Consuming and Persistent Types for Classical Logic

Delia Kesner
Université de Paris and Institut Universitaire de France
France

Pierre Vial
Inria (LS2N CNRS)
France

Abstract

We prove that type systems are able to capture exact measures related to *dynamic* properties of functional programs with *control operators*, which allow implementing intricate *continuations* and *backtracking*. Our type systems give the number of evaluation steps to normal form as well as the size of this normal form *without any evaluation being needed.*

We focus on Parigot's $\lambda\mu$-calculus, a computational interpretation of classical natural deduction, and our type systems are based upon non-idempotent *intersection* and *union* types. We introduce two kinds of arrows: *consuming* ones, which type function applications that are evaluated/destroyed during reduction, and *persistent* ones, which type those that are never consumed. These two forms of arrows are the essential tool of our typing systems to deal with control operators, as we are going to show.

The main contribution of this paper is a type framework capturing exact measures within the $\lambda\mu$-calculus for 3 different evaluation strategies, namely, head, leftmost-outermost and maximal, associated to the well-known notions of head, weak and strong normalization respectively. Moreover, this is done in a *parametric* way: we do not provide one type system for each reduction strategy, as it is usually done in the literature, but we give a *unique* typing system parametrizing key concepts and factorizing common proofs.

CCS Concepts: • **Theory of computation**;

Keywords: classical logic, lambda-mu calculus, intersection and union types, exact measures, natural deduction

ACM Reference Format:
Delia Kesner and Pierre Vial. 2020. Consuming and Persistent Types for Classical Logic. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20), July 8–11, 2020, Saarbrücken, Germany*. ACM, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/3373718.3394774

ACM ISBN 978-1-4503-7104-9/20/07…$15.00
https://doi.org/10.1145/3373718.3394774

1 Introduction

In contrast to *simple types*, which are only *sound* with respect to termination (a simple typable term is always terminating), **intersection types**, pioneered by [Coppo and Dezani-Ciancaglini 1978, 1980], provide *sound* and *complete* tools to **characterize** different notions of termination (a program is typable *iff* it is terminating). **Idempotent** intersection types were used in different frameworks to characterize *qualitative* properties of programs, such as for example solvability, normalization (head, linear-head, weak, strong), etc. Inspired from Linear Logic [Girard 1987] –a resource-aware logical system– **non-idempotent** types, pioneered by [Gardner 1994] and [Kfoury 2000], refined the original intersection type theory, which was idempotent, by providing an adequate tool to reason about *quantitative properties*, as enlightened by [de Carvalho 2007, 2018]. Indeed, for instance, it is not only the case that a term t is typable *if and only if* t is terminating, but, from the size of its typing derivation, it is also possible to obtain rough **upper bounds** for the *size* of the normal form of t and the *length* of the evaluation sequence to this normal form. Thus, a type system specified by means of non-idempotent types provides *quantitative* information about a *dynamic* property of the program itself. This quantitative information is provided without having to run the program: the typing rules are defined by induction on their tree structure, without making any reference to reduction.

Functional Programs with Control Operators. Even though the Curry-Howard correspondence was initially formulated for *intuitionistic* logic and pure functional programming, it was later extended to *classical* logic and lambda calculi with control operators by the discovery of [Griffin 1990], who remarked that the `callcc` operator, which captures evaluation contexts and allows restoring them later, could be naturally typed with Peirce's law. The $\lambda\mu$-calculus was introduced [Parigot 1992] thereafter as a computational interpretation of classical natural deduction. Following Griffin, the `Scheme` operator **call-cc** may be directly represented in the $\lambda\mu$-calculus –as well as other control operators [de Groote 1994; Laurent 2003a]–, and is typed with the Peirce Law $((A \to B) \to A) \to A$, which is sufficient to move from intuitionistic to classical logic. In particular, the $\lambda\mu$-calculus enables backtracking, *i.e.* to go back multiple times to any execution point of a program.

Some works have investigated intersection and union types to characterize semantic properties of classical term calculi [Dougherty et al. 2008; Kikuchi and Sakurai 2014; Laurent 2004; van Bakel 2010], but none of them provides quantitative information about normalizing terms. In [Kesner and Vial 2017], two intersection/union type systems are proposed to characterize head and strong normalization of $\lambda\mu$-terms, but they only provide *upper bounds* for evaluation lengths, while we focus in this paper on *exact measures*.

Exact Measures. In the case of the λ-calculus, a way to obtain exact measures from a typing system is to consider non-idempotent type derivations which are, in some sense, "minimal" [de Carvalho 2007, 2018], an approach that was later extended in [Bernadet and Lengrand 2013]. Given a term t, one thus obtains a derivation whose measure is equal to the *sum* $\ell_t + f_t$, where ℓ_t is the normalization length of t and f_t the size of its normal form. Thus, the measure is *unsplit*. But there can be an exponential gap between f_t and ℓ_f, and this suggests that these measures should be computed *independently*. Indeed, another approach is developed by [Accattoli et al. 2018] which not only gives *split* exact measures (*i.e.* ℓ_t and f_t are captured separately) but also *internalizes* the notion of minimal derivation. This approach has been adapted to call-by-need [Accattoli et al. 2019], call-by-value [Accattoli and Guerrieri 2018], and call-by-push-value [Bucciarelli et al. 2020]. And it is also the main approach in this paper. However, several concepts and methods used for intuitionistic logic do not trivially extend to the classical case, *e.g.* separability [Saurin 2005], realizability [Krivine 2007], proof-nets [Laurent 2003b], models [van Bakel 2010; Vaux 2007], etc. It is not then surprising that the technical tools used to obtain exact measures for intuitionistic logic do not scale up to classical logic, and thus for several reasons:

- ♦ In the λ-calculus, β-reduction simply decreases the **size** of non-idempotent derivations (the size of a derivation being defined as its number of typing judgments).
- ♦ The $\lambda\mu$-calculus features another reduction, called μ, which does not always decrease the size of derivations (and sometimes even increases it), even in the non-idempotent setting [Kesner and Vial 2017].
- ♦ In order to obtain decreasing (*i.e. quantitative*) measures in the classical setting, it is necessary to refine the notion of size, typically by recording the **arities** of continutation types inside the derivations.
- ♦ Yet, the system of [Accattoli et al. 2018] for the λ-calculus collapses the information pertaining to arities, and thus cannot be extended to the $\lambda\mu$-calculus (details in Sec. 3.1).

The solution we propose to capture exact measures in the $\lambda\mu$-calculus is to keep track of the different *nature* of the constructors involved during the evaluation process *within the type system*. Indeed, a term constructor is *consuming* if it is destroyed during evaluation, while a *persistent* constructor remains preserved until the normal form. For that, we use two kinds of functional arrows (consuming and persistent) in the typing system indicating whether they will be consumed or not during evaluation. This refines [Accattoli et al. 2018] so that arities are not collapsed anymore. We thus succeed in capturing exact measures with non-idempotent types in the classical case. The resulting types and derivations are called here **exact**.

Last but not least, in order to type classical constructors in the $\lambda\mu$-calculus, one must collect and store together different types, and we do this by using a non-idempotent *union* operator. But to precisely count the number of μ-evaluation steps by means of these types, one needs to transform persistent arrows into consuming ones, a transformation which can be done by using an **activation operator**. This is one of the crucial constructions that are necessary to provide exact measures for the $\lambda\mu$-calculus (details in Sec. 3.2).

Parametrization. In the intersection type discipline, it is often the case that different variants of the same type system lead to *different* characterization results using *similar* proofs, which are often skipped or painfully rewritten for several (but similar) cases. In this paper, we do not want to skip these subtleties, neither to fully prove again similar results. We thus define a unique parametrized type framework, which highlights the specific differences between the various characterization results. This allows us to give a very concise presentation, coming along with proofs which do not need to be developed independently for each particular case.

Structure of the Paper. We made the following choice of presentation to progressively show our contributions.

- ♦ Sec. 2 presents a type-theoretical characterization of exact bounds for the λ-calculus w.r.t. *head* evaluation and *head* normalization. Although this characterization is not new *per se*, our alternative presentation introduces the novel tools which are later crucial for the classical case.
- ♦ Sec. 3 recalls the $\lambda\mu$-calculus and gives its non-idempotent system $\mathscr{U}_{\mathsf{hd}}$ (Sec. 3.1) capturing upper bounds for head evaluation, based on [Kesner and Vial 2017].
- ♦ Sec. 3.2 proposes a new type system $\mathscr{X}_{\mathsf{hd}}$ capturing exact bounds for head evaluation in the $\lambda\mu$-calculus. The role of persistent arrows become crucial in the classical case, as well as their transformation into consuming arrows, what is achieved by means of the activation operator.
- ♦ Sec. 4 extends the technique used for head evaluation to two more strategies: leftmost-outermost and maximal, respectively related to weak and strong normalization. Actually, we give a *unique parametrized* system that can be instantiated for the 3 mentioned evaluations. This allows us to *factorize* the common proofs of the properties pertaining to the three associated strategies, and outlines the key ingredients necessary to obtain exact bounds in different frameworks.

2 Intuitionistic Computation

This section illustrates the main principles of our general development by treating the particular case of the head reduction strategy for the λ-calculus. Indeed, Sec. 2.2 *reformulates* the results in [Accattoli et al. 2018] within a new typing system introduced in Sec. 2.1.

We consider a countably infinite set of **variables** $x, y, z, \ldots$. The set of λ-**terms** is given by the following grammar:

$$(\textbf{terms}) \quad t, u, v \quad ::= \quad x \mid \lambda x.t \mid tu$$

We denote by $\mathtt{I}$ the identity function $\lambda x.x$ and we abbreviate an application $v = (\ldots((tu_1)u_2)\ldots u_n)$ as $t\, u_1 \ldots u_n$. Moreover, in this last case, when t is a variable or an abstraction, t is said to be the **head** of v. The predicate $\neg\mathtt{abs}(\_)$ characterizes all the terms that are not abstractions.

Free and **bound variables** of objects are defined as expected. We work modulo α-**conversion**, *i.e.* renaming of bound variables. **Application** of the **substitution** $\{x/u\}$ to t, written $t\{x/u\}$, may require α-conversion in order to avoid capture of free variables, it is defined as expected.

The λ-calculus is given by the set of λ-terms and the **reduction relation** $\rightarrow_\beta$, which is the closure by all contexts (defined in a natural way) of the following rewriting rule:

$$(\lambda x.t)u \quad \mapsto_\beta \quad t\{x/u\}$$

We use $\rightarrow^*_\beta$ (resp. $\rightarrow^+_\beta$) for the reflexive-transitive (resp. transitive) closure of the relation $\rightarrow_\beta$.

Ideally, in order to measure the evaluation lengths and the normal forms, we would like to distinguish at the term level the constructors which are consumed during evaluation and those which remain in the normal form. We call the former **consuming** and the latter **persistent**. For instance, if we temporarily materialize application with the constructor @ in the reduction step $t_0 = (\lambda x.w@x)@(\lambda z.z) \rightarrow_\beta w@(\lambda z.z)$, then persistent constructors in t_0 can be colored in red as follows $(\lambda x.w@x)@\lambda z.z$. Indeed, all the occurrences of the variable x and the outermost application are destroyed during the β-reduction step and thus are not colored. However, in general, the notions of persistence and consumption cannot be formalized at the term level. For instance, consider $(\lambda x.x\,x)(\lambda z.z)$, which reduces to $(\lambda z.z)(\lambda z.z)$ and then to $\lambda z.z$: one copy of $\lambda z.z$ is consumed whereas the other one persists in the normal form. Thus, the different nature of the constructors (consuming *vs.* persistent) cannot be distinguished on the untyped level, but they can be hopefully captured in the (non-idempotent) type derivations.

Head Normal Forms. Results of evaluation sequences are terms having a particular syntactical form. In what follows, a term of the form $\lambda x_1.\ldots\lambda x_n.xt_1\ldots t_m$ $(n, m \geq 0)$ is called a **head normal form** (HNF), and $t_1, \ldots, t_m$ its **head arguments**. HNFs are captured by the predicate $\mathtt{norm}_{\mathtt{hd}}(\_)$ in Fig. 1, where an auxiliary predicate $\mathtt{neut}_{\mathtt{hd}}(\_)$ is used to denote HNFs that are not abstractions (they are called **neutral** HNF). We measure HNFs by using the function hd-**size**: $|x|_{\mathtt{hd}} := 1$, $|\lambda x.t|_{\mathtt{hd}} := |t|_{\mathtt{hd}} + 1$ and $|tu|_{\mathtt{hd}} := |t|_{\mathtt{hd}} + 1$. Notice that hd-size does not count arguments of applications.

$$\frac{}{\mathtt{neut}_{\mathtt{hd}}(x)} \qquad \frac{\mathtt{neut}_{\mathtt{hd}}(t)}{\mathtt{neut}_{\mathtt{hd}}(tu)} \qquad \frac{\mathtt{neut}_{\mathtt{hd}}(t)}{\mathtt{norm}_{\mathtt{hd}}(t)} \qquad \frac{\mathtt{norm}_{\mathtt{hd}}(t)}{\mathtt{norm}_{\mathtt{hd}}(\lambda x.t)}$$

Figure 1. Head Neutral and Normal Terms

Head Normalization Versus Head Evaluation. A λ-term t is **head normalizing (HN)** if there is a reduction sequence from t to a HNF. We concentrate, however, on a particular *deterministic* relation, the **head strategy**, defined in Fig. 2.

$$\frac{}{(\lambda x.t)u \rightarrow_{\mathtt{hd}} t\{x/u\}}\,(\beta) \qquad \frac{t \rightarrow_{\mathtt{hd}} t'}{\lambda x.t \rightarrow_{\mathtt{hd}} \lambda x.t'}\,(\mathtt{abs}{-}\lambda)$$

$$\frac{\neg\mathtt{abs}(t) \quad t \rightarrow_{\mathtt{hd}} t'}{t\,u \rightarrow_{\mathtt{hd}} t'\,u}\,(\neg\mathtt{abs})$$

Figure 2. The Head Evaluation Strategy for the λ-Calculus

A term t is a hd-normal form, written $t \not\rightarrow_{\mathtt{hd}}$, if there is no t' s.t. $t \rightarrow_{\mathtt{hd}} t'$. The term t is **head terminating** if $t \rightarrow^*_{\mathtt{hd}} t'$ and $t' \not\rightarrow_{\mathtt{hd}}$. One easily proves that t is a HNF iff $t \not\rightarrow_{\mathtt{hd}}$.

Note that t head terminating clearly implies t head normalizing. The converse is also true, but the proof is non-trivial [Barendregt 1985; Krivine 1993]. In other words, the (deterministic) head strategy computes the head normal form when it exists, and thus provides an implementation *certification* of the abstract notion of head normalization. Intersection type systems provide elegant, short and *combinatorial* proofs of this kind of certification, thus avoiding the intricacies of syntactical proofs (based on residuals and complete developments).

2.1 Upper Bounds

We now present a type system based on *(non-idempotent) intersection* types, which is used to obtain upper bounds for the head strategy presented before. In particular, intersection is *associative*, *commutative*, and *non-idempotent*, so that intersection types can be represented as multisets.

Types are defined by means of the following grammar:

$$\begin{array}{llll} (\textbf{Intersection Types}) & \mathcal{I}, \mathcal{J} & ::= & [\sigma_k]_{k\in K} \\ (\textbf{Elementary Types}) & \sigma, \tau & ::= & \bullet \mid \mathcal{I} \rightarrow \sigma \mid \mathcal{I} \twoheadrightarrow \sigma \end{array}$$

The intersection type $[\sigma_k]_{k\in K}$ is a finite multiset whose elements are indexed by $k \in K$, where K may be empty. We use $\mathtt{card}(\mathcal{I})$ to denote the cardinal of an intersection type $\mathcal{I}$, the symbol $+$ to denote *multiset union*, *e.g.* $[\sigma, \tau] + [\sigma] = [\sigma, \sigma, \tau]$ and $\subseteq$ to denote *multiset inclusion*, *e.g.* $[\sigma, \tau] \subseteq [\sigma, \tau, \sigma]$. The constant $\bullet$ is called the **end type**, it can only be assigned to subterms that are never going to be applied, including *abstractions*, *e.g.* $\lambda x.x$ in the term $y(\lambda x.x)$. There are two kinds of arrows: $\rightarrow$ is a **consuming** arrow, which types application constructors that are going to be consumed during evaluation, while $\nrightarrow$ is a **persistent** arrow, typing application constructors which are persistent, *i.e.* remain in the normal form. We write $\mathcal{I} \Rightarrow \sigma$ when we do not want to distinguish between the two kinds of arrows. We define the **persistent order** function $\#_{\mathsf{p}}\_$ on elementary types as $\#_{\mathsf{p}}(\sigma)$ is 1 when $\sigma = \mathcal{I} \nrightarrow \tau$, and 0 otherwise.

Variable assignments (written Γ), are total functions from variables to intersection types; we use $\mathsf{supp}(\Gamma)$ for the **support** of Γ. The notation $\Gamma(x)$ stands for $[\,]$ when $x \notin \mathsf{supp}(\Gamma)$. We write $\Gamma \wedge \Gamma'$ for $x \mapsto \Gamma(x) + \Gamma'(x)$, where $\mathsf{supp}(\Gamma \wedge \Gamma') = \mathsf{supp}(\Gamma) \cup \mathsf{supp}(\Gamma')$. When $\mathsf{supp}(\Gamma)$ and $\mathsf{supp}(\Gamma')$ are disjoint we may write $\Gamma; \Gamma'$ instead of $\Gamma \wedge \Gamma'$. We write $x : [\sigma_k]_{k\in K}; \Gamma$, even when $K = \emptyset$, for the following variable assignment $(x : [\sigma_k]_{k\in K}; \Gamma)(x) = [\sigma_k]_{k\in K}$ and $(x : [\sigma_k]_{k\in K}; \Gamma)(y) = \Gamma(y)$ if $y \neq x$. We use $\Gamma' \subseteq \Gamma$ when $\Gamma'(x) \subseteq \Gamma(x)$ for all x.

We present the type system $\mathscr{U}^{\lambda}_{\mathsf{hd}}$ ($\mathcal{U}$ means $\mathcal{U}$pper) in Fig. 3, it is based on **regular** (resp. **auxiliary**) **judgments** of the form $\Gamma \vdash t : \sigma$ (resp. $\Gamma \Vdash t : \mathcal{I}$). In both cases t is said to be the **subject** of the judgment. A **(type) derivation** is a tree obtained by applying the (inductive) typing rules of the system. We write $\Phi \triangleright \mathcal{TJ}$ for a type derivation concluding with the judgment $\mathcal{TJ}$, and just $\triangleright \mathcal{TJ}$ if there exists Φ such that $\Phi \triangleright \mathcal{TJ}$, in which case the subject of the judgment is said to be **typable**. The **size of a type derivation** Φ, written $\mathsf{sz}(\Phi)$, is obtained by counting all the nodes of the tree, except those corresponding to the rule $\wedge$.

$$\frac{}{x : [\sigma] \vdash x : \sigma}\,(\mathsf{ax}) \qquad \frac{\Gamma; x : \mathcal{I} \vdash t : \sigma}{\Gamma \vdash \lambda x.t : \mathcal{I} \rightarrow \sigma}\,(\mathsf{abs})$$

$$\frac{(\Gamma_k \vdash t : \sigma_k)_{k\in K}}{\wedge_{k\in K}\Gamma_k \Vdash t : [\sigma_k]_{k\in K}}\,(\wedge) \qquad \frac{\Gamma_t \vdash t : \mathcal{I} \Rightarrow \sigma \quad \Gamma_u \Vdash u : \mathcal{I}}{\Gamma_t \wedge \Gamma_u \vdash t\,u : \sigma}\,(\mathsf{app})$$

Figure 3. System $\mathscr{U}^{\lambda}_{\mathsf{hd}}$: Upper Bounds for the λ-Calculus

System $\mathscr{U}^{\lambda}_{\mathsf{hd}}$ corresponds to the original non-idempotent intersection type system for λ-calculus [Gardner 1994], except that arrows are here of two kinds, consuming or persistent, a difference which does not play any role for the moment. A well-known result is the following:

Theorem 2.1 (Upper Bounds for Head Reduction)**.** *A term t is $\mathscr{U}^{\lambda}_{\mathsf{hd}}$ typable iff t is head-normalizing iff the head strategy terminates on t. Moreover, the size of the type derivation of t gives an upper bound to the sum $\ell + f$, where ℓ is the length of the head-evaluation of t and f is the size of the* HNF *of t.*

Theorem. 2.1 relies on *quantitative* subject reduction: not only typing is stable under reduction, but the **size** of derivations, given by the number of their judgments, **decreases**:

Proposition 2.2 (Quantitative Subject Reduction)**.** *If $\Pi \triangleright \Gamma \vdash t : \sigma$ and $t \rightarrow_{\beta} t'$, then there is $\Pi' \triangleright \Gamma \vdash t' : \sigma$ such that* $\mathsf{sz}(\Pi) \geq \mathsf{sz}(\Pi')$.

A few words should be said about the quantitative aspect of this last statement: since the type system is non-idempotent, then given a typed redex $(\lambda x.r)s$, the types of x match the types of s, *number-wise.* Once such a redex is fired, giving $r\{x\backslash s\}$, the (app) and the (abs)-rules typing the redex are destroyed, as well as all the (ax)-rules typing x, and no duplication takes place at the level of typing derivations because of non-idempotency: indeed, the size of the derivation decreases by $2 + k$, where $k \geq 0$ is the number of (ax)-rules typing x. Now, when this redex is fired *inside* a typed term t, *each* subderivation typing $(\lambda x.r)s$ inside the whole type derivation of t decreases w.r.t. size. However, it is also possible that $(\lambda x.r)s$ occurs *untyped* in t, and then the decrease of the size of the derivation typing t is null. This explains why, in system $\mathscr{U}^{\lambda}_{\mathsf{hd}}$, only upper bounds (and not exact measures) can be captured.

Note that, since head redexes are always typed, subject reduction is always *strictly* decreasing for the head-strategy.

2.2 Exact Measures

The types that we have defined so far are enough to characterize normalization properties ([Bucciarelli et al. 2017] for a survey) by providing the expected corresponding upper bounds. However, only some restricted form of typing, known as *tight types* in [Accattoli et al. 2018], provides *exact measures* for head reduction lengths and size of normal forms. In this paper, we refine the notion of tightness by introducing the one of **exact types**, which take into account the arities of types, while keeping a distinction between the consuming/persistent aspect of constructors. These two features of exact types are necessary to deal with the classical case in Sec. 3.2. A hd-**exact exact type** is given by the following grammar:

$$\mathsf{Ex}_{\mathsf{hd}} ::= \bullet \mid [\,] \nrightarrow \mathsf{Ex}_{\mathsf{hd}}$$

A hd-**exact intersection type** is a multiset containing only hd-exact elementary types. We also use the notations $\mathsf{Ex}_{\mathsf{hd}}(\_)$ and $\mathtt{IEx}_{\mathsf{hd}}(\mathcal{I})$ as predicates to characterize such hd-exact elementary and intersection types, respectively. A hd-**exact elementary type of length** d, written e_d, is the unique hd-exact elementary type of length d (d is the number of arrows), and is used to type hd-normal forms that can still be applied to d arguments without creating any new redex. Thus, while exact elementary types e_d ($d \geqslant 1$) are going to be assigned only to *neutral* terms, e_0 can be assigned to both *neutral* terms and *abstractions.* Notice that every hd-elementary exact type is necessarily identified with e_d for some $d \geq 0$.

We present the type system $\mathcal{X}^{\lambda}_{\mathsf{hd}}$ ($\mathcal{X}$ means e$\mathcal{X}$act) in Fig. 4, it is based on **regular** (resp. **auxiliary**) **judgments** of the form $\Gamma \vdash^{(\ell,f)} t : \sigma$ (resp. $\Gamma \Vdash^{(\ell,f)} t : \mathcal{I}$), where the **counters** ℓ and f are integers:

- ℓ counts the number of β-steps of the head strategy, this is done by counting the number of *consuming* abstractions.
- f counts the size of the normal form, and this is done by counting the rules typing constructors that are *persistent*.

$$\frac{}{x : [\sigma] \vdash^{(0,1)} x : \sigma}\ (\mathsf{ax})$$

$$\frac{\Gamma; x : \mathcal{I} \vdash^{(\ell,f)} t : \sigma}{\Gamma \vdash^{(\ell+1, f-\mathsf{card}(\mathcal{I}))} \lambda x.t : \mathcal{I} \rightarrow \sigma}\ (\mathsf{abs})$$

$$\frac{\Gamma; x : \mathcal{I} \vdash^{(\ell,f)} t : \mathsf{Ex}_{\mathsf{hd}} \qquad \mathtt{IEx}_{\mathsf{hd}}(\mathcal{I})}{\Gamma \vdash^{(\ell,f+1)} \lambda x.t : \bullet}\ (\bullet)$$

$$\frac{(\Gamma_k \Vdash^{(\ell_k,f_k)} t : \sigma_k)_{k\in K}}{\wedge_{k\in K}\Gamma_k \Vdash^{+_{k\in K}(\ell_k,f_k)} t : [\sigma_k]_{k\in K}}\ (\wedge)$$

$$\frac{\Gamma_t \vdash^{(\ell_t,f_t)} t : \mathcal{I} \Rightarrow \sigma \qquad \Gamma_u \Vdash^{(\ell_u,f_u)} u : \mathcal{I}}{\Gamma_t \wedge \Gamma_u \vdash^{(\ell_t+\ell_u, f_t+f_u+\#_{\mathsf{p}}(\mathcal{I}\Rightarrow\sigma))} t\,u : \sigma}\ (\mathsf{app})$$

Figure 4. System $\mathcal{X}^{\lambda}_{\mathsf{hd}}$: Exact Measures for the λ-Calculus

Let us now discuss a few aspects of the rules:

- The (ax)-rule adds 1 to the second counter f, which is designed to measure the size of the normal form. However, some variables are substituted during evaluation (they are consumed), and so their contribution to the value of f must be removed later. This is why (abs) decreases f by $\mathtt{card}(\mathcal{I})$.
- The ($\bullet$)-rule types abstractions that will never be applied during evaluation. Indeed, their type is *not* an arrow type.
- The (abs)-rule types abstractions that will be consumed during β-evaluation (so that a consuming arrow is introduced). That is why the first counter of the rule contributes to ℓ by adding 1, while the second counter removes $\mathtt{card}(\mathcal{I})$, which is the number of typed variables x bound by the abstraction that will be consumed during evaluation. Notice that there is no equivalent rule introducing persistent arrows, they can only be introduced via the (ax)-rule.
- The (app)-rule does not increase the number of evaluation steps: they are already recorded by the abstraction rule (abs). An application only increases counter f –representing the size of the head normal form– when the arrow type on the left premise is persistent.

Notice that, thanks to our consuming/persistent duality for arrows, there is *only one* rule to type all applications, in contrast to [Accattoli et al. 2018].

Exact derivations. Not all the derivations of system $\mathcal{X}^{\lambda}_{\mathsf{hd}}$ capture the exact measures of their subject. These exact measures are captured when the judgments are *exact*:

Definition 2.3 (Exact judgments and derivations)**.** A judgment $\Gamma \vdash t : \sigma$ is **exact** when σ and $\Gamma(x)$ (for all x) are exact. An **exact derivation** concludes with an exact judgment, in which case the typed term is said to be **exacty typable**.

A first observation to understand how *exact* derivations capture exact measures is that exact types ensure that the arguments of HNFs are never typed. Indeed, the head variable is always typed with an exact type, whose domain is empty. For instance, let t_1 and t_2 be two terms and consider the exact type $\tau = [\,] \nrightarrow [\,] \nrightarrow \bullet$ and the derivation below.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : [\tau] \vdash^{(0,1)} x : \tau}\ (\mathsf{ax}) \qquad \dfrac{}{\Vdash^{(0,0)} t_1 : [\,]}\ (\wedge)}{x : [\tau] \vdash^{(0,1+0+1=2)} x\,t_1 : [\,] \nrightarrow \bullet}\ (\mathsf{app}) \qquad \Vdash^{(0,0)} t_2 : [\,]}{x : [\tau] \vdash^{(0,2+0+1=3)} x\,t_1\,t_2 : \bullet}\ (\mathsf{app})}{x : [\tau] \vdash^{(0,3+1=4)} \lambda y.x\,t_1\,t_2 : \bullet}\ (\bullet)$$

The second counter increased in the two app-rules comes from $\#_{\mathsf{p}}(\tau) = \#_{\mathsf{p}}([\,] \nrightarrow \bullet) = 1$. The typed term $\lambda y.x\,t_1\,t_2$ is a HNF whose hd-size is 4. Thus, the final counter $(0, 4)$ has the expected value. Observe how the operator $\#_{\mathsf{p}}\_$, counting one for each persistent arrow, ensures that the number of (persistent) application constructors is computed correctly.

Another interesting example of exact derivation is given in Fig. 5 for the term $t = (\lambda x.x\,x)(\lambda z.z)$, whose HNF is $\mathtt{I}$, obtained in 2 hd-steps: $t \rightarrow_{\mathsf{hd}} (\lambda z.z)(\lambda z.z) \rightarrow_{\mathsf{hd}} \mathtt{I}$. Since $|\mathtt{I}|_{\mathsf{hd}} = 2$, the expected counter is $(2, 2)$, which is indeed the one obtained in Fig. 5. This time, the exact derivation has no persistent arrow, simply because the HNF does not contain any application. As a side remark, observe how the non-idempotent machinery works. During reduction the term $\lambda z.z$ is duplicated: one copy is persistent and the other one is consuming. Non-idempotency allows distinguishing in the type derivation the different behaviours of the copies of $\lambda z.z$ during reduction. More generally, non-idempotent types give a natural mechanism to type a subterm as many times as it is duplicated during evaluation, thus making it possible to discern the nature of all its different copies, *before* starting computation. This is why persistence and consumption, which are dynamic notions, can be captured by non-idempotent intersection types.

The following result can be seen as a reformulation of [Accattoli et al. 2018] in our new typing framework:

Theorem 2.4 (Exact Measures for Head Reduction)**.** *The term t is exactly-typable with counters (ℓ, f) (i.e. $\triangleright\Gamma \vdash^{(\ell,f)} t : \sigma$ is exact) iff the head strategy terminates on t in ℓ β-steps in a head normal form of size f.*

This theorem also relies on a quantitative subject reduction property: one may check that a *head* reduction step

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:[[\bullet]\to\bullet]\vdash^{(0,1)} x:[\bullet]\to\bullet}\,\mathsf{(ax)} \qquad \dfrac{\dfrac{}{x:[\bullet]\vdash^{(0,1)} x:\bullet}\,\mathsf{(ax)}}{x:[\bullet]\vdash^{(0,1)} x:[\bullet]}\,(\wedge)}{x:[\bullet,[\bullet]\to\bullet]\vdash^{(0,1+1+0=2)} x\,x:\bullet}\,\mathsf{(app)}}{\vdash^{(1,2-2=0)} \lambda x.x\,x:[\bullet,[\bullet]\to\bullet]\to\bullet}\,\mathsf{(abs)} \qquad \dfrac{\dfrac{\dfrac{}{z:[\bullet]\vdash^{(0,1)} z:\bullet}\,\mathsf{(ax)}}{\vdash^{(1,1-1=0)} \lambda z.z:[\bullet]\to\bullet}\,\mathsf{(abs)} \qquad \dfrac{\dfrac{}{z:[\bullet]\vdash^{(0,1)} z:\bullet}\,\mathsf{(ax)}}{\vdash^{(0,1+1=2)} \lambda z.z:\bullet}\,(\bullet)}{\Vdash^{(1,2)} \lambda z.z:[[\bullet]\to\bullet,\bullet]}\,(\wedge)}{\vdash^{(1+1=2,0+2+0=2)} (\lambda x.x\,x)(\lambda z.z):\bullet}\,\mathsf{(app)}$$

Figure 5. An Example of Exact Derivation in System $\mathcal{X}^{\lambda}_{\mathsf{hd}}$

always decreases the first counter ℓ by 1 (the situation will be trickier for leftmost-outermost and maximal evaluation in Sec. 4.3). Moreover, as noted in the above type derivation of $\lambda y.x\,t_1\,t_2$, exact judgments ensure that the arguments of head variables always remain untyped, so that the second counter f exactly captures the hd-size of the HNF.

Towards Weak and Strong Normalization. Other standard notions of normalization associated to the λ-calculus are *weak* and *strong* normalization, related, respectively, to the *leftmost-outermost* and the *maximal* deterministic reduction strategies. We will come back to the formal definitions of these notions in Sec. 4. As for now, it is worth mentioning that by just modifying some of the previous definitions, it is possible to obtain exact measures for these strategies too by using the same method. Moreover, this can be done within a **parametrized framework** which provides a unified presentation for the three standard strategies (head, leftmost, maximal) of λ-calculus, as well as to their associated notions of (head, weak, strong) normalization. The parametric formalism that we propose in Sec. 4.2 allows factorizing the various characterizations in the literature, as well as their proofs, in a concise and elegant way. For now, let us move forward by extending the method to the $\lambda\mu$-calculus.

3 Classical Computation

This section extends the technique of Sec. 2 to the head strategy of the $\lambda\mu$-calculus. We then introduce *exact* types: while tight types as defined in [Accattoli et al. 2018] cannot capture exact measures in classical logic, the types defined in Sec. 2 can be extended to do so in the $\lambda\mu$-calculus.

We consider a countable infinite set of **variables** $x, y, z, \ldots$ (resp. **names** $\alpha, \beta, \gamma, \ldots$). Objects, terms and commands of the $\lambda\mu$-calculus are given by the following grammars:

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

The grammar extends λ-terms with two new constructors: commands $[\alpha]t$ ("call continuation α with t as argument") and μ-abstractions $\mu\alpha.c$ ("record the current continuation as α and continue as c"). We say that t is **named** α in the command $[\alpha]t$. The predicate $\neg\mathsf{abs}(\_)$ now characterizes all the objects that are neither λ nor μ-abstractions.

Free and **bound variables** of objects are defined as expected, in particular $\mathsf{fv}(\mu\alpha.c) := \mathsf{fv}(c)$ and $\mathsf{fv}([\alpha]t) := \mathsf{fv}(t)$. **Free names** of objects are also defined as expected, in particular $\mathsf{fn}(\mu\alpha.c) := \mathsf{fn}(c) \setminus \{\alpha\}$ and $\mathsf{fn}([\alpha]t) := \mathsf{fn}(t) \cup \{\alpha\}$. **Bound names** are defined accordingly.

We work modulo α**-conversion**, *i.e.* renaming of bound variables/names, *e.g.* $[\delta]\mu\alpha.[\alpha]\lambda x.xz \equiv [\delta]\mu\beta.[\beta]\lambda y.yz$. **Application** of the **substitution** $\{x/u\}$ to the object o, written $o\{x/u\}$, may require α-conversion in order to avoid capture of free variables/names, and it is defined as expected. **Application** of the **replacements** $\{\alpha\backslash\!\!\backslash u\}$ to the object o, written $o\{\alpha\backslash\!\!\backslash u\}$, passes the term u as an argument to any command of the form $[\alpha]t$, *i.e.* every subcommand $[\alpha]t$ in o is replaced by $[\alpha](t\,u)$. This operation may require α-conversion. Thus, *e.g.* if $\mathtt{I} = \lambda z.z$, then $(x(\mu\alpha[\alpha]xy))\{x/\mathtt{I}\} = \mathtt{I}(\mu\alpha[\alpha]\mathtt{I}y)$, and $([\alpha]x(\mu\beta.[\alpha]y))\{\alpha\backslash\!\!\backslash\mathtt{I}\} = [\alpha](x(\mu\beta.[\alpha]y\mathtt{I}))\mathtt{I}$.

The $\lambda\mu$-calculus is given by the set of $\lambda\mu$-objects and the **reduction relation** $\to_{\lambda\mu}$, sometimes simply written $\to$, which is the closure by *all* contexts of the two rules:

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

defined by means of the substitution and replacement application notions presented above. A reduction step $\to_{\beta}$ (resp. $\to_{\mu}$) is **erasing** if $x \notin \mathsf{fv}(t)$ ($\alpha \notin \mathsf{fn}(c)$).

As a side remark, the operational semantics of the calculus often includes the rules $[\alpha]\mu\beta.c \to_{\rho} c\{\alpha/\beta\}$ or $\mu\alpha.[\alpha]c \to_{\eta} c$, when $\alpha \notin \mathsf{fv}(c)$. But the left and right-hand side terms of these rules can be seen as *structurally equivalent* terms from the proof-net perspective, as enlightened by [Laurent 2003b]. Therefore, they are not really contributing to the logical meaning of proof/term transformations, and this is why we have preferred to keep the reduction system as simple as possible.

Head Normal Forms. HNF s are extended by adding the predicates in Fig. 6 to those in Fig. 1. The hd**-size** function is extended to $\lambda\mu$-HNFs by: $|[\alpha]t|_{\mathsf{hd}} := |t|_{\mathsf{hd}}$ and $|\mu\alpha.c|_{\mathsf{hd}} := |c|_{\mathsf{hd}}+1$. Notice that we choose to ignore names of commands in the size: setting $|[\alpha]|_{\mathsf{hd}}t := |t|_{\mathsf{hd}} + 1$ would have been possible without compromising the results to come.

We also extend the head strategy for $\lambda\mu$-objects by adding the rules in Fig. 7 to those in Fig. 2. The $\lambda\mu$-object o is **head terminating** if $\rho : o \to^{*}_{\mathsf{hd}} o'$ and $o' \not\to_{\mathsf{hd}}$.

$$\frac{\mathsf{norm}_{\mathsf{hd}}(t)}{\mathsf{norm}_{\mathsf{hd}}([\alpha]t)} \qquad \frac{\mathsf{norm}_{\mathsf{hd}}(c)}{\mathsf{norm}_{\mathsf{hd}}(\mu\alpha.c)}$$

Figure 6. Extending Head Neutral and Normal Terms

$$\frac{}{(\mu\alpha.c)u \to_{\mathsf{hd}} \mu\alpha.c\{\alpha\backslash\!\!\backslash u\}}\ (\mu)$$

$$\frac{c \to_{\mathsf{hd}} c'}{\mu\alpha.c \to_{\mathsf{hd}} \mu\alpha.c'}\ (\mathsf{abs}\text{-}\mu) \qquad \frac{t \to_{\mathsf{hd}} t'}{[\alpha]t \to_{\mathsf{hd}} [\alpha]t'}\ (\mathsf{com})$$

Figure 7. Extending the Head Evaluation Strategy

3.1 Upper Bounds

We now slightly reformulate the type system in [Kesner and Vial 2017], written here $\mathscr{U}^{\lambda\mu}_{\mathsf{hd}}$, or simply $\mathscr{U}_{\mathsf{hd}}$, which provides upper bounds for the head strategy in the $\lambda\mu$-calculus.

The system is based on *intersection* and *union* types. Intersection is to type distinct occurrences of the same variable differently. Dually, union is used to type distinct occurrences of the same name differently. Indeed, while free and bound variables are typed with intersection types, free and bound names (and thus μ-abstractions, and actually all terms) must be assigned union types. Our intersection and union operators are *associative*, *commutative*, and *non-idempotent*, so that they can be represented as multisets. Consequently, we use two different kinds of multisets: $[\,]$ for intersection types (IT) and $\langle\,\rangle$ for union types (UT). Types are defined by mutual induction:

$$\begin{array}{lll}
\textbf{(UT)} & \mathcal{U},\mathcal{V} ::= & \langle\rangle \mid \mathcal{N} \\
\textbf{(Non-Empty UT)} & \mathcal{N} \quad ::= & \langle\sigma_k\rangle_{k\in K}\ (K \neq \emptyset) \\
\textbf{(IT)} & \mathcal{I},\mathcal{J} ::= & [\mathcal{N}_k]_{k\in K} \\
\textbf{(Elementary Types)} & \sigma,\tau \quad ::= & \bullet \mid \mathcal{I} \to \mathcal{N} \mid \mathcal{I} \twoheadrightarrow \mathcal{N}
\end{array}$$

Notice that union types used to build intersections are never empty. This will ensure that the empty union type $\langle\,\rangle$ cannot be assigned to any term. However, the empty intersection type $[\,]$ may occur anywhere as the domain of an arrow. We write $\langle\sigma\rangle_n$ ($n \geq 1$), to denote a multiset $\langle\,\rangle$ containing n occurrences of the same type σ, and we may use in particular simply $\vec{\bullet}$ to denote *any* multiset $\langle\bullet\rangle_k$ ($k \geq 1$). Moreover, $\langle\bullet\rangle$ may be written $\bar{\bullet}$. A special category of union types is given by the **blind types**, defined by the grammar:

$$\mathcal{B} ::= \vec{\bullet} \mid \langle[\,] \to \mathcal{B}\rangle$$

The **arity** of types and union types is defined by induction: $\mathsf{ar}(\bullet) = 0$, $\mathsf{ar}(\mathcal{I} \to \mathcal{N}) = \mathsf{ar}(\mathcal{N}) + 1$, $\mathsf{ar}(\mathcal{I} \twoheadrightarrow \mathcal{N}) = \mathsf{ar}(\mathcal{N}) + 1$; $\mathsf{ar}(\langle\sigma_k\rangle_{k\in K}) := \min_{k\in K}\ \mathsf{ar}(\sigma_k)$. Thus, the arity of a type counts the number of application arguments it can be fed with. The **(non-deterministic) choice operator** $\_^*$ on union types is defined as follows: if $\mathcal{U} \neq \langle\,\rangle$, then $\mathcal{U}^* = \mathcal{U}$, otherwise $\langle\,\rangle^* = \mathcal{B}$, where $\mathcal{B}$ is any *arbitrary* blind union type (we make a slight abuse of vocabulary by calling $\_^*$ an operator rather than a relation, and this without any technical consequence).

Left-hand sides of applications are generally typed with unions of the form $\mathcal{F} = \langle\mathcal{I}_1 \Rightarrow \mathcal{N}_1, \ldots, \mathcal{I}_k \Rightarrow \mathcal{N}_k\rangle$, called **function types**. To reason about the (app)-rule in a more synthetic way, we set $\mathsf{dom}(\mathcal{F}) = \wedge_{k=1\ldots n}\mathcal{I}_k$ and $\mathsf{codom}(\mathcal{F}) = \vee_{k=1\ldots n}\mathcal{N}_k$, the **domain** and **codomain** of $\mathcal{F}$. **Variable assignments** Γ are total functions from *variables* to *intersection* types and **name assignments** Δ are total functions from *names* to (possibly empty) *union* types. They come along with similar notations used before for the λ-calculus in Sec. 2, *e.g.* $\mathsf{supp}(\Delta)$, $\Delta(\alpha)$, $\Delta \vee \Delta'$, $\Delta;\Delta'$, $\Delta \subseteq \Delta'$, $\alpha : \mathcal{U}$, etc.

The syntax directed rules of the type system $\mathscr{U}_{\mathsf{hd}}$ are presented in Fig. 8. Terms can never be assigned *empty* union types (see [Kesner and Vial 2017; Laurent 2004] for a discussion), which is ensured in particular by the choice operator in the $(\mathsf{mu}_{\mathscr{U}})$-rule. Notice again that this system does not make any specific use of persistent arrows: they are still not necessary to obtain upper bounds.

Theorem 3.1 (Upper Bounds for Head Reduction,[Kesner and Vial 2017])**.** *A $\lambda\mu$-object o is $\mathscr{U}_{\mathsf{hd}}$-typable iff o is head-normalizing iff the head strategy terminates on o. Moreover, the "size"* (cf. *discussion below) of the derivation typing o gives an upper bound to the sum $\ell + m + f$, where ℓ, m are respectively the number of β- and μ-steps in the head-evaluation of t and f is the size of the* HNF *of t.*

A quantitative subject reduction property for the λ-calculus still holds for a suitable definition of derivation size (the same we refer to in the above theorem), *e.g.* the one in [Kesner and Vial 2017]. But such a notion cannot be a simple extension of that for the λ-calculus, which simply counts the number of judgments of a derivation. Indeed, μ-reduction creates several ($n \geqslant 0$) application nodes. As argued in *ibid.*, to obtain a quantitative measure for the $\lambda\mu$-calculus, **the size of a (derivation typing a) μ-abstraction $\mu\alpha.c$ must also measure the types that are stored by the name α inside the command c.**

Let us shortly explain why. Consider a μ-redex $(\mu\alpha.c)s$, where α has type $A \to B$. Then all subterms v named α are of type $A \to B$, they are replaced after μ-reduction by applications of the form $v\,s$, of type B, smaller than $A \to B$. That is, the type of α changes along μ-evaluation in the simply typed case. Thus, the size of the new type stored by α has been reduced (in particular, its arity has decreased by 1), and this suggests how a quantitative measure can be defined for the $\lambda\mu$-calculus. We will use this observation in Sec. 3.2 to define the counters measuring μ-evaluation within the forthcoming type system capturing exact measures.

As a consequence, arities must be measured to obtain exact bounds in a classical framework, but the *tight types* of [Accattoli et al. 2018] cannot measure arities, and thus, cannot be extended to the $\lambda\mu$-calculus. Indeed, tight types

$$\frac{}{x : [\mathcal{N}] \vdash x : \mathcal{N} \mid \emptyset}\ (\mathtt{ax}) \qquad \frac{\Gamma; x : \mathcal{I} \vdash t : \mathcal{N} \mid \Delta}{\Gamma \vdash \lambda x.t : \langle \mathcal{I} \to \mathcal{N} \rangle \mid \Delta}\ (\mathtt{abs}) \qquad \frac{\Gamma \vdash t : \mathcal{N} \mid \Delta}{\Gamma \vdash [\alpha]t \mid \Delta \vee \{\alpha : \mathcal{N}\}}\ (\mathtt{c}) \qquad \frac{\Gamma \vdash c \mid \Delta; \alpha : \mathcal{U}}{\Gamma \vdash \mu\alpha.c : \mathcal{U}^* \mid \Delta}\ (\mathtt{mu}_{\mathscr{U}})$$

$$\frac{(\Gamma_k \vdash t : \mathcal{N}_k \mid \Delta_k)_{k \in K}}{\wedge_{k \in K} \Gamma_k \Vdash t : [\mathcal{N}_k]_{k \in K} \mid \vee_{k \in K} \Delta_k}\ (\wedge) \qquad \frac{\Gamma_t \vdash t : \mathcal{F} \mid \Delta_t \quad \Gamma_u \Vdash u : \mathsf{dom}(\mathcal{F}) \mid \Delta_u}{\Gamma_t \wedge \Gamma_u \vdash t\,u : \mathsf{codom}(\mathcal{F}) \mid \Delta_t \vee \Delta_u}\ (\mathtt{app})$$

Figure 8. Upper System $\mathscr{U}_{\mathsf{hd}}$ for the $\lambda\mu$-Calculus (Head Evaluation Strategy)

are specified by means of two type constants $\mathtt{neut}$ and $\mathtt{abs}$ so that $t : \mathtt{neut}$ is derivable when t reduces to a *neutral* $\mathtt{HNF}$, and the system is equipped with a second rule to type (persistent) applications:

$$\frac{\Gamma \vdash t : \mathtt{neut}}{\Gamma \vdash t\,u : \mathtt{neut}}$$

This rule does not record how many arguments a head variable may be fed with (intuitively, $\mathtt{neut}$ behaves like a type with *infinitely* many persistent arrows), and thus, the resulting system cannot provide a decreasing measure for μ-reduction steps.

Before explaining how we capture exact measures in the $\lambda\mu$-calculus (next section), let us comment on the relation between our system and Laurent's encoding of classical natural deduction into Polarized Linear Logic.

Connection with Linear Logic. The translation of $\lambda\mu$-calculus into Polarized Linear Logic (PLL) in [Laurent 2003b] puts in evidence the non-trivial behaviors of the calculus. More precisely, duplication caused by the substitution of variables (resp. the replacement of names) in the term language is materialized by duplication of boxes (resp. tensor products) on the logical side, modelled by intersection (resp. union) multisets in our type system. On the other side, while the *arity* of types is used here to count μ-reduction steps, i.e. the length of the list of arguments of a μ-abstraction, the number of copies of tensors in PLL is rather captured by the notion of cardinality (different from arity) of union types. A last observation is that contraction is *admissible* for negative types in PLL, *i.e.* any negative type may be duplicated during cut elimination, while in our system, intersection/union types are strictly non-idempotent, so that they are *split* (instead of duplicated) during reduction.

3.2 Exact Measures

This section introduces the typing system $\mathcal{X}^{\lambda\mu}_{\mathsf{hd}}$, simply written $\mathcal{X}_{\mathsf{hd}}$, which refines and extends $\mathscr{U}_{\mathsf{hd}}$ so that exact derivations in $\mathcal{X}_{\mathsf{hd}}$ will capture exact measures for the head evaluation in the $\lambda\mu$-calculus. The notion of exact types in a classical framework is generalized as follows. A hd-**exact elementary type** is now given by the following grammar:

$$\mathsf{Ex}_{\mathsf{hd}} ::= \bullet \mid [\,] \not\twoheadrightarrow \langle \mathsf{Ex}_{\mathsf{hd}} \rangle$$

We use the notation e_d ($d \geq 0$) for an exact elementary type of arity d (d persistent arrows). An hd-**exact intersection type** is of the form $[\langle\sigma_1\rangle, \ldots, \langle\sigma_m\rangle]$, where every σ_i is a hd-exact elementary type. Thus, a hd-exact intersection type is an intersection of *singleton union* types. A hd-**exact union type** is of the form $\langle\sigma_1, \ldots, \sigma_n\rangle$, where every σ_i is a hd-exact elementary type. We may write $\mathsf{IEx}_{\mathsf{hd}}$ (resp. $\mathsf{UEx}_{\mathsf{hd}}$) for a hd-exact intersection (resp. union) type. We use the predicate $\mathsf{Ex}_{\mathsf{hd}}(\_)$ to characterize hd-exact types. Note that these notions generalize the ones introduced for the intuitionistic case in Sec. 2.2.

In order to state the main properties of system $\mathcal{X}_{\mathsf{hd}}$, we first need to generalize the notion of *exact* derivation. A variable (resp. name) **assignment** Γ (resp. Δ) is hd-**exact** when $\Gamma(x)$ (resp. $\Delta(\alpha)$) is a hd-exact intersection (resp. union) type for all $x \in \mathsf{supp}(\Gamma)$ (resp. all $\alpha \in \mathsf{supp}(\Delta)$). We then write $\mathsf{Ex}_{\mathsf{hd}}(\Gamma)$ (resp. $\mathsf{Ex}_{\mathsf{hd}}(\Delta)$). Judgments are now decorated with counters of the form (ℓ, m, f), where the new component m counts the number of μ-steps of the head strategy. A **regular** judgment $\Gamma \vdash^{(\ell,m,f)} t : \mathcal{N} \mid \Delta$ is hd-**exact** when $\mathsf{Ex}_{\mathsf{hd}}(\Gamma)$, $\mathsf{Ex}_{\mathsf{hd}}(\mathcal{N})$, $\mathsf{Ex}_{\mathsf{hd}}(\Delta)$. A **command** judgment $\Gamma \vdash^{(\ell,m,f)} c \mid \Delta$ is hd-**exact** when $\mathsf{Ex}_{\mathsf{hd}}(\Gamma)$ and $\mathsf{Ex}_{\mathsf{hd}}(\Delta)$. This extends to auxiliary judgments. A **derivation** is hd-**exact** when it concludes with a hd-exact judgment, in which case the typed object is said to be hd-**exactly typable**. Exactness of derivations pertains only to the conclusion of a derivation, so it is not a global condition on the derivation.

The definition of the rules of $\mathcal{X}_{\mathsf{hd}}$ in Fig. 9 is subtle, and makes use of different operators. The **(non-deterministic) activation operator** on elementary and union types is defined by: $\bullet^{\uparrow} = \bullet$, $(\mathcal{I} \to \mathcal{N})^{\uparrow} = \mathcal{I} \to \mathcal{N}^{\uparrow}$, $(\mathcal{I} \not\twoheadrightarrow \mathcal{N})^{\uparrow} = \mathcal{I} \to \mathcal{N}^{\uparrow}$, $\langle\,\rangle^{\uparrow} = \mathscr{B}$ (an *arbitrary* blind union type), and $\langle\sigma_k\rangle^{\uparrow}_{k \in K} = \langle\sigma^{\uparrow}_k\rangle_{k \in K}$ when $K \neq \emptyset$. Intuitively, the activation operator chooses a blind type for an empty union type, and replaces the top-level (*i.e.* those which are not nested in an intersection) persistent arrows by consuming ones. As we see in the $(\mathtt{mu}_{\mathcal{X}})$-rule, the activation operator prevents types of μ-abstractions to be empty. The **persistence order** of a non-empty union type $\mathcal{N} = \langle\sigma_k\rangle_{k \in K}$ counts the number of *root* persistent arrows and is defined by $\#_{\mathsf{p}}\mathcal{N} = \mathsf{card}\{k \in K \mid \exists \mathcal{I},\ \mathcal{N}',\ \sigma_k = \mathcal{I} \not\twoheadrightarrow \mathcal{N}'\}$, *e.g.* $\#_{\mathsf{p}}\langle[\bar{\bullet}] \not\twoheadrightarrow \langle[\vec{\bullet}] \not\twoheadrightarrow \vec{\bullet}\rangle\rangle = 1$ and $\#_{\mathsf{p}}\langle[\vec{\bullet}] \not\twoheadrightarrow \bar{\bullet}, \mathcal{I} \to \mathcal{N}, \mathcal{I} \not\twoheadrightarrow \mathcal{N}\rangle = 2$. The **cumulated persistence** of types counts the number of top-level persistent arrows, which will in turn help count persistent applications that μ-abstractions create during μ-evaluation, and is defined as follows: $\mathsf{c}_{\mathsf{p}}(\bullet) = 0$, $\mathsf{c}_{\mathsf{p}}(\mathcal{I} \to \mathcal{N}) = \mathsf{c}_{\mathsf{p}}(\mathcal{N})$,

$\mathsf{c_p}(\mathcal{I} \twoheadrightarrow \mathcal{N}) = \mathsf{c_p}(\mathcal{N}) + 1$ and $\mathsf{c_p}(\langle \sigma_k \rangle_{k \in K}) = +_{k \in K} \mathsf{c_p}(\sigma_k)$. All these operators are used in the typing rules. The $(\mathtt{mu}_{\mathscr{U}})$-rule of system $\mathscr{U}_{\mathsf{hd}}$ in Fig. 8 differs from the $(\mathtt{mu}_{\mathcal{X}})$-rule of system $\mathcal{X}_{\mathsf{hd}}$ in Fig. 9: the latter features the activation operator which transforms top-level persistent arrows into consuming ones. This is not a detail: the introduction of two kinds of arrows and the activation operator on persistent arrows are the key features allowing system $\mathcal{X}_{\mathsf{hd}}$ to provide exact measures. In what follows, we discuss these tools in detail, but let us also mention before some other salient features of the $(\mathtt{mu}_{\mathcal{X}})$-rule.

Counting Mechanisms of the $(\mathtt{mu}_{\mathcal{X}})$***-Rule.*** The $(\mathtt{mu}_{\mathcal{X}})$-rule is the key ingredient of the typing system for the classical case, it concludes with the counter

$$(\ell, m + \mathsf{ar}(\mathcal{N}), f + 1 + \mathsf{c_p}(\mathcal{U}))$$

where $\mu\alpha.c$ is typed with $\mathcal{N} = \mathcal{U}^{\uparrow}$. This means that the operator $\_^{\uparrow}$ is used only *once* on $\mathcal{U}$: it produces a new *non-empty* type $\mathcal{N}$ (which is blind) if $\mathcal{U} = \langle\,\rangle$, and activates the top-level persistent arrows of $\mathcal{U}$ when $\mathcal{U} \neq \langle\,\rangle$. Notice, however, that the output of this possibly non-deterministic choice is used *twice* in the conclusion of the rule (in the second counter, and in the type of $\mu.c$). Let us explain now why this rule appropriately counts consuming and persistent constructors, before discussing the crucial role of the activation operator.

- ♦ The size of μ-abstractions was defined as $|\mu\alpha.c|_{\mathsf{hd}} = |c|_{\mathsf{hd}} + 1$: hence the +1 on the third counter of the $(\mathtt{mu}_{\mathcal{X}})$-rule.
- ♦ μ-reduction may create persistent applications: for instance in $(\mu\alpha.[\alpha]x)\mathtt{I} \rightarrow \mu\alpha.[\alpha](x\,\mathtt{I})$. Therefore, we use the operator $\mathsf{c_p}(\_)$ in the third counter to record the number of persistent applications that will be created by each typed μ-abstraction. This aspect of the type system is illustrated in the forthcoming Example 4.1.
- ♦ The arity operator $\mathsf{ar}(\_)$ in the second counter captures how many times the μ-abstraction will be evaluated in a μ-reduction. More precisely, the activation operator $\mathcal{N} = \mathcal{U}^{\uparrow}$ (where $\alpha : \mathcal{U}$) used to type $\mu\alpha.c$ raises a *blind* union type when $\mathcal{U}$ is empty. As a consequence, $\mu\alpha.c$ is always typed with a non-empty union, whose arity must be recorded in the second counter, so that it is possible to count the number of μ-evaluation steps that are generated by this μ-abstraction $\mu\alpha.c$, whether they are erasing or not. This explains the use of $\mathsf{ar}(\mathcal{N})$ in the second counter of the $(\mathtt{mu}_{\mathcal{X}})$-rule.

As previously mentioned, a *blind* type $\mathcal{N}$ is raised by $\_^{\uparrow}$ when the type $\mathcal{U}$ of α is $\langle\,\rangle$, *e.g.* when α occurs in some untyped subterms of c only. When this blind type $\mathcal{N}$ is also functional (*i.e.* $\mathcal{N} \neq \vec{\bullet}$), its domain is empty: this ensures that the future arguments of the μ-abstraction $\mu\alpha.c : \langle [\,] \rightarrow \ldots \langle [\,] \rightarrow \vec{\bullet} \rangle \rangle$ are also going to be left untyped. This is essential to guarantee subject reduction for *erasing* evaluation steps.

Activation Operator. In order to capture exact measures in the classical setting, the crucial point is that persistent arrows can be *activated*, featured in the $(\mathtt{mu}_{\mathcal{X}})$-rule by means of the activation operation $\_^{\uparrow}$. Indeed, we introduced two kinds of arrows to capture arities, which are necessary to deal with classical logic, as explained in Sec. 3.1. However, as it turns out, even when α has a non-empty type $\mathcal{N}$ while typing c, the abstraction $\mu\alpha.c$ cannot have this same type $\mathcal{N}$ in all generality, as it is the case in system $\mathscr{U}_{\mathsf{hd}}$ or in [Laurent 2004; Parigot 1992]. Indeed, the top-level persistent arrows of $\mathcal{N}$ should be transformed into consuming ones.

But why transforming persistent arrows into consuming is necessary? To explain that, let us suppose that the typing rule for the μ-abstraction *does not* change persistent arrows into consuming ones, and let us call $(\mu^{\dagger})$ the corresponding modified typing rule. We can then construct the following exact derivation Φ, where $\gamma, \beta \neq \alpha$ are fresh names (recall notation e_d from Sec. 3.2).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : [\langle \mathsf{e}_2 \rangle] \vdash^{(0,0,1)} x : \langle \mathsf{e}_2 \rangle \mid \emptyset}\,(\mathsf{ax})}{x : [\langle \mathsf{e}_2 \rangle] \vdash^{(0,0,2)} x\,z : \langle \mathsf{e}_1 \rangle \mid \emptyset}\,(\mathsf{app})}{x : [\langle \mathsf{e}_2 \rangle] \vdash^{(0,0,2)} [\alpha]x\,z \mid \alpha : \langle \mathsf{e}_1 \rangle}\,(\mathsf{c})}{x : [\langle \mathsf{e}_2 \rangle] \vdash^{(0,0,2+1)} \mu\gamma.[\alpha]x\,z : \vec{\bullet} \mid \alpha : \langle \mathsf{e}_1 \rangle}\,(\mu^{\dagger})}{x : [\langle \mathsf{e}_2 \rangle] \vdash^{(0,0,3)} [\beta](\mu\gamma.[\alpha]x\,z) \mid \alpha : \langle \mathsf{e}_1 \rangle, \beta : \vec{\bullet}}\,(\mathsf{c})}{x : [\langle \mathsf{e}_2 \rangle] \vdash^{(0,1,3+1+1)} \mu\alpha.[\beta](\mu\gamma.[\alpha]x\,z) : \langle \mathsf{e}_1 \rangle \mid \beta : \vec{\bullet}}\,(\mu^{\dagger})$$

The counter of the top $(\mu^{\dagger})$-rule would be justified by $\mathsf{ar}(\vec{\bullet}) = 0$ and $\mathsf{c_p}(\langle\,\rangle) = 0$. The counter of the bottom $(\mu^{\dagger})$-rule would be justified by $\mathsf{ar}(\langle \mathsf{e}_1 \rangle) = 1$ (recall that $(\mu^{\dagger})$ does not change persistent to consuming arrows) and $\mathsf{c_p}(\langle \mathsf{e}_1 \rangle) = 1$. Remark that Φ concludes with an exact judgment, however, the resulting final counter $(0, 1, 5)$ turns out to be wrong: it does not only count one potential μ-step for a hd-normal form in an *exact* judgment, but the hd-size of this normal form is also wrong (5 instead of 4). This explains why the activation operator should change persistent to consuming arrows, so that μ-abstractions are never typed with functional persistent *exact* types. However, thanks to the operator $\_^{\uparrow}$, using $(\mathtt{mu}_{\mathcal{X}})$ instead of $(\mu^{\dagger})$ in the previous example, the derivation would conclude with $\mu\alpha.[\beta](\mu\gamma.[\alpha]x\,z) : \langle [\,] \rightarrow \vec{\bullet} \rangle$, which is not an exact type: indeed, the rule $(\mathtt{mu}_{\mathcal{X}})$ tells us that the above derivation *does not* capture exact measures (we should have actually assigned e_1 to x instead of e_2).

3.3 Main Properties

The lemma below gives some key properties of the exact system $\mathcal{X}_{\mathsf{hd}}$. The first one is dubbed exact spreading, which is a technical observation used to prove exact subject reduction/expansion as well as soundness and completeness for $\mathtt{NF}$s. The second point states that when a $\mathtt{HNF}$ is *exactly* typed, then its hd-size is captured by the typing. The last point ensures that *all* the $\mathtt{HNF}$s are indeed exactly typable. The three points are proved by structural induction.

Lemma 3.2 (Basic Properties)**.**

$$\frac{}{x : [\mathcal{N}] \vdash^{(0,0,1)} x : \mathcal{N} \mid \emptyset}\,(\mathsf{ax}) \qquad \frac{\Gamma; x : \mathcal{I} \vdash^{(\ell,m,f)} t : \mathcal{N} \mid \Delta}{\Gamma \vdash^{(\ell+1,m,f-\mathsf{card}(\mathcal{I}))} \lambda x.t : \langle \mathcal{I} \to \mathcal{N} \rangle \mid \Delta}\,(\mathsf{abs}) \qquad \frac{\Gamma \vdash^{(\ell,m,f)} t : \mathcal{N} \mid \Delta}{\Gamma \vdash^{(\ell,m,f)} [\alpha]t \mid \Delta \vee \alpha : \mathcal{N}}\,(\mathsf{c})$$

$$\frac{\Gamma; x : \mathcal{I} \vdash^{(\ell,m,f)} t : \mathsf{UEx}_{\mathsf{hd}} \mid \Delta \qquad \mathtt{IEx}_{\mathsf{hd}}(\mathcal{I})}{\Gamma \vdash^{(\ell,m,f+1)} \lambda x.t : \bar{\bullet} \mid \Delta}\,(\bullet) \qquad \frac{\Gamma \vdash^{(\ell,m,f)} c \mid \Delta; \alpha : \mathcal{U} \qquad \mathcal{N} = \mathcal{U}^{\uparrow}}{\Gamma \vdash^{(\ell,m+\mathsf{ar}(\mathcal{N}),f+1+\mathsf{c_p}(\mathcal{U}))} \mu\alpha.c : \mathcal{N} \mid \Delta}\,(\mathsf{mu}_{\mathcal{X}})$$

$$\frac{(\Gamma_k \vdash^{(\ell_k,m_k,f_k)} t : \mathcal{N}_k \mid \Delta_k)_{k \in K}}{\wedge_{k\in K}\Gamma_k \Vdash^{+_{k\in K}(\ell_k,m_k,f_k)} t : [\mathcal{N}_k]_{k\in K} \mid \vee_{k\in K} \Delta_k}\,(\wedge) \qquad \frac{\Gamma_t \vdash^{(\ell_t,m_t,f_t)} t : \mathcal{F} \mid \Delta_t \qquad \Gamma_u \Vdash^{(\ell_u,m_u,f_u)} u : \mathsf{dom}(\mathcal{F}) \mid \Delta_u}{\Gamma_t \wedge \Gamma_u \vdash^{(\ell_t+\ell_u,m_t+m_u,f_t+f_u+\#_{\mathsf{p}}\mathcal{F})} t\,u : \mathsf{codom}(\mathcal{F}) \mid \Delta_t \vee \Delta_u}\,(\mathsf{app}_{\mathsf{hd}})$$

Figure 9. Exact System $\mathcal{X}_{\mathsf{hd}}$ for the $\lambda\mu$-calculus (Head Strategy)

1. *(Exact Spreading) Let t be a neutral* HNF *and* $\Phi \triangleright \Gamma \vdash^{(\ell,m,f)} t{:}\mathcal{N} \mid \Delta$. *If* $\mathsf{Ex}_{\mathsf{hd}}(\Gamma)$*, then* $\mathcal{N} = \langle \mathsf{e}_d \rangle$ *for some* $d \geq 0$.
2. *(Soundness of* hd*-NF) Let t be a* HNF *and* $\Phi \triangleright \Gamma \vdash^{(\ell,m,f)} t : \mathcal{N} \mid \Delta$. *Then* $\ell = m = 0$ *and* $f = |t|_{\mathsf{hd}}$.
3. *(Completeness for* hd*-NF) Let t be a* HNF. *Then t is exactly typable.*

System $\mathcal{X}_{\mathsf{hd}}$ enjoys quantitative subject reduction and expansion in the same way system $\mathcal{U}_{\mathsf{hd}}$ does. However, in the case of exact derivations, the decrease is always equal to 1. This allows us to prove the following exact characterization:

Theorem 3.3 (Exact Measures for Head Reduction)**.** *A $\lambda\mu$-object o is exactly typable with counters (ℓ, m, f) iff the head strategy terminates on o in ℓ β-steps and m μ-steps in a head normal form of size f.*

4 Beyond Head Normalization

In this section, we extend the technique introduced in Sec. 3.2 for head-evaluation to another two evaluation strategies of the $\lambda\mu$-calculus: leftmost-outermost and maximal reduction, related to weak and strong normalization, respectively. We start by defining these strategies, then we present them in a parametrized way.

Full Normal Forms. A **full normal form**, or just a **normal form (**NF**)**, is an object which does not contain any β- or μ-redex. NFs can be characterized by the predicates in Fig. 10.

$$\frac{}{\mathsf{neut}_{\mathtt{full}}(x)} \qquad \frac{\mathsf{neut}_{\mathtt{full}}(t) \quad \mathsf{norm}_{\mathtt{full}}(u)}{\mathsf{neut}_{\mathtt{full}}(tu)} \qquad \frac{\mathsf{neut}_{\mathtt{full}}(t)}{\mathsf{norm}_{\mathtt{full}}(t)}$$

$$\frac{\mathsf{norm}_{\mathtt{full}}(t)}{\mathsf{norm}_{\mathtt{full}}(\lambda x.t)} \qquad \frac{\mathsf{norm}_{\mathtt{full}}(t)}{\mathsf{norm}_{\mathtt{full}}([\alpha]t)} \qquad \frac{\mathsf{norm}_{\mathtt{full}}(c)}{\mathsf{norm}_{\mathtt{full}}(\mu\alpha.c)}$$

Figure 10. Full Neutral and Normal Forms

We measure NFs by using the function `full`**-size**: $|x|_{\mathtt{full}} := 1$, $|\lambda x.t|_{\mathtt{full}} := |t|_{\mathtt{full}} + 1$, $|tu|_{\mathtt{full}} := |t|_{\mathtt{full}} + |u|_{\mathtt{full}} + 1$, $|[\alpha]t|_{\mathtt{full}} := |t|_{\mathtt{full}}$ and $|\mu\alpha.c|_{\mathtt{full}} := |c|_{\mathtt{full}} + 1$.

Weak Normalization and Leftmost-Outermost Evaluation. A $\lambda\mu$-object o is **weakly normalizing (WN)** if there is a reduction sequence from o to a NF. When o is not a normal form, it is **reducible**, and thus it contains in particular a **leftmost-outermost (**`lo`**)** redex: the one whose abstraction is the leftmost one when o is seen as a string of characters. The `lo` **strategy** $\to_{\mathtt{lo}}$ consists in reducing this `lo` redex (a formal definition later). An object o is `lo` **terminating (LOT)** if the lo strategy terminates on o. Clearly, LOT implies WN, and, as in the hd-case, the (non-trivial) converse implication also holds, *i.e.* LOT and WN are equivalent.

Strong Normalization and Maximal Evaluation. A $\lambda\mu$-object o is **strongly normalizing (SN)** if o is not the source of an infinite reduction sequence. In particular, if o is SN, then the lo strategy terminates on o, so o is WN. The converse is not true: if $o = (\lambda x.y)\Omega$ (where Ω is the non-terminating term $(\lambda z.z\,z)(\lambda z.zz)$), then $o \to y$, so that o is WN, but o is not SN, since $\Omega \to \Omega$, and thus also $o \to o$. It turns out that there is a *deterministic* evaluation strategy producing such a longest (and possibly infinite) sequence, which is called the **maximal evaluation strategy**, written $\to_{\mathtt{mx}}$. In particular, this strategy computes a NF when o is SN but does not terminate when o is not SN. It extends the maximal strategy of the λ-calculus [van Raamsdonk et al. 1999] in that it performs an erasing reduction step only when the argument has been fully evaluated. Otherwise, there is a risk to erase a non-SN term. For instance, if $t = (\lambda x.y)u \to_\beta y$, then t is SN iff u is SN, which means that the erasing reduction step $t \to_\beta y$ causes a **semantic loss**[1] **of information**. This is why the maximal strategy must first evaluate the subterm u before erasing it. In other words, the maximal evaluation refines the leftmost one by adding new steps to evaluate erasable arguments before consumption, while leftmost *blindly* erases them. For instance, if $o = (\mu\alpha.[\beta]x)((\lambda x.(\lambda y.z)(x\,x))(z\,z))$ with $\alpha \neq \beta$, $x, y \neq z$, then $o \to_{\mathsf{hd/lo}} \mu\alpha.[\beta]x$ while $o \to_{\mathtt{mx}} (\mu\alpha.[\beta]x)((\lambda y.z)((z\,z)(z\,z))) \to_{\mathtt{mx}} (\mu\alpha.[\beta]x)z \to_{\mathtt{mx}} \mu\alpha.[\beta]x$.

[1]In contrast, there is no semantic loss w.r.t. HN/WN, *i.e.* whether $t \to t'$ is erasing or not, t is HN (resp. WN) iff t' is HN (resp. WN).

Parametric Evaluation. A parametric presentation of the 3 strategies head, leftmost and maximal is given in Fig. 11. We write $\xrightarrow{e}_S$ for an S-reduction step ($S \in \{\mathtt{hd}, \mathtt{lo}, \mathtt{mx}\}$), where, intuitively, the integer e, called the **erasure measure**, denotes the quantity of information which is lost during reduction. Indeed, when $S = \mathtt{mx}$ and the step is non-erasing, then $e = 0$ (no semantic loss); when $S = \mathtt{mx}$ and the step is erasing, e is the size of the erased *normal form* ($\rightarrow_{\mathtt{mx}}$ erases only NFs); and $e = 0$ when $S \in \{\mathtt{hd}, \mathtt{lo}\}$ or the step is not erasing. The reflexive-transitive closure of $\xrightarrow{e}_S$ is defined as expected by *cumulating* erasure measures. The integer e will play a important role in the final Theorem. 4.4.

4.1 Towards Exact Measures

This section discusses the extension of the method from hd-evaluation to the lo/max cases.

Leftmost-Outermost Evaluation. The strategy lo first behaves like hd-evaluation, but once a HNF has been obtained, then it inductively evaluates all the head arguments: it is designed to compute NFs. In the case of system $\mathcal{X}_{\mathtt{hd}}$ for hd-evaluation, exact types ensure that the arguments of HNFs are left untyped. For lo-evaluation, however, one should instead ensure that the arguments of HNFs are typed exactly *once*: this is done by redefining exact types so that their domains are now of cardinal 1, *i.e.* of the form $[\vec{\bullet}] \not\twoheadrightarrow \ldots \not\twoheadrightarrow [\vec{\bullet}] \not\twoheadrightarrow \vec{\bullet}$. We thus now accept derivations of the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : [\vec{\bullet}] \not\twoheadrightarrow [\vec{\bullet}] \not\twoheadrightarrow \vec{\bullet}}\,(\mathsf{ax}) \qquad u_1 : \vec{\bullet}}{x\,u_1 : [\vec{\bullet}] \not\twoheadrightarrow \vec{\bullet}}\,(\mathsf{app}) \qquad u_2 : \vec{\bullet}}{x\,u_1\,u_2 : \vec{\bullet}}\,(\mathsf{app})}{\lambda y.x\,u_1\,u_2 : \vec{\bullet}}\,(\bullet)$$

where inductively, u_1 and u_2 are typed with $\vec{\bullet}$, which would mean here that u_1 and u_2 are weakly normalizing (compared with the typing of $\lambda y.x\,t_1\,t_2$ in the hd-case on page).

Notice that typing persistent abstractions with rules ($\bullet$) and ($\mathtt{mu}_{\mathcal{X}}$) should ensure that these abstractions can also be assigned the type $\vec{\bullet}$, so that the induction works. Modifying the definition of exact types is actually the only ingredient which is necessary in order to characterize lo-termination/weak normalization.

Maximal Evaluation. The strategy mx computes NFs, like lo. However, whereas erasable arguments do not matter regarding the termination of hd/lo, it is necessary to ensure that they are normalizing for mx to terminate, as explained above. Thus, in order to capture normalization for mx, it is necessary for erasable arguments to *be* typed. To do so, it turns out to be sufficient to change the standard definition of domain for arrow types (*cf.* Sec. 3.1) in order to force the arguments to be always typed. To do so, it is sufficient to change the definition of domains of arrow types by setting $\mathsf{dom}_{\mathsf{nb}}([\,] \Rightarrow \mathcal{U}) = [\vec{\bullet}]$, where nb stands for **n***on*-**b***lind erasing*, whereas the previous standard notion for domains will be henceforth indicated with b, standing for **b***lind erasing*: this choice ensures that erasable arguments are always typed, and more precisely that they are typed *once* with the exact type $\vec{\bullet}$. For instance, the system giving exact measures for mx accepts:

$$\dfrac{\dfrac{\dfrac{}{y : [\overline{\bullet}] \vdash^{(0,0,1)} y : \overline{\bullet} \mid}\,(\mathsf{ax})}{y : [\overline{\bullet}] \vdash^{(1,0,1)} \lambda x.y : [\,] \rightarrow \overline{\bullet} \mid}\,(\mathsf{abs}) \qquad \dfrac{\dfrac{}{x : [\overline{\bullet}] \vdash^{(0,0,1)} x : \overline{\bullet} \mid}\,(\mathsf{ax})}{\Vdash^{(0,0,2)} \mathtt{I} : [\overline{\bullet}] \mid}\,(\bullet{+}\wedge)}{y : [\overline{\bullet}] \vdash^{(1,0,3)} (\lambda x.y)\mathtt{I} : \overline{\bullet} \mid}\,(\mathsf{app})$$

4.2 Parametric Type System

We present in Fig. 12 a type system $\mathcal{X}_S$, which is parametric w.r.t. an evaluation strategy $S \in \{\mathtt{hd}, \mathtt{lo}, \mathtt{mx}\}$, each of them related to head, weak and strong normalization, respectively. As already discussed, each strategy $S \in \{\mathtt{hd}, \mathtt{lo}, \mathtt{mx}\}$ depends on a notion of hd or full normal form (Figures 1, 6 and 10), together with a notion of b or nb domain for arrow types (Fig. 13). Remark, however, that the notion of codomain is not parametric.

$$\begin{array}{lcl}
\mathsf{dom}_S(\langle \phi_k \rangle_{k \in K}) & := & \wedge_{k \in K} \mathsf{dom}(\phi_k) \\
\mathsf{dom}_S(\mathcal{I} \Rightarrow \sigma) & := & \mathcal{I} \quad \text{if } \mathcal{I} \neq [\,] \\
\mathsf{dom}_{\mathsf{b}}([\,] \Rightarrow \sigma) & := & [\,] \\
\mathsf{dom}_{\mathsf{nb}}([\,] \Rightarrow \sigma) & := & [\vec{\bullet}] \quad \text{(non-deterministic choice)} \\
\hline
\mathsf{codom}(\mathcal{I} \Rightarrow \sigma) & := & \sigma \\
\mathsf{codom}(\langle \phi_k \rangle_{k \in K}) & := & \vee_{k \in K} \mathsf{codom}(\phi_k)
\end{array}$$

Figure 13. Domains and Codomains

The table below summarizes the 3 case studies:

Strategy	Normal Forms	Domain
hd	hd	blind
lo	full	blind
mx	full	non-blind

Normal Forms. The kind of normal form (hd/full) also determines the appropriate notions of size ($|\_|_{\mathsf{hd}}/|\_|_{\mathtt{full}}$) and exact types ($\mathsf{Ex}_{\mathsf{hd}}/\mathsf{Ex}_{\mathtt{full}}$), which are defined in Fig. 14. Note also that rule ($\bullet_S$) depends on the notion of exact types.

$$\begin{array}{llll}
\mathsf{Ex}_{\mathsf{hd}} ::= \bullet \mid [\,] \not\twoheadrightarrow \langle \mathsf{Ex}_{\mathsf{hd}} \rangle & & \mathsf{UEx}_S ::= \langle \mathsf{Ex}^i_S \rangle_{i \in I} \\
\mathsf{Ex}_{\mathtt{full}} ::= \bullet \mid [\vec{\bullet}] \not\twoheadrightarrow \langle \mathsf{Ex}_{\mathtt{full}} \rangle & & \mathsf{IEx}_S ::= [\langle \mathsf{Ex}^i_S \rangle]_{i \in I}
\end{array}$$

Figure 14. Exact Types

Domains. The kind of domain (b/nb) determines if the argument of a function should be typed or not when the left-hand side of its arrow type is empty, this affects the rule (app_S).

For instance, when $S = \mathtt{lo}$, lo-normal form means full-normal form, lo-exact types means full-exact types, and $\mathsf{dom}_{\mathtt{lo}}()$ means $\mathsf{dom}_{\mathsf{b}}()$.

$$\frac{x \in \mathtt{fv}(t)}{(\lambda x.t)u \overset{0}{\rightarrow}_S t\{x/u\}}\,(\beta{-}\in) \qquad \frac{\alpha \in \mathtt{fn}(c)}{(\mu\alpha.c)u \overset{0}{\rightarrow}_S \mu\alpha.c\{\alpha\backslash\!\backslash u\}}\,(\mu{-}\in) \qquad \frac{\neg\mathtt{abs}(t) \quad t \overset{e}{\rightarrow}_S t'}{t\,u \overset{e}{\rightarrow}_S t'\,u}\,(\neg\mathtt{abs})$$

$$\frac{t \overset{e}{\rightarrow}_S t'}{\lambda x.t \overset{e}{\rightarrow}_S \lambda x.t'}\,(\mathtt{abs}{-}\lambda) \qquad \frac{c \overset{e}{\rightarrow}_S c'}{\mu\alpha.c \overset{e}{\rightarrow}_S \mu\alpha.c'}\,(\mathtt{abs}{-}\mu) \qquad \frac{t \overset{e}{\rightarrow}_S t'}{[\alpha]t \overset{e}{\rightarrow}_S [\alpha]t'}\,(\mathtt{com})$$

Common evaluation rules $S \in \{\mathtt{hd}, \mathtt{lo}, \mathtt{mx}\}$

$$\frac{\mathtt{neut}_{\mathtt{full}}(t) \quad u \overset{e}{\rightarrow}_S u'}{t\,u \overset{e}{\rightarrow}_S t\,u'}\,(\mathtt{neut})$$

Computing full normal forms ($S \in \{\mathtt{lo}, \mathtt{mx}\}$)

$$\frac{x \notin \mathtt{fv}(t)}{(\lambda x.t)u \overset{0}{\rightarrow}_S t}\,\beta{-}\notin{-}\mathrm{b} \qquad \frac{\alpha \notin \mathtt{fn}(c)}{(\mu\alpha.c)u \overset{0}{\rightarrow}_S \mu\alpha.c}\,\mu{-}\notin{-}\mathrm{b}$$

Blind erasure ($S \in \{\mathtt{hd}, \mathtt{lo}\}$)

$$\frac{u \overset{e}{\rightarrow}_{\mathtt{mx}} u' \quad x \notin \mathtt{fv}(t)}{(\lambda x.t)u \overset{e}{\rightarrow}_{\mathtt{mx}} (\lambda x.t)u'}\,(\mathtt{a}\text{-}\lambda) \qquad \frac{u \overset{e}{\rightarrow}_{\mathtt{mx}} u' \quad \alpha \notin \mathtt{fn}(c)}{(\mu\alpha.c)u \overset{e}{\rightarrow}_{\mathtt{mx}} (\mu\alpha.c)u'}\,(\mathtt{a}\text{-}\mu)$$

$$\frac{\mathtt{norm}_{\mathtt{full}}(u) \quad x \notin \mathtt{fv}(t)}{(\lambda x.t)u \overset{|u|_{\mathtt{mx}}}{\rightarrow}_{\mathtt{mx}} t}\,(\beta{-}\notin{-}\mathtt{nb}) \qquad \frac{\mathtt{norm}_{\mathtt{full}}(u) \quad \alpha \notin \mathtt{fn}(c)}{(\mu\alpha.c)u \overset{|u|_{\mathtt{mx}}}{\rightarrow}_{\mathtt{mx}} \mu\alpha.c}\,(\mu{-}\notin{-}\mathtt{nb})$$

Evaluating arguments before erasure ($\mathtt{mx}$)

Figure 11. A Parametric Definition of the Three Evaluation Strategies $\mathtt{hd}/\mathtt{lo}/\mathtt{mx}$

$$\frac{}{x : [\mathcal{N}] \vdash^{(0,0,1)} x : \mathcal{N} \mid \emptyset}\,(\mathsf{ax}) \qquad \frac{\Gamma; x : \mathcal{I} \vdash^{(\ell,m,f)} t : \mathcal{N} \mid \Delta}{\Gamma \vdash^{(\ell+1,m,f-\mathsf{card}(\mathcal{I}))} \lambda x.t : \langle \mathcal{I} \rightarrow \mathcal{N}\rangle \mid \Delta}\,(\mathsf{abs}) \qquad \frac{\Gamma \vdash^{(\ell,m,f)} t : \mathcal{N} \mid \Delta}{\Gamma \vdash^{(\ell,m,f)} [\alpha]t \mid \Delta \vee \alpha : \mathcal{N}}\,(\mathsf{c})$$

$$\frac{\Gamma; x : \mathcal{I} \vdash^{(\ell,m,f)} t : \mathsf{UEx}_S \mid \Delta \quad \mathsf{IEx}_S(\mathcal{I})}{\Gamma \vdash^{(\ell,m,f+1)} \lambda x.t : \overline{\bullet} \mid \Delta}\,(\bullet_S) \qquad \frac{\Gamma \vdash^{(\ell,m,f)} c \mid \Delta; \alpha : \mathcal{U} \quad \mathcal{N} = \mathcal{U}^{\uparrow}}{\Gamma \vdash^{(\ell,m+\mathsf{ar}(\mathcal{N}),f+1+\mathsf{c_p}(\mathcal{U}))} \mu\alpha.c : \mathcal{N} \mid \Delta}\,(\mathsf{mu}_{\mathcal{X}})$$

$$\frac{(\Gamma_k \vdash^{(\ell_k,m_k,f_k)} t : \mathcal{N}_k \mid \Delta_k)_{k\in K}}{\wedge_{k\in K}\Gamma_k \Vdash^{+_{k\in K}(\ell_k,m_k,f_k)} t : [\mathcal{N}_k]_{k\in K} \mid \vee_{k\in K} \Delta_k}\,(\wedge) \qquad \frac{\Gamma_t \vdash^{(\ell_t,m_t,f_t)} t : \mathcal{F} \mid \Delta_t \quad \Gamma_u \Vdash^{(\ell_u,m_u,f_u)} u : \mathsf{dom}_S(\mathcal{F}) \mid \Delta_u}{\Gamma_t \wedge \Gamma_u \vdash^{(\ell_t+\ell_u,m_t+m_u,f_t+f_u+\#_{\mathsf{p}}\mathcal{F})} t\,u : \mathsf{codom}(\mathcal{F}) \mid \Delta_t \vee \Delta_u}\,(\mathsf{app}_S)$$

Figure 12. Parametric System $\mathcal{X}_S$ for the $\lambda\mu$-Calculus

Exact Derivations. A **regular** judgment $\Gamma \vdash^{(\ell,m,f)} t : \mathcal{N} \mid \Delta$ is **S-exact** when $\mathsf{Ex}_S(\Gamma)$, $\mathsf{Ex}_S(\mathcal{N})$, $\mathsf{Ex}_S(\Delta)$. Similar definition holds for **command/auxiliary** judgments. An $\mathcal{X}_S$-**derivation** is S-**exact** when it concludes with an S-exact judgment, in which case the typed object is said to be S-**exactly typable**. Notice that this definition is local (pertaining only the bottom judgment), in contrast to [Accattoli et al. 2018] for the case $S = \mathtt{mx}$ [2].

Example 4.1. Consider $t = (\mu\alpha.[\beta](y(\mu\gamma.[\alpha]x)\mu\gamma.[\alpha]\mathtt{I}))\mathtt{I}$, so that $\mathtt{lo}$-evaluation gives:

$$\begin{array}{lll} t & \rightarrow_\mu & \mu\alpha.[\beta](y(\mu\gamma.[\alpha]x\,\mathtt{I})\mu\gamma.[\alpha](\mathtt{I}\,\mathtt{I})) \\ & \rightarrow_\beta & \mu\alpha.[\beta](y(\mu\gamma.[\alpha]x\,\mathtt{I})\mu\gamma.[\alpha]\mathtt{I} \end{array}$$

which is a $\mathtt{full}$-normal form of size 12, so the expected counters are $(1, 1, 12)$. Indeed, we sketch a $\mathtt{lo}$-exact typing in Fig. 15, where we use $(*)$ when several typing rules are applied without giving the full details of them. Since γ does not occur free, the $\mu\gamma$-abstraction raises blind types that we choose to be $\overline{\bullet}$. The cumulated persistence $\mathsf{c_p}(\cdot)$ is equal to 1 in the $(\mathsf{mu}_{\mathcal{X}})$-rule introducing $\mu\alpha$ and counts the persistent application node of $x\,\mathtt{I}$ which is created by the μ-step (whereas it is consuming in $\mathtt{I}\,\mathtt{I}$). This illustrates some of the remarks concluding Sec. 3.2. Interestingly, if we assign the $\mathtt{lo}$-exact type $\mathcal{N}_2 = \langle[\overline{\bullet}] \nrightarrow \langle[\overline{\bullet}] \not\twoheadrightarrow \overline{\bullet}\rangle\rangle$ to x instead of $\mathcal{N}_1 = \langle[\overline{\bullet}] \rightarrow \overline{\bullet}\rangle$, the derivation concludes with

$$x : [\mathcal{N}_2], y : [\mathcal{N}_2] \vdash^{(1,2,13)} t : \langle[\overline{\bullet}] \rightarrow \overline{\bullet}, \bullet\rangle \mid \beta : \overline{\bullet}$$

which does not have the good counter because it is not exact anymore: the activation operator has transformed top-level persistent arrows into consuming ones while typing $\mu\alpha$.

4.3 Main Properties

Lem. 3.2 generalizes to the parametric type system, and can be stated using all the parametric notions introduced before

[2] More precisely, arbitrary types are used in [Accattoli et al. 2018] to type argument of erasing functions in the case $S = \mathtt{mx}$, while only exact types can be chosen here.

$$\mathcal{N}_1 := \langle [\overline{\bullet}] \not\twoheadrightarrow \overline{\bullet} \rangle \text{ (}\mathtt{lo}\text{-exact)} \qquad \mathcal{N}_1^{\uparrow} = \langle [\overline{\bullet}] \to \overline{\bullet} \rangle \qquad \mathcal{N}_2 := \langle [\overline{\bullet}] \not\twoheadrightarrow \mathcal{N}_1 \rangle = \langle [\overline{\bullet}] \not\twoheadrightarrow \langle [\overline{\bullet}] \not\twoheadrightarrow \overline{\bullet} \rangle \rangle \text{ (}\mathtt{lo}\text{-exact)}$$

$$t_1 ::= \mu\gamma.[\alpha]x \qquad t_2 ::= \mu\gamma.[\alpha]\mathtt{I}$$

$$\dfrac{\dfrac{\dfrac{\vdots \quad \dfrac{\vdots}{x : [\mathcal{N}_1] \Vdash^{(0,0,2)} \mu\gamma.[\alpha]x : [\overline{\bullet}] \mid \alpha : \mathcal{N}_1}(*) \quad \dfrac{\vdots}{\Vdash^{(1,0,1)} \mu\gamma.[\alpha]\mathtt{I} : [\overline{\bullet}] \mid \alpha : \mathcal{N}_1^{\uparrow}}(*)}{x : [\mathcal{N}_1], y : [\mathcal{N}_2] \vdash^{(1,0,6)} (y\,t_1)t_2 : \overline{\bullet} \mid \alpha : \langle [\overline{\bullet}] \not\twoheadrightarrow \overline{\bullet}, [\overline{\bullet}] \to \overline{\bullet} \rangle}(*)}{x : [\mathcal{N}_1], y : [\mathcal{N}_2] \vdash^{(1,1,6+1+1=8)} \mu\alpha.[\beta](y\,t_1)\,t_2 : \langle [\overline{\bullet}] \to \overline{\bullet} \rangle_2 \mid \beta : \overline{\bullet}}\mathtt{c+mu}_{\mathcal{X}}) \quad \dfrac{\vdots}{\Vdash^{(0,0,4)} \mathtt{I} : [\overline{\bullet}, \overline{\bullet}] \mid \emptyset}(*)}{x : [\mathcal{N}_1], y : [\mathcal{N}_2] \vdash^{(1,1,8+4=12)} (\mu\alpha.[\beta]((y\,t_1)\,t_2))\mathtt{I} : \langle \bullet \rangle_2 \mid \beta : \overline{\bullet}}(\mathtt{app}_S)$$

Figure 15. Exact $\mathcal{X}_S$-Derivation ($S = \mathtt{lo}$)

Subject reduction holds for $S \in \{\mathtt{hd}, \mathtt{lo}\}$, *i.e.* variable/name assignments and types are stable under reduction, and it induces a decrease of $\ell + m + f$, which is strict when the redex is typed. However, subject reduction does not hold for $S = \mathtt{mx}$ when $\mathtt{mx}$-steps because some subderivation may type an erasable term, *e.g.* the one typing z in $(\lambda x.y)z$ (thus reflecting the so-called semantic loss). This difference between $\mathtt{hd}/\mathtt{lo}$ and $\mathtt{mx}$ is reflected in the statement of the forthcoming Lem. 4.2. Moreover, to obtain exact measures for every strategy $S \in \{\mathtt{hd}, \mathtt{lo}, \mathtt{mx}\}$, we expect a decrease of 1 for each S-step (w.r.t. S-exact derivations), which indeed holds. However, such a property cannot be proved directly on exact derivations: induction fails because a subderivation of an S-exact derivation is not necessarily S-exact. For instance, let $t\,u$ be exactly typed, $\neg\mathsf{abs}(t)$ but $t \to_S \lambda x.t'_0$, then $t\,u \to_S (\lambda x.t'_0)u \to_S t'_0\{x/u\}$. Thus, t and $\lambda x.t'_0$ are typed with a *consuming* functional type $\langle \mathcal{I} \to \mathcal{N} \rangle$: in particular, t is *not* exactly typed, whereas the induction hypothesis should be applied on the step $t \to_S \lambda x.t'_0$. This is why we need a weaker notion, called **S-quasi-exact derivation**: a regular **judgment** $\Gamma \vdash^{(\ell,m,f)} t : \mathcal{N} \mid \Delta$ is **S-quasi-exact** if $\mathsf{Ex}_S(\Gamma)$ and $\mathsf{Ex}_S(\Delta)$ and either $\mathsf{Ex}_S(\mathcal{N})$ or $\neg\mathsf{abs}(t)$. This generalizes to command/auxiliary judgments. An **$\mathcal{X}_S$-derivation** is **S-quasi-exact** when it concludes with an S-quasi-exact judgment. A left-hand side subderivation of an S-quasi-exact derivation is also S-quasi-exact, a property which enables the inductive proof of the statement below.

Lemma 4.2 (Quasi-Exact Subject Reduction). *Assume $\Phi \triangleright \Gamma \vdash_S^{(\ell,m,f)} t : \mathcal{N} \mid \Delta$ is S-quasi-exact and $t \xrightarrow{e}_S t'$ for $S \in \{\mathtt{hd}, \mathtt{lo}, \mathtt{mx}\}$. Then, there exists a derivation $\Phi' \triangleright \Gamma' \vdash_S^{(\ell',m',f-e)} t' : \mathcal{N} \mid \Delta'$ where $\Gamma' \subseteq \Gamma$ and $\Delta' \subseteq \Delta$, such that*

- ♦ *If $e = 0$ (*e.g.*, if $S \in \{\mathtt{hd}, \mathtt{lo}\}$), then $\Gamma = \Gamma'$ and $\Delta = \Delta'$.*
- ♦ *If $t \to_\beta t'$, then $\ell' = \ell - 1$ and $m' = m$.*
- ♦ *If $t \to_\mu t'$, then $\ell' = \ell$ and $m' = m - 1$.*

Moreover[3], *if Φ is S-exact, then Φ' is also S-exact.*

Furthermore, quasi-exact derivations also enjoy subject expansion when $S \in \{\mathtt{hd}, \mathtt{lo}\}$, but the following weaker form of subject expansion applies to the three cases:

Lemma 4.3 (Quasi-Exact Subject Expansion). *Assume $\Phi' \triangleright \Gamma' \vdash^{(\ell',m',f)} t' : \mathcal{N} \mid \Delta'$ is S-quasi-exact and $t \xrightarrow{e}_S t'$. Then there exists a $\mathcal{X}_S$-derivation $\Phi \triangleright \Gamma \vdash^{(\ell,m,f+e)} t : \mathcal{N} \mid \Delta$ where $\Gamma' \subseteq \Gamma$, $\Delta' \subseteq \Delta$, $\mathsf{Ex}_{\mathtt{mx}}(\Gamma)$, $\mathsf{Ex}_{\mathtt{mx}}(\Delta)$:*

- ♦ *If $e = 0$ (*e.g.* if $S \in \{\mathtt{hd}, \mathtt{lo}\}$), then $\Gamma = \Gamma'$ and $\Delta = \Delta'$.*
- ♦ *If $t \to_\beta t'$, then $\ell = \ell' + 1$ and $m = m'$;*
- ♦ *If $t \to_\mu t'$, then $\ell = \ell'$ and $m = m' + 1$.*

Moreover, if Φ' is S-exact, Φ is S-exact.

Using Lemmas 3.2-2 and 4.2 we prove soundness, and Lemmas 3.2-3 and 4.3 give completeness. We thus obtain:

Theorem 4.4 (Parametric Capture of Exact Measures). *Let $S \in \{\mathtt{hd}, \mathtt{lo}, \mathtt{mx}\}$ and t be a $\lambda\mu$-term. Then there is an S-exact derivation $\Phi \triangleright \Gamma \vdash^{(\ell,m,f)} t : \mathcal{N} \mid \Delta$ iff t S-evaluates with an erasure measure of e to an S-normal form of S-size $f - e$ in exactly ℓ β-steps and m μ-steps.*

5 Conclusion

We defined type systems based on non-idempotent intersections and unions, being able to capture exact measures for evaluation lengths and size of normal forms in the setting of a functional language with control-operators. Our case studies focus on three different evaluation strategies for the $\lambda\mu$-calculus: head, leftmost and maximal. We prove soundness and completeness properties for all of them, thus giving both directions of each characterization.

This contribution is based on the crucial notion of persistence and consumption. Moreover, it is synthesized in a unique parametric type system subsuming the three mentioned strategies, thus factorizing their common proofs. This approach is modular and gives a unified method to obtain exact measures for different cases.

It would also be interesting to study exact measures obtained by using CPS translations into the λ-calculus.

We expect to transfer the ideas in this paper to a classical sequent calculus system. We also plan to extend this framework to other evaluation strategies such as classical *call-by-value* and classical *call-by-need* [Ariola et al. 2011]. It would also be interesting to export the notion of persistence/consumption to a categorical/denotational setting.

[3]If Φ is only quasi-exact, then Φ' is not necessarily quasi-exact.

References

Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. 2018. Tight Typings and Split bounds. *PACMPL* 2, ICFP (2018), 94:1–94:30.

Beniamino Accattoli and Giulio Guerrieri. 2018. Types of Fireballs. In *APLAS (LNCS)*, Vol. 11275. Springer, 45–66.

Beniamino Accattoli, Giulio Guerrieri, and Maico Leberle. 2019. Types by Need. In *ESOP (LNCS)*, Vol. 11423. Springer, 410–439.

Zena M. Ariola, Hugo Herbelin, and Alexis Saurin. 2011. Classical Call-by-Need and Duality. In *the 10th International Conference on Typed Lambda Calculi and Applications (TLCA), Novi Sad, Serbia, June 1-3 (Lecture Notes in Computer Science)*, Luke Ong (Ed.), Vol. 6690. Springer-Verlag, 27–44.

Henk Barendregt. 1985. *The Lambda-Calculus: Its Syntax and Sematics*. Elsevier.

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

Antonio Bucciarelli, Delia Kesner, Alejandro Ríos, and Andrés Viso. 2020. The Bang Calculus Revisited. (2020). Submitted.

Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. 2017. Non-Idempotent Intersection Types for the Lambda-Calculus. *Logic Journal of the IGPL* 4, 25 (2017), 431–464.

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A new type assignment for lambda-terms. *Archive for Mathematical Logic* 19 (1978), 139–156.

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

Daniel de Carvalho. 2007. *Sémantiques de la logique linéaire et temps de calcul.* These de Doctorat. Université Aix-Marseille II.

Daniel de Carvalho. 2018. Execution time of λ-terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science* 28, 7 (2018), 1169–1203.

Philippe de Groote. 1994. On the Relation between the Lambda-Mu-Calculus and the Syntactic Theory of Sequential Control. In *the 5th International Conference on Logic Programming and Automated Reasoning (LPAR), Kiev, Ukraine, July 16-22 (Lecture Notes in Computer Science)*, Frank Pfenning (Ed.), Vol. 822. Springer-Verlag, 31–43.

Daniel J. Dougherty, Silvia Ghilezan, and Pierre Lescanne. 2008. Characterizing strong normalization in the Curien-Herbelin symmetric lambda calculus: Extending the Coppo-Dezani heritage. *Theoretical Computer Science* 398, 1-3 (2008), 114–128.

Philippa Gardner. 1994. Discovering Needed Reductions Using Type Theory. In *Theoretical Aspects of Computer Software, International Conference TACS '94, April 19-22 (Lecture Notes in Computer Science)*, Masami Hagiya and John C. Mitchell (Eds.), Vol. 789. Springer, 555–574.

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102.

Timothy Griffin. 1990. A Formulae-as-Types Notion of Control. In *17th Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 47–58.

Delia Kesner and Pierre Vial. 2017. Types as Resources for Classical Natural Deduction. In *the 2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017 (LIPIcs)*, Dale Miller (Ed.), Vol. 84. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 24:1–24:17.

Assaf Kfoury. 2000. A linearization of the Lambda-calculus and consequences. *Journal of Logic and Computation* 10, 3 (2000), 411–436.

Kentaro Kikuchi and Takafumi Sakurai. 2014. A Translation of Intersection and Union Types for the $\lambda\mu$-Calculus. In *the 12th Asian Symposium on Programming Languages and Systems (APLAS), Singapore, November 17-19 (Lecture Notes in Computer Science)*, Jacques Garrigue (Ed.), Vol. 8858. Springer-Verlag, 120–139.

Jean-Louis Krivine. 1993. *Lambda-calculus, types and models*. Ellis Horwood.

Jean-Louis Krivine. 2007. Realizability in classical logic. *Panoramas et synthèses* 27 (2007), 197–229.

Olivier Laurent. 2003a. Krivine's abstract machine and the lambda-mu calculus. (2003). https://perso.ens-lyon.fr/olivier.laurent/lmkamen.pdf.

Olivier Laurent. 2003b. Polarized proof-nets and $\lambda\mu$-calculus. *Theoretical Computer Science* 290, 1 (Jan. 2003), 161–188.

Olivier Laurent. 2004. On the denotational semantics of the untyped lambda-mu calculus. (2004). Unpublished note.

Michel Parigot. 1992. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *International Conference on Logic Programming and Automated Reasoning (Lecture Notes in Computer Science)*, Andrei Voronkov (Ed.), Vol. 624. Springer-Verlag, 190–201.

Alexis Saurin. 2005. Separation with Streams in the lambdaμ-calculus. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*. IEEE Computer Society, 356–365. DOI: http://dx.doi.org/10.1109/LICS.2005.48

Steffen van Bakel. 2010. Sound and Complete Typing for lambda-mu. In *the Fifth Workshop on Intersection Types and Related Systems, ITRS 2010, Edinburgh, U.K., 9th July 2010. (EPTCS)*, Elaine Pimentel, Betti Venneri, and Joe B. Wells (Eds.), Vol. 45. 31–44.

Femke van Raamsdonk, Paula Severi, Morten Heine Sørensen, and Hongwei Xi. 1999. Perpetual Reductions in Lambda-Calculus. *Inf. Comput.* 149, 2 (1999), 173–225.

Lionel Vaux. 2007. Convolution Lambda-Bar-Mu-Calculus. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings (Lecture Notes in Computer Science)*, Simona Ronchi Della Rocca (Ed.), Vol. 4583. Springer-Verlag, 381–395.