Types as Resources for Classical Natural Deduction

Delia Kesner and Pierre Vial

IRIF (CNRS and Université Paris-Diderot), France

Abstract

We define two resource aware typing systems for the λ_μ-calculus based on non-idempotent *intersection* and *union* types. The non-idempotent approach provides very simple combinatorial arguments –based on decreasing measures of type derivations– to characterize head and strongly normalizing terms. Moreover, typability provides upper bounds for the length of head-reduction sequences and maximal reduction sequences.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases lambda-mu-calculus, classical logic, intersection types, normalization

Digital Object Identifier 10.4230/LIPIcs.FSCD.2017.32

1 Introduction

A few years after Griffin [22] observed that Feilleisen's C operator can be typed with the double-negation elimination, Parigot [32] made a major step in extending the Curry-Howard from intuitionistic to classical logic by proposing the λ_μ-calculus as a simple term notation for classical natural deduction proofs. Other calculi were proposed since then, as for example Curien-Herbelin's $\lambda\mu\tilde{\mu}$-calculus [11] based on classical sequent calculus.

Simple types are known to be unable to type some normalizing term, for instance the normal form $\Delta = \lambda x.xx$. *Intersection* types, pioneered by Coppo and Dezani [9, 10], extend simple types by resorting to a new constructor $\cap$ for types, allowing the assignment of a type of the form $((\sigma \Rightarrow \sigma) \cap \sigma) \Rightarrow \sigma$ to the term Δ. The intuition behind a term t of type $\tau_1 \cap \tau_2$ is that t has both types τ_1 and τ_2. The intersection operator $\cap$ is to be understood as *idempotent* ($\sigma \cap \sigma = \sigma$), *commutative* ($\sigma \cap \tau = \tau \cap \sigma$), and *associative* ($(\sigma \cap \tau) \cap \delta = \sigma \cap (\tau \cap \delta)$) laws. Among other applications, intersection types have been used as a *behavioural* tool to reason about several operational and semantical properties of programming languages. 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.

This technology turns out to be a powerful tool to reason about *qualitative* properties of programs, but not about *quantitative* ones. Indeed, *e.g.* there is a type system assigning a type to a term t if and only if t is head normalizing, but the type derivations give no information about the number of head-reduction steps needed to head-normalize t, because of idempotency. In constrast, after the pioneering works of Gardner [19] and Kfoury [27], D. de Carvalho [14, 15] established 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. Non-idempotent types have recently received a lot of attention in the domain of semantics of programming languages from a quantitative perspective (see for example [6]), notably because they are closely related to Girard's translation of intuitionistic logic into linear logic (according to which $A \Rightarrow B$ becomes $!A \multimap B$).

The case of the λ_μ-calculus: The non-idempotent intersection and union types for lambda-mu-calculus that we present in this article can be seen as a quantitative refinement of Girard's translation of classical logic into linear logic. Different qualitative and/or quantitative models for classical calculi were proposed in [34, 37, 39, 3], thus limiting the characterization of operational properties to head-normalization. Intersection and union types were also studied in the framework of classical logic [30, 36, 28, 17], but no work adresses the problem from a quantitative perspective. Type-theoretical characterization of strong-normalization for classical calculi were provided both for λ_μ [38] and $\lambda\mu\tilde{\mu}$-calculus [17], but the (idempotent) typing systems do not allow to construct decreasing measures for reduction, thus a resource aware semantics cannot be extracted from those interpretations. Combinatorial strong normalization proofs for the λ_μ-calculus were proposed for example in [12], but they do not provide any explicit decreasing measure, and their use of structural induction on simple types does not work anymore with intersection types, which are more powerful than simple types as they do not only ensure termination but also characterize it. Different small step semantics for classical calculi were developed in the framework of neededness [4, 33], without resorting to any resource aware semantical argument.

In this paper we define a resource aware type system for the λ_μ-calculus based on non-idempotent *intersection* and *union* types. The non-idempotent approach provides very simple combinatorial arguments, only based on a decreasing *measure*, to characterize head and strongly normalizing terms by means of typability. In the well-known case of the λ-calculus, the measure $\mathtt{sz}(\Pi)$ of a derivation Π is simply given by the number of its nodes. This approach cannot be straightforwardly adapted to λ_μ, and we need now to take into account the structure (*multiplicity* and *size*) of certain types appearing in the types derivations.

By lack of space we cannot provide in this submission all the proofs of our results, but we refer the interested reader to the extended detailed version available at [26].

2 The λ_μ-Calculus

This section gives the syntax (Sec. 2.1) and the operational semantics (Sec. 2.2) of the $\lambda_\mu-$calculus [32]. But before this we first introduce some preliminary general notions of rewriting that will be used all along the paper, and that are applicable to any system $\mathcal{R}$. We denote by $\to_\mathcal{R}$ the (one-step) reduction relation associated to system $\mathcal{R}$. We write $\to_\mathcal{R}^*$ for the reflexive-transitive closure of $\to_\mathcal{R}$, and $\to_\mathcal{R}^n$ for the composition of n-steps of $\to_\mathcal{R}$, thus $t \to_\mathcal{R}^n u$ denotes a finite $\mathcal{R}$-reduction sequence of length n from t to u. A term t is in $\mathcal{R}$-normal form, written $t \in \mathcal{R}$-nf, if there is no t' s.t. $t \to_\mathcal{R} t'$; and t has an $\mathcal{R}$-normal form iff there is $t' \in \mathcal{R}$-nf such that $t \to_\mathcal{R}^* t'$. A term t is said to be strongly $\mathcal{R}$-normalizing, written $t \in \mathcal{SN}(\mathcal{R})$, iff there is no infinite $\mathcal{R}$-sequence starting at t.

2.1 Syntax

We consider a countable infinite set of **variables** $x, y, z, \ldots$ (resp. **continuation names** $\alpha, \beta, \gamma, \ldots$). The set of **objects** ($\mathcal{O}_{\lambda_\mu}$), **terms** ($\mathcal{T}_{\lambda_\mu}$) and **commands** ($\mathcal{C}_{\lambda_\mu}$) of the λ_μ-calculus are given by the following grammars

$$\begin{array}{lllll} (\textbf{objects}) & o & ::= & t \mid c \\ (\textbf{terms}) & t, u, v & ::= & x \mid \lambda x.t \mid tu \mid \mu\alpha.c \\ (\textbf{commands}) & c & ::= & [\alpha]t \end{array}$$

We write $\mathcal{T}_\lambda$ for the set of λ-terms. We abbreviate $(\ldots((tu_1)u_2)\ldots u_n)$ as $tu_1 \ldots u_n$ or $t\overline{u}$ when n is clear from the context. The grammar extends λ-terms with two new constructors:

commands $[\alpha]t$ and μ-abstractions $\mu\alpha.c$. **Free** and **bound variables** of objects are defined as expected, in particular $\mathtt{fv}(\mu\alpha.c) := \mathtt{fv}(c)$ and $\mathtt{fv}([\alpha]t) := \mathtt{fv}(t)$. **Free names** of objects are defined as expected, in particular $\mathtt{fn}(\mu\alpha.c) := \mathtt{fn}(c) \setminus \{\alpha\}$ and $\mathtt{fn}([\alpha]t) := \mathtt{fn}(t) \cup \{\alpha\}$. **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 (finite) functions from variables to terms specified by $\{x_1/u_1, \dots, x_n/u_n\}$ $(n \geq 0)$. **Application** of the **substitution** σ to the object o, written $o\sigma$, may require α-conversion in order to avoid capture of free variables/names, and it is defined as expected. **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 $\{\alpha /\!\!/ u\}$ passes the term u as an argument to any command of the form $[\alpha]t$. Formally, the **application** of the **replacement** Σ to the object o, written $o\Sigma$, may require α-conversion in order to avoid the capture of free variables/names, and is defined as:

$$\begin{array}{rlrl} x\{\alpha /\!\!/ u\} := & x & (\lambda z.t)\{\alpha /\!\!/ u\} := & \lambda z.t\{\alpha /\!\!/ u\} \\ ([\alpha]t)\{\alpha /\!\!/ u\} := & [\alpha](t\{\alpha /\!\!/ u\})u & (tv)\{\alpha /\!\!/ u\} := & t\{\alpha /\!\!/ u\}v\{\alpha /\!\!/ u\} \\ ([\gamma]t)\{\alpha /\!\!/ u\} := & [\gamma]t\{\alpha /\!\!/ u\} & (\mu\gamma.c)\{\alpha /\!\!/ u\} := & \mu\gamma.c\{\alpha /\!\!/ u\} \end{array}$$

For example, if $\mathtt{I} = \lambda z.z$, then $(x(\mu\alpha[\alpha]y)(\lambda z.zx))\{x/\mathtt{I}\} = \mathtt{I}(\mu\alpha[\alpha]y)(\lambda z.z\mathtt{I})$, and $[\alpha]x(\mu\beta.[\alpha]y)\{\alpha /\!\!/ \mathtt{I}\} = [\alpha](x\mu\beta.[\alpha]y\mathtt{I}))\mathtt{I}$.

2.2 Operational Semantics

The $\lambda\mu$-calculus is given by the set of objects introduced in Sec. 2.1 and the **reduction relation** $\to_{\lambda_\mu}$, which is the closure by all contexts of the following rewriting rules

$$\begin{array}{lll} (\lambda x.t)u & \mapsto_\beta & t\{x/u\} \\ (\mu\alpha.c)u & \mapsto_\mu & \mu\alpha.c\{\alpha /\!\!/ u\} \end{array}$$

defined by means of the substitution and replacement application notions given in Sec. 2.1. A **redex** is a term of the form $(\lambda x.t)u$ or $(\mu\alpha.c)u$. We write $t \to_{\lambda_\mu} t'$ (or simply $t \to t'$) to denote the closure by all contexts of the reduction relation generated by the previous set of rewriting rules.

A **head-context** is a context defined by the following grammar:

$$\begin{array}{lll} \mathcal{HO} & ::= & \mathcal{HT} \mid \mathcal{HC} \\ \mathcal{HT} & ::= & \Box t_1 \dots t_n \ (n \geqslant 0) \mid \lambda x.\mathcal{HT} \mid \mu\alpha.\mathcal{HC} \\ \mathcal{HC} & ::= & [\alpha]\mathcal{HT} \end{array}$$

A **head-normal form** is an object of the form $\mathcal{HO}[x]$, where x is any variable replacing the constant $\Box$. 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 $o \in \mathcal{HN}(\lambda_\mu)$, if $o \to^*_{\lambda_\mu} o'$, for some head-normal form o'. Remark that $o \in \mathcal{HN}(\lambda_\mu)$ does not imply $o \in \mathcal{SN}(\lambda_\mu)$ while the converse necessarily holds. We write $\mathcal{HN}(\lambda)$ and $\mathcal{SN}(\lambda)$ when t is restricted to be a λ-term and the reduction system is restricted to the β-reduction rule.

A redex r in a term of the form $t := \mathcal{HO}[r]$ is called the **head-redex** of t. The reduction step $t \to_{\lambda_\mu} t'$ contracting the head-redex of t 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 $o \in \mathcal{HN}(\lambda_\mu)$, while the converse will be stated later (*cf.* Thm. 7).

A typical example of expressivity in the λ_μ-calculus is the control operator [22] **call-cc** $:= \lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x)$ which gives raise to the following reduction sequence:

$$\textbf{call-cc}\ t\ u_1\ \dots\ u_n \to_\beta (\mu\alpha.[\alpha]t(\lambda x.\mu\beta.[\alpha]x))u_1 \dots u_n$$
$$\to_\mu (\mu\alpha.[\alpha]t(\lambda x.\mu\beta.[\alpha]xu_1)u_1)u_2 \dots u_n \to_\mu^* \mu\alpha.[\alpha]t(\lambda x.\mu\beta.[\alpha]xu_1 \dots u_n)u_1 \dots u_n$$

A reduction step $o \to o'$ is said to be **erasing** iff $o = (\lambda x.u)v$ and $x \notin \mathtt{fv}(u)$, or $o = (\mu\alpha.c)u$ and $\alpha \notin \mathtt{fn}(c)$. Thus *e.g.* $(\lambda x.z)y \to z$ and $(\mu\alpha.[\beta]x)\mathtt{I} \to_\mu \mu\alpha.[\beta]x$ are erasing steps. A reduction step $o \to o'$ which is not erasing is called **non-erasing**. Reduction is stable by substitution and replacement. More precisely, if $o \to o'$, then $o\{x/u\} \to o'\{x/u\}$ and $o\{\alpha /\!\!/ u\} \to o'\{\alpha /\!\!/ u\}$. These stability properties give the following corollary.

▶ **Corollary 1.** *If* $o\{x/u\} \in \mathcal{SN}(\lambda_\mu)$ *(resp.* $o\{\alpha /\!\!/ u\} \in \mathcal{SN}(\lambda_\mu)$*) , then* $o \in \mathcal{SN}(\lambda_\mu)$*.*

3 Quantitative Type Systems for the λ-Calculus

As mentioned before, our results rely on typability of λ_μ-terms in suitable systems with non-idempotent types. Since the λ_μ-calculus embeds the λ-calculus, we start by recalling the well-known [19, 14, 7] quantitative type systems for λ-calculus, called here $\mathcal{H}_\lambda$ and $\mathcal{S}_\lambda$. We then reformulate them, using a different syntactical formulation, resulting in the typing systems $\mathcal{H}'_\lambda$ and $\mathcal{S}'_\lambda$, that are the formalisms we adopt in Sec. 4 for λ_μ.

We start by fixing a countable set of **base types** $a, b, c \dots$, then we introduce two different categories of types specified by the following grammars:

$$\begin{array}{lllll} (\textbf{Intersection Types}) & \mathcal{I} & ::= & [\sigma_k]_{k\in K} \\ (\textbf{Types}) & \sigma,\tau & ::= & a \mid \mathcal{I} \Rightarrow \sigma \end{array}$$

An intersection type $[\sigma_k]_{k\in\{1..n\}}$ is a *multiset* that can be understood as a type $\sigma_1 \cap \dots \cap \sigma_n$, where $\cap$ is associative and commutative, but *non-idempotent*. The *non-deterministic* **choice** operation $\_^*$ is defined on intersection types as follows:

$$[\sigma_k]^*_{k\in K} := \begin{cases} [\tau] & \text{if } K = \emptyset \text{ and } \tau \text{ is any arbitrary type} \\ [\sigma_k]_{k\in K} & \text{if } K \neq \emptyset \end{cases}$$

Variable assignments (Γ) are functions from variables to intersection types. The **domain of** Γ is given by $\mathtt{dom}(\Gamma) := \{x \mid \Gamma(x) \neq [\,]\}$, where $[\,]$ is the empty intersection type. We write $x_1 : \mathcal{I}_1, \dots, x_n : \mathcal{I}_n$ for the assignment of domain $\{x_1, \dots, x_n\}$ mapping each x_i to $\mathcal{I}_i$. When $x \notin \mathtt{dom}(\Gamma)$, then $\Gamma(x)$ stands for $[\,]$. We write $\Gamma \wedge \Gamma'$ for $x \mapsto \Gamma(x) + \Gamma'(x)$, where $+$ is multiset union, and $\mathtt{dom}(\Gamma \wedge \Gamma') = \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Gamma')$. We write $\Gamma \setminus\!\!\setminus x$ for the assignment defined by $(\Gamma \setminus\!\!\setminus x)(x) = [\,]$ and $(\Gamma \setminus\!\!\setminus x)(y) = \Gamma(y)$ if $y \neq x$.

To present/discuss different typing systems, we consider the following derivability notions. A **type judgment** is a triple $\Gamma \vdash t : \sigma$, where Γ is a variable assignment, t a term and σ a type. A **(type) derivation** in system $\mathcal{X}$ is a tree obtained by applying the (inductive) rules of the type system $\mathcal{X}$. We write $\Phi \triangleright_\mathcal{X} \Gamma \vdash t : \sigma$ if Φ is a type derivation concluding with the type judgment $\Gamma \vdash t : \sigma$, and just $\triangleright_\mathcal{X} \Gamma \vdash t : \sigma$ if there exists Φ such that $\Phi \triangleright_\mathcal{X} \Gamma \vdash t : \sigma$. A term t is **$\mathcal{X}$-typable** iff there is a derivation in $\mathcal{X}$ typing t, *i.e.* if there is Φ such that $\Phi \triangleright_\mathcal{X} \Gamma \vdash t : \sigma$. We may omit the index $\mathcal{X}$ if the name of the system is clear from the context.

3.1 Characterizing Head β-Normalizing λ-Terms

We discuss in this section typing systems being able to characterize head β-normalizing λ-terms. We first consider system $\mathcal{H}_\lambda$ in Fig. 1, first appearing in [19], then in [14].

Notice that $K = \emptyset$ in rule $(\Rightarrow_\mathtt{e})$ allows to type an application tu without necessarily typing the subterm u. Thus for example, if $\Omega = (\lambda x.xx)(\lambda x.xx)$, then from the judgment $x : [\sigma] \vdash x : \sigma$ we can derive $x : [\sigma] \vdash (\lambda y.x)\Omega : \sigma$.

System $\mathcal{H}_\lambda$ characterizes head β-normalization:

$$\frac{}{x : [\tau] \vdash x : \tau}(\mathtt{ax}) \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \setminus\!\setminus x \vdash \lambda x.t : \Gamma(x) \Rightarrow \tau}(\Rightarrow_{\mathtt{i}}) \qquad \frac{\Gamma \vdash t : [\sigma_k]_{k\in K} \Rightarrow \tau \quad (\Gamma_k \vdash u : \sigma_k)_{k\in K}}{\Gamma \wedge_{k\in K} \Gamma_k \vdash tu : \tau}(\Rightarrow_{\mathtt{e}})$$

Figure 1 System $\mathcal{H}_\lambda$

Rule $(\mathtt{ax})$ Rule $(\Rightarrow_{\mathtt{i}})$ $\dfrac{(\Gamma_k \vdash t : \sigma_k)_{k\in K}}{\wedge_{k\in K}\Gamma_k \Vdash t : [\sigma_k]_{k\in K}}(\wedge)$ $\dfrac{\Gamma \vdash t : \mathcal{I} \Rightarrow \sigma \quad \Gamma' \Vdash u : \mathcal{I}}{\Gamma \wedge \Gamma' \vdash t\, u : \sigma}(\Rightarrow_{\mathtt{e}})$

Figure 2 System $\mathcal{H}'_\lambda$

▶ **Lemma 2.** *Let $t \in \mathcal{T}_\lambda$. Then t is $\mathcal{H}_\lambda$-typable iff $t \in \mathcal{HN}(\lambda)$.*

Moreover, the implication *typability implies normalization* can be shown by simple arithmetical arguments provided by the *quantitative* flavour of the typing system $\mathcal{H}_\lambda$, in contrast to classical reducibility arguments usually invoked in other cases [20, 29]. Actually, the arithmetical arguments give the following quantitative property:

▶ **Lemma 3.** *If t is $\mathcal{H}_\lambda$-typable with tree derivation Π, then the size (number of nodes) of Π gives an upper bound to the length of the head-reduction strategy starting at t.*

To reformulate system $\mathcal{H}_\lambda$ in a different way, 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* to terms.

An equivalent formulation of system $\mathcal{H}_\lambda$, called $\mathcal{H}'_\lambda$, is given in Fig. 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. Notice also that systems $\mathcal{H}_\lambda$ and $\mathcal{H}'_\lambda$ are *relevant*, *i.e.* they lack weakening. Equivalence between $\mathcal{H}_\lambda$ and $\mathcal{H}'_\lambda$ gives the following result:

▶ **Corollary 4.** *Let $t \in \mathcal{T}_\lambda$. Then t is $\mathcal{H}'_\lambda$-typable iff $t \in \mathcal{HN}(\lambda)$.*

Auxiliary judgments turn out to substantially lighten the notations and to make the statements (and their proofs) more readable.

3.2 Characterizing Strong β-Normalizing λ-Terms

We now discuss typing systems being able to characterize strong β-normalizing λ-terms. We first consider system $\mathcal{S}_\lambda$ in Fig. 3, which appears in [8] (slight variants appear in [13, 6, 24]). Rule $(\Rightarrow_{\mathtt{e}_1})$ forces the *erasable arguments* (the subterm u) to be typed, even if the type of u (*i.e.* σ) is not being used in the conclusion of the judgment. Thus, in contrast to system $\mathcal{H}_\lambda$, every subterm of a typed term is now typed. System $\mathcal{S}_\lambda$ characterizes strong β-normalization:

▶ **Lemma 5.** *Let $t \in \mathcal{T}_\lambda$. Then t is $\mathcal{S}_\lambda$-typable iff $t \in \mathcal{SN}(\lambda)$.*

As before, the implication *typability implies normalization* can be show by simple arithmetical arguments provided by the *quantitative* flavour of the typing system $\mathcal{S}_\lambda$.

An equivalent formulation of system $\mathcal{S}_\lambda$, called $\mathcal{S}'_\lambda$, is given in Fig. 4. As before, we use regular as well as auxiliary judgments. Notice that $I = \emptyset$ in rule $(\wedge)$ is still possible,

$$\frac{}{x : [\tau] \vdash x : \tau}\ (\mathtt{ax}) \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \setminus\!\setminus x \vdash \lambda x.t : \Gamma(x) \Rightarrow \tau}\ (\Rightarrow_{\mathtt{i}}) \qquad \frac{\Gamma \vdash t : [\,] \Rightarrow \tau \qquad \Delta \vdash u : \sigma}{\Gamma \wedge \Delta \vdash tu : \tau}\ (\Rightarrow_{\mathtt{e}_1})$$

$$\frac{\Gamma \vdash t : [\sigma_k]_{k\in K} \Rightarrow \tau \quad (\Delta_k \vdash u : \sigma_k)_{k\in K} \quad K \neq \emptyset}{\Gamma \wedge_{k\in K} \Delta_k \vdash tu : \tau}\ (\Rightarrow_{\mathtt{e}_2})$$

Figure 3 System $\mathcal{S}_\lambda$

$$\text{Rule } (\mathtt{ax}) \qquad \text{Rule } (\Rightarrow_{\mathtt{i}}) \quad \frac{(\Gamma_k \vdash t : \sigma_k)_{k\in K}}{\wedge_{k\in K}\Gamma_k \Vdash t : [\sigma_k]_{k\in K}}\ (\wedge) \qquad \frac{\Gamma \vdash t : \mathcal{I} \Rightarrow \tau \quad \Delta \Vdash u : \mathcal{I}^*}{\Gamma \wedge \Delta \vdash tu : \tau}\ (\Rightarrow_{\mathtt{e}})$$

Figure 4 System $\mathcal{S}'_\lambda$

but derivations of the form $\Vdash t : [\,]$, representing untyped terms, will never be used. The choice operation $\_^*$ (defined at the beginning of Sec. 3) in rule $(\Rightarrow_{\mathtt{e}})$ is used to impose an arbitrary type for an erasable term, *i.e.* when t has type $[\,] \Rightarrow \tau$, then u needs to be typed with an arbitrary type $[\sigma]$, thus the auxiliary judgment typing u on the right premise of $(\Rightarrow_{\mathtt{e}})$ cannot assign $[\,]$ to u. This should be understood as a sort of *controlled weakening*. Here is an example of type derivation in system $\mathcal{S}'_\lambda$:

$$\dfrac{\dfrac{\dfrac{}{x : [\sigma] \vdash x : \sigma}}{x : [\sigma] \vdash \lambda y.x : [\,] \Rightarrow \sigma} \qquad \dfrac{\dfrac{}{z : [\tau] \vdash z : \tau}}{z : [\tau] \Vdash z : [\tau]}}{x : [\sigma], z : [\tau] \vdash (\lambda y.x)z : \sigma}$$

Since $\mathcal{S}_\lambda$ and $\mathcal{S}'_\lambda$ are equivalent, we also have:

▶ **Corollary 6.** *Let $t \in \mathcal{T}_\lambda$. Then t is $\mathcal{S}'_\lambda$-typable iff $t \in \mathcal{SN}(\lambda)$.*

4 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. 4.2) and $\mathcal{S}_{\lambda_\mu}$ (Sec. 4.3), 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, $\mathcal{H}'_\lambda$ and $\mathcal{S}'_\lambda$, introduced in Sec. 3.

4.1 Types

We consider a countable set of **base types** $a, b, c \ldots$ and the following categories of types:

(**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	$::=$	$a \mid \mathcal{I} \Rightarrow \mathcal{U}$

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, $\sigma_1 \cap \ldots \cap \sigma_n$ and $\sigma_1 \cup \ldots \cup \sigma_n$, 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. The *non-deterministic* **choice** operation $\_^*$ is now defined on intersection *and* union types:

$$[\mathcal{U}_k]^*_{k\in K} := \begin{cases} [\mathcal{U}] & \text{if } K = \emptyset\ \mathcal{U} \neq \langle\,\rangle \text{ is any arbitrary non-empty union type} \\ [\mathcal{U}_k]_{k\in K} & \text{if } K \neq \emptyset \end{cases}$$

$$\langle\sigma_k\rangle^*_{k\in K} := \begin{cases} \langle\sigma\rangle & \text{if } K = \emptyset \text{ and } \sigma \text{ is any arbitrary blind type} \\ \langle\sigma_k\rangle_{k\in K} & \text{if } K \neq \emptyset \end{cases}$$

where a blind type is a type of the form $[\,] \to \ldots \to [\,] \to a$. 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 9) (2) subject reduction is guaranteed in system $\mathcal{H}_{\lambda_\mu}$ for erasing steps $(\mu\alpha.c)u \to \mu\alpha.c$ ($\alpha \notin \mathtt{fn}(c)$).

The **arity** of types and union multiset types is defined by induction: for types σ, if $\sigma = \mathcal{I} \Rightarrow \mathcal{U}$, then $\mathtt{ar}(\sigma) := \mathtt{ar}(\mathcal{U}) + 1$, otherwise, $\mathtt{ar}(\sigma) := 0$; for union multiset types, $\mathtt{ar}(\langle\sigma_k\rangle_{k\in K}) := \Sigma_{k\in K}\ \mathtt{ar}(\sigma_k)$. The **cardinality of multisets** is defined by $|[\mathcal{U}_k]_{k\in K}| = |\langle\sigma_k\rangle_{k\in K}| := |K|$.

Variable assignments (Γ), are, as before, 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 $\mathtt{dom}(\Delta) := \{\alpha \mid \Delta(x) \neq \langle\,\rangle\}$, where $\langle\,\rangle$ is the empty union multiset. We may write $\emptyset$ to denote the name assignment that associates the empty union type $\langle\,\rangle$ to every name. When $\alpha \notin \mathtt{dom}(\Delta)$, then $\Delta(x)$ stands for $\langle\,\rangle$. We write $\Delta \vee \Delta'$ for $\alpha \mapsto \Delta(\alpha) + \Delta'(\alpha)$, where $\mathtt{dom}(\Delta \vee \Delta') = \mathtt{dom}(\Delta) \cup \mathtt{dom}(\Delta')$.

When $\mathtt{dom}(\Gamma)$ and $\mathtt{dom}(\Gamma')$ are disjoint we may write $\Gamma;\Gamma'$ instead of $\Gamma \wedge \Gamma'$. We write $x : [\mathcal{U}_k]_{k\in K};\Gamma$, even when $K = \emptyset$, for the following variable assignment $(x : [\mathcal{U}_k]_{k\in K};\Gamma)(x) = [\mathcal{U}_k]_{k\in K}$ and $(x : [\mathcal{U}_k]_{k\in K};\Gamma)(y) = \Gamma(y)$ if $y \neq x$. Similar concepts apply to name assignments, so that $\alpha : \langle\sigma_k\rangle_{k\in K};\Delta$ and $\Delta \setminus\!\setminus \alpha$ 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. This makes our proofs much simpler than those arising with idempotent types which are based on long generation lemmas (*e.g.* [6, 36]).

4.2 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. 4.3, 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. 5. Rule $(\Rightarrow_{\mathtt{e}})$ is to be understood as a *logical admissible* rule: if union (resp. intersection) is interpreted as the `OR` (resp. `AND`) logical connective, then $\mathtt{OR}_{k\in K}$ $(\mathcal{I}_k \Rightarrow \mathcal{U}_k)$ and $(\mathtt{AND}_{k\in K}\ \mathcal{I}_k)$ implies $(\mathtt{OR}_{k\in K}\ \mathcal{U}_k)$. As in the simply typed λ_μ-calculus [32], the $(\#_i)$ rule saves a type $\mathcal{U}$ for the

$$\frac{\mathcal{U} \neq \langle\,\rangle}{x : [\mathcal{U}] \vdash x : \mathcal{U} \mid \emptyset}\ (\mathtt{ax}) \qquad \frac{\Gamma \vdash t : \mathcal{U} \mid \Delta}{\Gamma \setminus\!\setminus x \vdash \lambda x.t : \langle \Gamma(x) \Rightarrow \mathcal{U} \rangle \mid \Delta}\ (\Rightarrow_{\mathtt{i}}) \qquad \frac{\Gamma \vdash t : \mathcal{U} \mid \Delta}{\Gamma \vdash [\alpha]t : \# \mid \Delta \vee \{\alpha : \mathcal{U}\}}\ (\#_i)$$

$$\frac{\Gamma \vdash c : \# \mid \Delta}{\Gamma \vdash \mu\alpha.c : \Delta(\alpha)^* \mid \Delta \setminus\!\setminus \alpha}\ (\#_e) \qquad \frac{(\Gamma_k \vdash t : \mathcal{U}_k \mid \Delta_k)_{k\in K}}{\wedge_{k\in K}\Gamma_k \Vdash t : [\mathcal{U}_k]_{k\in K} \mid \vee_{k\in K}\Delta_k}\ (\wedge)$$

$$\frac{\Gamma_t \vdash t : \langle \mathcal{I}_k \Rightarrow \mathcal{U}_k \rangle_{k\in K} \mid \Delta_t \quad \Gamma_u \Vdash u : \wedge_{k\in K}\mathcal{I}_k \mid \Delta_u}{\Gamma_t \wedge \Gamma_u \vdash t\,u : \vee_{k\in K}\mathcal{U}_k \mid \Delta_t \vee \Delta_u}\ (\Rightarrow_{\mathtt{e}})$$

Figure 5 System $\mathcal{H}_{\lambda_\mu}$

$$\dfrac{\Phi_y \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : [\mathcal{U}_a] \vdash x : \mathcal{U}_a \mid}}{x : [\mathcal{U}_a] \vdash [\alpha]x : \# \mid \alpha : \mathcal{U}_a}}{x : [\mathcal{U}_a] \vdash \mu\beta.[\alpha]x : \mathcal{U}_b \mid \alpha : \mathcal{U}_a}}{\vdash \lambda x.\mu\beta.[\alpha]x : \langle [\mathcal{U}_a] \Rightarrow \mathcal{U}_b \rangle \mid \alpha : \mathcal{U}_a}}{\Vdash \lambda x.\mu\beta.[\alpha]x : [\langle [\mathcal{U}_a] \Rightarrow \mathcal{U}_b \rangle] \mid \alpha : \mathcal{U}_a}}{y : [\mathcal{U}_y] \vdash y(\lambda x.\mu\beta.[\alpha]x) : \mathcal{U}_a \mid \alpha : \mathcal{U}_a}}{y : [\mathcal{U}_y] \vdash [\alpha]y(\lambda x.\mu\beta.[\alpha]x) : \# \mid \alpha : \langle \mathcal{U}_a, \mathcal{U}_a \rangle}}{y : [\mathcal{U}_y] \vdash \mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x) : \langle \mathcal{U}_a, \mathcal{U}_a \rangle \mid}}{\vdash \lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x) : \langle [\mathcal{U}_y] \Rightarrow \langle \mathcal{U}_a, \mathcal{U}_a \rangle \rangle \mid}$$

Figure 6 Typing **call-cc**

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 $(\#_e)$ rule is not deterministic since $\Delta(\alpha)^*$ denotes an arbitrary union type, a choice that is now discussed.

In simply typed λ_μ, **call-cc** $= \lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x)$ would be typed with $((a \Rightarrow b) \Rightarrow a) \Rightarrow a$ (Peirce's Law), so that the fact that α is *used* twice in the type derivation would not be explicitely materialized (same comment applies to idempotent intersection/union types). This makes a strong contrast with the derivation in Fig. 6, where $\mathcal{U}_a := \langle a \rangle$, $\mathcal{U}_b := \langle b \rangle$, $\mathcal{U}_y := \langle [\langle [\mathcal{U}_a] \Rightarrow \mathcal{U}_b \rangle] \Rightarrow \mathcal{U}_a \rangle$ and $\Phi_y \triangleright y : [\mathcal{U}_y] \vdash y : \mathcal{U}_y \mid$. Indeed, we can distinguish two different uses of names :

- The name α is saved twice by a $(\#_i)$ 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 previoulsy stored by α. A similar phenomenon occurs with λ-abstractions, which restore the types of the free ocurrences 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 handside of the sequent. Consequently, λ and 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 $(\#_e)$ rule, at least when $\alpha \notin \mathtt{fn}(c)$, 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 $(\#_e)$-rule). Suppose also an object o containing 2 occurrences of the subterm $[\gamma]t$, 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 loosing the *quantitative* flavour of the system (see also a formal argument just after Lem. 9).

We define now the notion of **size derivation**, which is a natural number representing the amount of information in a tree derivation. For any type derivation Φ, $\mathtt{sz}(\Phi)$ is inductively defined by the following rules, where we use an abbreviated notation for the premises.

$$\mathtt{sz}\left(\frac{}{x : [\mathcal{U}] \vdash x : \mathcal{U} \mid \emptyset}\ (\mathtt{ax})\right) := 1$$

$$\mathtt{sz}\left(\frac{\Phi_t \rhd t}{\Gamma \setminus\!\setminus x \vdash \lambda x.t : \langle \Gamma(x) \Rightarrow \mathcal{U}\rangle \mid \Delta}\ (\Rightarrow_{\mathtt{i}})\right) := \mathtt{sz}(\Phi_t) + 1$$

$$\mathtt{sz}\left(\frac{\Phi_t \rhd t}{\Gamma \vdash [\alpha]t : \# \mid \Delta \vee \{\alpha : \mathcal{U}\}}\ (\#_i)\right) := \mathtt{sz}(\Phi_t) + \mathtt{ar}(\mathcal{U})$$

$$\mathtt{sz}\left(\frac{\Phi_c \rhd c}{\Gamma \vdash \mu\alpha.c : \Delta(\alpha)^* \mid \Delta \setminus\!\setminus \alpha}\ (\#_e)\right) := \mathtt{sz}(\Phi_c) + 1$$

$$\mathtt{sz}\left(\frac{(\Phi_k \rhd t)_{k\in K}}{\wedge_{k\in K}\Gamma_k \Vdash t : [\mathcal{U}_k]_{k\in K} \mid \vee_{k\in K}\Delta_k}\ (\wedge)\right) := \Sigma_{k\in K}\ \mathtt{sz}(\Phi_k)$$

$$\mathtt{sz}\left(\frac{\Phi_t \rhd t \qquad \Phi_u \rhd u}{\Gamma \vdash t\,u : \vee_{k\in K}\mathcal{V}_k \mid \Delta}\ (\Rightarrow_{\mathtt{e}})\right) := \mathtt{sz}(\Phi_t) + \mathtt{sz}(\Phi_u) + |K|$$

System $\mathcal{H}_{\lambda_\mu}$ behaves as expected, in particular, typing is stable by reduction (Subject Reduction) and anti-reduction (Subject Expansion). Moreover,

▶ **Theorem 7.** *Let $o \in \mathcal{O}_{\lambda_\mu}$. Then o is $\mathcal{H}_{\lambda_\mu}$-typable iff $o \in \mathcal{HN}(\lambda_\mu)$ iff the head-strategy terminates on o. Moreover, if o is $\mathcal{H}_{\lambda_\mu}$-typable with tree derivation Π, then $\mathtt{sz}(\Pi)$ 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 normalization. Notice that Thm. 7 ensures that the head-strategy is complete for head-normalization in λ_μ.

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 $(\#_e)$ and $(\#_i)$ 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. 2.

4.3 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. As in system $\mathcal{S}'_\lambda$, the operation $\_^*$ is used to choose arbitrary types for erasable terms, so that no subterm is untyped, thus ensuring strong λ_μ-normalization. While the use of $\_^*$ in the $(\#_e)$-rule can be seen as a *weakening* on the right hand-sides of sequents, its use in rule $(\Rightarrow_{\mathtt{e}})$ corresponds to a form of *controlled weakening* on the left hand-sides. We still consider the definition of size given before, as the choice operator does not play any particular role.

As in system $\mathcal{H}_{\lambda_\mu}$, a term is typed with a non-empty union type:

$$\frac{\mathcal{U} \neq \langle\,\rangle}{x : [\mathcal{U}] \vdash x : \mathcal{U} \mid \emptyset}\ (\mathtt{ax}) \qquad \frac{\Gamma \vdash t : \mathcal{U} \mid \Delta}{\Gamma \setminus\!\setminus x \vdash \lambda x.t : \langle \Gamma(x) \Rightarrow \mathcal{U} \rangle \mid \Delta}\ (\Rightarrow_{\mathtt{i}}) \qquad \frac{\Gamma \vdash t : \mathcal{U} \mid \Delta}{\Gamma \vdash [\alpha]t : \# \mid \Delta \vee \{\alpha : \mathcal{U}\}}\ (\#_i)$$

$$\frac{\Gamma \vdash c : \# \mid \Delta}{\Gamma \vdash \mu\alpha.c : \Delta(\alpha)^* \mid \Delta \setminus\!\setminus \alpha}\ (\#_e) \qquad \frac{(\Gamma_k \vdash t : \mathcal{U}_k \mid \Delta_k)_{k\in K}}{\wedge_{k\in K}\Gamma_k \Vdash t : [\mathcal{U}_k]_{k\in K} \mid \vee_{k\in K}\Delta_k}\ (\wedge)$$

$$\frac{\Gamma_t \vdash t : \langle \mathcal{I}_k \Rightarrow \mathcal{U}_k \rangle_{k\in K} \mid \Delta_t \qquad \Gamma_u \Vdash u : \wedge_{k\in K}\ (\mathcal{I}_k^*) \mid \Delta_u}{\Gamma_t \wedge \Gamma_u \vdash t\,u : \vee_{k\in K}\mathcal{U}_k \mid \Delta_t \vee \Delta_u}\ (\Rightarrow_{\mathtt{e}})$$

Figure 7 System $\mathcal{S}_{\lambda_\mu}$

▶ **Lemma 8.** *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. 4.2 we obtain the typing system $\mathcal{S}'_\lambda$ in Fig. 4 that characterizes β-strong normalization.

A key property of system $\mathcal{S}_{\lambda_\mu}$ is known as *relevance*:

▶ **Lemma 9 (Relevance).** *If* $\Phi \triangleright \Gamma \vdash o : \mathcal{A} \mid \Delta$, *then* $\mathtt{dom}(\Gamma) = \mathtt{fv}(o)$ *and* $\mathtt{dom}(\Delta) = \mathtt{fn}(o)$.

Relevance holds thanks to the choice operator $\_^*$: indeed, if $\Delta(\alpha)^*$ is replaced by $\Delta(\alpha)$ in the $(\#_e)$-rule, then the following derivations gives a counter-example to the relevance property, where $\alpha \in \mathtt{fn}([\alpha]\mu\beta.[\gamma]x)$ but $\alpha \notin \mathtt{dom}(\gamma : \langle a \rangle)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : [\langle a\rangle] \vdash x : \langle a\rangle \mid}}{x : [\langle a\rangle] \vdash [\gamma]x : \# \mid \gamma : \langle a\rangle}}{x : [\langle a\rangle] \vdash \mu\beta.[\gamma]x : \langle\,\rangle \mid \gamma : \langle a\rangle}}{x : [\langle a\rangle] \vdash [\alpha]\mu\beta.[\gamma]x : \# \mid \gamma : \langle a\rangle}$$

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. 5). Notice that $\mathtt{sz}(\Phi) \geq 1$ holds for any *regular* derivation Φ, whereas, by definition, the derivation of the empty auxiliary judgment $\Vdash t : [\,] \mid$ has size 0.

5 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. 5.1 we analyse the *subject reduction (SR)* property, and we prove that reduction preserves typing and decreases the size of type derivations (that is why we call it weighted SR). The proof of this property makes use of two fundamental properties (Lem. 11 and 12) guaranteeing well-typedness of the meta-operations of substitution and replacement. Sec. 5.2 is devoted to *subject expansion (SE)*, which states that non-erasing anti-reduction preserves types. The proof uses the fact that reverse substitution (Lem. 13) and reverse replacement (Lem. 14) preserve types.

We start by stating an interesting property, to be used in our forthcoming lemmas, that allows us to split and merge auxilary derivations:

▶ **Lemma 10.** *Let* $\mathcal{I} = \wedge_{k\in K}\mathcal{I}_k$. *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.* $(\Phi_k \triangleright \Gamma_k \Vdash t : \mathcal{I}_k \mid \Delta_k)_{k\in K}$, $\Gamma = \wedge_{k\in K}\Gamma_k$ *and* $\Delta = \vee_{k\in K}\Delta_k$. *Moreover,* $\mathtt{sz}(\Phi) = \Sigma_{k\in K}\mathtt{sz}(\Phi_k)$.

5.1 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 11 (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* $\Phi_{o\{x/u\}}$ *such that*

- $\Phi_{o\{x/u\}} \triangleright \Gamma \wedge \Gamma_u \vdash o\{x/u\} : \mathcal{A} \mid \Delta \vee \Delta_u$.
- $\mathtt{sz}\left(\Phi_{o\{x/u\}}\right) = \mathtt{sz}\left(\Phi_o\right) + \mathtt{sz}\left(\Theta_u\right) - |\mathcal{I}|$.

Proof. By induction on Φ_o using Lem. 9 and 10. ◀

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 12 (Replacement).** *Let* $\Theta_u \triangleright \Gamma_u \Vdash u : \wedge_{k\in K}\ (\mathcal{I}_k^*) \mid \Delta_u$ *where* $\alpha \notin \mathtt{fn}(u)$. *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* $\Phi_{o\{\alpha /\!\!/ u\}}$ *such that :*

- $\Phi_{o\{\alpha /\!\!/ u\}} \triangleright \Gamma \wedge \Gamma_u \vdash o\{\alpha /\!\!/ u\} : \mathcal{A} \mid \alpha : \vee_{k\in K}\mathcal{V}_k; \Delta \vee \Delta_u$.
- $\mathtt{sz}\left(\Phi_{o\{\alpha /\!\!/ u\}}\right) = \mathtt{sz}\left(\Phi_o\right) + \mathtt{sz}\left(\Theta_u\right)$.

Proof. By induction on Φ using Lem. 9 and 10. ◀

Notice that the type of α in the conclusion of the derivation $\Phi_{o\{\alpha /\!\!/ u\}}$ (which is $\vee_{k\in K}\mathcal{V}_k$) is strictly smaller than that of the conclusion of the derivation Φ_o (which is $\langle \mathcal{I}_k \Rightarrow \mathcal{V}_k \rangle_{k\in K}$) if and only if $K \neq \emptyset$. Lemmas 11 and 12 are used in the proof of the following key property.

▶ **Property 1 (Weighted Subject Reduction for λ_μ).** *Let* $\Phi \triangleright \Gamma \vdash o : \mathcal{A} \mid \Delta$. *If* $o \to o'$ *is a non-erasing step, then there exists a derivation* $\Phi' \triangleright \Gamma \vdash o' : \mathcal{A} \mid \Delta$ *such that* $\mathtt{sz}\left(\Phi\right) > \mathtt{sz}\left(\Phi'\right)$.

Proof. By induction on $o \to o'$ using Lem. 9, 11 and 12. ◀

Discussion. A first remark about the property above is that variable and name assignments are not necessarily preserved by erasing reductions. Thus for example, consider $t = (\lambda y.x)z \to x = t'$. 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 $\mathcal{S}'_\lambda$ system appears on page 6), 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 (see how we deal with the erasing steps in the proof of Lem. 16).

A second remark is that the consideration of arities of names in the definition of the size of derivations (third case $(\#_i)$) is crucial to guarantee that μ-reduction decreases $\mathtt{sz}\left(\_\right)$. This is perfectly reflected in Lem. 12, where the type of α in the conclusion of the derivation $\Phi_{o\{\alpha /\!\!/ u\}}$ is strictly smaller than that of the conclusion of the derivation Φ_o.

A third point is about the use of the choice operator in the typing rule $(\#_e)$, which does not allow for the type $\langle\,\rangle$ to be assigned to α when $\alpha \notin \mathtt{fn}(c)$. More precisely, assume, just temporarily, that the $(\#_e)$ 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 Lem. 9, simply because $\alpha \notin \mathtt{fn}(\Delta)$ but α has two free occurrences in c. This has heavy consequences that can be illustrated by the reduction sequence $t = (\mu\alpha.c)x \to \mu\alpha.[\alpha](\mu\delta.[\alpha](\mu\beta.[\gamma]y)x)x \to^* \mu\alpha.c = t'$. 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.

5.2 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 $o\{x/u\}$; similarly, the second one, called reverse replacement, gives type information for a command c and a term u from the type derivation of $c\{\alpha/\!\!/u\}$. Both of them are proved by induction on derivations using Lem. 9 and 10. Formally,

▶ **Lemma 13 (Reverse Substitution).** *Let* $\Phi'\triangleright\Gamma' \vdash o\{x/u\} : \mathcal{A} \mid \Delta'$ *Then* $\exists\Gamma, \exists\Delta, \exists\mathcal{I}, \exists\Gamma_u, \exists\Delta_u$ *such that:* $\Gamma' = \Gamma \wedge \Gamma_u$, $\Delta' = \Delta \vee \Delta_u$, $\triangleright\Gamma; x : \mathcal{I} \vdash o : \mathcal{A} \mid \Delta$ *and* $\triangleright\Gamma_u \Vdash u : \mathcal{I} \mid \Delta_u$.

▶ **Lemma 14 (Reverse Replacement).** *Let* $\Phi' \triangleright \Gamma' \vdash o\{\alpha/\!\!/u\} : \mathcal{A} \mid \alpha : \mathcal{V}; \Delta'$, *where* $\alpha \notin \mathtt{fn}(u)$. *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:* $\Gamma' = \Gamma \wedge \Gamma_u$, $\Delta' = \Delta \vee \Delta_u$, $\mathcal{V} = \vee_{k\in K}\mathcal{V}_k$, $\triangleright\Gamma \vdash o : \mathcal{A} \mid \alpha : \langle\mathcal{I}_k \to \mathcal{V}_k\rangle_{k\in K}; \Delta$, *and and* $\triangleright\Gamma_u \Vdash u : \wedge_{k\in K}\mathcal{I}_k^* \mid \Delta_u$.

The following property will be used in Sec. 6 to show that normalization implies typability.

▶ **Property 2 (Subject Expansion for λ_μ).** *Assume* $\Phi' \triangleright \Gamma' \vdash o' : \mathcal{A} \mid \Delta'$. *If* $o \to o'$ *is a non-erasing step, then there is* $\Phi \triangleright \Gamma' \vdash o : \mathcal{A} \mid \Delta'$.

Proof. By induction on $\to$ using Lem. 9, 13 and 14. ◀

6 Strongly Normalizing λ_μ-Objects

In this section we show the characterization of strongly-normalizing terms of the $\lambda\mu$-calculus by means of the typing system introduced in Sec. 4, *i.e.* we show that an object o is strongly-normalizing iff t is typable.

The proof of our main result (Thm. 18) relies on the following two ingredients:

- Every $\mathcal{S}_{\lambda_\mu}$-typable object is in $\mathcal{SN}(\lambda_\mu)$ (Lem. 16).
- Every object in $\mathcal{SN}(\lambda_\mu)$ is $\mathcal{S}_{\lambda_\mu}$-typable (Lem. 17).

First, we inductively reformulate the set of strongly normalizing objects: the set $\mathcal{I}(\lambda_\mu)$ is defined as the smallest subset of $\mathcal{O}_{\lambda_\mu}$ satisfying the following closure properties:

(1) If $t_1, \dots, t_n$ $(n \geq 0) \in \mathcal{I}(\lambda_\mu)$, then $xt_1 \dots t_n \in \mathcal{I}(\lambda_\mu)$.
(2) If $t \in \mathcal{I}(\lambda_\mu)$, then $\lambda x.t \in \mathcal{I}(\lambda_\mu)$.
(3) If $c \in \mathcal{I}(\lambda_\mu)$, then $\mu\alpha.c \in \mathcal{I}(\lambda_\mu)$.
(4) If $t \in \mathcal{I}(\lambda_\mu)$, then $[\alpha]t \in \mathcal{I}(\lambda_\mu)$.
(5) If $u, t\{x/u\}\overline{v}$ $(n \geq 0) \in \mathcal{I}(\lambda_\mu)$, then $(\lambda x.t)u\overline{v} \in \mathcal{I}(\lambda_\mu)$.
(6) If $u, (\mu\alpha.c\{\alpha/\!\!/u\})\overline{v}$ $(n \geq 0) \in \mathcal{I}(\lambda_\mu)$, then $(\mu\alpha.c)u\overline{v} \in \mathcal{I}(\lambda_\mu)$.

The sets $\mathcal{SN}(\lambda_\mu)$ and $\mathcal{I}(\lambda_\mu)$ turn out to be equal, as expected:

▶ **Lemma 15.** $\mathcal{SN}(\lambda_\mu) = \mathcal{I}(\lambda_\mu)$.

Proof. $\mathcal{SN}(\lambda_\mu) \subseteq \mathcal{I}(\lambda_\mu)$ is proved by induction on the pair $\langle\eta(o), |o|\rangle$ endowed with the lexicographic order, where $\eta(o)$ denotes the maximal length of an λ_μ-reduction sequence starting at o and $|o|$ denotes the size of o. $\mathcal{I}(\lambda_\mu) \subseteq \mathcal{SN}(\lambda_\mu)$ is proved by induction on $\mathcal{I}(\lambda_\mu)$ using Cor. 1. No reducibility argument is then needed in this proof. ◀

▶ **Lemma 16.** *If o is $\mathcal{S}_{\lambda_\mu}$-typable, then $o \in \mathcal{SN}(\lambda_\mu)$.*

Proof. We proceed by induction on $\mathtt{sz}(\Phi)$, where $\Phi \triangleright \Gamma \vdash o : \mathcal{A} \mid \Delta$. When Φ does not end with the rule $(\Rightarrow_{\mathtt{e}})$ the proof is straightforward, so we consider Φ ends with $(\Rightarrow_{\mathtt{e}})$, where $\mathcal{A} = \mathcal{U}$ and $o = xt_1 \dots t_n$ or $o = (\mu\alpha.c)t_1 \dots t_n$ or $o = (\lambda x.u)t_1 \dots t_n$, where $n \geq 1$.
By construction there are subderivations $(\Phi_{t_i})_{i\in\{1\dots n\}}$ such that $(\mathtt{sz}(\Phi_{t_i}) < \mathtt{sz}(\Phi))_{i\in\{1\dots n\}}$ so that the *i.h.* gives $(t_i \in \mathcal{I}(\lambda_\mu))_{i\in\{1\dots n\}}$. There are three different cases:
If $o = xt_1 \dots t_n$, then from $t_i \in \mathcal{I}(\lambda_\mu)$ $(1 \leq i \leq n)$ we conclude directly $xt_1 \dots t_n \in \mathcal{I}(\lambda_\mu)$.
If $o = (\mu\alpha.c)t_1 \dots t_n$, there are two cases:

- $\alpha \in \mathtt{fn}(c)$. Using Prop. 1 we get $\Phi' \triangleright \Gamma \vdash (\mu\alpha.c\{\alpha /\!\!/ t_1\})t_2 \dots t_n : \mathcal{U} \mid \Delta$ and $\mathtt{sz}(\Phi') < \mathtt{sz}(\Phi)$. Then the *i.h.* gives $(\mu\alpha.c\{\alpha /\!\!/ t_1\})t_2 \dots t_n \in \mathcal{I}(\lambda_\mu)$. This, together with $t_1 \in \mathcal{I}(\lambda_\mu)$ gives $o \in \mathcal{I}(\lambda_\mu)$.
- $\alpha \notin \mathtt{fn}(c)$. Then it is easy to build a type derivation $\Phi' \triangleright \Gamma' \vdash (\mu\alpha.c)t_2 \dots t_n : \mathcal{U} \mid \Delta'$ verifying $\mathtt{sz}(\Phi') < \mathtt{sz}(\Phi)$, so that $(\mu\alpha.c)t_2 \dots t_n \in \mathcal{I}(\lambda_\mu)$ holds by the *i.h.* This, together with with $t_1 \in \mathcal{I}(\lambda_\mu)$ gives $o \in \mathcal{I}(\lambda_\mu)$.

If $o = (\lambda x.u)t_1 \dots t_n$, we reason similarly to the previous one. ◀

Normalization also implies typability:

▶ **Lemma 17.** *If $o \in \mathcal{SN}(\lambda_\mu)$, then o is $\mathcal{S}_{\lambda_\mu}$-typable.*

Proof. Thanks to Lem. 15 we can reason by induction on $o \in \mathcal{I}(\lambda_\mu) = \mathcal{SN}(\lambda_\mu)$. The four first cases are straightforward.

Let $o = (\lambda x.u)vt_1 \dots t_n \in \mathcal{I}(\lambda_\mu)$ coming from $u\{x/v\}t_1 \dots t_n, v \in \mathcal{I}(\lambda_\mu)$. By the *i.h.* $u\{x/v\}t_1 \dots t_n$ and v are both typable. We consider two cases. If $x \in \mathtt{fv}(u)$, then $(\lambda x.u)vt_1 \dots t_n$ is typable by Prop. 2. Otherwise, by construction, we get typing derivations for $u, t_1 \dots, t_n$ which can easily be used to build a typing derivation of $(\lambda x.u)vt_1 \dots t_n$.

Let $o = (\mu\alpha.c)vt_1 \dots t_n \in \mathcal{I}(\lambda_\mu)$ coming from $(\mu\alpha.c\{\alpha /\!\!/ v\})t_1 \dots t_n, v \in \mathcal{I}(\lambda_\mu)$. By the *i.h.* $(\mu\alpha.c\{\alpha /\!\!/ v\})t_1 \dots t_n$ and v are both typable. We consider two cases. If $\alpha \in \mathtt{fn}(c)$, then $(\mu\alpha.c)vt_1 \dots t_n$ is typable by Prop. 2. Otherwise, by construction, we get typing derivations for $c, t_1 \dots, t_n$ which can easily be used to build a typing derivation of $(\mu\alpha.c)vt_1 \dots t_n$. ◀

Lem. 16 and 17 allow us to conclude with the main result of this paper which is the equivalence between typability and strong-normalization for the λ_μ-calculus. Notice that no reducibility argument was used in the whole proof.

▶ **Theorem 18.** *Let $o \in \mathcal{O}_{\lambda_\mu}$. Then o is typable in system $\mathcal{S}_{\lambda_\mu}$ iff $o \in \mathcal{SN}(\lambda_\mu)$. Moreover, if o is $\mathcal{S}_{\lambda_\mu}$-typable with tree derivation Π, then $\mathtt{sz}(\Pi)$ 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. This modification, which does not recover subject expansion, is however sufficient to guarantee *weighted* subject reduction in all the cases (erasing and non-erasing steps) without changing the original measure of the derivations in system $\mathcal{S}_{\lambda_\mu}$.

7 Conclusion

This paper 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 here, and implementing a small step operational semantics for classical natural deduction. Unfortunately we cannot provide here the details of such development due to lack of space, but they can be found in [26]. Such a calculus can be seen as an extension of the *substitution at a distance paradigm* [2, 1] to the classical case.

Quantitative types are a powerful tool to provide relational models for λ-calculus [14, 3]. 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 [18].

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 [25].

The fact that idempotent types were already used to show observationally equivalence between call-by-name and call-by-need [23] in intuitionistic logic suggests that our typing system $\mathcal{S}_{\lambda_{\mu r}}$ 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 [33].

Moreover, it is possible to obtain *exact* bounds (as in [5]) for the lengths of the head-reduction and the perpetual reduction sequences. 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, which is one of the contributions of this paper.

The inhabitation problem for λ-calculus is known to be undecidable for idempotent intersection types [35], but decidable for the non-idempotent ones [7]. We may conjecture that inhabitation is also decidable for $\mathcal{H}_{\lambda_\mu}$.

Acknowledgment: We would like to thank Vincent Guisse, who started a reflexion on quantitative types for the λ_μ-calculus during his M1 stage in Univ. Paris-Diderot.

References

1 B. Accattoli, E. Bonelli, D. Kesner, and C. Lombardi. A nonstandard standardization theorem. In P. Sewell, editor, *Proceedings of the 41st Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2014.

2 B. Accattoli and D. Kesner. The structural *lambda*-calculus. In A. Dawar and H. Veith, editors, *Proceedings of 24th EACSL Conference on Computer Science Logic*, volume 6247 of *Lecture Notes in Computer Science*, pages 381–395. Springer-Verlag, Aug. 2010.

3 S. Amini and T. Erhard. On Classical PCF, Linear Logic and the MIX Rule. In S. 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.

4 Z. M. Ariola, H. Herbelin, and A. Saurin. Classical call-by-need and duality. In Ong [31], pages 27–44.

5 A. Bernadet and S. Lengrand. Complexity of strongly normalising λ-terms via non-idempotent intersection types. In M. Hofmann, editor, *Foundations of Software Science and Computation Structures (FOSSACS)*, volume 6604 of *Lecture Notes in Computer Science*. Springer-Verlag, 2011.

6 A. Bernadet and S. Lengrand. Non-idempotent intersection types and strong normalisation. *Logical Methods in Computer Science*, 9(4), 2013.

7 A. Bucciarelli, D. Kesner, and S. Ronchi Della Rocca. The inhabitation problem for non-idempotent intersection types. In Díaz et al. [16], pages 341–354.

8 A. Bucciarelli, D. Kesner, and D. Ventura. Non-idempotent intersection types for the lambda-calculus. Submitted.

9 M. Coppo and M. Dezani-Ciancaglini. A new type assignment for lambda-terms. *Archive for Mathematical Logic*, 19:139–156, 1978.

10 M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 4:685–693, 1980.

11 P. Curien and H. Herbelin. The duality of computation. In M. Odersky and P. 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.

12 R. David and K. Nour. A short proof of the strong normalization of classical natural deduction with disjunction. *J. Symb. Log.*, 68(4):1277–1288, 2003.

13 E. De Benedetti and S. Ronchi Della Rocca. Bounding normalization time through intersection types. In Graham-Lengrand and Paolini [21], pages 48–57.

14 D. de Carvalho. *Sémantiques de la logique linéaire et temps de calcul*. These de doctorat, Université Aix-Marseille II, 2007.

15 D. de Carvalho. Execution time of lambda-terms via denotational semantics and intersection types. *CoRR*, abs/0905.4251, 2009.

16 J. Díaz, I. Lanese, and D. Sangiorgi, editors. *Proceedings of the 8th International Conference on Theoretical Computer Science (TCS)*, volume 8705 of *Lecture Notes in Computer Science*. Springer-Verlag, 2014.

17 D. J. Dougherty, S. Ghilezan, and P. 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.

18 T. Ehrhard. Collapsing non-idempotent intersection types. In P. Cégielski and A. 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.

19 P. Gardner. Discovering needed reductions using type theory. In M. Hagiya and J. 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*, pages 555–574. Springer, 1994.

20 J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1990.

21 S. Graham-Lengrand and L. 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.

22 T. 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.

23 D. Kesner. Reasoning about call-by-need by means of types. In B. Jacobs and C. 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.

24 D. Kesner and D. Ventura. Quantitative types for the linear substitution calculus. In Díaz et al. [16], pages 296–310.

25 D. Kesner and D. Ventura. A resource aware computational interpretation for herbelin's syntax. In M. Leucker, C. Rueda, and F. 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.

26 D. Kesner and P. Vial. Types as resources for classical natural deduction, 2017. Available at `www.irif.fr/~kesner/papers/lambda-mu.pdf`.

27 A. Kfoury. A linearization of the lambda-calculus and consequences. Technical report, Boston Universsity, 1996.

28 K. Kikuchi and T. Sakurai. A translation of intersection and union types for the $\lambda\mu$-calculus. In J. 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.

29 J.-L. Krivine. *Lambda-calculus, types and models*. Ellis Horwood, 1993.

30 O. Laurent. On the denotational semantics of the untyped lambda-mu calculus, 2004. Unpublished note.

31 C. L. 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*. Springer-Verlag, 2011.

32 M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In A. 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.

33 P. Pédrot and A. Saurin. Classical by-need. In P. 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.

34 P. Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001.

35 P. Urzyczyn. The emptiness problem for intersection types. *Journal of Symbolic Logic*, 64(3):1195–1215, 1999.

36 S. van Bakel. Sound and complete typing for lambda-mu. In E. Pimentel, B. Venneri, and J. 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.

37 S. van Bakel, F. Barbanera, and U. de'Liguoro. A filter model for the $\lambda\mu$-calculus - (extended abstract). In Ong [31], pages 213–228.

38 S. van Bakel, F. Barbanera, and U. de'Liguoro. Characterisation of strongly normalising lambda-mu-terms. In Graham-Lengrand and Paolini [21], pages 1–17.

39 L. Vaux. Convolution lambda-bar-mu-calculus. In S. R. D. 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.