow{a_0} \theta_1 \xleftrightarrow{a_1} \theta_2 \dots \theta_{n-1} \xleftrightarrow{a_{n-1}} \theta_n$ associated to an operable derivation P .

It is easy to see to prove that, if no thread is left-consumed negatively in the chain (in that case, we say that the chain is a **normal brother chain**), then it is of the form $\theta_0^\oplus \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} \theta_n$ and that a_1, a_2, \dots are resp. nested in an argument of a_0, a_1, \dots . This entails that $\text{ad}(a_0) < \text{ad}(a_1) < \text{ad}(a_2) \dots$. Since $\theta_0 : \mathbf{e}_0 \xrightarrow{\phi_{a_0}}$ and $\xrightarrow{\phi_{a_{n-1}}} \mathbf{e}'_n : \theta_n$ implies that the edges of the thread θ_0 are above or below $a_0 \cdot 1$ whereas some edges of θ_n are above $a_{n-1} \cdot k$ (for some $k \geq 2$). From that and $\text{ad}(a_0) < \text{ad}(a_n \cdot k)$, we deduce that θ_0 and θ_n cannot be brother, which is a contradiction.

Thus, if a brother chain existed, then it cannot be normal and at least one thread should be left-consumed negatively. But left consumption is associated to the left premise of an $\text{app}_\mathbf{p}$ -rule *i.e.* the left-hand side of an application, and negative polarity is related to abstractions. It is thus not a surprise that negative left-consumption hints at the presence of a redex: more precisely, if $\theta_i^\ominus \xrightarrow{\phi_{a_i}} \theta_{i+1}$, there is a finite reduction sequence $t \rightarrow^* t'$ such that $t'|_a$ is a *redex*. If we fire that redex, then we have $\theta_i = \theta_{i+1}$ instead of $\theta_i^\ominus \xrightarrow{\phi_{a_i}} \theta_{i+1}$: the problematic link of the chain is destroyed. This process is also the *collapsing strategy* (as in Sec. 12.4.2). If a brother chain \mathcal{C} existed, then the collapsing strategy would produce from \mathcal{C} another brother chain \mathcal{C}' that is normal. Since *normal* brother chains do not exist, we may conclude that brother chains do not exist at all, as expected. By the end of Sec. 13.4.3, this is enough to conclude that any operable derivation is isomorphic to a trivial one, which ends the proof of Theorem 13.2.

Remark 13.7. Of course, a lot is implicit in the last section: for the collapsing to be a sound process, we must first check that threads also have residuals under β -reduction in an operable derivation (as positions and bipoositions do) and that β -reduction preserves consumption. We are going to do that now.

Before giving back our attention to the technical details, let us sum up what remains to be done and the main tools used in the final proof:

- Defining properly the applicative depth of a thread and syntactic polarity.
- Proving that applicative depth increases in case of *positive* left-consumption *i.e.* if $\theta_1^\oplus \xrightarrow{\phi_{a_1}} \theta_2$, then $\text{ad}(\theta_1) < \text{ad}(\theta_2)$ (Lemma 13.11). This is the main argument to be used in the final stage of the proof (Sec. 13.5.3), the basic idea being that if $\text{ad}(\theta_1) < \text{ad}(\theta_2)$, then θ_1 and θ_2 cannot be brother threads.
- We must then discard *negative* left-consumption. For that, we implement the technique informally presented in Sec. 11.2.1 (as it is also done in Chapter 12). This demands that we reduce some redexes and thus, that we properly define *residuation* for edge threads (Sec. 13.5.1).
- In particular, we must check a few preservation properties (w.r.t. residuation) on brother threads (*e.g.*, Lemma 13.14), consumption and formalize some observations of Chapter 11

13.4.5 Top Ascendants, Syntactic Polarity and Referents

In order to prove that the hypothesis of Proposition 13.2 is satisfied by any operable derivation (and thus proving Theorem 13.2), we must describe edge threads in a more

precise way. In this section, we formalize some observations of Sec. 11.1.1 about syntactic polarity or the possible forms of a thread.

Notice that \rightarrow_{asc} is functional: if $\mathbf{e}_1 \rightarrow_{\text{asc}} \mathbf{e}_2$, we write $\mathbf{e}_2 = \text{asc}(\mathbf{e}_1)$. Notice also that asc is injective. Thus, $\mathbf{e}_1 \equiv_{\text{asc}} \mathbf{e}_2$ iff $\exists i \geq 0$, $\mathbf{e}_2 = \text{asc}^i(\mathbf{e}_1)$ or $\mathbf{e}_1 = \text{asc}^i(\mathbf{e}_2)$ (as in Sec. 12.3.1).

Given an edge $\mathbf{e} = (a, c) \in \mathbf{E}(P)$, asc may only add the prefix 0 or 1 to a and add/remove the prefix 1 to c , by induction:

Lemma 13.5. If $(a_1, c_1) \equiv_{\text{asc}} (a_2, c_2)$ then $\exists a_3 \in \{0, 1\}^*$, $(a_2 = a_1 \cdot a_3$ or $a_1 = a_2 \cdot a_3)$ and $\exists i \geq 0$, $(c_2 = 1^i \cdot c_1$ or $c_1 = 1^i \cdot c_2)$.

Lemma 13.5 corresponds to Lemma 12.2. As in Sec. 11.1.3, we notice that, in a thread, every occurrence has the same applicative depth, which is a consequence of Lemma 13.5 along with the definition below:

Definition 13.14. Let P a hybrid derivation $\mathbf{e} \in \mathbf{E}(P)$. We define the **applicative depth** of \mathbf{e} (denoted $\text{ad}(\mathbf{e})$).

- If $\mathbf{e} = (a, c)$, then $\text{ad}(\mathbf{e}) = \text{ad}(a)$.
- If $\mathbf{e} = a \in \text{supp}_{\text{mut}}(P)$, then $\text{ad}(\mathbf{e}) = \text{ad}(a)$.

We set, for all $\mathbf{e} \in \text{bisupp}_{\text{mut}}(P)$, $\text{Asc}(\mathbf{e}) = \text{asc}^i(\mathbf{e})$, where i is maximal (*i.e.* $\text{asc}^i(\mathbf{e})$ is defined, but not $\text{asc}^{i+1}(\mathbf{e})$). Thus, $\text{Asc}(\mathbf{e})$ is the **top ascendant** of \mathbf{e} . Remark that if $\mathbf{e} = (a, \varepsilon)$ with $a \in \text{Ax}^P$ or $\mathbf{e} = a \in \text{supp}_{\text{mut}}(P)$, $\text{Asc}(\mathbf{e}) = \mathbf{e}$. As noted in Observation 11.1, p. 241, a top ascendant is either located in an **ax**-node or an **abs**-node (asc is total on **app**-nodes), motivating the notion of syntactic polarity (see Sec. 11.1.2 for examples) for mutable edges:

Definition 13.15.

- Let $\mathbf{e} \in \mathbf{E}(P)$. We define the **polarity** of \mathbf{e} as follows:
 - If $\text{Asc}(\mathbf{e}) = (a_0, c_0) \in \text{ref}_{\text{in}}(P) \cup \text{Ax} \times \{\varepsilon\}$ with $t(a_0) = x$ (resp. $t(a_0) = \lambda x$) for some $x \in \mathcal{V}$, then we set $\text{Pol}(\mathbf{e}) = \oplus$ (resp. $\text{Pol}(\mathbf{e}) = \ominus$).
 - If $\mathbf{e} = a \in \text{supp}_{\text{mut}}(P)$, then $\text{Pol}(\mathbf{e}) = \oplus$.
- If $\text{thr}(\mathbf{e}) = \theta$ and $\text{Pol}(\mathbf{e}) = \oplus/\ominus$, we say that θ occurs positively/negatively at \mathbf{e} .
- If θ is left/right-consumed at \mathbf{e} and $\text{Pol}(\mathbf{e}) = \oplus$ (resp. $\text{Pol}(\mathbf{e}) = \ominus$), we say that θ is left/right-consumed positively (resp. . negatively) at edge \mathbf{e} .

Then, we write for instance $\theta_1 \oplus \xrightarrow{a} \ominus \theta_2$ to mean that θ_1 is left-consumed positively and θ_2 is right-consumed negatively in the **app**-rule at position a . We can compare Definition 13.15 is quite similar to Definition 12.3.

Since \rightarrow_{pi} also defines an injective function and $\mathbf{p}_1 \rightarrow_{\text{pi}} \mathbf{p}_2$ implies that \mathbf{p}_1 (in a λx) and \mathbf{p}_2 (in an axiom) do not have ascendants, giving Lemma 13.6 (compare with Lemma 12.3):

Lemma 13.6. For all $\mathbf{e}_1, \mathbf{e}_2 \in \mathbf{E}(P)$ such that $\mathbf{e}_1 \equiv \mathbf{e}_2$, we have $\text{Pol}(\mathbf{e}_1) = \oplus$ and $\text{Pol}(\mathbf{e}_2) = \ominus$ iff $\text{Asc}(\mathbf{e}_1) \rightarrow_{\text{pi}} \text{Asc}(\mathbf{e}_2)$.

Lemmas 13.5 and 13.6 may be illustrated by Fig. 11.1, p. 240). Lemma 13.6 means an edge thread can have *at most* two connex components (ascendant threads) *e.g.*, the green or the purple threads in Fig. 11.1. Note that an argument thread contains only one element (an argument does not have ascendants or descendants).

Remark 13.8.

- If the top occurrence of an ascendant thread is in an **ax**-rule typing a variable x s.t. x is not bound in t , then the thread has one (positive) “connex component” *e.g.*, the red thread in Fig. 11.1.
- Since we consider threads of mutable edges, no thread can have only negative occurrences (contrary to the blue thread in Fig. 11.1), because if $(a, c) \in \mathbf{E}(P)$, $t(a) = \lambda x$ and (a, c) does not have an ascendant, then (1) 1 is not a prefix of c (if we had $c = 1 \cdot c_0$, then $(a \cdot 0, c_0)$ would be an ascendant of (a, c)) (2) $c \neq \varepsilon$ (because $(a, \varepsilon) \notin \mathbf{E}(P)$ when $a \notin \mathbf{Ax}^P$). Thus, $c = k \cdot c_0$ for some $k \in \mathbb{N} \setminus \{0, 1\}$ and (a, c) has a *positive* polar inverse which is $(\text{pos}(a \cdot 0, x, k), c_0)$.
- Note that a thread containing an axiom edge cannot be consumed positively: indeed, if $\mathbf{e} = (a, \varepsilon)$ with $a \in \mathbf{Ax}^P$, then \mathbf{e} cannot be in the domain of the consumption relation \rightarrow (by definition of \rightarrow) and does not have a descendant

By Remark 13.8, we can define the following notion, intuitively presented in Sec. 11.1.3:

Definition 13.16. Let P be a hybrid derivation and $\theta \in \mathbf{Thr}_{\mathbf{E}}(P)$.

- The **referent** of θ , denoted $\mathbf{ref}(\theta)$ is the top ascendant of the positive ascendant thread included in θ .
- A thread whose referent is an argument referent (resp. an axiom referent resp. an inner referent) is said to be an **argument thread** (resp. an **axiom thread**, resp. an **inner thread**).
- The **applicative depth** of θ , denoted $\mathbf{ad}(\theta)$, is defined by $\mathbf{ad}(\theta) = \mathbf{ad}(\mathbf{ref}(\theta))$.

Since $(a, k \cdot c) \rightarrow_{\text{pi}} (a', c)$ implies $a \leq a'$ and so $\mathbf{ad}(a) \leq \mathbf{ad}(a')$, we have, for all $\mathbf{e} \in \theta$, $\mathbf{ad}(\mathbf{e}) \leq \mathbf{ad}(\theta)$, as observed in Sec. 11.1.3.

The referent of a thread θ is the unique element of the intersection $\theta \cap \mathbf{ref}(P)$. Given a relabelling Θ of the hybrid derivation P (Definition 13.9), the referent of a mutable edge \mathbf{e} may be seen as the unique representative of $\mathbf{thr}(\mathbf{e})$ that allows us to directly compute $\Theta(\mathbf{e})$ from Θ , according to Lemma 13.3.

Since a consumed biposition does not have a descendant, Lemma 13.5 and 13.6 imply Lemma 13.7, which is the formal version⁶ of Observation 11.3, p. 11.3:

Lemma 13.7 (Uniqueness of Consumption). Let $\otimes \in \{\oplus, \ominus\}$ and $\theta \in \mathbf{Thr}$, $\theta \neq \theta_{\perp}$. Then, there is a most one θ' s.t. $(\theta^{\otimes} \dot{\rightarrow} \theta'$ or $\theta^{\otimes} \dot{\leftarrow} \theta')$.

⁶As Lemma 12.4 also is.

Brotherhood, Threads and Consumption Lemma 13.8 below means that axiom threads may only occur at the root of sources of arrows and that consumption, when the left-hand side is negative, identifies some axiom threads with argument threads:

Lemma 13.8.

- If $\theta_L \xrightarrow{\sim} \theta_R$ and θ_L is an axiom thread, then θ_R is an argument thread.
- If $\theta_L^{\ominus} \xrightarrow{\sim} \theta_R$ and θ_R is an argument thread, then θ_L is an axiom thread.

Proof. See the proof of Lemma B.4 in Appendix B.3. \square

Lemma 13.9. Let P be a hybrid derivation and $e_1, e_2 \in \mathbf{E}(P)$ two brother edges.

- If $e_1 \equiv e'_1 \in \mathbf{E}(P)$, then there is a brother edge $e'_2 \in \mathbf{E}(P)$ of e'_1 such that $e_2 \equiv e_1$.
- In particular, $\mathbf{ref}(\mathbf{thr}(e_1))$ and $\mathbf{ref}(\mathbf{thr}(e_2))$ are brother edge.
- Moreover, if $\mathbf{thr}(e_1)$ and $\mathbf{thr}(e_2)$ are not axiom threads, then they have the same applicative depth.

Proof. Straightforward by induction on \equiv_{asc} . Note that, by Definition 13.11, two brother edges have the same applicative depth except perhaps when they are axiom edges. \square

Moreover, thread brotherhood is compatible with consumption in the following sense, which captures Observation 11.7:

Lemma 13.10. Let θ_1 and θ_2 be two brother edges and $a \in \mathbf{supp}_{\text{@}}(P)$. Then θ_1 is consumed at position a iff θ_2 is. Moreover:

- If $\theta_1 \xrightarrow{a} \theta_1^{\mathbf{R}}$, then, there is $\theta_2^{\mathbf{R}}$, brother with $\theta_1^{\mathbf{R}}$, such that $\theta_2 \xrightarrow{a} \theta_2^{\mathbf{R}}$.
- If $\theta_1^{\mathbf{L}} \xrightarrow{a} \theta_1$, then, there is $\theta_2^{\mathbf{L}}$, brother with $\theta_1^{\mathbf{L}}$, such that $\theta_2^{\mathbf{L}} \xrightarrow{a} \theta_2$.

A monotonicity property relates consumption and applicative depth, provided the left thread is consumed with a positive polarity. Lemma 13.11 is the formal counterpart of Observation 11.6:

Lemma 13.11 (Monotonicity). If $\theta_L^{\oplus} \xrightarrow{\sim} \theta_R$, then $\mathbf{ad}(\theta_L) < \mathbf{ad}(\theta_R)$.

Proof. We have $\theta_L : \mathbf{e}_L^{\oplus} \xrightarrow{a} \mathbf{e}_R : \theta_R$ for some $\mathbf{e}_L = (a \cdot 1, k \cdot c)$, $\mathbf{e}_R = (a \cdot k, c) \in \mathbf{E}(P)$, $a \in \mathbf{supp}_{\text{@}}(P)$, $k \geq 2$ and $c \in \mathbb{N}^*$. By the sentence just above Lemma 13.14, $\mathbf{ad}(\theta_L) = \mathbf{ad}(\mathbf{e}_L)$, so that $\mathbf{ad}(\mathbf{e}_L) = \mathbf{ad}(a \cdot 1) = \mathbf{ad}(a)$. We conclude with $\mathbf{ad}(\theta_R) \geq \mathbf{ad}(\mathbf{e}_R) = \mathbf{ad}(a \cdot k) = \mathbf{ad}(a) + 1$. \square

13.5 Residuation of Threads and the Collapsing Strategy

In order to apply the monotonicity lemma (Lemma 13.11) and conclude the proof of Theorem 13.2 in Sec. 13.5.3, we must avoid the cases of *negative* left-consumption.

13.5.1 Edges and Residuation

In this section, we informally discuss the behavior of residuation of edges and threads, and we state the main properties of residuation w.r.t. threads.

Residuation is needed because, in order to discard negative left-consumption in a brother chain related to a term t , one will actually (finitarily) reduce the term t . This implies that the notion of residual of thread must be properly defined. Since threads are sets of mutable edges, we must first define residuation for edges.

In this section, we only give the main properties of residuation. See Appendix B.4 for the details and the formal definitions. In particular, quasi-residuation is defined for mutable edges in Appendix B.4.1.

First, let us observe informally that a distinction should be made between residuation for nodes and residuation for edges.

Consider an operable derivation $P \triangleright C \vdash t : T$ and assume that $t|_b = (\lambda x.r)s$, $t \xrightarrow{b} t'$ (so that $t'|_b = r[s/x]$), $P \xrightarrow{b} P'$ so that P' is a residual operable derivation of P . Assume that $a \in \text{supp}_{\text{@@}}(P)$, $\bar{a} = b$ and $a \cdot k \in \text{supp}_{\text{mut}}(P)$ i.e. $\overline{a \cdot k} = b \cdot 1$, which is the position of subterm s in t .

- As a *position*, $a \cdot k$ points to the argument of the redex. This position has a residual since the argument derivation at position argument derivation will be moved. For instance, in Fig. 13.2, the subderivation P_5 at position $a \cdot 5$ is moved.
- As an *edge*, $a \cdot k$ represents the argument edge from a judgment typing the redex to an argument derivation. This edge is destroyed. For instance, in Fig. 13.2, the edge from the root of the derivation typing $(\lambda x.r)s$ to P_5 is destroyed and does not have a residual.

Thus, as a position, $a \cdot k$ has a residual, but it does not as an edge.

Relations \rightarrow_{asc} and \rightarrow_{pi} are compatible with reduction. This entails, by induction on \equiv :

Lemma 13.12. If $e_1 \equiv e_2$, then $\text{QRes}_b^E(e_1)$ is defined iff $\text{QRes}_b^E(e_2)$ is. In that case, $\text{QRes}_b^E(e_1) = \text{QRes}_b^E(e_2)$.

Lemma 13.12 allows us to define **(quasi)-residuals** for *edge threads*. We set $\text{Res}_b(\theta) = \text{thr}'(\text{QRes}_b^E(e))$ for any $e : \theta$ (where $\text{thr}'(\cdot)$ denotes threads in P'). Residuation is compatible with consumption in the following sense:

Lemma 13.13. Let P be an operable derivation whose interface is ϕ . Assume that $\theta_1 \rightarrow \theta_2$.

- Then $\text{Res}_b(\theta_1)$ is defined iff $\text{Res}_b(\theta_2)$ is.
- In that case, $\text{Res}_b(\theta_1) \xrightarrow{\sim} \text{Res}_b(\theta_2)$ or $\text{Res}_b(\theta_1) = \text{Res}_b(\theta_2)$.
- Moreover, $\text{Res}_b(\theta_1) = \text{Res}_b(\theta_2)$ iff $(\theta_1$ occurs in the left key of the redex and is not an axiom thread).

Proof. A complete statement can be found in Lemma B.6. □

The 3rd point of Lemma 13.13 means that θ_1 must occur in the source of the type of $\lambda x.r$, the abstraction of the reduced redex. Thus, the threads of the left key of a redex (which are not axiom threads) can be *collapsed* by reduction, which is generalized in Sec. 13.5.2.

Likewise, (strict) brotherhood is compatible with residuation:

Lemma 13.14. Let P be a hybrid derivation and θ_1, θ_2 be two strict brother threads. Then $\mathbf{Res}_b(\theta_1)$ is defined iff $\mathbf{Res}_b(\theta_2)$ is. In that case, $\mathbf{Res}_b(\theta_1) = \mathbf{Res}_b(\theta_2)$.

13.5.2 The Collapsing Strategy for Operable Derivations

Now that residuation is defined for threads, we can explain how to discard negative left-consumption by using Lemma 13.13. This will allow us to invoke the monotonicity lemma (Lemma 13.11), as suggested in the end of Sec. 13.4.4.

Let P be an operable derivation whose interface is denoted ϕ . Assume that we have $\theta_L^\ominus \xrightarrow{a} \theta_R$ for some $\theta_L, \theta_R \in \mathbf{Thr}_E(P)$ and that θ_L is *not* an axiom thread.

Thus, $\theta_L : \mathbf{e}_L^\oplus \xrightarrow{a} \mathbf{e}_R : \theta_R$ for some $\mathbf{e}_1 := (\alpha \cdot 1, k \cdot c)$, $\mathbf{e}_2 := (\alpha \cdot k', c')$ with $\alpha \in \mathbf{supp}_@ (P)$, $k, k' \geq 2$ and $c, c' \in \mathbb{N}^*$. Let $(\alpha_\lambda, c_\lambda) = \mathbf{Asc}(\mathbf{e}_L)$, so that $t(\alpha_\lambda) = \lambda x$ for some x since \mathbf{e}_L is negative. Then (see Sec. 11.2.1, particular, Fig. 11.4) there is a maximal $a \leq \alpha_\lambda$ such that $t(a) = @$. Note that $t|_a$ is a redex. We define the **collapsing strategy w.r.t.** θ_L^\ominus by induction on $h := |\alpha_\lambda| - |a|$:

- Case $h = 1$: then $\alpha_\lambda = a \cdot 1$ and $a = \alpha$ and there is a redex in t at position \bar{a} . We fire it and the strategy is completed. By Lemma 13.13, $\mathbf{Res}_b(\theta_L) = \mathbf{Res}_b(\theta_R)$ (both members of this equality are defined since θ_L is not an axiom thread).
- Case $h > 1$: We fire the redex at position \bar{a} , so that, by Lemma 13.13, $\mathbf{Res}_b(\theta_L) \xrightarrow{a} \mathbf{Res}_b(\theta_R)$, but the height h has decreased by 2, and we go on with the strategy.

All this is related to the notion of redex tower and can be formalized, as in Sec. 12.4.2.

Let \mathbf{rs} be the sequence of reductions representing the collapsing strategy. Lemma 13.13 entails that $\mathbf{Res}_{\mathbf{rs}}^{\mathbf{ref}}(\theta_L) = \mathbf{Res}_{\mathbf{rs}}^{\mathbf{ref}}(\theta_R)$. Thus:

Lemma 13.15. If $\theta_L^\ominus \xrightarrow{\sim} \theta_R$ and θ_L is not an axiom thread, then, there is a reduction sequence \mathbf{rs} such that $\mathbf{Res}_{\mathbf{rs}}(\theta_L) = \mathbf{Res}_{\mathbf{rs}}(\theta_R)$.

Lemma 13.15 is extremely important: it entails that we can discard negative left-consumption by a finite sequence of reduction.

13.5.3 Conclusion of the Proof

In Sec. 13.5.3, we prove the two final steps to obtain Theorem 13.2, namely:

Definition 13.17. A brother chain is said to be **normal** if no threads is left-consumed negatively in it.

Proposition 13.4. If a brother chain exists w.r.t. some operable derivation P typing a term t , then there exists a *normal* brother chain w.r.t. a derivation P' typing a reduct of t .

Proposition 13.5. There is no *normal* brother chain.

Indeed, Propositions 13.4, 13.5 and 13.3 entail Theorem 13.2 (and thus, Theorem 13.1).

Proof of Proposition 13.4. Assume that there is a normal brother chain $\theta_0 \xrightarrow{a_0} \theta_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} \theta_n$ w.r.t. a an operable derivation typing a term t , where θ_0 and θ_n are brother threads.

We describe now an algorithm to produce from \mathcal{C} a normal brother chain \mathcal{C}' that is minimal.

By Lemma 13.7. There may not be “crossings” of the form $\theta_{i-1} \xrightarrow{a_{i-1}} \oplus \theta_i \xrightarrow{a_i} \theta_{i+1}$. Moreover, if there is a crossing $\theta_{i-1} \xleftarrow{a_{i-1}} \oplus \theta_i \xrightarrow{a_i} \oplus \theta_{i+1}$, then (still by Lemma 13.7), $a_{i-1} = a_i$ and $\theta_{i-1} = \theta_{i+1}$, so that we may assume that this kind of crossing never occurs (if a some point these is one, we immediately remove it).

Likewise, we may remove every crossing of the form $\theta_{i-1} \xleftarrow{a_{i-1}} \ominus \theta_i \xrightarrow{a_i} \theta_{i+1}$, since this implies that $\theta_{i-1} = \theta_{i+1}$.

Since argument threads only occur positively and axiom threads can be only be consumed negatively (3rd point of Remark 13.8), Lemmas 13.7 and 13.8 imply that no θ_i may be an argument thread and that no θ_i may be an axiom thread that is left-consumed negatively: if not, we would have $n = 1$ and \mathcal{C} would be $\theta_0 \xrightarrow{a_0} \theta_1$ with θ_0 axiom thread and θ_1 (or $\theta_0 \xleftarrow{a_0} \theta_1 \dots$), which is not a brother chain!

Thus, there are no axiom threads in \mathcal{C} . This allows us to resort to the collapsing strategy (Lemma 13.15): we can reduce t so that we obtain a normal brother chain \mathcal{C}' associated to a reduct t' of t . Notice that, by Lemma 13.13, this process terminates and the length of \mathcal{C}' is smaller than that of \mathcal{C} . \square

We prove now Proposition 13.5.

Proof of Proposition 13.5. We proceed *ad absurdum* and consider a *normal* brother chain $\theta_0 \xrightarrow{a_0} \theta_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} \theta_n$ where θ_0 and θ_n are brother threads. By the proof of Proposition 13.4, there is no axiom thread in the chain and we can also assume that they are no “crossing” of the form $\xleftarrow{\oplus} \theta_i \xrightarrow{\ominus}$.

Thus, there may be only four kinds of crossings in \mathcal{C} . $\theta_{i-1} \oplus \xrightarrow{\alpha_{i-1} \ominus} \theta_i \oplus \xrightarrow{\alpha_i} \theta_{i+1}$ (right-right) or $\theta_{i-1} \xleftarrow{\alpha_{i-1} \oplus} \theta_i \ominus \xleftarrow{\alpha_i} \oplus \theta_{i+1}$ (left-left) or $\theta_{i-1} \oplus \xrightarrow{\alpha_{i-1} \ominus} \theta_i \oplus \xleftarrow{\alpha_i} \oplus \theta_{i+1}$ (minus-plus) or $\theta_{i-1} \oplus \xrightarrow{\alpha_{i-1} \oplus} \theta_i \ominus \xleftarrow{\alpha_i} \oplus \theta_{i+1}$ (plus-minus)

We cannot have only crossings of kind right-right. If it were, by Lemma 13.11, we would have $\text{ad}(\theta_0) < \text{ad}(\theta_n)$. But, as it has been observed above, θ_0 and θ_n cannot be axiom threads. By Lemma 13.9, we should have $\text{ad}(\theta_0) = \text{ad}(\theta_n)$, which is a contradiction. For the same reason, we cannot only have crossings of kind left-left.

Thus, there must be at least a crossing of kind minus-plus or of kind plus-minus. This is easy to see that there can only be one (there cannot be more that one “change of direction” in \mathcal{C}' : if not, we would have a crossing on the form $\theta_{i-1} \xleftarrow{\ominus} \theta_i \xrightarrow{\oplus} \theta_{i+1}$ and this case is not possible for \mathcal{C}). In both cases, the chain starts with $\theta_0 \oplus \xrightarrow{a_0} \theta_1$ and ends with $\theta_{n-1} \xleftarrow{a_{n-1}} \oplus \theta_n$. Lemmas 13.10 and 13.7 entail that $a_0 = a_{n-1}$ and θ_1 and θ_{n-1} are brother

threads. Thus, $\theta_1 \xleftrightarrow{a_1} \dots \xleftrightarrow{a_{n-2}} \theta_{n-1}$ is also a brother chain. By induction on $i \leq n/2$, we show that θ_i and θ_{n-i} are brother threads, so that n is even and $i_0 := n/2$ is a natural number and θ_{i_0} is brother with itself, which is impossible (by Lemma 13.9). Thus, \mathcal{C} cannot exist. This concludes the proof of Proposition 13.5 and of Theorem 13.2. \square

13.6 Conclusion

Theorem 13.2 establishes that every \mathcal{R} -derivation Π (*i.e.* any derivation based on multiset constructions and a coinductive type grammar) is the collapse of an \mathbf{S} -derivation P and that we even could internalize every sequence of reduction choices of \mathcal{R} in \mathbf{S} .

From Sec. 13.4.3 and 13.5.3, we may remember that the technique to deal with non-productive reduction (*i.e.* reduction that does not normalize) introduced here is the following: instead of trying to normalize terms, we try to normalize *proofs* (these proofs are called *brother chains* here, whereas we normalized another kind called *nihilating chains* in Chapter 12), as it was announced in the presentation p. 235.

The same technique was actually later used *twice* in Chapter 12 (first, to prove that every term was \mathbf{S} -typable, second, to prove that \mathbf{S} captured the order of the terms) but by considering another first order theory.

The question of the surjectivity of the collapse of the irrelevant version of \mathbf{S} on $\mathcal{D}_{\mathbf{w}}$ (*i.e.* is every $\mathcal{D}_{\mathbf{w}}$ -derivation the collapse of a possibly irrelevant \mathbf{S} -derivation?) remains open.

Conclusion

Aussi ne peut-il y avoir d'autre terme que l'épuisement du voyageur explorant ce paysage inépuisable, contemplant la carte approximative qu'il en a dressée et à demi rassuré seulement d'avoir obéi de son mieux dans sa marche à certains élans, certaines pulsions. [...]. À sa recherche, il progresse laborieusement, tâtonne en aveugle, s'engage dans des impasses, s'embourbe, repart — et, si l'on veut à tout prix tirer un enseignement de sa démarche, on dira que nous avançons toujours sur des sables mouvants..

Claude Simon, *Discours de Stockholm*

Now that the time has come to part company, we hope that the reader is convinced of the use of non-idempotent intersection and union types in general, and of sequential intersection in particular: they give syntax-direction, easy proofs of termination in the finite case, nice combinatorial features without duplication, possibility to express a validity criterion in the infinite case or to study non-productive reduction. We also hope that the methodological motto has proved⁷ its worth and its utility:



This thesis's thesis

The introduction of a type system should always come along with a figure representing subject reduction from the perspective of derivations trees.

This maxim, that will, to be sure, become a classic in every kindergarten textbook within a few years, was illustrated by Figures 3.1, 3.2, 3.3, 6.8, 7.8, 10.5 and 13.2, and are all akin to Figure 2.6, representing β -reduction.

Besides drawing figures, we used non-idempotent types to characterize head and strong normalization in the λ_μ -calculus, infinitary weak normalization and to prove that every term can be typed in a non-trivial way with infinitary types. Incidentally, on the way, we showed that we did not lose anything by considering sequential intersection instead of multiset intersection. In general, non-idempotent types have not yet shown all their possible applications as good-behaviors certificates or as quantitative descriptors. There are also probably many *operational* or *observational* properties of the λ -calculus or of one of its variant, that could be given simpler *semantic* proofs involving them.

As a conclusion of this document, we give a summary of our work along with interesting perspectives, as well as a few insights on another contribution regarding intersection type theory, which takes a different direction from what has been presented here: namely, Melliès and Zeilberger's "type as functors" approach.

⁷It certainly helped the author penetrate the jungle of intersection type theory in any case and, as a somewhat more important consequence, secured the prospective buffet following his PhD defense.

Perspectives on Part II: the λ_μ -Calculus

We extended the non-idempotent intersection types of the λ -calculus into non-idempotent intersection and *union* types for the λ_μ -calculus. This allowed us to provide type-theoretic characterizations of head and strong normalization in the λ_μ -calculus, along with upper bounds to the length of the head reduction strategy and of all reductions sequences in the former and in the latter case respectively. We also introduced a small-step operational semantics $\lambda_{\mu\tau}$ extending that of the λ_μ -calculus. The characterization of strong normalization in the λ_μ -calculus (by means of system $\mathcal{S}_{\lambda_\mu}$) was then extended characterizing strong normalization in $\lambda_{\mu\tau}$ (by means of system $\mathcal{S}_{\lambda_{\mu\tau}}$). This work can be furthered in different directions (conclusion on p. 184):

The λ_μ -calculus and classical logic (future work)

- Obtaining exact bounds for normalizing reduction sequence (instead of just upper bounds) *à la* B-L [13].
- Proving that inhabitation is decidable as for non-idempotent intersection [18].
- Studying the models associated *e.g.*, to $\mathcal{H}_{\lambda_\mu}$, investigating the quantitative to qualitative collapse as in [41].
- Providing quantitative inter. and union. types for other classical calculi, *e.g.*, the $\lambda\mu\tilde{\mu}$ [32].

Perspective on Part III: Infinitary Normalization

We provided a type-theoretic characterization of the set of hereditary head normalizing λ -terms, thus answering positively to Klop's question. Simultaneously, we gave a semantic proof that the hereditary head reduction strategy is complete for (infinitary) weak normalization in the infinitary calculus Λ^{001} , which is an extension of arguments that were hitherto used in the finite case. Last, we characterized the set of hereditary permutations by means of **S**-types, which gives a positive answer to TLCA Problem # 20. Many natural extensions of these contributions come to mind (conclusion on p. 232):

Infinitary normalization and system **S**, beyond Klop's question

- Identifying other sets of Böhm trees (besides hereditary permutations) that can be characterized with system **S**.
- Characterizing strong normalization in Λ^{001} .
- Extending the characterization of WN (and possibly SN) to the other infinitary calculi Λ^{111} and Λ^{101} .
- Relations of system **S** with Bucciarelli-Ehrhard's indexed Linear Logic [16], with Grellois-Melliès' infinitary model of LL [51].

Perspective on Part IV: Non-Productive Reduction

We proved that (1) every term is typable in system \mathcal{R} and in system **S** (when **S** is not endowed with the approximability criterion) (2) the order of a term t could be extracted from the set of \mathcal{R} -types that can be assigned to t (3) every derivation of system \mathcal{R} is the collapse of a derivation system **S** (4) every reduction choice sequence w.r.t. a derivation of system \mathcal{R} can be encoded in a derivation of system **S**. The method used in Part IV allows us to work with completely unstable terms (mute terms) and to emancipate our-

selves from productivity. If for any term t , we set $\llbracket t \rrbracket = \{(\Gamma, \tau) \mid \triangleright_{\mathcal{R}} \Gamma \vdash t : \tau\}$, then we obtain a relational model [17] of the λ -calculus, that can also be denoted \mathcal{R} . Following the conclusions on p. 284 and 317, we suggest exploring next:



The model \mathcal{R} (open questions)

- Studying the equational theory of \mathcal{R} .
- Does infinitary subject expansion hold (without approximability)?
- Can we extract a tree-like semantics from \mathcal{R} ?
- Applications to observational theories (TLCA Problem # 18), possible new semantic proofs, etc.
- Is the collapse from \mathcal{R}_w to \mathcal{D}_w (irrelevant non-idempotent intersection to irrelevant idempotent intersection) surjective?
- Investigating whether the proofs of Chapters 12 and 13 could be reformulated into Girard’s Geometry of Interaction or Ludics [48].

Polyadic Approximations and Fibrations and Intersection Types: a categorical interpretation of intersection type theory

We conclude by saying a few words about a work that was not presented in this thesis, that provides a categorical understanding of intersection type systems (whereas we have been more focused on the “concrete machinery” of typing throughout this document).

Starting from an exact correspondence between **polyadic affine approximations** and non-idempotent intersection types, we have developed, with Mazza and Pellissier, a general framework [81] for building intersection types systems characterizing normalization properties in the finite case. This construction, which uses in a fundamental way Melliès and Zeilberger’s “type systems as functors” viewpoint [82], allows us to recover equivalent versions of every well-known intersection type system (including Coppo and Dezani’s original system, as well as its non-idempotent variants independently introduced by Gardner and de Carvalho).

An intersection type system is then defined as a **colored operad** (*i.e.* a symmetric multicategory with possibly more than one object) endowed with 2-arrows, that is, arrows between arrows: a multimorphism generalizes the notion of typed judgment $\Gamma \vdash t : B$, where t is now thought as a multimorphism from Γ to B . Intuitively, there is a 2-arrow from $\Gamma \vdash t : B$ to $\Gamma \vdash t' : B$ if the latter judgment is obtained from the former by subject reduction steps.

An intersection type system is built from a given restriction of the λ -calculus (*e.g.*, the set of λ -terms endowed with head reduction) and a given subset of the simply typed polyadic calculus by a **pullback construction**, inspired from the **Grothendieck construction**. This framework enables us to almost automatically build new systems of intersection types in this way. The proofs of the termination properties (*i.e.* properties of the form “if t is typable, then t is normalizing”) in the most known type systems (*e.g.*, $\mathcal{D}\Omega$, $\mathcal{D}_{0,w}$, \mathcal{R}_0 , \mathcal{G}) are now subsumed by the proof of strong normalization of propositional multiplicative exponential linear logic. When operads are extended into *cyclic* operads (*i.e.* an operad in which categorical rules allow permuting inputs and outputs, in order to handle the fact that, intuitively, some λ_μ -terms have several outputs), the type system $\mathcal{H}_{\lambda_\mu}$ from Chapter 7 can be recovered with the pullback construction.

Related contribution

- Extending Melliès and Zeilberger’s “type systems as functors” to intersection type theory.
- A Grothendieck-like construction allowing us to recover most well-known intersection type systems and to build new ones.
- A universal proof of termination resorting to the strong normalization of MELL.

Interestingly, infinitary normalization (as in Chapter 10) does not fit in this framework, notably because of the approximability criterion, but also because the notion of productivity is difficult to capture from a categorical perspective.

Thus, giving a categorical definition of productivity is currently the main challenge to provide an abstract account of typing and its semantical consequences, which was also left unaddressed in Melliès and Zeilberger’s work.

Future work on the “Types as functors” viewpoint

- Capturing the notion of productivity in the categorical framework.

Bibliography

- [1] Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Inf. Comput.*, 105(2):159–267, 1993.
- [2] Beniamino Accattoli, Eduardo Bonelli, Delia Kesner, and Carlos Lombardi. A nonstandard standardization theorem. In Peter Sewell, editor, *Proceedings of the 41st Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2014.
- [3] Beniamino Accattoli and Delia Kesner. The structural *lambda*-calculus. In Anuj Dawar and Helmut Veith, editors, *Proceedings of 24th EACSL Conference on Computer Science Logic*, volume 6247 of *Lecture Notes in Computer Science*, pages 381–395. Springer-Verlag, August 2010.
- [4] Shahin Amini and Thomas Erhard. On Classical PCF, Linear Logic and the MIX Rule. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*, volume 41 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 582–596. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.
- [5] Y. Andou. Church-rosser property of a simple reduction for full first-order classical natural deduction. *Annals of Pure Applied Logic*, 119(1-3):225–237, 2003.
- [6] Zena M. Ariola, Hugo Herbelin, and Alexis Saurin. Classical call-by-need and duality. In C.-H. Luke Ong, editor, *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, volume 6690 of *Lecture Notes in Computer Science*, pages 27–44. Springer-Verlag, 2011.
- [7] David Baelde, Amina Doumane, and Alexis Saurin. Infinitary proof theory: the multiplicative additive case. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*, volume 62 of *LIPIcs*, pages 42:1–42:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
- [8] Henk Barendregt. *The Lambda-Calculus: Its Syntax and Semantics*. Ellis Horwood series in computers and their applications. Elsevier, 1985.
- [9] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Bulletin of Symbolic Logic*, 48:931–940, 1983.
- [10] Henk Barendregt and Jan Willem Klop. Applications of infinitary lambda calculus. *Inf. Comput.*, 207(5):559–582, 2009.

- [11] A. Berarducci. Infinite lambda-calculus and non-sensible models. *Lecture Notes in Pure and Applied Mathematics*, 180:339–377, 1996.
- [12] Alessandro Berarducci and Benedetto Intrigila. Some new results on easy lambda-terms. *Theor. Comput. Sci.*, 121(1&2):71–88, 1993.
- [13] Alexis Bernadet and Stéphane Lengrand. Complexity of strongly normalising λ -terms via non-idempotent intersection types. In Martin Hofmann, editor, *Foundations of Software Science and Computation Structures (FOSSACS)*, volume 6604 of *Lecture Notes in Computer Science*. Springer-Verlag, 2011.
- [14] Alexis Bernadet and Stéphane Lengrand. Non-idempotent intersection types and strong normalisation. *Logical Methods in Computer Science*, 9(4), 2013.
- [15] Antonio Bucciarelli, Alberto Carraro, Giordano Favro, and Antonino Salibra. Graph easy sets of mute lambda terms. *Theor. Comput. Sci.*, 629:51–63, 2016.
- [16] Antonio Bucciarelli and Thomas Ehrhard. On phase semantics and denotational semantics: the exponentials. *Ann. Pure Appl. Logic*, 109(3):205–241, 2001.
- [17] Antonio Bucciarelli, Thomas Ehrhard, and Giulio Manzonetto. Not enough points is enough. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 298–312. Springer-Verlag, 2007.
- [18] Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. The inhabitation problem for non-idempotent intersection types. In Díaz et al. [37], pages 341–354.
- [19] Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Strong normalization through intersection types and memory. *Electr. Notes Theor. Comput. Sci.*, 323:75–91, 2016.
- [20] Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Mathematical Structures in Computer Science.*, 2017.
- [21] Felice Cardone and J. Roger Hindley. *Lambda-Calculus and Combinators in the 20th Century*, volume 5 of *Handbook of the History of Logic*. Elsevier, 2009.
- [22] Daniel De Carvalho. *Sémantique de la logique linéaire et temps de calcul*. PhD thesis, Université Aix-Marseille, November 2007.
- [23] Daniel De Carvalho. Execution time of lambda terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, 2017.
- [24] Alonzo Church. A Note on the Entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936.
- [25] Alonzo Church. An unsolvable problem of elementary number theory. *Amer. J. of Math.*, (58):345–363, 1936.
- [26] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

- [27] Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type assignment for lambda-terms. *Archive for Mathematical Logic*, 19:139–156, 1978.
- [28] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the λ -calculus. *Notre Dame Journal of Formal Logic*, 4:685–693, 1980.
- [29] Thierry Coquand. *Une théorie des constructions. Thèse de Troisième Cycle*. PhD thesis, Université Paris 7, 1985.
- [30] Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In Bruno Buchberger, editor, *EUROCAL '85, European Conference on Computer Algebra, Linz, Austria, April 1-3, 1985, Proceedings Volume 1: Invited Lectures*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184. Springer, 1985.
- [31] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88 (LNCS 41.7)*. Springer-Verlag, 1990.
- [32] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 233–243. ACM, 2000.
- [33] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North-Holland Co., Amsterdam, 1958. (3rd edn. 1974).
- [34] Lukasz Czajka. A coinductive confluence proof for infinitary lambda-calculus. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8560 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2014.
- [35] Ferruccio Damiani and Paola Giannini. A decidable intersection type system based on relevance. In Hagiya and Mitchell [54], pages 707–725.
- [36] Erika De Benedetti and Simona Ronchi Della Rocca. Bounding normalization time through intersection types. In Graham-Lengrand and Paolini [50], pages 48–57.
- [37] Josep Díaz, Ivan Lanese, and Davide Sangiorgi, editors. *Proceedings of the 8th International Conference on Theoretical Computer Science (TCS)*, volume 8705 of *Lecture Notes in Computer Science*. Springer-Verlag, 2014.
- [38] Daniel J. Dougherty, Silvia Ghilezan, and Pierre Lescanne. Characterizing strong normalization in the curien-herbelin symmetric lambda calculus: Extending the coppo-dezani heritage. *Theoretical Computer Science*, 398(1-3):114–128, 2008.
- [39] Andrej Dudenhefner and Jakob Rehof. Intersection type calculi of bounded dimension. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 653–665. ACM, 2017.

- [40] Andrej Dudenhefner and Jakob Rehof. Typability in bounded dimension. In Ouaknine [87], pages 1–12.
- [41] Thomas Ehrhard. Collapsing non-idempotent intersection types. In Patrick Cégielski and Arnaud Durand, editors, *Proceedings of 26th EACSL Conference on Computer Science Logic*, volume 16 of *LIPICs*, pages 259–273. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
- [42] Jörg Endrullis, Helle Hvid Hansen, Dimitri Hendriks, Andrew Polonsky, and Alexandra Silva. A coinductive framework for infinitary rewriting and equational reasoning. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPICs*, pages 143–159. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
- [43] Philippa Gardner. Discovering needed reductions using type theory. In Hagiya and Mitchell [54], pages 555–574.
- [44] Gerhard Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, (39):405–431, 1934.
- [45] Jean-Yves Girard. Une extension de l’interprétation de Gödel l’analyse, et son application à l’élimination des coupures dans l’analyse et la théorie des types. In J.E. Fenstad, editor, *Proceedings of the Scandinavian Logic Symposium, North-Holland*, 1971.
- [46] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l’arithmétique d’ordre supérieur*. PhD thesis, Université Paris 7, 1972.
- [47] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [48] Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.
- [49] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1990.
- [50] Stéphane Graham-Lengrand and Luca Paolini, editors. *Proceedings of the Sixth Workshop on Intersection Types and Related Systems (ITRS), Dubrovnik, Croatia, 2012*, volume 121 of *Electronic Proceedings in Theoretical Computer Science*, 2013.
- [51] Charles Grellois and Paul-André Melliès. An infinitary model of linear logic. In Andrew M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2015.
- [52] Charles Grellois and Paul-André Melliès. Relational semantics of linear logic and higher-order model checking. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*, volume 41 of *LIPICs*, pages 260–276. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

- [53] Timothy Griffin. A Formulae-as-Types Notion of Control. In *17th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–58. ACM Press, 1990.
- [54] Masami Hagiya and John C. Mitchell, editors. *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, volume 789 of *Lecture Notes in Computer Science*. Springer, 1994.
- [55] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1980.
- [56] Giuseppe Jacopini. A condition for identifying two elements of whatever model of combinatory logic. In Corrado Böhm, editor, *Lambda-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, March 25-27, 1975*, volume 37 of *Lecture Notes in Computer Science*, pages 213–219. Springer, 1975.
- [57] Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. Infinitary lambda calculus. *Theor. Comput. Sci.*, 175(1):93–125, 1997.
- [58] Delia Kesner. A theory of explicit substitutions with safe and full composition. *Logical Methods in Computer Science*, 5(3:1):1–29, 2009.
- [59] Delia Kesner. Reasoning about call-by-need by means of types. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9634 of *Lecture Notes in Computer Science*, pages 424–441. Springer-Verlag, 2016.
- [60] Delia Kesner and Daniel Ventura. Quantitative types for the linear substitution calculus. In Díaz et al. [37], pages 296–310.
- [61] Delia Kesner and Daniel Ventura. A resource aware computational interpretation for herbelin’s syntax. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, volume 9399 of *Lecture Notes in Computer Science*, pages 388–403. Springer-Verlag, 2015.
- [62] Delia Kesner and Pierre Vial. Types as Resources for Classical Natural Deduction. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 4-7, 2017, Oxford, England*, pages 1–15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
- [63] Assaf Kfoury. A linearization of the lambda-calculus and consequences. Technical report, Boston University, 1996.
- [64] Assaf Kfoury and Joe Wells. Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science*, 311(1-3):1–70, 2004.
- [65] Kentaro Kikuchi and Takafumi Sakurai. A translation of intersection and union types for the $\lambda\mu$ -calculus. In Jacques Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *Lecture Notes in Computer Science*, pages 120–139. Springer-Verlag, 2014.

- [66] Stephen C. Kleene. λ -definability and recursiveness. *Duke Mathematical Journal*, (2):340–353, 1936.
- [67] Stephen Cole Kleene. On the interpretation of intuitionistic number theory. *J. Symb. Log.*, 10(4):109–124, 1945.
- [68] Jean-Louis Krivine. *Lambda-calculus, types and models*. Ellis Horwood series in computers and their applications. Masson, 1993.
- [69] Toshihiko Kurata. A type theoretical view of böhm-trees. In *Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4, 1997, Proceedings*, pages 231–247, 1997.
- [70] J. Lambek and P.J. Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge University Press, 1986.
- [71] Peter J. Landin. Correspondence between ALGOL 60 and church’s lambda-notation: part I. *Commun. ACM*, 8(2):89–101, 1965.
- [72] Peter J. Landin. A correspondence between ALGOL 60 and church’s lambda-notations: Part II. *Commun. ACM*, 8(3):158–167, 1965.
- [73] Olivier Laurent. On the denotational semantics of the untyped lambda-mu calculus, 2004. Unpublished note.
- [74] Jean-Jacques Lévy. An algebraic interpretation of equality in some models of the lambda calculus. *Lambda Calculus and Computer Science Theory (LNCS No. 37)*, 1975.
- [75] G. Longo. Set-theoretical models of lambda calculus: Theories, expansions and isomorphisms. *Annals of Pure and Applied Logic*, 1983.
- [76] Giulio Manzonetto. *Lambda Calculus, Linear Logic and Symbolic Computation*. Mémoire d’habilitation à diriger des recherches, Université Paris-Nord, 2017.
- [77] Betti Venneri Mario Coppo, Mariangiola Dezani-Ciancaglini. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27:45–58, 1981.
- [78] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. Rose and J. Shepherdson, editors, *Proceedings of the Logic Colloquium '73, North Holland Co., Amsterdam*, 1975.
- [79] Per Martin-Löf. An Intuitionistic Theory of Types (technical report, 1972). In Giovanni Sambin and Jan Smith, editors, *Twenty-Five Years of Constructive Type Theory*, pages 127–172. Clarendon Press, Oxford, 1998.
- [80] Damiano Mazza. An infinitary affine lambda-calculus isomorphic to the full lambda-calculus. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 471–480. IEEE Computer Society, 2012.

- [81] Damiano Mazza, Luc Pellissier, and Pierre Vial. Polyadic Approximations, Fibrations and Intersection types. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2018, Los Angeles, USA, January 8-13, 2017 (to appear)*, pages 1–26, 2018.
- [82] Paul-André Melliès and Noam Zeilberger. Functors are type refinement systems. In *Proceedings of POPL*, pages 3–16, 2015.
- [83] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
- [84] Hiroshi Nakano. A modality for recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*, pages 255–266. IEEE Computer Society, 2000.
- [85] Peter Møller Neergaard and Harry G. Mairson. Types, potency, and idempotency: why nonlinearity and amnesia make a type system work. In Chris Okasaki and Kathleen Fisher, editors, *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 138–149. ACM Press, 2004.
- [86] Luke Ong and Steven J. Ramsay. Verifying higher-order functional programs with pattern matching algebraic data type s. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 587–598. ACM Press, 2011.
- [87] Joel Ouaknine, editor. *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 2017.
- [88] Michele Pagani and Simona Ronchi Della Rocca. Solvability in resource lambda-calculus. In Luke Ong, editor, *Foundations of Software Science and Computation Structures*, volume 6014 of *Lecture Notes in Computer Science*, pages 358–373. Springer-Verlag, 2010.
- [89] Michel Parigot. $\lambda\mu$ -calculus: an algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer-Verlag, July 1992.
- [90] Michel Parigot. Classical proofs as programs. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Computational Logic and Proof Theory, Third Kurt Gödel Colloquium, KGC'93, Brno, Czech Republic, August 24-27, 1993, Proceedings*, volume 713 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 1993.
- [91] Michel Parigot. Proofs of strong normalisation for second order classical natural deduction. *J. Symb. Log.*, 62(4):1461–1479, 1997.
- [92] Pierre-Marie Pédrot and Alexis Saurin. Classical by-need. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 616–643. Springer-Verlag, 2016.

- [93] Emmanuel Polonovski. *Substitutions explicites, logique et normalisation*. Thèse de doctorat, Université Paris 7, 2004.
- [94] Dag Prawitz. Natural deduction: A proof-theoretical study. In *Acta Universitatis Stockholmiensis, Stockholm studies in philosophy*, volume 3. Almqvist & Wicksell, 1965.
- [95] Christophe Raffalli. Data types, infinity and equality in system af_2 . In Egon Börger, Yuri Gurevich, and Karl Meinke, editors, *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers*, volume 832 of *Lecture Notes in Computer Science*, pages 280–294. Springer, 1993.
- [96] Luigi Santocanale. A calculus of circular proofs and its categorical semantics. Technical Report RS-01-15, BRICS, Dept. of Computer Science, University of Aarhus, May 2001.
- [97] Helmut Schwichtenberg. Definierbare Funktionen im λ -Kalkül mit Typen. *Arch. Math Logik* 17, pages 113 – 114, 1976.
- [98] Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001.
- [99] Robert I. Soare. *Turing Computability - Theory and Applications*. Theory and Applications of Computability. Springer, 2016.
- [100] William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967.
- [101] Makoto Tatsuta. Types for hereditary head normalizing terms. In Jacques Garrigue and Manuel V. Hermenegildo, editors, *Functional and Logic Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14-16, 2008. Proceedings*, volume 4989 of *Lecture Notes in Computer Science*, pages 195–209. Springer, 2008.
- [102] Makoto Tatsuta. Types for hereditary permutators. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 83–92. IEEE Computer Society, 2008.
- [103] Alan Turing. On computable numbers with an application to the "Entscheidungsproblem". *Proceeding of the London Mathematical Society*, 1936.
- [104] Alan M. Turing. Computability and λ -definability. *J. Symb. Log.*, 2(4):153–163, 1937.
- [105] Pawel Urzyczyn. The emptiness problem for intersection types. *Journal of Symbolic Logic*, 64(3):1195–1215, 1999.
- [106] Steffen van Bakel. Complete restrictions of the intersection type discipline. *Theor. Comput. Sci.*, 102(1):135–163, 1992.
- [107] Steffen van Bakel. Intersection type assignment systems. *Theor. Comput. Sci.*, 151(2):385–435, 1995.

- [108] Steffen van Bakel. Completeness and partial soundness results for intersection and union typing for lambda- $\mu\mu$ -. *Ann. Pure Appl. Logic*, 161(11):1400–1430, 2010.
- [109] Steffen van Bakel. Sound and complete typing for lambda-mu. In Elaine Pimentel, Betti Venneri, and Joe B. Wells, editors, *Proceedings Fifth Workshop on Intersection Types and Related Systems, ITRS 2010, Edinburgh, U.K., 9th July 2010.*, volume 45 of *EPTCS*, pages 31–44, 2010.
- [110] Steffen van Bakel, Franco Barbanera, and Ugo de'Liguoro. Characterisation of strongly normalising lambda-mu-terms. In Graham-Lengrand and Paolini [50], pages 1–17.
- [111] Steffen van Bakel, Franco Barbanera, and Ugo de'Liguoro. Intersection types for the lambda-mu calculus. *CoRR*, abs/1704.00272, 2017.
- [112] Femke van Raamsdonk, Paula Severi, Morten Heine Sørensen, and Hongwei Xi. Perpetual reductions in lambda-calculus. *Inf. Comput.*, 149(2):173–225, 1999.
- [113] Lionel Vaux. Convolution lambda-bar-mu-calculus. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 381–395. Springer-Verlag, 2007.
- [114] Pierre Vial. Infinitary intersection types as sequences: A new answer to Klop's problem. In Ouaknine [87], pages 1–12.
- [115] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1925–1927.

Appendix A

Complements to Klop's Problem

A.1 Expanding the Π'_n and Π'

- $\rho_1 = [] \rightarrow o$
- $\rho_{n+1} = [\rho_k]_{1 \leq k \leq n} \rightarrow o$
- A term t typed with ρ_{n+1} can be fed with an argument u typed with ρ_1, \dots, ρ_n to give the term $t u$ of type o .
- $\Gamma_n = x : ([o] \rightarrow o)_{n-1} + [o] \rightarrow o$.

We set:

$$\Psi_1 = \frac{\frac{f : [[]] \rightarrow o \vdash f : [] \rightarrow o}{f : [[]] \rightarrow o \vdash f(x x) : o}}{f : [[]] \rightarrow o \vdash \Delta_f : \rho_1}$$

and, for all $k \geq 2$:

$$\Psi_k = \frac{\frac{f : [[o] \rightarrow o] \vdash f : [o] \rightarrow o}{f : [[o] \rightarrow o]; x : [\rho_i]_{1 \leq i \leq k} \vdash f(x x) : o}}{f : [[o] \rightarrow o] \vdash \Delta_f : \rho_k} \quad \left(\frac{x : [\rho_{k-1}] \vdash x : \rho_k}{x : [\rho_i]_{1 \leq i \leq k-1} \vdash x x : o} \right)_{1 \leq i < k-2}$$

We may then define:

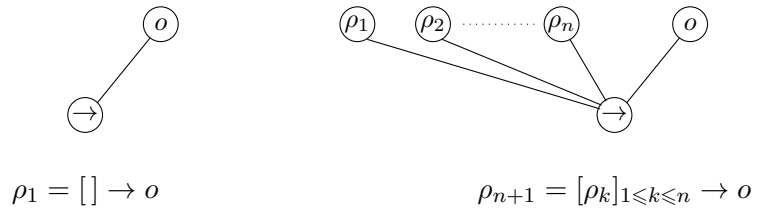
$$\Pi_1 = \frac{\Psi_1 \triangleright f : [[]] \rightarrow o \vdash \Delta_f : [[]] \rightarrow o}{\Gamma_1 : \Upsilon_f}$$

and, for all $n \geq 2$:

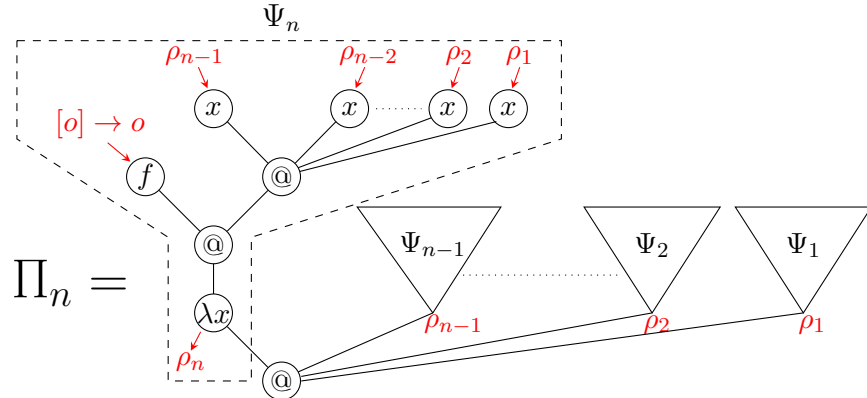
$$\Pi_n = \frac{\Psi_n \triangleright f : [[o] \rightarrow o] \vdash \Delta_f : \rho_n \quad (\Psi_k \triangleright f : [[\dots] \rightarrow o] \vdash \Delta_f : \rho_k)_{1 \leq k \leq n-1}}{\Gamma_n \vdash \Upsilon_f : o}$$

By induction, on n , we prove that:

Lemma. For all $n \geq 1$, the derivation Π_n is obtained from Π'_n by n steps of subject expansion.

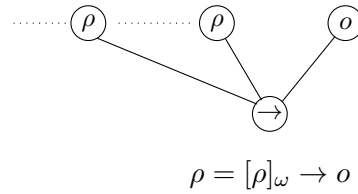


The derivation Π_n below is obtained from Π_n^k (Fig. 10.2) after k steps of expansion.
 The subderivation Ψ_n types Δ_f with ρ_n .



Remark: Π_1 is a bit different, it just assigns $[] \rightarrow o$ to f .

Intuitively, the family $(\rho_n)_{n \geq 1}$ is “directed” (ρ_n is a truncation of ρ_{n+1}).
 The infinite type ρ below is the “join” of this family.



Since $(\rho_n)_{n \geq 1}$ is “directed”, $(\Psi_n)_{n \geq 0}$ and then, $(\Phi_n)_{n \geq 0}$ are also directed
 (Ψ_n and Φ_n are truncations of Ψ_{n+1} and Φ_{n+1} resp.).
 Their resp. joins are then Ψ and Π below.

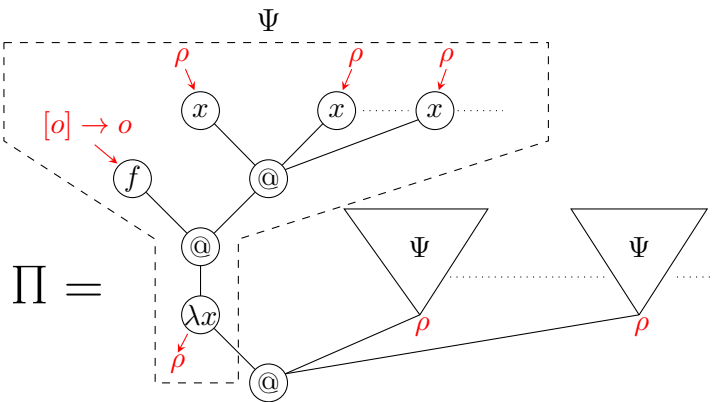


Figure A.1: Expanding the Π_n^k and Π'

The last part of this appendix is more informal. We explain how the derivation Π' typing f^ω can be expanded into a derivation Π typing Y_f by taking the infinite reduction sequence $Y_f \rightarrow^\infty f^\omega$ backwards. The derivations Π_n play a key role and Π will be obtained by taking their join them. First, we observe that, for all $k < n$:

- The type ρ_k is a truncation of ρ_n since $[\rho_i]_{1 \leq i < k}$ is obviously a truncation of $[\rho_i]_{1 \leq i < n}$ (so that $\rho_k = [\rho_i]_{1 \leq i < k} \rightarrow o$ is a truncation $\rho_n = [\rho_i]_{1 \leq i < n} \rightarrow o$). Thus, the family (ρ_n) is “directed”.
- The derivation Ψ_k is a truncation of Ψ_n thanks to the first observation.
- The derivation Π_k is a truncation of Π_n .

Those observations are corroborated by drawing ρ_n, Ψ_n, Π_n for small values of n . We then observe that the type ρ coinductively defined by $\rho = [\rho]_\omega \rightarrow o$ is the “join” of the types ρ_n (the family (ρ_n) is directed and every finite truncation of ρ is “included” in a ρ_n for some great enough n). We define

$$\Psi = \frac{\frac{f : [[o] \rightarrow o] \vdash f : [o] \rightarrow o}{x : [\rho] \vdash x : \rho} \quad \left(\frac{x : [\rho] \vdash x : \rho}{x : [\rho] \vdash x : \rho} \right)_\omega}{\frac{f : [[o] \rightarrow o]; x : [\rho]_\omega \vdash f(xx) : o}{x : [\rho]_\omega \vdash xx : o}}{\frac{f : [[o] \rightarrow o] \vdash \Delta_f : \rho}{f : [[o] \rightarrow o] \vdash \Delta_f : \rho}}$$

Then Ψ is the “infinitary join” of the Ψ_n , notably because the ρ is the infinitary join of the ρ_n .

$$\Pi = \frac{\Psi_n \triangleright f : [[o] \rightarrow o] \vdash \Delta_f : \rho \quad (\Psi_k \triangleright f : [[o] \rightarrow o] \vdash \Delta_f : \rho)_\omega}{\Gamma \vdash Y_f : o}$$

We note likewise that, intuitively, Π is the “infinitary join” of the Π_n , and Π conclude, as expected, with $\Gamma \vdash Y_f : o$.

An example of Unsound Derivation

Using the type ρ , we set:

$$\Pi_\Delta := \frac{\frac{x : [\rho] \vdash x : \rho}{x : [\rho] \vdash x : \rho} \quad \left(\frac{x : [\rho] \vdash x : \rho}{x : [\rho] \vdash x : \rho} \right)_\omega}{\frac{x : [\rho]_\omega \vdash xx : o}{\vdash \Delta : \rho}}$$

Then Ω can be typed with o :

$$\Pi_\Omega := \frac{\Pi_\Delta \triangleright \vdash \Delta : \rho \quad (\Pi_\Delta \triangleright \vdash \Delta : \rho)_\omega}{\vdash \Omega :}$$

A.2 Equinecessity, Reduction and Approximability

The rigid constructions presented here ensure "trackability", contrary to multiset constructions of system \mathcal{R} . We show here a few applications useful to prove that approximability is stable under reduction or expansion (Lemma 10.5). We consider a derivation P , with the usual associated notations (including $A = \text{supp}(P)$).

A.2.1 Equinecessary bipoositions

Definition A.1. Let p_1, p_2 two bipoositions of P .

- We say p_1 **subjugates** p_2 if, for all finite ${}^fP \leq P$, $p_1 \in {}^fP$ implies $p_2 \in {}^fP$.
- We say p_1 and p_2 are **equinecessary** (written $p_1 \leftrightarrow p_2$) if, for all finite ${}^fP \leq P$, $p_1 \in {}^fP$ iff $p_2 \in {}^fP$.

Moreover, let $B_1, B_2 \subseteq P$. Then B_1 and B_2 are **equinecessary** if every $p_1 \in B_1$ (resp. $p_2 \in B_2$) is equinecessary to some $p_2 \in B_2$ (resp. $p_1 \in B_1$).

There are many elementary equinecessity cases that are easy to observe. We need only a few ones and we define $\text{asc}(\mathbf{p})$ and $\text{Asc}(\mathbf{p})$ s.t. $\mathbf{p} \leftrightarrow \text{asc}(\mathbf{p})$ and $\mathbf{p} \leftrightarrow \text{Asc}(\mathbf{p})$ for all $\mathbf{p} \in \text{bisupp}(P)$.

- $\text{asc}(\mathbf{p})$ is defined for any $\mathbf{p} \in \text{bisupp}(P)$ which is not in an axiom leaf.
 - *Left bipoositions*: $\text{asc}(a, x, k \cdot c) = (a \cdot \ell, x, k \cdot c)$, where $\ell \geq 0$ is the unique integer s.t. $(a \cdot \ell, x, k \cdot c) \in \text{bisupp}(P)$.
 - *Right bipoositions (abs)*: if $t(\bar{a}) = \lambda x$, $\text{asc}(a, \varepsilon) = (a \cdot 0, \varepsilon)$, $\text{asc}(a, 1 \cdot c) = (a \cdot 0, c)$ and $\text{asc}(a, k \cdot c) = (a \cdot 0, x, k \cdot c)$ if $k \geq 2$.
 - *Right bipoositions (app)*: if $t(\bar{a}) = @$, $\text{asc}(a, c) = (a \cdot 1, 1 \cdot c)$.
- $\text{Asc}(\mathbf{p})$ is a *right bipoosition* and is defined as the highest right bipoosition related to \mathbf{p} by asc .
 - *Right bipoositions*: if $\mathbf{p} = (a, c)$, let h be maximal such that $\text{asc}^h \mathbf{p}$ is defined. In that case, we set $\text{Asc}(\mathbf{p}) = \text{asc}^h(\mathbf{p})$.
 - *Left bipoositions*: if $\mathbf{p} = (a, x, k \cdot c)$, let h be maximal (if it exists) such that $\text{asc}^h(\mathbf{p})$ exists. In that case, $\text{asc}^h(\mathbf{p})$ is of the form $(a_0, x, k \cdot c)$ with $t(a_0) = x$. We set then $\text{Asc}(\mathbf{p}) = (a_0, c)$.

Since $t \in \Lambda^{001}$ (and not in $\Lambda^{111} - \Lambda^{001}$), $\text{Asc}(\mathbf{p})$ is defined for any right bipoosition \mathbf{p} . If \mathbf{p} is quantitative, then $\text{Asc}(\mathbf{p})$ is also defined for any left bipoosition.

An examination of the **app**-rule shows that, if $t(a) = @$, for all $k \geq 2$ and $c \in \mathbb{N}^*$, $(a \cdot 1, k \cdot c) \leftrightarrow (a \cdot k, c)$.

Assume $t|_b = (\lambda x.r)s$. A very important case of equinecessity is this one (with the same notations as in Sec. 10.3.5 e.g., $\bar{a} = b$): $(a \cdot 1, k \cdot c) \leftrightarrow (a \cdot 10 \cdot a_k, x, k \cdot c)$ and $(a \cdot 1, k \cdot c) \leftrightarrow (a \cdot k, c)$. Thus, $(a \cdot 10 \cdot a_k, c) \leftrightarrow (a \cdot k, c)$.

A.2.2 Approximability is stable under (anti)reduction

We assume here that $P \rightarrow P'$ (P is still assumed to be quantitative). Let ${}^0B \subseteq \mathbf{bissupp}(P)$ a finite subset. We notice that \mathbf{Res}_b is defined for any right biposition which is on an axiom leaf typing $y \neq x$.

So, let 0B_* be the set obtained from $\mathbf{Asc}(B)$ by replacing any $(a \cdot 10 \cdot a_k, c)$ by $(a \cdot k, c)$. Then $|{}^0B_*| \leq |{}^0B|$ (so that 0B_* is finite) and any $\mathbf{p} \in {}^0B$ is equinecessary with a $\tilde{\mathbf{p}} \in {}^0B_*$. So the partial proof of Lemma 10.5 is valid for 0B_* . By equinecessity, it entails it is also valid for 0B .

For the converse implication, we just have to replace ${}^0B'$ by $\mathbf{Asc}({}^0B')$.

Remark A.1. This also allows us to prove that a derivation P is approximable iff P is quantitative and, for all finite set of right bipositions ${}^0B \subseteq \mathbf{bissupp}(P)$, there exists ${}^fP \leq P$ such that ${}^0B \subseteq \mathbf{bissupp}({}^fP)$. The implication \Rightarrow is obvious by Definition 10.4 and Lemma 10.5 (first point).

Let us prove the converse implication. Let ${}^0B \subseteq \mathbf{bissupp}({}^fP)$ be a finite set of bipositions. Since P is quantitative, for all $\mathbf{p} \in {}^0B$, $\mathbf{Asc}(b)$ is defined. We set ${}^0B_* = \mathbf{Asc}({}^0B)$, so that 0B_* is a set of *right* bipositions equinecessary to 0B . By hypothesis, there is ${}^fP \leq P$ such that ${}^0B_* \subseteq \mathbf{bissupp}({}^fP)$. By equinecessity, ${}^0B \subseteq \mathbf{bissupp}({}^fP)$.

A.2.3 Equinecessity and Bipositions of Null Applicative Depth

One may conjecture that every biposition (in a quantitative derivation P) is equinecessary with a **root biposition** *i.e.* a biposition that is located in the root judgment of P . Such a biposition is of the form $\mathbf{p} = (0, c)$ with $c \in \mathbf{supp}(\mathbf{T}^P(0))$ or $\mathbf{p} = (0, x, k \cdot c)$ with $k \cdot c \in \mathbf{supp}(\mathbf{C}^P(0)(x))$.

This would imply that, for a derivation P to be approximable, it is enough to have: “ P is quantitative and, for all ${}^0B \subseteq \mathbf{bissupp}(P)$ finite set of *root* bipositions, there exists ${}^fP \leq P$ such that ${}^0B \subseteq \mathbf{bissupp}({}^fP)$ ”. We call this condition **root approximability**. $\mathbf{p} \in {}^0B$,

This is actually true in the finite case and so, for the approximable derivation. However, this is not true for *any* derivation. We exhibit a counter-example of this conjecture in this section *i.e.* a derivation that root approximable but not approximable. We present this counter example with a \mathcal{R} -derivation whereas it should be a \mathcal{S} -derivation (since approximability is only an informal notion for System \mathcal{R} , Sec. 10.3.4), but it is easier to understand that way. Corresponding \mathcal{S} -derivations are not difficult to define from our presentation.

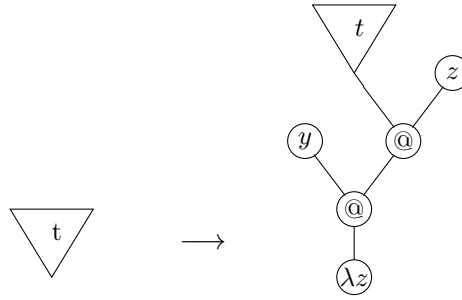
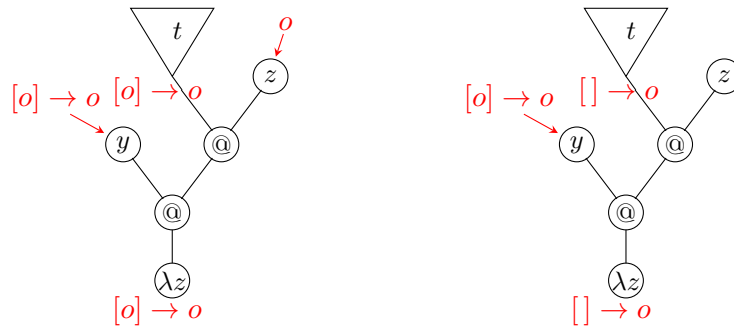
The idea is to use a strongly converging sequence $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \dots \rightarrow^\infty t'$ such that no reduction step is erasing but there is a variable $x \in \mathbf{fv}(t_0) = \mathbf{fv}(t_1) = \dots$ such that $x \notin \mathbf{fv}(t')$ (*i.e.* there is an asymptotic erasure).

Let $\Delta_* = \lambda x.(\lambda z.y(x x z))$ and $t = \Delta_* \Delta_*$, so that $t \rightarrow \lambda z.y(t z)$ (see Fig. A.2). We have $t \equiv_\beta Y(\lambda t x.(\lambda z.y(t x)))$.

Note that $t f$ strongly converges to the term $t' = y^\omega$, which does not contain f . Indeed, $t f \rightarrow (\lambda z.y(t z))f \rightarrow y(t f)$ (non-erasing steps).

There are two derivations Ψ_1 and Ψ_2 respectively concluding with $y : [[o] \rightarrow o]_\omega \vdash \lambda z.y(t z) : [o] \rightarrow o$ and $y : [[o] \rightarrow o]_\omega \vdash \lambda z.y(t z) : [] \rightarrow o$. They are obtained from Ψ'_1 and Ψ'_2 from Fig A.2 by a one-step expansion:

Let Π_Ω a derivation concluding with $\vdash \Omega : o$. This derivation is unsound (it types the mute term Ω) and is intuitively not approximable: it is impossible to find a finite

Figure A.2: Reduction of t 

$$\Psi'_1 \triangleright y : [[o] \rightarrow o]_\omega \vdash \lambda z.y(tz) : [o] \rightarrow o \quad \Psi'_2 \triangleright y : [[o] \rightarrow o]_\omega \vdash \lambda z.y(tz) : [] \rightarrow o$$

Figure A.3: Two Derivations typing $t' := \lambda z.y(tz)$

derivation (of \mathcal{R}_0) concluding with $\vdash \Omega : o$. Then using Ψ_1 and Π_Ω and an **app**-rule we can build a derivation Π_1 concluding with $y : [[o] \rightarrow o]_\omega \vdash t\Omega : o$, in which the subterm Ω is typed. We can also build a derivation Π_2 from Ψ_2 that also concludes with $y : [[o] \rightarrow o]_\omega \vdash t\Omega : o$. This times, the subter Ω is not typed.

It is not difficult to see that Π_2 is intuitively approximable, whereas Π_1 is 0-approximable but not fully approximable. Roughly speaking, the subderivation of Π_1 typing Ω is the only non-approximable part of Π_1 . Indeed:

- Π_2 is not approximable since it contains a subderivation typing the mute term Ω .
- Every (finite) approximation of Π_2 is an approximation of Π_1 . Thus, the join of the finite approximations of Π_1 is actually Π_2 .

This proves that depth 0 approximability is not equivalent to approximability.

A.3 Lattices of (finite or not) approximations

In this Appendix, we prove Theorem 10.3, stating that the set of derivations typing a given term t is a complete semi-lattice. We implicitly use some of the concepts of Part IV (*e.g.*, ascendance) but in a more basic way.

Let t be a term, $\mathbb{A} = \{a \in \mathbb{N}^* \mid \bar{a} \in \mathbf{supp}(t)\}$ and $\mathbb{B} = \mathbb{A} \times (\mathbb{N} \setminus \{0\})^* \cup \mathbb{A} \times \mathcal{V} \times ((\mathbb{N} \setminus \{0, 1\}) \cdot (\mathbb{N} \setminus \{0\})^*)$. Thus, \mathbb{A} (resp. \mathbb{B}) holds the potential positions (resp. right and left bipoitions) of a derivation P typing t *i.e.* if P types t , then $\mathbf{supp}(P) \subseteq \mathbb{A}$ and $\mathbf{bisupp}(P) \subseteq \mathbb{B}$.

Let $\mathbf{p} \in \mathbb{B}$. Whether \mathbf{p} is of the form (a, c) or $(a, x, k \cdot c)$, we set $\text{out}(\mathbf{p}) = a$.

In order to prove Theorem 10.3, we first need to characterize the set of function $P : \mathbb{B} \longrightarrow \mathcal{O} \cup \{\rightarrow\}$ which actually define a derivation typing t .

Thus, let $P : \mathbb{B} \longrightarrow \mathcal{O} \cup \{\rightarrow\}$ a function. We set $B = \text{dom}(P)$ and $A = \{\text{out}(a) \mid \mathbf{p} \in B\}$.

For all $a \in A$, we write $T(a) : (\mathbb{N} \setminus \{0\})^* \rightarrow \mathcal{O} \cup \{\rightarrow\}$ for the function defined by $T(a)(c) = P(a, c)$ and for all $x \in \mathcal{V}$, we write $C(a)(x) : ((\mathbb{N} \setminus \{0, 1\}) \cdot (\mathbb{N} \setminus \{0\})^*) \rightarrow \mathcal{O} \cup \{\rightarrow\}$ for the function defined by $C(a)(x)(c) = P(a, x, c)$. We write $P(a)$ for $P(a) = C(a) \vdash t|_a : T(a)$.

We also set $\text{Lves}(B) = \{(a, c) \in B \mid (a, c \cdot 1) \notin B\} \cup \{(a, x, c) \in B \mid (a, c \cdot 1) \notin B\}$.

Let us first observe that A and B must satisfy the following conditions, for P to be a derivation:

- *Non-Vacuity (c1)*: $(\varepsilon, \varepsilon) \in B$, for all $a \in A$, $(a, \varepsilon) \in B$.
- *Axiom Rule (c2)*: for all $a \in A$ s.t. $t(a) = x$:
 - There is a unique $k \in \mathbb{N}$ such that $(a, x, k) \in B$. This integer k may be then denoted $\text{tr}^P(a)$.
 - For all $y \in \mathcal{V}$ s.t. $y \neq x$, for all $k \geq 2$, $c \in \mathbb{N}^*$, $(a, y, c) \notin B$.
- *Abstraction (c3)*: For all $a \in A$ such that $t(a) = \lambda x$:
 - For all $k \geq 2$, $c \in \mathbb{N}^*$, $(a, x, k \cdot c) \notin B$.
 - $(a, 1) \in B$.
- *Application (c4)*: for all $a \in A$ such that $t(a) = @$, for all $x \in \mathcal{V}$, $k \geq 2$ such that $(a, x, k) \in B$, there is a unique $\ell \geq 1$ such that $(a \cdot \ell, x, k) \in B$. This integer ℓ may be then denoted $\text{uptr}^P(a, x, k)$ (uptr stands for “up” and “track”).
- *001-Restriction (c5)*: for all $(a, c) \in \mathbb{B}$ (resp. $(a, x, c) \in \mathbb{B}$), there is $n \geq 0$ such that $(a, c \cdot 1^n) \notin B$ (resp. $(a, x, c \cdot 1^n) \notin B$).

Let us briefly explain the meaning of those conditions. Condition (c1) states that a derivation is non-empty and requires that there is a type on the right-hand side of \vdash whenever there is a context on its left-hand side (without (c1), it is not difficult to have define a “derivation” typing f^ω without right-hand sides under the hypotheses of Proposition A.1 below). Condition (c2) states that, if $t(a) = x$ (*i.e.* if $a \in A$ necessarily corresponds to an **ax**-rule typing x), then x is typed with a singleton sequence type (first point) and that no other variable is typed (second point), modulo some conditions to be introduced later ((*tf1*) and (*tf2*)). Condition (c3) means that, if $t(a) = \lambda x$, then x is not typed in the context of $t|_a$ (first point) and that the type of $t|_a$ is an arrow type and not a type variable ($\text{support} = \{\varepsilon\}$): its support must contain 1 (second point). Condition (c4) means that, if $t(a) = @$ (*i.e.* $a \in A$ corresponds to an **app**-rule), then there are no track conflicts (see Sec. 10.2.3). Condition (c5) ensures that types are in Typ (and not in $\text{Typ}^{111} \setminus \text{Typ}$).

Conditions (c1), (c2), (c3), (c4), (c5) are necessary but not sufficient for P to be a derivation typing t . Assume that P satisfies (c1), (c2), (c3), (c4), (c5). This allows us to define some binary relations on \mathbb{B} , that will be denoted by an arrow. Notice that those relations are defined on \mathbb{B} (and not on $B \subseteq \mathbb{B}$), but that they depend on B e.g., condition (la@) below is well-founded and unambiguously defined because P satisfies (c4), condition (axf) depends on (c2).

- *Type Formation (tf1)*: For all $a \in \mathbb{A}$, $x \in \mathcal{V}$, $c \in \mathbb{N}^*$, $k \in \mathbb{N}$, $(a, c \cdot k) \rightarrow_{t1} (a, c)$
 $(a, x, c \cdot k) \rightarrow_{t1} (a, x, c)$
- *Type Formation (tf2)*: For all $a \in \mathbb{A}$, $x \in \mathcal{V}$, $c \in \mathbb{N}^*$, $k \geq 2$, $(a, c \cdot k) \rightarrow_{t1} (a, c \cdot 1)$
and $(a, c \cdot k) \rightarrow_{t1} (a, x, c \cdot 1)$.
- *Axiom Formation (axf)*: For all $a \in A$ such that $t(a) = x$, for all $c \in \mathbb{N}^*$,
 $(a, x, k \cdot c) \rightarrow_{ax} (a, c)$, where $k = \text{tr}^P(a)$ (see (c2)).
- *Right Ascendance λ (ra λ)*: For all $a \in \mathbb{A}$ s.t. $t(a) = \lambda x$, for all $c \in \mathbb{N}^*$,
- *Right Ascendance @ (ra@)*: For all $a \in \mathbb{A}$ s.t. $t(a) = @$, for all $c \in \mathbb{N}^*$, $(a, c) \rightarrow_{asc} (a \cdot 1, c)$
- *Left Ascendance λ (la λ)*: For all $a \in \mathbb{A}$ s.t. $t(a) = \lambda x$:
 - (la λ 1) For all $k \geq 2$, $c \in \mathbb{N}^*$, $(a, k \cdot c) \rightarrow_{asc} (a \cdot 0, x, k \cdot c)$
 - (la λ 2) For all $y \in \mathcal{V}$, $k \geq 2$, $c \in \mathbb{N}^*$ s.t. $y \neq x$, $(a, y, k \cdot c) \rightarrow_{asc} (a \cdot 0, y, k \cdot c)$
- *Left Ascendance @ (la@)*: For all $a \in A, x \in \mathcal{V}, k \geq 2$ s.t. $t(a) = @$ and
 $(a, x, k) \in B$, for all $c \in \mathbb{N}^*$, $(a, x, k \cdot c) \rightarrow_{asc} (a \cdot \ell, x, k \cdot c)$, where $\ell = \text{uptr}^P(a, x, k)$
(see (c4)).
- *Consumption (con)*: For all $a \in \mathbb{A}$ s.t. $t(a) = @$, for all $k \geq 2$, $c \in \mathbb{N}^*$, $(a \cdot 1, k \cdot c) \rightarrow (a \cdot k, c)$

Notice that if $p_1 \rightarrow_{asc} p_2$, then $p_2 = \text{asc}p_1$, with the notation of Sec. A.2.1. But if $t(a) = \lambda x$, we have $(a \cdot 0, \varepsilon) = \text{asca}, \varepsilon$, but not $(a, \varepsilon) \rightarrow_{asc} (a \cdot 0, \varepsilon)$.

We will consider $\equiv_{@}$, the reflexive transitive symmetric closure of $\rightarrow_{ax} \cup \rightarrow_{asc} \cup \rightarrow_{pi} \cup \rightarrow$: relation $\equiv_{@}$ helps us to track a type symbol through a derivation. For instance, in P_{ex} , let us describe the equivalence class of $(03, \varepsilon)$ (a biposition pointing to o') by $\equiv_{@}$. We have $(\varepsilon, 1) \rightarrow_{asc} (0, \varepsilon) \rightarrow_{asc} (03, \varepsilon)_{ax} \leftarrow (03, x, 2)_{asc} \leftarrow (0, x, 2)_{asc} \leftarrow (0, 2)$ but also $(03, \varepsilon) \leftarrow (01, 3)_{ax} \leftarrow (01, x, 43)_{asc} \leftarrow (0, x, 43)_{asc} \leftarrow (\varepsilon, 43)$. All those bipositions also point to type variable o' .

An attentive observation of the relations above shows that, for P to be a derivation typing t , then B must be closed under \rightarrow_{t1} , \rightarrow_{t2} , $\equiv_{@}$. Type formation ensures that the supports of the types (on the right-hand sides of \vdash) are well formed e.g., closure under \rightarrow_{t1} ensures that the supports of types are trees and closure under \rightarrow_{t2} means that any non terminal node in a type has a son on track 1 (because a non terminal node of a type is an arrow and has its right hand-side placed on track 1, see Sec. 10.2.2). Left ascendance is mostly about how the context of a rule may be computed from the contexts of its premises. Right ascendance explains how the type given in a rule is partially computed from the type given in one of its premises. Relation \rightarrow_{ax} ensures that axiom are of the form $x : (k \cdot T) \vdash x : T$ where k is the axiom track given by (c2)

at position a . Consumption is related to the **app**-rule: the left-hand side of the arrow type typing the left-hand side of the application (bipositions of the form $(a \cdot 1, k \cdot c)$ must be equal to the sequence of types given to the argument (bipositions of the form $(a \cdot k, c)$).

Conditions (c1), (c2), (c3), (c4), (c5) and closure under $\rightarrow_{\mathbf{t}_1}, \rightarrow_{\mathbf{t}_2}, \equiv_{\textcircled{a}}$ are not enough to guarantee that P is a derivation. There are also labelling conditions (for (lab3), see the example above with $(03, \varepsilon)$).

- *Non terminal nodes are arrow (lab1)*: For all $\mathbf{p} \in B \setminus \text{Lves}(B)$, $P(\mathbf{p}) = \rightarrow$.
- *Leaves are type variables (lab2)*: For all $\mathbf{p} \in \text{Lves}(B)$, $P(\mathbf{p}) \in \mathcal{O}$.
- *Matching leaves labels (lab3)*: for all $\mathbf{p}_1, \mathbf{p}_2 \in \text{Lves}(B)$ such that $\mathbf{p}_1 \equiv_{\textcircled{a}} \mathbf{p}_2$, $P(\mathbf{p}_1) = P(\mathbf{p}_2)$.

We claim and prove:

Proposition A.1. The function P is a **S**-derivation typing t iff it satisfies (c1), (c2), (c3), (c4), (c5), (lab1), (lab2), (lab3) and is closed under $\rightarrow_{\mathbf{t}_1}, \rightarrow_{\mathbf{t}_2}$ and $\equiv_{\textcircled{a}}$.

Proof. The necessity of those conditions has been discussed. Let us prove that they are sufficient and consider a function P satisfying the hypotheses of the proposition.

First, thanks to (tf1), (tf2), (lab1), (lab2), it is easy to check that, for all $a \in A$ and $x \in \mathcal{V}$, $T(a)$ is a type and $C(a)(x)$ is sequence type of Typ^{111} . Using (c5), we check that they are in **Typ**. Thus, the $P(a) = C(a) \vdash t|_a : T(a)$ are well-formed **S**-judgments.

- *Correctness of ax-rules*: let $a \in A$ such that $t(a) = x$. By (r1), let $k \geq 2$ be the unique integer such that $(a, x, k) \in B$. By (axf), closure by $\equiv_{\textcircled{a}}$ and (lab3) (w.r.t. $\rightarrow_{\mathbf{ax}}$), the function $(c \mapsto C(a)(x)(c))$ and $(c \mapsto T(a)(c))$ are equal. Moreover, by (r1) again, if $y \neq x$, then $C(a)(y)$ is empty. Thus, $P(a) = x : k \cdot T(a) \vdash x : T(a)$ is a correct **ax**-rule.

- *Correctness of abs-rules*: let $a \in A$ such that $t(a) = \lambda x$. By (r2), $(\varepsilon, 1) \in B$, so $(a, \varepsilon) \notin \text{Lves}(B)$ and $1 \in \text{supp}(T(a))$ so $T(a)$ is an arrow type (if it were a type variable, we would have $\text{supp}(T(a)) = \{\varepsilon\}$).

The fact that $C(a \cdot 0) = C(a); x : \text{Sc}(T(a))$ comes from the definition of left ascendance (see (la λ 1) and (la λ 2)) and (lab3) (w.r.t. $\rightarrow_{\mathbf{asc}}$). By (ra λ) and (lab3) (w.r.t. $\rightarrow_{\mathbf{asc}}$), the functions $c \mapsto T(a)(1 \cdot c)$ and $c \mapsto T(a \cdot 0)(x)$ are equal *i.e.* $T(a \cdot 0) = \text{Tg}(T(a))$.

Thus, we have a correct **abs**-rule.

- *Correctness of app-rules*: let $a \in A$ such that $t(a) = x$. The fact that $C(a) = \bigcup_{\ell \in \{1\} \cup K} C(a \cdot \ell)$ comes from the definition of left ascendance (see (laapp)) and (lab3) (w.r.t. $\rightarrow_{\mathbf{asc}}$), but also from (c4) and the fact that every $C(a \cdot \ell)(x)$ is a correct sequence judgment.

By (c1), $(a, \varepsilon) \in B$, so, by (raapp), $(a \cdot 1, 1) \in B$, so $(a, \varepsilon) \notin \text{Lves}(B)$ and $P(a, \varepsilon) = \rightarrow$.

By (ra \textcircled{a}) and (lab3) (w.r.t. $\rightarrow_{\mathbf{asc}}$), the functions $c \mapsto T(a)(c)$ and $c \mapsto T(a \cdot 1)(1 \cdot c)$

are equal *i.e.* $T(a) = \mathbf{Tg}(T(a \cdot 1))$. By (*con*) and (*lab3*) (w.r.t. \rightarrow), the functions $(k, c) \mapsto T(a \cdot 1)(k \cdot c)$ and $(k, c) \mapsto T(a \cdot k)(c)$ (from $(\mathbb{N} \setminus \{0, 1\}) \times \mathbb{N}^*$) are equal *i.e.* $\mathbf{Sc}(T(a \cdot 1)) = \uplus_{k \geq 2, a \cdot k \in A} (k \cdot T(a \cdot k))$.

Thus, we have a correct **app**-rule.

□

A.3.1 Meets and Joins of Derivations Families

Lemma A.1. Let $(P_i)_{i \in I}$ be a non-empty family of derivations typing the same term t , such that $\forall i, j \in I, \exists P \in \mathbf{Deriv}, P_i, P_j \leq_\infty P$.

We define P by $\mathbf{bisupp}(P) = \cap_{i \in I} \mathbf{bisupp}(P_i)$ and $P(\mathbf{p}) = P_i(\mathbf{p})$ for any $i \in I$.

Then, P defines a correct derivation (that is finite if one of the P_i is finite). We write $P = \wedge_{i \in I} P_i$.

Proof sketch.

- We first check that if one of the P_i satisfies (*c1*), (*c2*), (*c3*), (*c4*), (*c5*), then P also does.
- The stability of P under $\rightarrow_{\mathbf{t1}}$, $\rightarrow_{\mathbf{t2}}$ and $\equiv_{\textcircled{a}}$ comes from the stability of under those for the P_i . However, $\equiv_{\textcircled{a}}$ depends on $B := \mathbf{bisupp}(P)$, because of (*axf*) and (*la* \textcircled{a}). This works because:
 - For all $a \in A := \mathbf{out}(B)$ such that $t(a) = x$, for all $i \in I$, $\mathbf{tr}^P(a) = \mathbf{tr}^{P_i}(a)$.
 - For all $a \in A$, $x \in \mathcal{V}$ and $k \geq 2$ such that $t(a) = \textcircled{a}$ and $(a, x, k) \in B$, for all $i \in I$, $\mathbf{uptr}^P(a, x, k) = \mathbf{uptr}^{P_i}(a, x, k)$.

Thus, the relation $\equiv_{\textcircled{a}}$ of P is the intersection of those of the P_i (i ranging over I).

- Condition $\forall i, j \in I, \exists P \in \mathbf{Deriv}, P_i, P_j \leq_\infty P$ implies that $\mathbf{p} \in \mathbf{Lves}(P)$ iff $\mathbf{p} \in \mathbf{Lves}(P_i)$ for all i . Thus, P also satisfies (*lab1*) and (*lab2*). Finally, P satisfies (*lab3*) because of the remark about $\equiv_{\textcircled{a}}$ in the last point.

□

Lemma A.2. Let $(P_i)_{i \in I}$ be a non-empty family of derivations typing the same term, such that $\forall i, j \in I, \exists P \in \mathbf{Deriv}, P_i, P_j \leq_\infty P$ (*e.g.*, $(P_i)_{i \in I}$ is directed).

We define the labelled tree P by $\mathbf{bisupp}(P) = \cup_{i \in I} \mathbf{bisupp}(P_i)$ and $P(\mathbf{p}) = P_i(\mathbf{p})$ for any i such that $\mathbf{p} \in \mathbf{bisupp}(P_i)$.

Then P defines a correct derivation (that is finite if I is finite and all the P_i are). We write $P = \vee_{i \in I} P_i$.

Proof sketch.

- We first check that, since P_i satisfies (*c1*), (*c2*), (*c3*), (*c4*), (*c5*) for all i , then P also does. For instance, for (*c5*), we need to also use (*lab2*) and condition $\forall i, j \in I, \exists P \in \mathbf{Deriv}, P_i, P_j \leq_\infty P$, which entail that if $\mathbf{p} \in \mathbf{Lves}(P_i)$ for some i , then $P_i(\mathbf{p}) = o$ (for some $o \in \mathcal{O}$), so, for all $j \in I$ such that $\mathbf{p} \in \mathbf{bisupp}(P_j)$, $P_j(\mathbf{p}) = o$ *i.e.* $\mathbf{p} \in \mathbf{Lves}(P_j)$. Thus, $\mathbf{p} \in \mathbf{Lves}(B)$. In particulier, this easily implies that P also satisfies (*lab1*) and (*lab2*).

- The stability of P under \rightarrow_{t_1} , \rightarrow_{t_2} and $\equiv_{@}$ comes from the stability of under those for the P_i . However, $\equiv_{@}$ depends on $B = \mathbf{bisupp}(P)$, because of (axf) and $(la@)$. This works because:

- For all $a \in A := \mathbf{out}(B)$ such that $t(a) = x$, then for all $i \in I$ such that $a \in A_i := \mathbf{supp}(P_i)$, we have $\mathbf{tr}^P(a) = \mathbf{tr}^{P_i}(a)$.
- For all $a \in A$, $x \in \mathcal{V}$ and $k \geq 2$ such that $t(a) = @$ and $(a, x, k) \in B$, then for all $i \in I$ such that $(a, k, x) \in B_i := \mathbf{bisupp}(P_i)$, we have $\mathbf{uptr}^P(a, x, k) = \mathbf{uptr}^{P_i}(a, x, k)$.

Thus, the relation $\equiv_{@}$ of P is the union of those of the P_i (i ranging over I).

- Finally, P satisfies $(lab3)$ because of the remark about $\equiv_{@}$ in the last point. □

The previous lemmas define the join and the meet of derivations (under the same derivation) as their set-theoretic union and intersection. More precisely, they entail Theorem 10.3:

Theorem. The set of derivations typing a same term t endowed with \leq_{∞} is a directed complete semi-lattice.

- If D is a directed set of derivations typing t :
 - The **join** $\vee D$ of D is the function P defined by $\mathbf{dom}(P) = \cup_{P_* \in D} \mathbf{bisupp}(P_*)$ and $P(\mathbf{p}) = P_*(\mathbf{p})$ (for any $P_* \in D$ s.t. $\mathbf{p} \in \mathbf{bisupp}(P_*)$), which also is a derivation.
 - The **meet** $\wedge D$ of D is the function P defined by $\mathbf{dom}(P) = \cap_{P_* \in D} \mathbf{bisupp}(P_*)$ and $P(\mathbf{p}) = P_*(\mathbf{p})$ (for all $P_* \in D$), which also is a derivation.
- If P is a derivation typing t , $\mathbf{Approx}_{\infty}(P)$ is a complete lattice and $\mathbf{Approx}(P)$ is a lattice.

A.3.2 Reach of a derivation

Definition A.2. • For any derivation P , we set $\mathbf{Reach}(P) = \{\mathbf{p} \in \mathbf{bisupp}(P) \mid \exists {}^fP \leq P, \mathbf{p} \in {}^fP\}$.

- If $\mathbf{p} \in \mathbf{Reach}(P)$, we say \mathbf{p} is **reachable**.
- If $B \subset \mathbf{bisupp}(P)$, we say B is **reachable** if there is ${}^fP \leq P$ s.t. $B \subset \mathbf{bisupp}({}^fP)$.

Since $\mathbf{Approx}(P)$ is a complete lattice and the bisupports of its elements are finite, we can write $P\langle \mathbf{p} \rangle$ (resp. $P\langle B \rangle$) for the smallest fP containing \mathbf{p} (resp. containing B), for any $\mathbf{p} \in \mathbf{Reach}(P)$ (resp. for any reachable $B \subset \mathbf{bisupp}(P)$).

Proposition A.2. Let $B \subset \mathbf{bisupp}(P)$. Then B is reachable iff B is finite and $B \subset \mathbf{Reach}(P)$.

In that case, $P\langle B \rangle = \vee_{\mathbf{p} \in B} P\langle \mathbf{p} \rangle$.

Definition A.3. If $\mathbf{Reach}(P)$ is non-empty, we define $P\langle \mathbf{Reach} \rangle$ as the induced labelled tree by P on $\mathbf{Reach}(P)$.

We have actually $P\langle\text{Reach}\rangle = \bigvee_{p \in \text{Reach}(P)} P\langle p \rangle$, so P is a derivation. By construction, P is approximable.

A.3.3 Proof of the subject expansion property

We reuse the notations and assumptions of Sec. 10.5.4 and set $A' = \text{supp}(P')$. expansion of a term inside demands to choose new axiom tracks. We will do this *uniformly* i.e. we fix an injection $\lfloor \cdot \rfloor$ from \mathbb{N}^* to $\mathbb{N} - \{0, 1\}$ and any axiom rule created at position a will use the axiom track value $\lfloor a \rfloor$.

Assume ${}^f P' \leq P'$. Let $N \in \mathbb{N}$ s.t., for all $n \geq N$, $b_n \notin \overline{{}^f A'}$ with ${}^f A' = \text{supp}({}^f P')$. For $n \geq N$, we write ${}^f P'(n)$ for the derivation replacing t' by t_n in ${}^f P'$. This derivation is correct according to the subject substitution lemma (section 10.4.4), since $t_n(\bar{a}) = t'(\bar{a})$ for all $a \in {}^f A'$.

We then write ${}^f P'(n, k)$ (with $0 \leq k \leq n$) the derivation obtained by performing k expansions (w.r.t. our reduction sequence and $\lfloor \cdot \rfloor$). Since b_n is not in A , we observe that ${}^f P'(n+1, 1) = {}^f P'(t_n)$. Therefore, for all $n \geq N$, ${}^f P'(n, n) = {}^f P'(N, N)$. Since we could replace N by any $n \geq N$, ${}^f P$ is morally ${}^f P'(\infty, \infty)$. We write $P = P'(\text{init})$ to refer to this deterministic construction.

We set $\mathcal{D} = \{{}^f P'(\text{init}) \mid {}^f P' \leq P'\}$. Let us show that \mathcal{D} is a directed set.

Let ${}^f P'_1, {}^f P'_2 \leq P'$. We set ${}^f P' = {}^f P'_1 \vee {}^f P'_2$. Let N be great enough so that $\forall n \geq N$, $b_n \notin \overline{{}^f A'}$ with ${}^f A' = \text{supp}({}^f P')$.

We have ${}^f P'_i \leq {}^f P'$, so ${}^f P'_i(N) \leq {}^f P(N)$, so, the by monotonicity of *uniform* expansion, ${}^f P'_i(N, N) \leq {}^f P(N, N)$ i.e. ${}^f P'_i(\text{init}) \leq {}^f P(\text{init})$.

Since \mathcal{D} is directed, we can set $P = \bigvee_{{}^f P' \leq P'} {}^f P'(\text{init})$. Since for any ${}^f P' \leq P$ and the associated usual notation, ${}^f C(\varepsilon) = {}^f C'(\varepsilon)$, ${}^f T(\varepsilon) = {}^f T'(\varepsilon)$ and $C(\varepsilon)$, $C'(\varepsilon)$, $T(\varepsilon)$, $T'(\varepsilon)$ are the respective infinite join of ${}^f C(\varepsilon)$, ${}^f C'(\varepsilon)$, ${}^f T(\varepsilon)$, ${}^f T'(\varepsilon)$ when ${}^f P'$ ranges over $\text{Approx}(P')$, we conclude that $C(\varepsilon) = C'(\varepsilon)$ and $T(\varepsilon) = T'(\varepsilon)$.

A.4 Approximability of the quantitative NF-derivations

We show in this Appendix that every quantitative derivation typing a normal form t is approximable. We use the same notations as in Section 10.5: we consider a derivation P built as in Subsection 10.5.2, from a normal form t , a support candidate A of t and a type $\hat{T}(\hat{a})$ given for each unconstrained position $\hat{a} \in \hat{A}$. It yields a family of contexts $(\mathcal{C}(a))_{a \in A}$ and of types $(\mathbb{T}(a))_{a \in A}$ such that $P(a)$ is $\mathcal{C}(a) \vdash t|_a : \mathbb{T}(a)$ for all $a \in A$.

A.4.1 Degree of a position inside a type in a derivation

For $a \in \mathbb{N}^*$, we define the rank $\text{rk}(a)$ of x by $\text{rk}(a) = \max(\text{ad}(a), \max(a))$. Thus, $\text{rk}(a)$ bounds the “width” and applicative depth of a .

- For each a in A and each position c in $\text{Cal}(a)$ such that $\text{Cal}(a)(c) \neq X_i$, we define the number $\text{rk}_{\text{in}}(a, c)$ by:

- When a is a unconstrained node, $\text{d}_{\text{out}}(c)$ is $\text{rk}(a)$.

- When a is a non zero position: the value of $\mathbf{rk}_{\text{in}}(a, c)$ for the positions colored in red is the applicative depth is $\mathbf{rk}(a, c)$ (and of \hat{a}).

$$\mathbf{E}(\hat{a})(x_1) \rightarrow \mathbf{E}(\hat{a})(x_2) \rightarrow \dots \rightarrow \mathbf{E}(\hat{a})(x_n) \rightarrow \mathbf{T}(\hat{a})$$

- When a is partial: the value of $\mathbf{rk}_{\text{out}}, c$ for the positions colored in red is the applicative depth of a .

$$\mathbf{R}_1(a) \rightarrow \dots \rightarrow \mathbf{R}_n(a) \rightarrow \mathbf{T}(\hat{a})$$

From Sec. 10.5.1, we recall that n is also denoted $\mathbf{rdeg}(a)$.

For each $a \in A$ and each position c in $\mathbf{T}(a)$, we define the number $\mathbf{deg}_{\text{out}}(a, c)$ (that is the applicative depth of the position a' on which c depends) by extending $\mathbf{rank}_{\text{out}}$ via substitution (this is formally done in the next section).

Again, for each $a \in A$, each variable x and each position c in $\mathbf{C}(a)(x)(c)$ we define $\mathbf{deg}_{\text{out}}(a, x, c)$ by extending $\mathbf{rank}_{\text{out}}$ via substitution.

The definition of $\mathbf{deg}_{\text{out}}(a, x, c)$ in $\mathbf{T}(a)$ and $\mathbf{C}(a)(x)$ is sound, because in $E(a)(x)$ and $F_j(x)$, there a no symbol other than the X_i . But the X_i disappear thanks to the coinductive definition of $\mathbf{T}(a)$: every biposition (a, c) will receive a value for $\mathbf{deg}_{\text{out}}(a, c)$.

- For each a in A and each position c in $S(a)$ such that $S(a)(c) \neq X_i$, we define the number $\mathbf{rk}_{\text{in}}(a, c)$ by:

- When a is a unconstrained node: $\mathbf{rk}_{\text{in}}(a, c) = \mathbf{rk}(c)$ (c is a position of $\mathbf{T}(a)$).
- When a is an abstraction node: if c is a position colored in red, $\mathbf{rk}_{\text{in}}(a, c) = \mathbf{rk}(c)$:

$$\mathbf{E}(\hat{a})(x_1) \rightarrow \mathbf{E}(\hat{a})(x_2) \rightarrow \dots \rightarrow \mathbf{E}(\hat{a})(x_n) \rightarrow \mathbf{T}(\hat{a})$$

- When a is partial: if c is a position colored in red, $\mathbf{rk}_{\text{in}}(c) = \mathbf{rk}(c)$

$$\mathbf{R}_1(a) \rightarrow \dots \rightarrow \mathbf{R}_n(a) \rightarrow \mathbf{T}(\hat{a})$$

As for \mathbf{rk}_{out} , we extend $\mathbf{rk}_{\text{in}}(a, c)$ into $\mathbf{deg}_{\text{in}}(a, c)$ for inner positions of $\mathbf{T}(a)$ or in $\mathbf{C}(a)(x)$ via substitution.

Definition A.4. If $\mathbf{p} \in \mathbf{bisupp}(P)$, the **degree** of \mathbf{p} is defined by $\mathbf{deg}(\mathbf{p}) = \max(\mathbf{deg}_{\text{out}}(\mathbf{p}), \mathbf{deg}_{\text{in}}(\mathbf{p}))$.

A.4.2 Truncation of degree n

We present more formally the definitions of the last section and we recall that $\mathbf{rk}(a) = \max(\mathbf{ad}(a), \max(a))$ for $a \in \mathbb{N}^*$.

For all $a \in A$ and $k \in \mathbb{N}$, we set $\mathbf{T}^0(a) = X_a$ and $\mathbf{T}^{k+1}(a) = \mathbf{T}^k(a)[\mathbf{Cal}(a')/X_{a'}]_{a' \in \mathbb{N}^*}$.

For all $k \in \mathbb{N}$, we set $\mathbf{supp}_*(\mathbf{T}^k(a)) = \{c \in \mathbf{supp}(\mathbf{T}^k(a)) \mid \mathbf{T}^k(a)(c) \neq X_{a'}\}$.

If $c \in \mathbf{supp}(\mathbf{T}(a))$, there is a minimal $k \in \mathbb{N}$ s.t. $c \in \mathbf{supp}_*(\mathbf{T}^k(a))$. We denote it $\mathbf{cd}(a)(c)$ (call-depth of c at pos. a).

In that case, there are unique $c' \in \mathbf{supp}(\mathbf{T}(a))$, $c' \in \mathbb{N}^*$ and $a' \in A$ s.t. $c = c' \cdot c''$, $\mathbf{T}^{k-1}(a)(c') = X_{a'}$ (we have necessarily $a \leq a'$) and $c'' \in \mathbf{supp}(\mathbf{Cal}(a'))$.

We write $a' = \text{cop}(a, c)$ (calling outer position of c at position a), $c' = \text{cip}_n(a, c)$ (calling inner position of c at position a) and $c'' = \text{pf}(a, c)$ (postfix of c at position a). Then, we set $\text{deg}_{\text{out}}(a, c) := \text{rk}(a')$ and $\text{deg}_{\text{in}}(a, c) = \text{rk}(c'')$. Finally, we may define $\text{deg}(a, c)$ by $\text{deg}(a, c) = \max(\text{deg}_{\text{out}}(a, c), \text{deg}_{\text{in}}(a, c))$.

- We set $A_n = \{a \in A \mid \text{rk}(a) \leq n\}$.
- For all $\hat{a} \in \hat{A}_n$, we define $\hat{T}_n(\hat{a})$ by removing all the positions c such that $\text{rk}(c) > n$ i.e. $\hat{T}_n(\hat{a})$ is (as a function) the restriction of $\hat{T}(\hat{a})$ on $\{c \in \text{supp}(\hat{T}(\hat{a})) \mid \text{rk}(c) \leq 0\}$.

Since t is in Λ^{001} (and not in $\Lambda^{111} \setminus \Lambda^{001}$), A_n is finite. Since, for all $a \in A$, $\mathbb{T}(a)$ is in Typ (and not in $\text{Typ}^{111} \setminus \text{Typ}$), the type $\hat{T}_n(\hat{a})$ is finite for all $\hat{a} \in \hat{A}_n$.

We define P_n as the natural extension of (A_n, \hat{T}_n) . We retrieve contexts C_n and types \mathbb{T}_n such that, for all $a \in A_n$, $P_n = C_n(a) \vdash t|_a : \mathbb{T}_n(a)$.

Lemma A.3. For all $k \in \mathbb{N}$, $a \in A$, $c \in \mathbb{N}^*$, we have $\text{rk}(a) \leq n$ and $c \in \text{supp}(\mathbb{T}_n^k(a))$ iff $c \in \text{supp}(\mathbb{T}^k(a))$ and $\text{deg}(a, c) \leq n$.

In that case, $\mathbb{T}^k(a)(c) = \mathbb{T}_n^k(a)(c)$.

Proof. By a simple but tedious induction on k . We write cop_n , cip_n , pf_n and \mathbb{T}_n^k w.r.t. P_n .

- Case $k = 0$:
If $\text{rk}(a) \leq n$ and $c \in \text{supp}(\mathbb{T}_n^0(a))$, then $\text{rk}(a) \leq n$ and $c = \varepsilon$. By definition, $\text{deg}_{\text{out}}(a, \varepsilon) = \text{rk}(a) \leq n$ and $\text{deg}_{\text{in}}(a)(\varepsilon) = \text{rk}(\varepsilon) = 0$. Thus, $\text{deg}(a, c) \leq n$.
Conversely, if $c \in \text{supp}(\mathbb{T}^0(a))$ and $\text{deg}(a, c) \leq n$, we have likewise $c = \varepsilon$ and $\text{cop}(a, \varepsilon) = a$, so $\text{rk}(a) = \text{deg}_{\text{out}}(a, c) \leq n$. So $c = \varepsilon \in \mathbb{T}_n^0(a)$.
- Case $k + 1$:
If $\text{rk}(a) \leq n$ and $c \in \text{supp}(\mathbb{T}_n^{k+1}(a))$, we assume that $a \notin \text{supp}_*(\mathbb{T}_n^k(a))$ (case already handled by induction hypothesis).

Assume first that $c \in \text{supp}(\mathbb{T}_n^k(a))$ and $\mathbb{T}_n^k(a) = X_{a'}$. Then $\mathbb{T}_n^{k+1}(a) = \text{Cal}_n(a')(\varepsilon)$. By IH, we have $\text{deg}(a, c) \leq n$ and $\mathbb{T}^k(a)(c) = X_{a'}$, so $\mathbb{T}^{k+1}(a)(c) = \text{Cal}(a')(\varepsilon)$ and $\text{rk}(a') = \text{deg}_{\text{out}}(a, c) \leq n$, so that $\text{rk}(a) \leq n$ since $a \leq a'$. Since $\text{rk}(\varepsilon) = 0$, we have $\text{Cal}_n(a')(\varepsilon) = \text{Cal}(a')(\varepsilon)$. So $\mathbb{T}^{k+1}(a) = \mathbb{T}_n^{k+1}(a, \varepsilon)$.

We assume now that $c \notin \text{supp}(\mathbb{T}_n^k(a))$. We set $a' = \text{cop}_n(a, c)$ and $c' = \text{cip}_n(a, c)$ (thus, $\mathbb{T}_n^k(a)(c') = X_{a'}$) and $c'' = \text{pf}_n(a, c)$. By IH, we also have $a' = \text{cop}(a, c)$ and $c' = \text{cip}(a, c)$. We have two subcases, depending if $\mathbb{T}_n^{k+1}(a)(c) = X_{a''}$ holds or not.

- If $S_n^{k+1}(a)(c) = X_{a''}$ (with necessarily $\text{rk}(a'') = \text{deg}_{\text{out}}(a, c) \leq n$), then $c'' = 1^j \cdot \ell$ with $j < \text{rdeg}(a')$ and $\ell = [a''] \geq 2$ and, by IH, $c' \in \text{supp}(\mathbb{T}^k(a))$ and $\mathbb{T}^k(a)(c') = X_{a'}$.
Then $c = c' \cdot c'' = c' \cdot 1^j \cdot \ell \in \text{supp}(\mathbb{T}^{k+1}(a))$, $\text{deg}_{\text{out}}(a, c) = \text{ad}(a'') \leq n$ (since $\text{rk}(a'') \leq n$) and $\text{deg}_{\text{in}}(a, c) = \text{rk}(\varepsilon) = 0$. So we have $\text{deg}(a, c) \leq n$.

- If $\mathbb{T}_n^{k+1}(a)(c) \neq X_{a''}$ for all a'' , then $c'' \in \text{supp}_*(\text{Cal}_n(a')) \subseteq \text{Cal}(a')$. Thus, we also have $\mathbb{T}^{k+1}(a)(c) = \text{Cal}_n(a')(c) = \mathbb{T}_n^{k+1}(a, c)$. Moreover, we have $\text{deg}_{\text{out}}(a, c) = \text{rk}(a') \leq n$ and $\text{deg}_{\text{in}}(a, c) = \text{rk}(c'') \leq n$, so that $\text{deg}(a, c) \leq n$.

Conversely, if $a \in A$, $c \in \text{supp}(\mathbb{T}^{k+1}(a))$ and $\text{deg}(a, c) \leq n$, we assume that $a \notin \text{supp}_*(S^k(a))$ (case already handled by IH).

Assume first that $c \in \text{supp}(\mathbb{T}^k(a))$ and $\mathbb{T}^k(a) = X_{a'}$. Then $\text{rk}(a') = \text{deg}_{\text{out}}(a, c) \leq n$, so that $a' \in A_n$, and $\mathbb{T}^{k+1}(a) = \text{Cal}_n(a')(\varepsilon)$. By IH, we have $c \in \text{supp}(\mathbb{T}_n^k(a))$ and $\mathbb{T}_n^k(a)(c) = X_{a'}$, so $\mathbb{T}_n^{k+1}(\varepsilon) = \text{Cal}_n(a')(\varepsilon) = \text{Cal}(a')(\varepsilon) = \mathbb{T}^{k+1}(a)(c)$.

We assume now that $c \notin \text{supp}(\mathbb{T}^k(a))$. We set $a' = \text{cop}(a, c)$ and $c' = \text{cip}(a, c)$ (thus, $\mathbb{T}^k(a)(c') = X_{a'}$) and $c'' = \text{pf}(a, c)$. By IH, we also have $a' = \text{cop}_n(a, c)$ and $c' = \text{cip}_n(a, c)$. We have two subcases, depending if $\mathbb{T}^{k+1}(a)(c) = X_{a''}$ holds or not.

- If $\mathbb{T}^{k+1}(a)(c) = X_{a''}$, then, by definition of $\text{deg}_{\text{out}}(a, c)$, we have $\text{rk}_{\text{in}}(a, c) = \text{rk}(a'')$, so $\text{rka}'' = \text{deg}_{\text{out}}(a, c) \leq \text{deg}(a, c) \leq n$, so $a'' \in A_n$. Since $a \leq a'$, $\text{rk}(a) \leq n$.
Moreover, $c'' = 1^j \cdot \ell$ with $j < \text{rdeg}(a')$ and $\ell = \lfloor a'' \rfloor \geq 2$. Since $\text{rk}(a'') \leq n$, we have $c'' \in \text{supp}(\text{Cal}_n(a'))$, so that $c \in \text{supp}(\mathbb{T}_n^{k+1}(a))$ and $\mathbb{T}_n^{k+1}(a)(c) = X_{a''}$.
- If $\mathbb{T}^{k+1}(a)(c) \neq X_{a''}$ for all a'' , then $c'' \in \text{supp}_*(\text{Cal}(a'))$, $\text{deg}_{\text{out}}(a, c) = \text{rk}(a')$ and $\text{deg}_{\text{in}}(a, c) = \text{rk}(c'')$. Thus, $\text{rk}(a'), \text{rk}(c'') \leq n$ by $\text{deg}(a, c) \leq n$ (in particular, $a' \in A_n$). So $a'' \in A_n$ and $c'' \in \text{supp}_*(\text{Cal}_n(a'))$, so that $c \in \text{supp}_*(\mathbb{T}_n^{k+1}(a))$ and $\mathbb{T}_n^{k+1}(a)(c) = \mathbb{T}^{k+1}(a)(c)$.

□

A.4.3 A Complete Sequence of Derivation Approximations

Proposition A.3. If P is a quantitative derivation typing a Normal Form t , then P is approximable.

Proof. Let ${}^0B \subseteq \text{bisupp}(P)$ a finite subset. We set $n = \max\{\text{deg}(\mathfrak{p}) \mid \mathfrak{p} \in B\}$. Then ${}^0B \subseteq \text{bisupp}(P_n)$.

In order to conclude, it is enough to prove that P_n is a *finite* derivation. For that, we notice that $\mathbb{T}_n^{k+1}(a) = \text{tt}T_n^{n+1}(a)$ for all $k \geq n + 1$, since $\mathbb{T}_n^{n+1}(a)$ does not hold any $X_{a'}$. □

This proves the claims of Sec. 10.5.3.

A.5 Isomorphisms between S-Derivations

Let P_1 and P_2 be two S-derivations typing the same term t . We write A_i, C_i, T_i for their respective supports, contexts and types.

A **derivation isomorphism** ϕ from P_1 to P_2 is given by:

- ϕ_{supp} , a 01-isomorphism (of unlabelled tree) from A_1 to A_2 (Definition 13.1).

- For each $a_1 \in A_1$:
 - A type isomorphism $\phi_{a_1} : T_1(a_1) \rightarrow T_2(\phi_{\text{supp}}(a_1))$
 - For each $x \in \mathcal{V}$, a sequence type isomorphism $\phi_{a_1|x} : C_1(a_1)(x) \rightarrow C_2(\phi_{\text{supp}}(a_1))(x)$

such that the following "rules compatibility" conditions hold:

- If $t(\overline{a_1}) = \lambda x$, then:
 - $\phi_{a_1}(1 \cdot c) = 1 \cdot \phi_{a_1 \cdot 0}(c)$ and $\phi_{a_1}(k \cdot c) = \phi_{a_1 \cdot 1|x}(k \cdot c)$ for any $k \geq 2$ and $c \in \mathbb{N}^*$
 - $\phi_{a_1|y} = \phi_{a_1 \cdot 0|y}$ for any $y \in \mathcal{V}$, $y \neq x$.
- If $t(\overline{a_1}) = @$:
 - $\phi_{a_1}(c) = \text{Sc}(\phi_{a_1 \cdot 1}(1 \cdot c))$, for any $c \in \mathbb{N}^*$, where $\text{Sc}(k \cdot c) = c$ (removal of the first integer in a finite sequence).
 - $\phi_{a_1|x} = \bigcup_{\ell \geq 1} \phi_{a_1 \cdot \ell}$ (the functional join must be defined because of the app-rule).

The above rules means that ϕ must respect different occurrences of the "same" (from a moral point of view) biposition. For instance:

- Assume $t(\overline{a_1}) = \lambda x$, then $T_1(a_1) = C_1(a_1 \cdot 0)(x) \rightarrow T_1(a_1 \cdot 0)$. So, any inner position c_1 inside $T(a_1 \cdot 0)$ can be "identified" to the inner position $1 \cdot c_1$ inside $T_1(a_1)$. Thus (forgetting about the indexes), if ϕ maps c_1 on c_2 (inside $T_2(a_2)$), then ϕ should map $1 \cdot c_1$ on $1 \cdot c_2$.
- Assume $t(\overline{a_1}) = @$. Then, the sequence type $C(a_1)(x)$ is the union of the $C(a_1 \cdot \ell)(x)$ (for ℓ spanning over $\mathbb{N} - \{0\}$). Then ϕ should map every inner position $k \cdot c$ inside $C(a_1)(x)$ according to the unique $C(a_1 \cdot \ell)$ to which it belong.

Lemma A.4. If $P_1 \xrightarrow{b} P'_1$, $P_2 \xrightarrow{b} P'_2$, then $P_1 \equiv P_2$ iff $P'_1 \equiv P'_2$.

Proof. Let $\alpha'_1 \in A'_1$. We set $\alpha_1 = \text{Res}_b^{-1}(\alpha'_1)$, $\alpha_2 = \phi_{\text{supp}}(\alpha_1)$, $\alpha'_2 = \text{Res}_b(\alpha_2)$ (Res_b is meant w.r.t. P_1 or P_2 according to the cases). Then we set $\phi'_{\text{supp}} = \text{Res}_b \circ \phi_{\text{supp}} \circ \text{Res}_b^{-1}$. Thus, $\alpha'_2 = \phi'_{\text{supp}}(\alpha'_1)$.

We set $\phi'_{\alpha'_1} = \phi_{\alpha_1}$. Observing the form of $C_1(\alpha_1)(y)$ (for $y \neq x$) given in Section 10.3.5, we set $\phi'_{\alpha'_1|y} = \phi_{\alpha_1|y} \cup \bigcup_{k \in K} \phi_{a(k)|x}$ with $K = \{\text{tr}(a_0) \mid a_0 \in \text{Ax}_{\alpha_1}(x)\}$ and $a(k) = \text{pos}((a_1, x, k))$. □

Notice that ϕ' is deterministically defined from ϕ .

Proposition A.4. If P_1 and P_2 are isomorphic and type the term t (we do not assume them to be approximable), $t \rightarrow^\infty t'$, yielding two derivation P'_1 , P'_2 according to section 10.4.4, then P'_1 and P'_2 are also isomorphism.

Proof. We reuse all the hypotheses and notations of Sec. 10.4.4 and we consider an isomorphism $\phi : P_1 \rightarrow P_2$.

For all $n \in \mathbb{N}$, let P_1^n , P_2^n and ϕ^n be the derivations and derivation isomorphisms obtained after n steps of reduction from P_1 , P_2 and ϕ . Let $\alpha'_1 \in A'_1$ and $N \in \mathbb{N}$ such that, for all $n \geq N$, $|b_n| > |\alpha'_1|$. But then, for any $n \geq N$, $C_i^n(\alpha')(x) = C_i'(\alpha')(x)$, $T_i'(\alpha') = T_i^n(\alpha')$. So we can set $\phi'_{\text{supp}}(\alpha_1) = \phi^n_{\text{supp}}(\alpha_1)$, $\phi'_{\alpha'} = \phi^n_{\alpha'}$. □

A.6 Approximability cannot be defined by means of Multisets

A.6.1 Quantitativity in System \mathcal{R}

Let Γ be *any* context. Using the infinite branch of f^ω , we notice we can give the following variant of derivation Π' (subsection 10.1.3), which still respects the rules of system \mathcal{R} :

$$\frac{\overline{f : [[o] \rightarrow o] \vdash f : [o] \rightarrow o} \quad \Pi'_\Gamma \triangleright f : [[o] \rightarrow o]_\omega + \Gamma \vdash f^\omega : o}{f : [[o] \rightarrow o]_\omega + \Gamma \vdash f^\omega : o}$$

If, for instance, we choose the context Γ to be $x : \tau$, from a quantitative/relevant point of view, the variable x (that is not in the typed term f^ω) should not morally be present in the context. We have been able to “call” the type τ by the mean of an infinite branch (*i.e.* we have performed a weakening). Thus, we can enrich the type of any variable in any part of a derivation, as long it is below an infinite branch (neglecting the bound variables). It motivates the following definition:

Definition A.5. • A semi-rigid derivation P is **quantitative** if, for all $a \in \text{supp}(P)$, $\Gamma(a)(x) = [\tau(a')]_{a' \in \text{Ax}(a)(x)}$.

- A \mathcal{R} -derivation Π is quantitative if any of its semi-rigid representatives is (in that case, all of them are quantitative).

In the next subsection, we show that a derivation Π from system \mathcal{R} can have both quantitative and unquantitative, approximable and not approximable representatives in System \mathcal{S} . It once again shows that rigid constructions allow a more fine-grained control than system \mathcal{R} does on derivations.

A.6.2 Representatives and Dynamics

A rigid derivation P (with the usual notations \mathbb{C} , t , \mathbb{T}) **represents** a derivation Π if the semi-rigid derivation P_* defined by $\text{supp}(P_*) = \text{supp}(P)$ and $P_*(a) = \overline{\mathbb{C}(a)} \vdash t|_a : \overline{\mathbb{T}(a)}$, is a representative of Π . We write $P_1 \stackrel{\mathcal{R}}{=} P_2$ when P_1 and P_2 both represent the same derivation Π .

Proposition A.5. If a rigid derivation P is quantitative, then the derivation \overline{P} (in system \mathcal{R}) is quantitative.

Using natural extensions (Section 10.5.2), it easy to prove:

Proposition A.6. If Π is a quantitative derivation typing a normal form, then, there is a quantitative rigid derivation P s.t. $\overline{P} = \Pi$.

Proof. Let $P(*)$ be a semi-rigid derivation representing Π . We set $A = \text{supp}(P(*))$ and for all full position $a \in A$, we choose a representative $\mathbb{T}(\hat{a})$ of $\tau(a)$. We apply then the special construction, which yields a rigid derivation P such that $P_* = P(*)$ (we show that, for all $a \in A$, $\mathbb{T}(a)$ represents $\tau(a)$). \square

$$P' = \frac{\frac{f : ((2 \cdot o) \rightarrow o)_2 \text{ [1]}}{\frac{f : ((2 \cdot o) \rightarrow o)_3 \text{ [1]} \quad P' \triangleright f : ((2 \cdot o) \rightarrow o)_{k \geq 4} \vdash f^\omega : o \text{ [2]}}{f : ((2 \cdot o) \rightarrow o)_{k \geq 3} \vdash f^\omega : o \text{ [2]}}}}{f : ((2 \cdot o) \rightarrow o)_{k \geq 2} \vdash f^\omega : o}$$

$$\tilde{P}' = \frac{\frac{f : ((2 \cdot o) \rightarrow o)_2 \text{ [1]}}{\frac{f : ((2 \cdot o) \rightarrow o)_4 \text{ [1]} \quad P' \triangleright f : ((2 \cdot o) \rightarrow o)_{k=3 \vee k \geq 5} \vdash f^\omega : o \text{ [2]}}{f : ((2 \cdot o) \rightarrow o)_{k \geq 3} \vdash f^\omega : o \text{ [2]}}}}{f : ((2 \cdot o) \rightarrow o)_{k \geq 2} \vdash f^\omega : o}$$

Figure A.4: Two Representatives of Π'

We can actually prove that every quantitative derivation can be represented with a quantitative rigid derivation and that we can endow it with every possible infinitary reduction choice (Theorem 13.2). However, a quantitative derivation can also have an unquantitative rigid representative (see below Π' and \tilde{P}').

Actually, whereas $P_1 \equiv P_2$ (Sec. A.5) means that P_1 and P_2 are isomorphic in every possible way, $P_1 \stackrel{\mathcal{R}}{\equiv} P_2$ is far weaker: we explicit in this subsection big differences in the dynamical behaviour between two rigid representatives of the derivations Π and Π' of Subsection 10.1.3.

We omit the right side of axiom rules *e.g.*, $f : ((2 \cdot o) \rightarrow o)_2$ stands for $f : ((2 \cdot o) \rightarrow o)_2 \vdash f : (2 \cdot o) \rightarrow o$.

- Let P_k ($k \geq 2$) and P be the following rigid derivations:

$$P_k = \frac{\frac{f : ((2 \cdot o) \rightarrow o)_k \text{ [1]} \quad \frac{x : (\rho)_2 \text{ [1]} \quad x : (\rho)_i \text{ [i-1]}}{x : (\rho)_{i \geq 2} \vdash xx : o \text{ [2]}}}{f : ((2 \cdot o) \rightarrow o)_k \vdash f(xx) : o \text{ [0]}}}{f : ((2 \cdot o) \rightarrow o)_k \vdash \Delta_f : \rho}$$

$$P = \frac{P_2 \text{ [1]} \quad (P_k \text{ [k-1]})_{k \geq 3}}{f : ((2 \cdot o) \rightarrow o)_{k \geq 2} \vdash \Delta_f \Delta_f}$$

- Let \tilde{P}_k ($k \geq 2$) and \tilde{P} be the following rigid derivations:

$$\tilde{P}_k = \frac{\frac{f : ((2 \cdot o) \rightarrow o)_k \text{ [1]} \quad \frac{x : (\rho)_3 \text{ [1]} \quad x : (\rho)_2 \text{ [2]} \quad (x : (\rho)_i \text{ [i-1]})_{i \geq 4}}{x : (\rho)_{i \geq 2} \vdash xx : o \text{ [2]}}}{f : ((2 \cdot o) \rightarrow o)_k \vdash f(xx) : o \text{ [0]}}}{f : ((2 \cdot o) \rightarrow o)_k \vdash \Delta_f : \rho}$$

$$\tilde{P} = \frac{\tilde{P}_2 [1] \quad (\tilde{P}_k [k-1])_{k \geq 3}}{f : ((2 \cdot o) \rightarrow o)_{k \geq 2} \vdash \Delta_f \Delta_f}$$

• The rigid derivations P and \tilde{P} both represent Π . Morally, subject reduction in P will consist in taking the first argument P_3 , placing it on the first occurrence of x in $f(xx)$ (in P_2) and putting the other P_k ($k \geq 4$) in the different axiom rules typing the second occurrence of x in the same order. There is a simple decrease on the track number and we can go this way towards f^ω .

The rigid derivation \tilde{P} process the same way, except it will always skip \tilde{P}_3 (\tilde{P}_3 will remain on track 2). Morally, we perform subject reduction "by-hand" while avoiding to ever place P_3 in head position.

The definitions of Sec. 10.4.4 show that infinitary reductions performed in P and \tilde{P} yield respectively to P' and \tilde{P}' of Figure A.4.

Thus, P' and \tilde{P}' both represent Π' (from subsec. 10.1.3), but P' is quantitative whereas \tilde{P}' is not (the track 3 w.r.t. f does not end in an axiom leaf). Thus, quantitativity is not stable under s.c.r.s.

Moreover, it is easy to check that P and P' are approximable (reuse the finite derivations of Sec.; 10.1.3) or infinitary subject reduction/expansion. Thus, Π and Π' have both approximable and not approximable representatives. This provides a new argument for the impossibility of formulating approximability in system \mathcal{R} .

A.7 A Positive Answer to TLCA Problem 20

For all $n \in \mathbb{N}$, we denote by \mathfrak{S}_n the set of permutation of $\{1, \dots, n\}$.

Definition A.6.

- For all $x \in \mathcal{V}$, the sets HP^x of x -Hereditary Permutations (x -HP) ($x \in \mathcal{V}$) are defined by mutual coinduction:

$$p^x := \lambda y_1 \dots y_n. x p^{y_{\sigma(1)}} \dots p^{y_{\sigma(n)}} \quad (n \geq 0, \sigma \in \mathfrak{S}_n)$$

Moreover, the head variable x of p_x is written $x = \text{hv}(p_x)$ and its order n is written $n = \text{or}(p_x)$.

- A **Hereditary Permutation (HP)** is a term of the form $p = \lambda x. p^x$.

We now want to define the *permutation pairs* $((2 \cdot S), T)$ (with S, T types of system \mathbf{S}) so that the judgments of the form $x : (2 \cdot S) \vdash t : T$ characterize the x -HP (*i.e.* there is an approximable $P \triangleright x : (2 \cdot S) \vdash t : T$ iff t is a x -HP). Informally, if $p^x = \lambda y_1 \dots y_n. x p^{y_{\sigma(1)}} \dots p^{y_{\sigma(n)}}$ and we type p^x with a type of order n , then we have:

- Type of $p^x = (\text{type of } y_1) \rightarrow \dots (\text{type of } y_n) \rightarrow o$
- Type of $x = (\text{type of } p^{y_{\sigma(1)}}) \rightarrow \dots (\text{type of } p^{y_{\sigma(n)}}) \rightarrow o$

Since y_1, \dots, y_n are the respective head variable of the $*$ -permutations p^{y_1}, \dots, p^{y_n} , this suggests the following definition:

Definition A.7.

- For all $o \in \mathcal{O}$, the sets PP^o of o -**permutation pairs** $((2 \cdot S), T)$ (where S and T are \mathbf{S} -types) are defined by mutual coinduction:

$$\frac{(S_1, T_1) \in \text{PP}^{o_1}, \dots, (S_n, T_n) \in \text{PP}^{o_n} \quad o_1, \dots, o_n, o \text{ pairwise distinct} \quad \sigma \in \mathfrak{S}_n}{((2 \cdot (2 \cdot T_{\sigma(1)})) \rightarrow \dots \rightarrow (2 \cdot T_{\sigma(n)}) \rightarrow o), (2 \cdot S_1) \rightarrow \dots (2 \cdot S_n) \rightarrow o) \in \text{PP}^o}$$

In the case above, we also say that o is the **last type variable** of (S, T) (for short, $o = \text{ltv}(S, T)$), and that n is the **order** of (S, T) (for short, $n = \text{or}(S, T)$).

- A pair $(S, T) \in \text{PP}^o$ is said to be **proper** (written $(S, T) \in \text{PPP}^o$) for all $o' \in \mathcal{O}$, o' occurs at most once in S and in T .
- A **permutation type** (metavariable U) is a type U of the form $(2 \cdot S) \rightarrow T$ where (S, T) is a proper permutation pair.

The condition of properness is here to ensure that every variable occurs at a level deeper than its binder and to distinguish them from one another (see the proof of Claim A.2). The first implication of the characterization is quite natural to prove:

Claim A.1. Let $x \in \mathcal{V}$ and p^x be a x -permutation pair. Then there is an approximable \mathbf{S} -derivation P and a permutation pair $((2 \cdot S), T)$ such that $P \triangleright x : (2 \cdot S) \vdash p^x : T$.

Proof sketch. Claim A.1 is proved by using a method similar to that of natural extensions (see Sec. 10.5.1, 10.5.2 and A.4).

Let us set $t = p^x$. Let $y \mapsto o_y$ be an injection from $B = \text{supp}(t)$ to \mathcal{O} . We associate to each $b \in \text{supp}(t)$ two indeterminates X_b and Y_b . The idea is that X_b is a placeholder for the types of head variables and Y_b is a placeholder for the types of the sub-hereditary permutations of t .

We denote by B^p the set of positions b of subterms of t that are y -HP for some $y \in \mathcal{V}$ and, for all $b \in B^p$, $\text{hvp}(b)$ denotes the position of the head variable of $t|_b$ (hvp stands for “head variable position”).

Formally, we set $B^p = \{\varepsilon\} \cup \{b \cdot 2 \mid b \in \{0, 1, 2\}^*\}$ and, for all $b \in B^p$, $\text{hvp}(b)$ is the longest b_0 such $b_0 \in b \cdot \{0, 1\}^*$.

For all $b \in B^p$, we abusively denote by $\text{or}(b)$ the order $\text{or}(t|_b)$ of $t|_b$ and by x_b the head variable of $\text{hv}(t|_b)$ (e.g., $x_\varepsilon = x$). Observe that, if $b \in B^p$ and $n = \text{or}(b)$, then $\text{hvp}(b) = b \cdot 0^n \cdot 1^n$. We just write o_b instead of

Moreover, for $b \in B^p$, then $t|_b$ is of the form $\lambda y_1 \dots y_n. y p^{y\sigma_1} \dots p^{y\sigma_n}$ with $n = \text{or}(b) \geq 0$, $y = x_b$ and $\sigma \in \mathfrak{S}_n$. We then denote by σ_b the permutation σ and we set $b(k) = b \cdot 0^n \cdot 1^{k-1} \cdot 2$ for $1 \leq k \leq n$, so that $b(k)$ is the position of $p^{y\sigma_n}$. For $1 \leq k \leq n$, we also abusively write $b(\sigma(k))$ instead of $b(\sigma_b(k))$.

We then set, for all $b \in B^p$:

$$\begin{aligned} \mathbf{F}(b) &= (2 \cdot Y_{b(\sigma(1))}) \rightarrow \dots \rightarrow (2 \cdot Y_{b(\sigma(n))}) \rightarrow o_b \\ \mathbf{G}(b) &= (2 \cdot X_{b(1)}) \rightarrow \dots \rightarrow (2 \cdot X_{b(n)}) \rightarrow o_b \end{aligned}$$

We then coinductively define, for all $b \in B^p$,

$$\begin{aligned} S(b) &= \mathbf{F}(b)[S(b')/X_{b'}, T(b')/Y_{b'}]_{b' \in \text{supp}(t)} \\ T(b) &= \mathbf{G}(b)[S(b')/X_{b'}, T(b')/Y_{b'}]_{b' \in \text{supp}(t)} \end{aligned}$$

By proceeding as in Sec. A.4, we prove that, for all $b \in B^p$, $(S(b), T(b))$ is a proper permutation pair. From there, it is not difficult to build a (quantitative) \mathbf{S} -derivation P , such that, for all $b \in B^p$, $P(b) = x_b : (2 \cdot S(b)) \vdash t|_b : T(b)$. Since t is a normal form, by Lemma 10.13, P is approximable, which concludes the proof. \square

Claim A.2. Let $t \in \Lambda^{001}$ be a term, P an approximable \mathbf{S} -derivation and U a permutation type such that $P \triangleright \vdash t : U$. Then t is a hereditary permutation.

Proof sketch. By Definition A.7, let (S, T) the proper permutation pair such that $U = (2 \cdot S) \rightarrow T$. We can write $S = (2 \cdot T_{\sigma(1)}) \rightarrow \dots \rightarrow (2 \cdot T_{\sigma(n)}) \rightarrow o$ and $T = (2 \cdot S_1) \rightarrow \dots \rightarrow (2 \cdot S_n) \rightarrow o$.

Moreover, also by Definition A.7, the empty sequence type $()$ does not occur in U , so that $\vdash t : U$ is unforgetful. Since P is approximable, this entails that t is weakly normalizing by Proposition 10.4. Since the context in $\vdash t : U$ is empty, t is closed and of order ≥ 1 (since, in particular, t is HN). Thus, $t \rightarrow^\infty t' = \lambda x_0. \lambda y_1 \dots y_p. x t_1 \dots t_q$ with t_1, \dots, t_q normal forms whose respective head variables are denoted y_1, \dots, y_q (and by Proposition 10.5, $\triangleright \vdash t' : U$).

Since (S, T) is proper, o does not occur in S_1, \dots, S_n , so necessarily, $x = x_0$ and $x : (2 \cdot S) \vdash \lambda y_1 \dots y_p. x t_1 \dots t_q : T$ is derivable by means of an approximable derivation P_* . Since T and S are types of order n , we must have $p = q \leq n$ (the **abs**-rule creates an arrow whereas the **app**-rule destroys one). Moreover, S_1, \dots, S_n are the respective types of y_1, \dots, y_n . Let us denote o_1, \dots, o_n the respective last type variables of S_1, \dots, S_n (as such, they occur at applicative depth 0 in S_1, \dots, S_k and at applicative depth 1 in S and T).

Since system \mathbf{S} is relevant and $(2 \cdot S_k)$ is a *singleton* sequence type, y_1, \dots, y_n must be typed exactly typed once in the derivation. Moreover:

- o_1, \dots, o_p occur respectively in T_1, \dots, T_n .
- $o_{\sigma(p)}, \dots, o_{\sigma(1)}$ are the respective last type variable of $T_{\sigma(1)}, \dots, T_{\sigma(p)}$, which are respectively the unique types of $t_1 \dots t_p$.

Now, if a y_k was typed at applicative depth ≥ 2 (*i.e.* if there was a $a \in \text{supp}(P_*)$ such that $t(a) = y_k$ and a bound by λy_k at position 0^{k-1}), then o_k would occur at applicative depth ≥ 2 in S or in T by Lemma A.5. This would contradict the properness of (S, T) . Thus, y_1, \dots, y_p occur at applicative depth 1 in $\lambda y_1 \dots y_p. x t_1 \dots t_q$ *i.e.* they are the head variables of the t_1, \dots, t_p . Since the o_1, \dots, o_p are pairwise distinct, we can even assert that y_1, \dots, y_p are respective head variable of $t_{\sigma^{-1}(1)}, \dots, t_{\sigma^{-1}(p)}$.

This easily implies that $y_1 : S_1 \vdash t_{\sigma^{-1}(1)} : T_1, \dots, y_p : S_p \vdash t_{\sigma^{-1}(p)} : T_p$ are approximably derivable judgments. We conclude by coinduction. \square

Lemma A.5 expresses the fact that, in an unforgetful derivation typing a normal form, every type nested in a subterm at applicative depth n occurs at applicative depth $\geq n$ in the context or the type in the root of the derivation:

Lemma A.5. Let P be an approximable derivation concluding with $C \vdash t : T$, where t is a normal form, and $a \in \text{supp}(P)$ such that $\text{ad}(a) \geq 1$.

Then there is $c_0 \in \text{supp}(T)$ such that $\text{ad}(c_0) \geq \text{ad}(a)$ and $T|_{c_0} = \mathbf{T}^P(a)$ or there is $x \in \mathcal{V}$ and $k \cdot c_0 \in \text{supp}(C(x))$ such that $\text{ad}(c_0) \geq 1$ and $C(x)|_{k \cdot c_0} = \mathbf{T}^P(a)$.

Proof sketch. We proceed by induction on $\text{ad}(a)$. Let us just informally explain the case $\text{ad}(a) = 1$.

Say that $t = \lambda x_1 \dots x_p. x t_1 \dots t_q$. Let T_0 be the type assigned to the head variable x . Then T_0 is an arrow type of order $\geq q$ whose sources are the types assigned to t_1, \dots, t_q (see the right-hand side of Fig. 3.4 for an example in system \mathcal{R}_0). Thus, the types of the arguments t_1, \dots, t_q occur at applicative depth 1 in T_0 .

Moreover, the types of all subterms occurring at applicative depth 1 are nested in the

types of the t_1, \dots, t_q . Thus, the types of the subterms at applicative depth 1 occur at applicative depth 1 in T_0 . To conclude, we just notice that T_0 is nested at applicative depth 1 in T if x is bound (*i.e.* $x = x_k$ for some k) or in $C(x)$ is x is free.

In the case $\text{ad}(a) = 2$, the previous argument shows that the types of the terms occurring at applicative depth occur in types of the terms at applicative depth ≥ 1 . . . \square

We can now positively answer to TLCA Problem # 20 by giving a type-theoretical characterization of the terms whose normal form is a hereditary permutation in system \mathbf{S} :

Theorem A.1. Let $t \in \Lambda^{001}$. Then t is β -equivalent to a hereditary permutation iff $\vdash t : U$ is approximably derivable for some permutation type U .

Proof.

- The implication \Leftarrow is given by Claim A.2.
- Implication \Rightarrow : assume that $t \rightarrow^\infty p$ with p hereditary permutation – say that $p = \lambda x.p_x$. By Claim A.1, there is a proper permutation pair (S, T) and an approximable derivation P such that $P \triangleright x : (2 \cdot S) \vdash p_x : T$. Let us set $U = (2 \cdot S) \rightarrow T$, so that $\vdash p : U$ is approximably derivable. By Proposition 10.7, $\vdash t : U$ is also approximably derivable.

\square

Appendix B

Residuation, Threads and Isomorphisms in System S_{op}

B.1 Subject Reduction

In Sec. B.1, we prove the first point¹ of Proposition 13.1 *i.e.* “pseudo-subject reduction” for system S_{h} , as well as all the claims of Sec. 13.2.2. From Sec. B.1.1 to Sec. B.1.3, we formally prove that interfaces provide a sound way to produce a derivation typing a reduct (Corollary B.1). In Sec. B.1.4, we formally prove that every sequence of reduction choices can be built-in inside an interface, as claimed by Lemma 13.1. In particular, we define properly residuals (Sec. B.1.1) and quasi-residuals, and some associated notions. Note that (quasi-)residuals are more complicated to define than in Sec. 10.3.5 and Sec. 12.4.1 because of interface.

We still assume that $t|_b = (\lambda x.r)s$, $t \xrightarrow{b} t'$ and $P \triangleright C \vdash t : T$. The hybrid derivation P comes along with the usual associated notations *e.g.*, \mathbf{C} for \mathbf{C}^P , \mathbf{T} for \mathbf{T}^P , pos for pos^P (see Sec. 10.3.1 and 10.3.2).

An **operable derivation** is a hybrid derivation endowed with a total interface. If P is an operable derivation whose interface is $(\phi_a)_{a \in \text{supp}_{\text{a}}(P)}$, we usually only write² ρ_a for $\text{Rt}(\phi_a)$ (so that ρ_a is a root interface) and we set $\mathbf{L}^P = \{(a \cdot 1, k \cdot c) \in \text{bisupp}(P) \mid k \in \mathbb{N} \setminus \{0, 1\}\}$ and $\mathbf{R}^P = \{(a \cdot k, c) \in \text{bisupp}(P) \mid k \in \mathbb{N} \setminus \{0, 1\}\}$. For all $\mathbf{p} = (a \cdot 1, k \cdot c) \in \mathbf{L}^P$, we just write $\phi(\mathbf{p})$ for $(a \cdot k', c')$ with $k' \in \mathbb{N} \setminus \{0, 1\}$, $c' \in \mathbb{N}^*$ and $k' \cdot c' = \phi_a(k \cdot c)$.

Notation $\mathbf{Ax}_a^P(x)$ In Appendix B.1, we reuse the notation $\mathbf{Ax}_a^P(x)$ from Sec. 10.3.2: if $a \in \text{supp}(P)$ and $x \in \mathcal{V}$, we have $\mathbf{Ax}_a^P(x) = \{a_0 \in \text{supp}(P) \mid a \leq a_0, t(a) = x, \nexists a'_0, a \leq a'_0 \leq a_0, t(a'_0) = \lambda x\}$ (positions of **ax**-rules in P above a typing occurrences of x that are not bound w.r.t. a).

Remark B.1. Let us call a tree $A \subset \mathbb{N}^*$ such that, for all infinite branch a of A , $\text{ad}(a) = \infty$, a **001-tree**. Let \mathcal{P} be a predicate on such a tree A . In order to prove “for all $a \in A$, $\mathcal{P}(a)$ ”, we can reason by **001-induction**: we prove that $\mathcal{P}(a)$ for all leaf a of A and then, for all $a \in \mathbb{N}^*$ such $a \cdot 0$ or $a \cdot 1$ is in A , we prove that $\mathcal{P}(a \cdot 0)$ or $\mathcal{P}(a \cdot 1)$ implies $\mathcal{P}(a)$.

¹ The proofs can be adapted for the second point (“pseudo-subject expansion”).

² See Sec. 13.1.1 for notation $\text{Rt}(\phi)$.

B.1.1 Residual Derivation (Hybrid)

We assume that P is endowed with total root interface $(\rho_a)_{\bar{a}=b}$ at position b . Using this root interface, we will now build a hybrid derivation P' concluding with $C \vdash t : T'$ with $T \equiv T'$, as it was announced at the beginning of Sec. 13.2.

Conventions on metavariables a and α As in Sec. 10.3.5 and 12.4.1, the letter a will stand for a position of P that corresponds to the root of the redex (*i.e.* $a \in \text{supp}(P)$ and $\bar{a} = b$) and the letter α for other positions in $\text{supp}(P)$ or even in A . We set $\text{Ax}_\lambda(a) = \text{Ax}_{a \cdot 1 \cdot 0}^P(x)$ and $\text{Tr}_\lambda(a) = \{\text{tr}^P(\alpha_0) \mid \alpha_0 \in \text{Ax}_\lambda(a)\} = \text{Rt}(T^P(a \cdot 1))$. Thus, $\text{Ax}_\lambda(a)$ is the set of positions of the redex variable (to be substituted) above a and $\text{Tr}_\lambda(a)$ is the set of the axiom tracks that have been used for them. For instance, in Fig. 13.2, $\text{Ax}_\lambda(a) = \{a \cdot 1 \cdot 0 \cdot a_2, a \cdot 1 \cdot 0 \cdot a_7\}$ and $\text{Tr}_\lambda(a) = \{2, 7\}$.

First, we define the residual position $\text{Res}_b(\alpha)$ for each $\alpha \in \text{supp}(P)$ except when α is of the form a , $a \cdot 1$ or $a \cdot 1 \cdot 0 \cdot a_k$ (for some a satisfying $\bar{a} = b$). We begin with discussing the symbols \heartsuit and \clubsuit in Fig.13.2. In Fig. 13.2, \heartsuit represents a judgment nested in P_7 . Thus, its position must be of the form $a \cdot 1 \cdot 0 \cdot \alpha_\heartsuit$. After reduction, the **app**-rule and **abs**-rule at positions a and $a \cdot 0$ have been destroyed and the position of this judgment \heartsuit will be $a \cdot \alpha_\heartsuit$. We set then $\text{Res}_b(a \cdot 1 \cdot 0 \cdot \alpha_\heartsuit) = a \cdot \alpha_\heartsuit$.

Likewise, \clubsuit represents a judgment nested in the argument derivation P_8 on track 8 w.r.t. a . Thus, its position must be of the form $a \cdot 8 \cdot \alpha_\clubsuit$ where $a \cdot 8$ is the root of P_8 . After reduction, P_8 will replace the **ax**-rule typing x on track $\rho_a^{-1}(8)$ *i.e.* 2, so its root will be at $a \cdot a_2$ (by definition of a_2). Thus, after reduction, the position of judgment \clubsuit will be $a \cdot a_2 \cdot \alpha_\clubsuit$. We set then $\text{Res}_b(a \cdot 7 \cdot \alpha_\clubsuit) = a \cdot a_2 \cdot \alpha_\clubsuit$.

- Paradigm \clubsuit : if $\alpha = a \cdot k_R \cdot \alpha_0$ where $\bar{a} = b$ and $k_R \in \text{Arg}(a)$, then $\text{Res}_b(\alpha) = a \cdot a_{k_L} \cdot \alpha_0$ with $k_L = \rho_a^{-1}(k_R)$.
- Paradigm \heartsuit : if $\alpha = a \cdot 1 \cdot 0 \cdot \alpha_0$ where $a \in \text{supp}(P)$, $\bar{a} = b$ and $\alpha_0 \neq a_k$, then $\text{Res}_b(\alpha) = a \cdot \alpha_0$.
- Outside the redex: if $b \not\leq \bar{a}$, then $\text{Res}_b(\alpha) = \alpha$

We set $A' = \{\text{Res}_b(\alpha) \mid \alpha \in A\} = \text{codom}(\text{Res}_b)$ and we call A' the **residual support** of P (w.r.t. reduction at position b and root interface $(\rho_a)_{\bar{a}=b}$). By case analysis, we check that A' is a tree and that Res_b is an *injection* from $\text{dom}(\text{Res}_b)$ to A' . Moreover, we set $A'_\heartsuit = \{\text{Res}_b(\alpha) \mid \alpha \in \text{dom}(\text{Res}_b), \bar{\alpha} \geq b \cdot 1 \cdot 0\}$, $A'_\clubsuit = \{\text{Res}_b(\alpha) \mid \alpha \in \text{dom}(\text{Res}_b), \bar{\alpha} \geq b \cdot 2\}$ and $A'_{\text{out}} = \{\alpha \in A \mid \bar{\alpha} \not\geq b\}$, so that A' is the disjoint union of A'_\heartsuit , A'_\clubsuit and A'_{out} .

Remark B.2 (Induction and reduction). Assume that $\text{Res}_b(\alpha_i) = \alpha'_i$ ($i = 1, 2$) and $\alpha_1 \leq \alpha'_2$.

- If $\alpha'_1 \in A'_\clubsuit$, then $\alpha'_2 \in A'_\clubsuit$.
- If $\alpha'_1 \in A'_\heartsuit$, then $\alpha'_2 \in A'_\heartsuit \cup A'_\clubsuit$.
- If $\alpha'_1 \in A'_{\text{out}}$, then $\alpha'_2 \in A'_{\text{out}} \cup A'_\heartsuit \cup A'_\clubsuit = A'$.

So, a 001-induction on A' should be split in three 001-inductions: first, one on A'_\clubsuit , then one on A'_\heartsuit , then, one on A'_{out} . See for instance the proof of Lemma B.2.

Conversely, we check that the converse injection Res_b^{-1} from A' to $\text{dom}(\text{Res}_b)$ satisfies:

- If $b \not\leq \bar{\alpha}'$, we set $\mathbf{Res}_b^{-1}(\alpha') = \alpha'$.
- If $\alpha' = a \cdot \alpha'_0$ where $a \in \mathbf{supp}(P)$, $\bar{a} = b$ but there is no k s.t. $b \geq a \cdot a_k$, we set $\mathbf{Res}_b^{-1}(\alpha') = a \cdot 1 \cdot 0 \cdot \alpha'_0$.
- If $\alpha' = a \cdot a_k \cdot \alpha'_0$ where $a \in \mathbf{supp}(P)$, $\bar{a} = b$, we set $\mathbf{Res}_b^{-1}(\alpha') = a \cdot \rho_a(k) \cdot \alpha'_0$.

If A is a tree (resp. T a labelled tree) and $a \in A$ or $a \in \mathbf{supp}(T)$, then $\mathbf{child}^A(a) = \{k \in \mathbb{N} \mid a \cdot k \in A\}$ and $\mathbf{child}^T(a) = \{k \in \mathbb{N} \mid a \cdot k \in \mathbf{supp}(T)\}$. By case analysis:

Lemma B.1. For all $\alpha' \in A'$ and α such that $\mathbf{Res}_b(\alpha) = \alpha'$:

- $\bar{\alpha}' \in \mathbf{supp}(t')$
- $t'(\bar{\alpha}') = t'(\bar{\alpha})$.
- $\mathbf{child}'(\alpha') = \mathbf{child}(\alpha)$ (where $\mathbf{child} = \mathbf{child}^A$ and $\mathbf{child}' = \mathbf{child}^{A'}$).

We set $A'_{\textcircled{a}} = \{a \in A' \mid t'(a) = \textcircled{a}\}$. By Lemma B.1, $A'_{\textcircled{a}} = \{\mathbf{Res}_b(a) \mid a \in \mathbf{supp}_{\textcircled{a}}(P)\}$.

B.1.2 Residual Types and Contexts (Hybrid Derivations)

In this paragraph, we define the residual derivation P' of P (w.r.t. reduction at position b and the root interface $(\rho_a)_{\bar{a}=b}$). The residual support A' will be the support of P' (i.e. $A' = \mathbf{supp}(P')$), but we must also define the contexts $\mathbf{C}^{P'}(\alpha')$ and types $\mathbf{T}^{P'}(\alpha')$ for all $\alpha' \in A'$. Since P' is not built yet, we will respectively denote these contexts and types $\mathbf{C}'(\alpha')$ and $\mathbf{T}(\alpha')$ when we define them.

We assume that t satisfies Barendregt convention i.e. for all $y \in \mathcal{V}$, λy occurs at most once in t and the sets of free variables and of bound variables of t are disjoint.

Let \mathbf{Ax} and \mathbf{Ax}' the respective sets of leaves of A and A' . For all $\alpha' \in A'$ and $y \in \mathcal{V}$, $y \neq x$, we set $\mathbf{Ax}'_{\alpha'}(y) = \{\alpha'_0 \in A' \mid \alpha'_0 \geq \alpha' \text{ and } \mathbf{Res}_b^{-1}(\alpha'_0) \in \mathbf{Ax}(y)\}$.

We observe that, for all $\alpha' \in \mathbf{Ax}'$, $\mathbf{Res}_b^{-1}(\alpha') \in \mathbf{Ax}$. Then, we set $\mathbf{tr}'(\alpha') = \mathbf{tr}^P(\mathbf{Res}_b^{-1}(\alpha'))$ and, for all $\alpha' \in A'$ and $y \in \mathcal{V}$, $y \neq x$, $\mathbf{C}'(\alpha')(y) = (\mathbf{tr}'(\alpha'_0) \cdot \mathbf{T}(\mathbf{Res}_b^{-1}(\alpha'_0)))_{\alpha'_0 \in \mathbf{Ax}'_{\alpha'}(y)}$. This definition is sound, because, if $\alpha_1, \alpha_2 \in \mathbf{Ax}_{\alpha'}(y)$ for some $\alpha' \in A'$ and $x \in \mathcal{V}$, then $\mathbf{tr}'(\alpha_1) = \mathbf{tr}'(\alpha_2)$ implies $\alpha_1 = \alpha_2$ (case analysis).

When $t'(\bar{\alpha}') = \lambda y$ (with $\alpha' \in A'$), a case analysis shows that $\mathbf{Ax}_{\alpha'}(y) = \{\mathbf{Res}_b(\alpha_0) \mid \alpha_0 \in \mathbf{Ax}_{\alpha}(y)\}$ where $\alpha' \in A'$. Thus, in that case, $\mathbf{C}(\alpha')(y) = (\mathbf{tr}'(\alpha'_0) \cdot \mathbf{T}(\mathbf{Res}_b^{-1}(\alpha'_0)))_{\alpha'_0 \in \mathbf{Ax}'_{\alpha'}(y)} = (\mathbf{tr}'(\alpha_0) \cdot \mathbf{T}(\alpha_0))_{\alpha_0 \in \mathbf{Ax}_{\alpha}(y)} = \mathbf{C}(\alpha)(y)$.

By a 001-induction on $\alpha' \in A'$, we define now $\mathbf{T}'(\alpha')$:

- When $\alpha' \in \mathbf{Ax}'$, $\mathbf{T}'(\alpha') = \mathbf{T}(\mathbf{Res}_b^{-1}(\alpha'))$.
- When $t'(\bar{\alpha}') = \lambda y$, we set $\mathbf{T}'(\alpha) = \mathbf{C}(\alpha \cdot 0)(y) \rightarrow \mathbf{T}'(\alpha \cdot 0)$.
- When $t'(\bar{\alpha}') = \textcircled{a}$, we set $\mathbf{T}'(\alpha') = \mathbf{Tg}(\mathbf{T}'(\alpha' \cdot 1))$.

We define then P' as the labelled tree s.t. $\text{supp}(P') = A'$ and for all $\alpha' \in A'$, $P'(\alpha') = \mathcal{C}'(\alpha') \vdash t'_{|\alpha'} : \mathbf{T}'(\alpha')$ (so that $\mathcal{C}' = \mathcal{C}^{P'}$ and $\mathbf{T}' = \mathbf{T}^{P'}$ as expected). We intend to prove that P' is a correct hybrid derivation typing t' .

As hinted at Sec. 13.2, we must check that a type $\mathbf{T}(\alpha)$ (where $\alpha \in A$) may only be replaced with a type $\mathbf{T}'(\alpha')$ (where $\alpha' = \text{Res}_b(\alpha)$) that is isomorphic to $\mathbf{T}(\alpha)$ in the residual derivation typing t' that we are going to define.

Quasi-Residuation in the Hybrid Setting In order to check that, it is convenient to extend residuation into *quasi-residuation*: namely, we define **quasi-residual** $\text{QRes}_b(\alpha)$ for any $\alpha \in A$ such that $\bar{a} \neq b \cdot 1$ by setting $\text{QRes}_b(\alpha) = \text{Res}_b(\alpha)$ when $\text{Res}_b(\alpha)$ is defined, $\text{QRes}_b(a) = a$ and $\text{QRes}_b(a \cdot 1 \cdot 0 \cdot a_k) = a \cdot a_k$ when $\bar{a} = b$ and $k \in \text{RedTr}(a)$.

Remark B.3.

- We do not necessarily have $t(\alpha) = t'(\alpha')$ or $\text{child}(\alpha) = \text{child}'(\alpha')$ when $\alpha' = \text{QRes}_b(\alpha)$ (compare with Lemma B.1) and QRes_b is usually not injective. For instance, if $t = (\lambda x.y)y$, $t' = y$, $b = \varepsilon = \alpha = \alpha'$, then $t \xrightarrow{b} t'$, $\alpha' = \text{QRes}_b(\alpha)$ but $t(\alpha) = @ \neq y = t'(\alpha')$ and $\text{child}(\alpha)$ contains at least 1 but $\text{child}'(\alpha')$ is empty.
- However, quasi-residuals will be useful to define the isomorphisms $\text{Res}_{b|\alpha}$, $\text{ResR}_{b|\alpha}$ and $\text{ResL}_{b|\alpha}$ below.

Lemma B.2. This lemma is also a definition: that of the **quasi-residuation**.

- For all $\alpha' \in A'$, $\mathbf{T}'(\alpha') \equiv \mathbf{T}(\alpha)$ where $\alpha = \text{Res}_b^{-1}(\alpha')$. Besides, if $\alpha' \in \text{Ax}'$ or $\alpha' \geq a \cdot a_k$ (for $\bar{a} = b$), then $\mathbf{T}'(\alpha') = \mathbf{T}(\alpha)$.
- More precisely, if P is endowed with an interface $(\phi_a)_{\bar{a}=b}$ at position b (extending the root-interface $(\rho_a)_{\bar{a}=b}$), then, for all $\alpha \in A$ and $\alpha' \in A'$ such that $\text{QRes}_b(\alpha) = \alpha'$, we can define a type isomorphism $\text{QRes}_{b|\alpha} : \mathbf{T}(\alpha) \rightarrow \mathbf{T}'(\alpha')$ by 001-induction on α' .
- When $\text{Res}_b(\alpha) = \alpha'$, we write $\text{Res}_{b|\alpha}$ instead of $\text{QRes}_{b|\alpha}$. Moreover, $\text{Res}_{b|\alpha}$ is the identity if $\alpha' \in \text{Ax}'$ or $\alpha' \geq a \cdot a_k$ for some $a \in A, \bar{a} = b$ and $k \in \text{AxTr}(a \cdot 1 \cdot 0)(x)$.

Proof. We proceed by 001-induction on $\alpha' \in A' := \text{supp}(P')$ and split the cases as suggested in Remark B.2.

- Paradigm ♣: $\text{Res}_b(\alpha) = \alpha'$ and $\alpha' \geq a \cdot a_k$ (for some $a \in A, \bar{a} = b$ and $k \in \text{RedTr}(a)$).
 - Subcase $t(\bar{a}) = y$: here, $t(\bar{a}) = y \neq x$ and $\mathbf{T}'(\alpha') = \mathbf{T}(\alpha)$.
 - Subcase $t(\bar{a}) = \lambda y$: $\alpha \cdot 0 \in A$, $\text{Res}_b(\alpha \cdot 0) = \alpha' \cdot 0$ and by IH, we have $\mathbf{T}'(\alpha' \cdot 0) = \mathbf{T}(\alpha \cdot 0)$ and $\text{Res}_{b|\alpha \cdot 0}$ is the identity $\text{id}_{\mathbf{T}(\alpha \cdot 0)}$. Since $\mathbf{T}(\alpha) = \mathcal{C}(\alpha \cdot 0)(y) \rightarrow \mathbf{T}(\alpha \cdot 0)$ and $\mathbf{T}'(\alpha') = \mathcal{C}(\alpha \cdot 0)(y) \rightarrow \mathbf{T}'(\alpha' \cdot 0)$, we also have $\mathbf{T}(\alpha) = \mathbf{T}'(\alpha)$ and we set $\text{Res}_{b|\alpha} = \text{id}_{\mathbf{T}(\alpha)}$.
 - Subcase $t(\bar{a}) = @$: $\alpha \cdot 1 \in A$, $\text{Res}_b(\alpha \cdot 1) = \alpha' \cdot 1$ and by IH, we have $\mathbf{T}'(\alpha \cdot 1) = \mathbf{T}(\alpha' \cdot 1)$ and $\text{Res}_{b|\alpha \cdot 1}$ is the identity $\text{id}_{\mathbf{T}(\alpha \cdot 1)}$. Moreover, $\mathbf{T}(\alpha) = \text{Tg}(\mathbf{T}(\alpha \cdot 1))$ and $\mathbf{T}'(\alpha') = \text{Tg}(\mathbf{T}'(\alpha' \cdot 1))$. So $\mathbf{T}(\alpha) = \mathbf{T}(\alpha')$ and we set $\text{Res}_{b|\alpha} = \text{id}_{\mathbf{T}(\alpha)}$.

- Paradigm \heartsuit : $\alpha \geq a \cdot 1 \cdot 0$ and $\alpha' \geq a$ (for some $a \in A, \bar{a} = b$):
 - Subcase $\alpha = a \cdot 1 \cdot 0 \cdot a_{k_L}$ and $\alpha' = a \cdot a_{k_L}$: $\alpha' = \mathbf{Res}_b(a \cdot k_R)$ (where $k_R = \rho_a(k_L)$) and by IH, $T'(\alpha') = T(a \cdot k_R)$. Moreover, since $T(\alpha) = L(\alpha)|_{k_L}$, we can set $\mathbf{QRes}_{b|\alpha} = \phi_{a|k_L}$.
 - Subcase $t(\bar{\alpha}) = y \neq x$: $\mathbf{Res}_b(\alpha) = \alpha'$ and $T'(\alpha') = T(\alpha)$.
 - Subcase $t(\bar{\alpha}) = \lambda y$: $\alpha \cdot 0 \in A$, $\mathbf{Res}_b(\alpha \cdot 0) = \alpha' \cdot 0$: we set $\mathbf{Res}_{b|\alpha} = C(\alpha \cdot 0)(y) \rightarrow \mathbf{QRes}_{b|\alpha \cdot 0}$.
 - Subcase $t(\bar{\alpha}) = @$: $\alpha \cdot 1 \in A$, $\mathbf{Res}_b(\alpha \cdot 1) = \alpha' \cdot 1$: we set $\mathbf{Res}_{b|\alpha} = \mathbf{Tg}(\mathbf{QRes}_{b|\alpha \cdot 1})$.
- Outside the redex: $\bar{\alpha} \not\geq b$:
 - Subcase $\alpha' \in \mathbf{Ax}'$: here, $t(\bar{\alpha}) = y \neq x$ and $T'(\alpha') = T(\alpha)$.
 - Subcase $\alpha = \alpha' = a$ (for some $a \in A, \bar{a} = b$): $a = \mathbf{Res}_b(a \cdot 1 \cdot 0)$ and by IH, we have an type isomorphism $\mathbf{QRes}_{b|a \cdot 1 \cdot 0} : T(a \cdot 1 \cdot 0) \rightarrow T(a)$. Since $T(a \cdot 1 \cdot 0) = T(a)$, we can set $\mathbf{QRes}_{b|a} = \mathbf{QRes}_{b|a \cdot 1 \cdot 0}$.
 - Subcase $t(\bar{\alpha}) = \lambda y$: $\alpha \cdot 0 \in A$, $\mathbf{QRes}_b(\alpha \cdot 0) = \alpha' \cdot 0$ and we set $\mathbf{Res}_{b|\alpha} = C(\alpha \cdot 0)(y) \rightarrow \mathbf{QRes}_{b|\alpha \cdot 0}$.
 - Subcase $t(\bar{\alpha}) = @$: $\alpha \cdot 1 \in A$, $\mathbf{QRes}_b(\alpha \cdot 1) = \alpha' \cdot 1$: we set $\mathbf{Res}_{b|\alpha} = \mathbf{Tg}(\mathbf{QRes}_{b|\alpha \cdot 1})$.

□

Remark B.4.

- It is far easier to define the residual of a biposition for a derivation of \mathbf{S} : if P is trivial, whenever $\alpha' := \mathbf{Res}_b(\alpha)$ is defined, the residual biposition of $\mathbf{p} := (\alpha, \gamma) \in \mathbf{bisupp}(P)$ is $\mathbf{Res}_b(\mathbf{p}) = (\alpha', \gamma)$ (Sec. 10.3.5).
- Thus, quasi-residuation is defined for all $(\alpha, c) \in \mathbf{bisupp}(P)$ such that $\bar{\alpha} \neq b \cdot 1$, whereas in Sec. 12.4.1, it was also defined for $\alpha = a \cdot 1$. Actually, we will extend $\mathbf{QRes}_b(\alpha, c)$ in this case (see Remark B.7), but now, it is not useful.

B.1.3 Residual Interface

We notice that if $\alpha \in \mathbf{supp}_{@}(P)$ and $\bar{\alpha} \neq b$, then $\mathbf{Res}_b(\alpha)$ is defined. So, for $\alpha' \in A'_{@}$ (Sec. B.1.1), we set $L'(\alpha') = \mathbf{Sc}(T'(\alpha' \cdot 1))$, $\mathbf{ArgTr}'(\alpha') = \{k \geq 2 \mid \alpha' \in A'\}$ and $R'(\alpha') = (k \cdot T'(\alpha' \cdot k))_{k \in \mathbf{ArgTr}'(\alpha')}$. We write then $\mathbf{Inter}'(\alpha')$ for the set of sequence type isomorphisms from $L'(\alpha')$ to $R'(\alpha')$.

Assume that $\alpha' \in A'_{@}$. Let us write $\alpha = \mathbf{Res}_b^{-1}(\alpha')$, so that $\alpha \in \mathbf{supp}_{@}(P)$, $\alpha \cdot 1 \in A$, $\bar{\alpha} \neq b \cdot 1$, $\mathbf{QRes}_b(\alpha \cdot 1) = \alpha' \cdot 1$ and $\mathbf{child}(\alpha) = \mathbf{child}'(\alpha')$. Thanks to Lemma B.1:

- Since $\mathbf{QRes}_{b|\alpha \cdot 1}$ is a type isomorphism from $T(\alpha \cdot 1)$ to $T'(\alpha' \cdot 1)$ and $L'(\alpha') = \mathbf{Sc}(T'(\alpha' \cdot 1))$, then $T'(\alpha' \cdot 1)$ is an arrow type (since $T(\alpha \cdot 1)$ is) and we define the sequence type isomorphism $\mathbf{ResL}_{b|\alpha}$ by $\mathbf{ResL}_{b|\alpha} = \mathbf{Sc}(\mathbf{Res}_{b|\alpha \cdot 1})$.
- We can define $\mathbf{ResR}_{b|\alpha}$ by $\mathbf{ResR}_{b|\alpha}(k \cdot \gamma) = k \cdot \mathbf{Res}_{b|\alpha \cdot k}(\gamma)$. It is a sequence type isomorphism from $L(\alpha)$ to $L'(\alpha')$.

Thus, for each application node $\alpha \in \text{supp}_{\textcircled{a}}(P)$ such that $\bar{\alpha} \neq b$, the residual $\alpha' = \text{Res}_b(\alpha)$ is defined and we can define a bijection $\text{ResI}_{b|\alpha}$ from $\text{Inter}(\alpha)$ to $\text{Inter}'(\alpha')$ by $\text{ResI}_{b|\alpha}(\phi) = \text{ResR}_{b|\alpha} \circ \phi \circ \text{ResL}_{b|\alpha}^{-1}$, so that the following diagram is commuting:

$$\begin{array}{ccc}
 \text{L}(\alpha) & \xrightarrow{\phi} & \text{R}(\alpha) \\
 \downarrow \text{ResL}_{b|\alpha} & & \downarrow \text{ResR}_{b|\alpha} \\
 \text{L}'(\alpha') & \xrightarrow{\text{ResI}_{b|\alpha}(\phi)} & \text{R}'(\alpha')
 \end{array}$$

It means that the set of interfaces at position $\alpha' = \text{Res}_b(\alpha)$ in P' can also be seen as the residual bijective image of the set of interfaces at position α in P . This observation is pivotal to prove the Representation Lemma in the next subsection.

Assume P is endowed with a complete interface $(\phi_\alpha)_{\alpha \in \text{supp}(P)}$ (i.e. P is an operable derivation). For all $\alpha' \in A'_{\textcircled{a}}$, we set $\phi'_{\alpha'} = \text{ResI}_{b|\alpha}(\phi_\alpha)$, where $\alpha = \text{Res}_b^{-1}(\alpha')$. Notice again that we can retrieve ϕ_α from $\phi'_{\alpha'}$ since $\text{ResI}_{b|\alpha}$ is a bijection. We have enough to ensure:

Proposition B.1. This proposition is also a definition.

- The labelled tree P' defined at end of Sec. B.1.2 is a hybrid derivation.
- When P is endowed with an interface $(\phi_\alpha)_{\alpha \in \text{supp}_{\textcircled{a}}(P)}$, then for all $\alpha' \in A'$, $\phi'_{\alpha'}$ is an interfaces at pos. α' .
- Thus, the family $(\phi'_{\alpha'})_{\alpha' \in \text{supp}_{\textcircled{a}}(P')}$ is a total interface for P' . We call it the **residual interface** of $(\phi_\alpha)_{\alpha \in \text{supp}(P)}$ after firing the redex at position b . When P' is endowed with the residual interface, it is an operable derivation

This entails, as expected:

Corollary B.1 (Pseudo Subject Reduction). If $t \rightarrow t'$ and $\triangleright_{\text{S}_h} C \vdash t : T$, then $\triangleright_{\text{S}_h} C \vdash t' : T'$ for some $T' \equiv T$.

Thus, if needed, we can apply a new β -reduction in t' according to this residual interface without having to define a new one. It allows us to define deterministically the way we perform reduction (inside a derivation) in any reduction sequence of length $\ell \leq \omega$. Now, we can assert that, for instance, $A' = \text{supp}(P')$, $A'_{\textcircled{a}} = \text{supp}_{\textcircled{a}}(P')$, for all $\alpha' \in A'_{\textcircled{a}}$, $\text{L}'(\alpha') = \text{L}^{P'}(\alpha')$, $\text{R}'(\alpha') = \text{R}^{P'}(\alpha')$. To sum up:

Remark B.5.

- We need only a total root interface at position b to define the hybrid derivation P' .
- If we have a total interface at position b , we may define the isomorphisms $\text{Res}_{b|\alpha} : \text{T}(\alpha) \rightarrow \text{T}'(\alpha')$ (resp. $\text{QRes}_{b|\alpha} : \text{T}(\alpha) \rightarrow \text{T}'(\alpha')$) for all α such that $\alpha' := \text{Res}_b(\alpha)$ is defined (resp. such that $\alpha' := \text{QRes}_b(\alpha)$ is defined), as well as $\text{ResI}_{b|\alpha}$ for all $\alpha \in \text{supp}_{\textcircled{a}}(P)$ s.t. $\bar{\alpha} \neq b$.
- It allows us to choose other interfaces (at positions different from b) *after* firing the redex at position b , as suggested in the end of Sec. 13.2.2. This observation is only one we need to prove the Representation Lemma.

B.1.4 Proof of the Representation Lemma

We prove here that every sequence of subject-reduction steps that we perform "by hand" – so called a **reduction choice sequence** – starting from a derivation Π can be built-in inside an operable derivation.

Let $\Pi \triangleright \Gamma \vdash t : \tau$ be a derivation, $P \triangleright C \vdash t : T$ a hybrid derivation collapsing on Π and $t = t_0 \xrightarrow{b_0} t_1 \xrightarrow{b_1} t_2 \xrightarrow{b_2} \dots \xrightarrow{b_{i-1}} t_i \xrightarrow{b_i} \dots$ a sequence of reduction of length $\ell \leq \omega$ (when $\ell = \omega$, we do not need to assume strong convergence [57]).

We write \mathbf{rs} for the sequence $(b_i)_{i < \ell}$ and \mathbf{rs}_n for the sequence $(b_i)_{i < n}$ for all $n < \ell$. If we perform reduction on P along with \mathbf{rs} , we get a sequence of hybrid derivations P_0 (with $P_0 = P$), P_1, P_2, \dots such that P_i concludes with $C \vdash t_i : T_i$ for some $T_i \equiv T$.

More precisely, for each step $i < \ell$ of \mathbf{rs} , we have to choose a root-interface $(\rho_a^i)_{\bar{a}=b_i}$ at position b_i in P_i (typing t_i) corresponding to our reduction choice step, then to reduce P_i w.r.t. $(\rho_a^i)_{\bar{a}=b_i}$, which yields a new hybrid derivation P_{i+1} . We proceed by induction on i .

Those reduction choices are heuristically made step-by-step. This raised the following question (Sec. 13.2.1): is the notion of operable derivation expressive enough? That is: can we endow P with a complete interface, such that performing \mathbf{rs} on P follows exactly our step-by-step choices of substitutions? The answer is positive, as stated in Lemma 13.1. We will prove that now.

We set $A_i = \text{supp}(P_i)$ for all $i < \ell$ and we define by induction on $i < \ell$ a partial function $\mathbf{Res}_{\mathbf{rs}(i)}$ from A to A_i :

- $\mathbf{Res}_{\mathbf{rs}(0)}$ is the identity on A .
- $\mathbf{Res}_{\mathbf{rs}(i+1)} = \mathbf{Res}_{b_i}^{\rho^i} \circ \mathbf{Res}_{\mathbf{rs}(i)}$, where $\mathbf{Res}_{b_i}^{\rho^i}$ is the residual function on $A_i = \text{supp}(P_i)$ defined w.r.t. reduction at position b_i and the root-interface $(\rho_a^i)_{\bar{a}=b_i}$ (see Sec. B.1.1).

For all $i < \ell$, let $A_{\mathbf{rs}(i)}$ denote the domain of $\mathbf{Res}_{\mathbf{rs}(i)}$. Thus, $A_{\mathbf{rs}(0)} = A_0 = \text{supp}(P)$, $(A_{\mathbf{rs}(i)})_{i < \ell}$ is a decreasing sequence (w.r.t. \subseteq) and, by induction on i , $\mathbf{Res}_{\mathbf{rs}(i)}$ is a bijection from $A_{\mathbf{rs}(i)}$ to A_i . We write $\mathbf{Res}_{\mathbf{rs}(i)}^{-1}$ for the converse bijection from A_i to $A_{\mathbf{rs}(i)}$.

To lighten notations, we write $A_{(i)}$ and $\mathbf{Res}_{(i)}$ instead of $A_{\mathbf{rs}(i)}$ and $\mathbf{Res}_{\mathbf{rs}(i)}$. We also set $A_{(i)}^{\textcircled{a}} = A_{(i)} \cap \text{supp}_{\textcircled{a}}(P)$. Thus, $\mathbf{Res}_{(i)}$ induces a bijection from $A_{(i)}$ to A_i .

Now, for all $i < \ell$, we chose an interface (ϕ_a^i) at position b_i in P_i such that $\text{Rt}(\phi_a^i) = \rho_a^i$ for all $a \in A_i$, $\bar{a} = b_i = b_i$.

We define by induction on i a type isomorphism $\mathbf{Res}_{(i)|\alpha}$ from $\mathbf{T}(\alpha)$ to $\mathbf{T}_i(\mathbf{Res}_{(i)}(\alpha))$ for all $\alpha \in A_{(i)}$ and a bijection $\mathbf{ResI}_{(i)|\alpha}$ from $\mathbf{Inter}(\alpha)$ to $\mathbf{Inter}_i(\mathbf{Res}_{(i)}(\alpha))$ for all $\alpha \in A_{(i)|\alpha}$ by:

- $\mathbf{Res}_{(0)|\alpha}$ and $\mathbf{Res}_{(i)|\alpha}$ are respectively the identity functions on $\mathbf{T}(\alpha)$ and $\mathbf{Inter}(\alpha)$.
- $\mathbf{Res}_{(i+1)|\alpha} = \mathbf{Res}_{b_i|\alpha_i} \circ \mathbf{Res}_{(i)|\alpha}$, where $\alpha_i = \mathbf{Res}_{(i)}(\alpha)$ and $\mathbf{Res}_{b_i|\alpha_i} : \mathbf{T}_i(\alpha_i) \rightarrow \mathbf{T}_{i+1}(\alpha_{i+1})$ (with $\alpha_{i+1} = \mathbf{Res}_{(i+1)}(\alpha)$) is the residual type isomorphism (in the sense of Sec. B.1.2) w.r.t. the interface (ϕ_a^i) at position b in P_i .
We set likewise $\mathbf{ResI}_{(i+1)|\alpha} = \mathbf{ResI}_{b_i|\alpha_i} \circ \mathbf{ResI}_{(i)|\alpha}$, where $\mathbf{ResI}_{b_i|\alpha_i}$ is the bijection (in the sense of Sec. B.1.4) w.r.t. the interface (ϕ_a^i) at position b in P_i .

To conclude, let $a \in \text{supp}_{\textcircled{a}}(P)$. There are two cases:

- $\text{Res}_{(i)}(a)$ is defined for all $i < \ell$. In that case, we choose an arbitrary $\phi_a \in \text{Inter}(\alpha)$.
- There is a unique $0 \leq i < \ell$ such that $\alpha_i = \text{Res}_{(i)}(\alpha)$ is defined, but $\text{Res}_{(i+1)}(\alpha)$ is not. In that case, $\bar{\alpha}_i = b_i$ and we have already chosen an interface $\phi_{\alpha_i}^i \in \text{Inter}_i(\alpha_i)$ (that extends $\rho_{\alpha_i}^i$). We set then $\phi_a = \text{ResI}_{(i)|a}^{-1}(\phi_{\alpha_i}^i)$

By construction, the complete interface (ϕ_a) emulates the reduction w.r.t. the family $(\rho_a^i)_{\bar{a}=b_i}$. Thus, Lemma 13.1 is proved:

Lemma. Every reduction choice sequence in a quantitative derivation Π can be built-in in an operable derivation representing Π .

B.2 Isomorphisms and Relabelling of Derivations

In Section B.2, we formally define by induction isomorphisms associated to isomorphisms of hybrid or operable derivations (Sec. B.2.1). We also properly define the image of a derivation by a resetting (Sec. B.2.2). We proceed again by *001-induction* (Remark B.1).

B.2.1 Isomorphisms of Operable Derivations

In this section, we formally define the isomorphisms of types/sequence types canonically induced by an isomorphism of hybrid derivation $\Psi : P_1 \rightarrow P_2$, that are hinted at in Sec. 13.3.1.

- For all $a_1 \in A^1$, we set $\Psi_{\text{tr}}(a_1) = \text{tr}_2(\Psi(a_1))$. Thus, $\Psi_{\text{tr}}(a_1)$ is the axiom track (in P_2) of the axiom rule a_2 corresponding to a_1 *via* Ψ .
- Since Ψ_{supp} induces a bijection from $\text{Ax}_1(a_1)(x)$ to $\text{Ax}_2(a_2)(x)$ for all $a_1 \in A_1$, $a_2 = \Psi(a_1)$ and $x \in \mathcal{V}$, we can define a context isomorphism $\Psi_{a_1,x}$ from $\text{C}_1(a_1)(x)$ to $\text{C}_2(a_2)(x)$ by $\Psi_{a_1,x}(k_1 \cdot \gamma_1) = \Psi_{\text{tr}}(a_{01}) \cdot \Psi_{a_{01}}(\gamma_1)$ for all $\gamma_1 \in \text{supp}(\text{T}_1(a_1))$, where $a_{01} = \text{pos}_1(a_1, x, k_1)$.
- By 001-induction, this allows us to define a type isomorphism Ψ_{a_1} from $\text{T}_1(a_1)$ to $\text{T}_2(a_2)$ for any $a_1 \in A_1$ and $a_2 = \Psi(a_1)$ (not only for $a_1 \in \text{Ax}_1$ and $a_2 \in \text{Ax}_2$).
 - Ψ_{a_1} is already defined when $a_1 \in \text{Ax}_1$.
 - If $t(a_1) = \lambda x$, we set $\Psi_{a_1} = \Psi_{a_1 \cdot 0, x} \rightarrow \Psi_{a_1 \cdot 0}$
 - If $t(a_1) = \textcircled{a}$, we set $\Psi_{a_1} = \text{Tg}(\Psi_{a_1 \cdot 1})$.
- We assume here that $a_1 \in \text{supp}_{\textcircled{a}}(P_1)$ and $a_2 = \Psi(a_1)$.
 - We set $\Psi_{a_1}^L = \text{Sc}(\Psi_{a_1})$.

- Since Ψ_{supp} induces a bijection from $\text{ArgTr}_1(a_1)$ to $\text{ArgTr}_2(a_2)$ (notation ArgTr is defined on p. 294), we define a sequence type isomorphism $\Psi_{a_1}^R$ from $R_1(a_1)$ to $R_2(a_2)$ by $\Psi_{a_1}^R(k_1 \cdot \gamma_1) = k_2 \cdot \gamma_2$, where $a_2 \cdot k_2 = \Psi_{\text{supp}}(a_1 \cdot k_1)$ and $\gamma_2 = \Psi_{a_1 \cdot k_1}(\gamma_1)$.
- We define now Ψ on bisupports (with $a_2 = \Psi_{\text{supp}}(a_1)$).
 - Left case: $\Psi(a_1, x, k_1 \cdot \gamma_2) = (a_2, x, \Psi_{a_1, x}(k_1 \cdot \gamma_1))$
 - Right case: $\Psi(a_1, \gamma_1) = (a_2, \Psi_{a_1}(\gamma_1))$.

Remark B.6. The isomorphisms between \mathbf{S} -derivations defined in Appendix A.5 are isomorphisms of operable derivations.

B.2.2 Resetting an Operable Derivation

Let P be a hybrid derivation and Ψ a resetting of P . We may now define step by step the $\Psi(P)$ mentioned in Proposition 13.1 *i.e.* we define the derivation P_0 such that Ψ actually defines an operable derivation isomorphism from P to P_0 .

- For all $a_0 \in \mathbf{Ax}_0$, we set $T_0(a_0) = \Psi_a(T(a))$, where $a = \Psi_{\text{supp}}^{-1}(a_0)$. Thus, Ψ_a induces a type isomorphism from $T(a)$ to $T_0(a_0)$
- Let $\mathbf{Ax}_0(\Psi(a))(x) = \{\Psi_{\text{supp}}(a_0) \mid a_0 \in \mathbf{Ax}_\alpha(x)\}$ for all $a \in A$ and $x \in \mathcal{V}$. Since Ψ_{supp} induces a bijection from \mathbf{Ax} to \mathbf{Ax}_0 , we can set $\text{tr}_0(\Psi(a)) = \Psi_{\text{tr}}(a)$ for all $a \in \mathbf{Ax}$ and $\mathbf{AxTr}_0(\Psi(a))(x) = \Psi_{\text{tr}}(\mathbf{Ax}_\alpha(x))$. Since Ψ_{tr} is an injection whose domain is \mathbf{Ax} , we can define $\text{pos}_0 : \text{codom}(\Psi_{\text{tr}}) \rightarrow \mathbf{Ax}'$ with $\text{pos}_0(k_0) = a_0$, where $a_0 \in \mathbf{Ax}_0$ is the unique leaf of A_0 such that $\text{tr}_0(a_0) = k_0$ (by the injectivity of Ψ_{tr} discussed above, the function pos_0 requires only one argument, contrary to pos).
- We define then $C_0(\Psi(a))(x) = (\text{tr}_0(\Psi(a_0)) \cdot T(\Psi(a_0)))_{a_0 \in \mathbf{Ax}_\alpha(x)}$. So we can write $\Psi_{a, x}$ for the context isomorphism from $C(a)(x)$ to $C_0(a_0)(x)$ such that $\Psi_{a, x}(k \cdot \gamma) = \Psi_{\text{tr}}(a_0) \cdot \Psi_{a_0}(\gamma)$, where $a_0 = \text{pos}(a, x, k)$.
- We can now define a type $T_0(\Psi(a))$ and a type isomorphism Ψ_a from $T(a)$ to $T_0(\Psi(a))$ for all $a \in A$ by 001-induction.
 - $T(a_0)$ and Ψ_a are already defined when $a \in \mathbf{Ax}$.
 - If $t(a) = \lambda x$, we set $T_0(a_0) = C_0(a_0)(x) \rightarrow T_0(a_0 \cdot 0)$ and $\Psi_a = \Psi_{a \cdot 0, x} \rightarrow \Psi_{a \cdot 0}$.
 - If $t(a) = @$, we set $T_0(a_0) = \text{Tg}(T_0(a_0))$ and $\Psi_a = \text{Tg}(\Psi_{a \cdot 1})$.
- We assume here that $t(a) = @$ and $a_0 = \Psi(a)$. We set then $\text{ArgTr}_0(\Psi(a)) = \{k_0 \geq 2 \mid \exists k \in \text{ArgTr}(a), a_0 \cdot k_0 = \Psi(a \cdot k)\}$, $L_0(a) = \text{Sc}(T_0(a_0 \cdot 1))$ and $R_0(a_0) = (k_0 \cdot T_0(a_0 \cdot k_0))_{k_0 \in \text{ArgTr}_0(a_0)}$.
 - We set $\Psi_a^L = \text{Sc}(\Psi_a)$.

- We define a sequence type isomorphism Ψ_a^R from $R(a)$ to $R_0(a_0)$ by $\Psi_a^R(k \cdot \gamma) = k_0 \cdot \gamma_0$, where $a_0 \cdot k_0 = \Psi(a_1 \cdot k_1)$ and $\gamma_0 = \Psi_{a \cdot k}(\gamma_0)$.

Moreover, if P is endowed with an interface ϕ , we can set $\phi_{0,a_0} = \Psi_a^R \circ \phi_a \circ (\Psi_a^L)^{-1}$. Thus, $\phi_{0,a_0} : L_0(a_0) \rightarrow R_0(a_0)$ is a sequence type isomorphism.

The following proposition, which is the formal counterpart of Proposition 13.1, stems from the previous constructions:

Proposition B.2. With the above notations, let P_0 be the labelled tree such that $\text{supp}(P_0) = A_0$ and $P_0(a_0) = C_0(a_0) \vdash t|_{a_0} : T_0(a_0)$.

Then P_0 is a hybrid derivation and (ϕ_{0,a_0}) is a complete interface that makes P_0 isomorphic to P as an operable derivation *via* the isomorphism Ψ . Thus, we may naturally denote P_0 by $\Psi(P)$.

B.3 Edge Threads, Brotherhood and Consumption

Lemma B.3. Let P be an operable derivation P and Θ a relabelling of P . If, for all $p \in L^P$, $\Theta(p) = \Theta(\phi(p))$, then P^Θ is a trivial derivation.

Proof. Let Ψ the resetting induced by Θ . Let $a_0 \in \text{supp}_\Theta(P^\Theta)$ and $a = \Psi^{-1}(a_0)$. From Sec. B.1.3, we recall that the interface $\phi_{a_0}^\Theta$ of P^Θ is defined by $\phi_{a_0}^\Theta = \Psi_a^R \circ \phi_a \circ (\Psi_a^L)^{-1}$.

Let $k_0 \cdot c_0 \in \text{supp}_{\text{mut}}(L^\Theta(a_0))$. We set $p_0 = (a_0, k_0 \cdot c_0)$ and $p = \Psi^{-1}(p_0)$. Since $\Theta(\theta) = \Theta(\phi(\theta))$, we have $\text{lab}(\phi_{a_0}(k_0 \cdot c_0)) = \text{lab}(k_0 \cdot c_0)$.

By induction on $|k_0 \cdot c_0|$, we show then that, for all $k_0 \cdot c_0 \in \text{supp}(L^\Theta(a_0))$, $\phi_{a_0}(k_0 \cdot c_0) = k_0 \cdot c_0$. □

Lemma B.4.

- If $\theta_L \xrightarrow{\sim} \theta_R$ and θ_L is an axiom thread, then θ_R is an argument thread.
- If $\theta_L^\ominus \xrightarrow{\sim} \theta_R$ and θ_R is an argument thread, then θ_L is an axiom thread.

Proof.

- Assume that $\theta_L : e_L \xrightarrow{a} e_R : \theta_R$ with θ_L axiom thread, $e_1 = (a \cdot 1, k \cdot c)$. Then let $(a_0, c_0) = \text{Asc}(e_L)$. The 3rd point of Remark 13.8 implies $\text{Pol}((a_0, c_0)) = \ominus$ and (a_0, c_0) is the polar inverse of an axiom edge. In particular, c_0 is of the form $k_0 \in \mathbb{N} \setminus \{0, 1\}$. Then, by Lemma 13.5, there is $i \geq 0$ such that $k \cdot c = 1^i \cdot k_0$ or $k \cdot c = 1^i \cdot k_0$. Since $k, k_0 \geq 2$, we have $i = 0$, $k = k_0$ and $c = \varepsilon$. Thus, by definition of \rightarrow , we have $e_2 = a \cdot k'$ with $k' = \phi_a(k)$.
- Assume that $\theta_L : e_L^\ominus \xrightarrow{a} e_R : \theta_R$ with e_1 negative and $e_2 = a \cdot k' \in \text{supp}_{\text{mut}}(P)$. Thus, there is $k \geq 2$ and $c \in \mathbb{N}^*$ such that $e_1 = (a \cdot 1, k \cdot c)$. Let $e_0 := \text{Asc}(e_1)$. Since e_L is negative, then $e_0 = (a_0, k_0 \cdot c_0)$ for some a_0, k_0, c_0 with $t(a_0) = \lambda x$ (for some x), $k_0 \geq 2$ and $c_0 \in \mathbb{N}^*$, so that $e_0 \rightarrow_{\text{pi}} (a_*, c_0) =: e_*$ for some $a_* \in \text{Ax} \cdot P$. By Lemma 13.5, there is $i \geq 0$ such that $k_0 \cdot c_0 = 1^i \cdot k$ or $k_0 \cdot c_0 = 1^i \cdot k$. Since $k, k_0 \geq 2$, we have $i = 0$, $k = k_0$ and $c_0 = \varepsilon$. In particular, $e_* = (a_*, \varepsilon)$ *i.e.* e_* is an axiom referent and θ_L is an axiom thread. □

B.4 Residuation for Mutable Edges and Threads

In Sec. B.4, we detail the constructions suggested in Sec. 13.5. In particular, we define residuation for edges and edges threads.

B.4.1 Edges and Residuation

Let us observe now that a distinction should be made between residuation for nodes and residuation for edges. Consider an operable derivation $P \triangleright C \vdash t : T$ (coming along with the usual notations) and assume that $t|_b = (\lambda x.r)s$, $t \xrightarrow{b} t'$ (so that $t'|_b = r[s/x]$), $P \xrightarrow{b} P'$ so that P' is a residual operable derivation of P . We use the same metavariable conventions as in Sec. B.1.1 *i.e.* metavariable a will now denote only positions in \mathbb{A} s.t. $\bar{a} = b$. Metavariables α and c range over \mathbb{N}^* . For instance, $\alpha \neq a$ means that $\bar{\alpha} \neq b$. Abusively, when $\bar{a} = b$, then a is an **app**-rule typing the root of the redex and $(S_k)_{k \in K}$ (resp. $(S'_k)_{k \in K'}$) denotes the left key (resp. the right key) of the **app**-rule (note that $(S_k)_{k \in K}$ and $(S'_k)_{k \in K'}$ depend on P and a), *cf.* p. 294.

If $a \in \mathbb{N}^*$ and $k' \in \mathbb{N} \setminus \{0, 1\}$ are such that $\bar{a} = b$ and $a \cdot k' \in \text{supp}(P)$, then $P(a)$ is a judgment of the form $D_{k'} \vdash s : S_{k'}$, more precisely, $a \cdot k'$ points to a *node* of P labelled with a judgment typing the argument s of the reduced redex. After reduction, this judgment will be located at position $a \cdot a_k$ where $k = \rho_a^{-1}(k')$. Thus, it was natural to define $\text{Res}_b(a \cdot k')$ as $a \cdot a_k$ in Sec. B.1.1 (paradigm \clubsuit). This point of view that used throughout Sec. 13.2 to define residuation for derivation.

But, as seen in Sec. 13.3.3, we also use $a \cdot k$ to denote the *edge* of P from a to $a \cdot k$. This is an edge that joins the **app**-rule of the redex to an argument derivation of the redex. When we reduce the redex at position b , this edge is destroyed and should *not* have a residual. This leads us to make a distinction between residuation for nodes (as in Sec. 13.2) and residuation for edges. Quasi-residuation for mutable edges should be then defined properly. For that, we proceed as in Sec. 12.4.1 except that residuation must take interfaces into account in system \mathbf{S}_{op} .

If $k \in \text{Tr}_\lambda(a)$, a_k is the unique position s.t. $\text{pos}(a \cdot 10, x, k) = a \cdot 10 \cdot a_k$ (see a_2 and a_7 in Fig. 13.2).

We define now the quasi-residuation QRes_b^E on $E(P)$ (note that QRes_b^E depends on P and in particular, on the interface on P , but that the notation does not mention them).

- *Argument edges:* Let $\alpha \in \text{supp}_{\text{mut}}(P)$.
 - If $\bar{\alpha} = b \cdot 2$ (*i.e.* α is an argument edge of the redex), then α is destroyed, so that $\text{QRes}_b^E(\alpha)$ is left undefined
 - If $\bar{\alpha} \neq b \cdot 2$, then α is moved during reduction. We set $\text{QRes}_b^E(\alpha) = \text{Res}_b(\alpha)$.
- *Axiom edges:* Let $(\alpha, \varepsilon) \in \mathbf{Ax}^P$.
 - If $t(\alpha) = x$ (assuming Barendregt convention), then (α, ε) corresponds to an axiom root of a type assigned to the variable of the redex. This axiom root is destroyed during reduction, so that $\text{QRes}_b^E(\alpha)$
 - If $t(\alpha) = y \neq x$, then (α, ε) is moved during reduction and we set $\text{QRes}_b^E(\alpha, \varepsilon) = (\text{Res}_b(\alpha), \varepsilon)$.
- *Inner Edges:* Let $(\alpha, c) \in \text{bisupp}_{\text{mut}}(P)$ with $\alpha \in \mathbf{Ax}^P$.
 - If $\bar{\alpha} \neq b \cdot 1$, then we set $\text{QRes}_b^E(\alpha, c) = \text{QRes}_b(\alpha, c)$ (which is defined).

- If $\alpha = a \cdot 1$ with $\bar{a} = b$. Then $\mathsf{T}(a \cdot 1) = (S_k)_{k \in K} \rightarrow T$ where $T = \mathsf{T}(a)$ and $(S_k)_{k \in K}$ is the left key of the **app**-rule at position a . Let $(S'_k)_{k \in K'}$ the right key of this **app**-rule.
 - * If $c = 1 \cdot c_0$, then c points inside T . Since $(a, c_0) \rightarrow_{\text{asc}} (a \cdot 1, c)$, we set $\mathsf{QRes}_b^E(a \cdot 1, c) = \mathsf{QRes}_b(a, c_0) = (a, c_0)$.
 - * If $c = k$ with $k \geq 2$, then, since $(a \cdot 1, k)$ points to an axiom tracks of the variable x of the redex, which is destroyed. So we leave $\mathsf{QRes}_b^E(a \cdot 1, k)$ undefined.
 - * If $c = k \cdot c_0$ with $k \geq 2$ and $c_0 \neq \varepsilon$, then c points inside $(S_k)_{k \in K}$. Since $(a \cdot 1, k \cdot c_0) \xrightarrow{\alpha} (a \cdot k', c'_0)$ with $k' \cdot c_0 = \phi_a(k \cdot c_0)$, we set $\mathsf{QRes}_b^E(a \cdot 1, c) = \mathsf{Res}_b(a \cdot k', c'_0) = (a \cdot a_k, c'_0)$.

Remark B.7.

- Note again that QRes_b^E and QRes_b are not always equal *e.g.*, if $\bar{a} = b$ and $a \cdot k \in \text{supp}_{\text{mut}}(P)$, then $\mathsf{QRes}_b(a \cdot k') = \mathsf{Res}_b(a \cdot k) = a \cdot a_k$ (with $\rho_a(k) = k'$), but $\mathsf{QRes}_b^E(a \cdot k)$ is not defined. This is because $a \cdot k$ denotes the *edge* from a to $a \cdot k$ in $\mathsf{QRes}_b^E(a \cdot k)$ and not the *node* at position $a \cdot k$, as in $\mathsf{QRes}_b(a \cdot k)$.
- The last case can also be used (with minor adaptation) to extend quasi-residuation for all right bipositions that are not of the form $(a \cdot 1, \varepsilon)$.

Remark B.8 (Edges without Quasi-Residuals). Note that $\mathsf{QRes}_b^E(\mathbf{e})$ is undefined only in the 3 followings cases: (1) \mathbf{e} is an axiom root associated to the variable of the redex (*i.e.* $\mathbf{e} = \text{pos}(a \cdot 0, x, k)$) (2) \mathbf{e} in the root of the source of the abstraction of the redex (*i.e.* $\mathbf{e} = (a \cdot 1, k)$) (3) \mathbf{e} is an argument edge of the redex *i.e.* $\mathbf{e} = a \cdot k$ (with $k \geq 2$). By definition, we have $\mathbf{e} \rightarrow_{\text{pi}} \mathbf{e}'$ and $\mathbf{e} \xrightarrow{\alpha} \mathbf{e}''$ with $\mathbf{e} = (a \cdot 1, k)$, $\mathbf{e}' = \text{pos}(a \cdot 0, x, k)$ and $\mathbf{e}'' = a \cdot k$. Moreover, observe that \mathbf{e}'' does not have ascendants or descendants and that \mathbf{e} and \mathbf{e}_2 do not have descendant *i.e.* $\text{thr}(\mathbf{e}) = \text{thr}(\mathbf{e}') = \{\mathbf{e}, \mathbf{e}'\}$. This justifies the first equivalence in Lemma 13.12 and that of Lemma 13.13 below.

B.4.2 Residuals of Edges Threads

In this section, we define residuation for edge threads and give some of its properties.

Relations \rightarrow_{asc} and \rightarrow_{pi} and thus \equiv are compatible with reduction. By case analysis (still guided by Fig. 13.2):

- Assume $\mathbf{e}_1 = (\alpha, c) \rightarrow_{\text{asc}} \mathbf{e}_2$. If $\alpha \neq a, a \cdot 1$, then $\mathsf{QRes}_b^E(\mathbf{e}_1) \rightarrow_{\text{asc}} \mathsf{QRes}_b^E(\mathbf{e}_2)$. If $\alpha = a, a \cdot 1$, then $\mathsf{QRes}_b(\mathbf{e}_1) = \mathsf{QRes}_b(\mathbf{e}_2)$.
- If $\mathbf{e}_1 = (\alpha, k \cdot c) \rightarrow_{\text{pi}} \mathbf{e}_2$. If $\alpha \neq a \cdot 1$, then $\mathsf{QRes}_b^E(\mathbf{e}_1) \rightarrow_{\text{pi}} \mathsf{Res}_b(\mathbf{e}_2)$. If $\alpha = a \cdot 1$ and $c = k \geq 2$, then nor $\mathsf{QRes}_b(\mathbf{e}_1)$ or $\mathsf{QRes}_b(\mathbf{e}_2)$ are defined. If $\alpha = a \cdot 1$ and $c \neq k \in \mathbb{N}$, then \mathbf{e}_1 would not have $\mathsf{QRes}_b(\mathbf{e}_1) = \mathsf{QRes}_b(\mathbf{e}_2)$.

This entails, by induction on \equiv :

Lemma B.5. If $\mathbf{e}_1 \equiv \mathbf{e}_2$, then $\mathsf{QRes}_b^E(\mathbf{e}_1)$ is defined iff $\mathsf{QRes}_b^E(\mathbf{e}_2)$ is. In that case, $\mathsf{QRes}_b^E(\mathbf{e}_1) = \mathsf{QRes}_b^E(\mathbf{e}_2)$.

Lemma 13.12 allows us to define (**quasi**)-residuals for *edge threads*. We set $\mathsf{Res}_b(\theta) = \text{thr}'(\mathsf{QRes}_b(\mathbf{e}))$ for any $\mathbf{e} : \theta$ (where $\text{thr}'(\cdot)$ denotes threads in P').

Assume that $\mathbf{e}_1 \rightarrow \mathbf{e}_2$. Then, by case analysis, $\mathbf{QRes}_b^E(\mathbf{e}_1)$ is defined iff $\mathbf{QRes}_b^E(\mathbf{e}_2)$. In that case, either $\mathbf{QRes}_b^E(\mathbf{e}_1) \rightarrow \mathbf{QRes}_b^E(\mathbf{e}_2)$ (\rightarrow is taken w.r.t. P' , the reduced derivation) or $\mathbf{QRes}_b^E(\mathbf{e}_1) = \mathbf{QRes}_b^E(\mathbf{e}_2)$. This entails, since :

Lemma B.6. Let P be an operable derivation whose interface is ϕ . Assume that $\theta_1 \rightarrow \theta_2$.

- Then $\mathbf{Res}_b(\theta_1)$ is defined iff $\mathbf{Res}_b(\theta_2)$ is.
- In that case, $\mathbf{Res}_b(\theta_1) \xrightarrow{\sim} \mathbf{Res}_b(\theta_2)$ or $\mathbf{Res}_b(\theta_1) = \mathbf{Res}_b(\theta_2)$.
- Moreover, $\mathbf{Res}_b(\theta_1) = \mathbf{Res}_b(\theta_2)$ iff $\theta_1 : \mathbf{e}_1 \xrightarrow{a} \mathbf{e}_2 : \theta_2$ for some $\mathbf{e}_1 = (a \cdot 1, k \cdot c)$, $\mathbf{e}_2 = (a \cdot k', c')$ with $a \in \mathbf{supp}_@ (P)$, $\bar{a} = b$ and $c \neq \varepsilon$. In particular, θ_1 is not an axiom thread.

Proof. The first part is justified by case analysis. The second part of the claim stems from the same case analysis: we notice that $\mathbf{QRes}_b^E(\mathbf{e}_1) = \mathbf{QRes}_b^E(\mathbf{e}_2)$ iff there are $a \in \mathbf{supp}_@ (P)$, $k \geq 2$, $c \in \mathbb{N}^*$ such that $\bar{a} = b$, $c \neq \varepsilon$, ($\mathbf{e}_1 = (a \cdot 1, k \cdot c)$ or $\mathbf{e}_1 = (a \cdot 10 \cdot a_k, c)$) and $\mathbf{e}_2 = (a \cdot k', c')$ with $\phi_a(k \cdot c) = k' \cdot c'$. We then observe that $\mathbf{thr}((a \cdot 1, k \cdot c)) = \mathbf{thr}((a \cdot 10 \cdot a_k, c)) = \{(a \cdot 1, k \cdot c), (a \cdot 10 \cdot a_k, c)\}$ \square

Likewise, (strict) brotherhood is compatible with residuation: a case analysis shows that if \mathbf{e}_1 and \mathbf{e}_2 are (strict) brother edges, then $\mathbf{QRes}_b(\mathbf{e}_1)$ is defined iff $\mathbf{QRes}_b(\mathbf{e}_2)$ is, and that in that case, $\mathbf{QRes}_b(\mathbf{e}_1)$ and $\mathbf{QRes}_b(\mathbf{e}_2)$ also are brother edges. This entails:

Lemma B.7. Let P be a hybrid derivation and θ_1, θ_2 be two strict brother threads. Then $\mathbf{Res}_b(\theta_1)$ is defined iff $\mathbf{Res}_b(\theta_2)$ is.

In that case, $\mathbf{Res}_b(\theta_1) = \mathbf{Res}_b(\theta_2)$.