




The Spirit of Node Replication

Delia Kesner^{1,2} , Loïc Peyrot ¹, and Daniel Ventura³ *

¹ Université de Paris, CNRS, IRIF, Paris, France
{kesner,lpeyrot}@irif.fr

² Institut Universitaire de France, France

³ Univ. Federal de Goiás, Goiânia, Brazil
ventura@ufg.br

Abstract. We define and study a term calculus implementing higher-order node replication. It is used to specify two different (weak) evaluation strategies: call-by-name and fully lazy call-by-need, that are shown to be observationally equivalent by using type theoretical technical tools.

1 Introduction

Computation in the λ -calculus is based on higher-order substitution, a complex operation being able to erase and copy terms during evaluation. Several formalisms have been proposed to model higher-order substitution, going from explicit substitutions (ES) [1] (see a survey in [41]) and labeled systems [15] to pointer graphs [60] or optimal sharing graphs [49]. The model of copying behind each of these formalisms is not the same.

Indeed, suppose one wants to substitute all the free occurrences of some variable x in a term t by some term u . We can imagine at least four ways to do that. (1) A drastic solution is a one-shot substitution, called *non-linear* (or *full*) *substitution*, based on simultaneously replacing *all* the free occurrences of x in t by the whole term u . This notion is generally defined by induction on the structure of the term t . (2) A refined method substitutes *one* free occurrence of x at a time, the so-called *linear* (or *partial*) *substitution*. This notion is generally defined by induction on the number of free occurrences of x in the term t . An orthogonal approach can be taken by replicating *one* term-constructor of u *at a time*, instead of replicating u as a whole, called here *node replication*. This notion can be defined by induction on the structure of the term u , and also admits two versions: (3) non-linear, *i.e.* by simultaneously replacing all the occurrences of x in t , or (4) linear. The linear version of the node replication approach can be formally defined by combining (2) and (3).

It is not surprising that different notions of substitution give rise to different evaluation strategies. Indeed, linear substitution is the common model in well-known abstract machines for call-by-name and call-by-value (see *e.g.* [3]), while (linear) node replication is used to implement fully lazy sharing [60]. However, node replication, originally introduced to implement optimal graph reduction in

* Supported by CNPq grant Universal 430667/2016-7.

a graphical formalism, has only been studied from a Curry-Howard perspective by means of a term language known as the atomic λ -calculus [33].

The Atomic Lambda-Calculus. The Curry-Howard isomorphism uncovers a deep connection between logical systems and term calculi. It is then not surprising that different methods to implement substitution correspond to different ways to normalize logical proofs. Indeed, full substitution (1) can be explained in terms of natural deduction, while partial substitution (2) corresponds to cut elimination in Proof-Nets [2]. Replication of nodes (3)-(4) is based on a Curry-Howard interpretation of deep inference [32,33]. Indeed, the logical aspects of intuitionistic deep inference are captured by the atomic λ -calculus [33], where copying of terms proceeds *atomically*, *i.e.* node by node, similar to the optimal graph reduction of Lamping [49].

The atomic λ -calculus is based on *explicit control of resources* such as erasure and duplication. Its operational semantics explicitly handles the structural constructors of weakening and contraction, as in the calculus of resources $\lambda\mathbf{1}\mathbf{xr}$ [43,44]. As a result, comprehension of the meta-properties of the term-calculus, in a higher-level, and its application to concrete implementations of reduction strategies in programming languages, turn out to be quite difficult. In this paper, we take one step back, by studying the paradigm of *node replication* based on *implicit*, rather than *explicit*, weakening and contraction. This gives a new concise formulation of node replication which is simple enough to model different programming languages based on reduction strategies.

Call-by-Name, Call-by-Value, Call-by-Need. *Call-by-name* is used to implement programming languages in which arguments of functions are first copied, then evaluated. This is frequently expensive, and may be improved by *call-by-value*, in which arguments are evaluated first, then consumed. The difference can be illustrated by the term $t = \Delta(\mathbf{II})$, where $\Delta = \lambda x.xx$ and $\mathbf{I} = \lambda z.z$: call-by-name first duplicates the argument \mathbf{II} , so that its evaluation is also duplicated, while call-by-value first reduces \mathbf{II} to (the value) \mathbf{I} , so that duplications of the argument do not cause any duplicated evaluation. It is not always the best solution, though, because evaluating erasable arguments is useless.

Call-by-need, instead, takes the best of call-by-name and call-by-value: as in call-by-name, erasable arguments are not evaluated at all, and as in call-by-value, reduction of arguments occurs at most once. Furthermore, call-by-need implements a *demand-driven* evaluation, in which erasable arguments are never needed (so they are not evaluated), and non-erasable arguments are evaluated only if needed. Technically, some sharing mechanism is necessary, for example by extending the λ -calculus with explicit substitutions/let constructs [7]. Then β -reduction is decomposed in at least two steps: one creating an explicit (pending) substitution, and the other ones (linearly) substituting *values*. Thus for example, $(\lambda x.xx)(\mathbf{II})$ reduces to $(xx)[x\backslash\mathbf{II}]$, and the substitution argument is thus evaluated in order to find a value before performing the linear substitution.

Even when adopting this wise evaluation scheme, there are still some unnecessary copies of redexes: while only *values* (*i.e.* abstractions) are duplicated,

they may contain redexes as subterms, *e.g.* $\lambda z.z(\text{II})$ whose subterm II is a redex. Duplication of such values might cause redex duplications in *weak* (*i.e.* when evaluation is forbidden inside abstractions) call-by-need. This happens in particular in the *confluent* variant of weak reduction in [52].

Full laziness. Alas, it is not possible to keep all values shared forever, typically when they potentially contribute to the creation of a future β -reduction step. The key idea to gain in efficiency is then to keep the subterm II as a *shared* redex. Therefore, the (full) value $\lambda z.z(\text{II})$ to be copied is split into two separate parts. The first one, called *skeleton*, contains the minimal information preserving the bound structure of the value, *i.e.* the linked structure between the binder and each of its (bound) variables. In our example, this is the term $\lambda z.zy$, where y is a fresh variable. The second one is a multiset of *maximal free expressions* (MFE), representing all the shareable expressions (here only the term II). Only the skeleton is then copied, while the problematic redex II remains shared:

$$(\lambda x.xx)(\lambda z.z(\text{II})) \rightarrow (xx)[x \setminus \lambda z.z(\text{II})] \rightarrow ((\lambda z.zy)x)[x \setminus \lambda z.zy][y \setminus \text{II}]$$

When the subterm II is needed ahead, it is first reduced inside the ES, as it is usual in (standard) call-by-need, thus avoiding to compute the redex twice. This optimization is called *fully lazy sharing* and is due to Wadsworth [60].

In the confluent weak setting evoked earlier [52], the fully lazy optimization is even optimal in the sense of Lévy [51]. This means that the strategy reaches the weak normal form in the same number of β -steps as the shortest possible weak reduction sequence in the usual λ -calculus without sharing. Thus, fully lazy sharing turns out to be a *decidable* optimal strategy, in contrast to other weak evaluation strategies in the λ -calculus without sharing, which are also optimal but not decidable [11].

Contributions. The first contribution of this paper is a term calculus implementing (full) node replication and internally encoding skeleton extraction (Sec. 2). We study some of its main operational properties: termination of the substitution calculus, confluence, and its relation with the λ -calculus.

Our second contribution is the use of the node replication paradigm to give an alternative specification of two evaluation strategies usually described by means of full or linear substitution: call-by-name (Sec. 4.1) and weak fully lazy reduction (Sec. 4.2), based on the key notion of skeleton. The former can be related to (weak) head reduction, while the latter is a fully lazy version of (weak) call-by-need. In contrast to other implementations of fully lazy reduction relying on (external) meta-level definitions, our implementation is based on formal operations internally defined over the term syntax of the calculus.

Furthermore, while it is known that call-by-name and call-by-need specified by means of full/linear substitution are observationally equivalent [7], it was not clear at first whether the same property would hold in our case. Our third contribution is a proof of this result (Sec. 6) using semantical tools coming from proof theory –notably intersection types. This proof technique [42] considerably

simplifies other approaches [7,54] based on syntactical tools. Moreover, the use of intersection types has another important consequence: standard call-by-name and call-by-need turn out to be observationally equivalent to call-by-name and call-by-need with node replication, as well as to the more semantical notion of neededness (see [45]).

Intersection types provide quantitative information about fully lazy evaluation so that a fourth contribution of this work is a measure based on type derivations which turns out to be an upper bound to the length of reduction sequences to normal forms in a fully lazy implementation.

More generally, our work bridges the gap between the Curry-Howard theoretical understanding of node replication and concrete implementations of fully lazy sharing. Related works are presented in the concluding Sec. 7.

2 A Calculus for Node Replication

We now present the syntax and operational semantics of the $\lambda\mathbf{R}$ -calculus (\mathbf{R} for Replication), as well as a notion of *level* playing a key role in the next sections.

Syntax. Given a countably infinite set \mathcal{X} of variables x, y, z, \dots , we consider the following grammars.

$$\begin{array}{ll}
 \text{(Terms)} & t, u ::= x \mid \lambda x.t \mid tu \mid t[x \setminus u] \mid t[x \setminus\!\!\setminus \lambda y.u] \\
 \text{(Pure Terms)} & p, q ::= x \mid \lambda x.p \mid pq \\
 \text{(Term Contexts)} \mathbf{C} & ::= \square \mid \lambda x.\mathbf{C} \mid \mathbf{C}t \mid t\mathbf{C} \mid \mathbf{C}[x \setminus t] \mid \mathbf{C}[x \setminus\!\!\setminus \lambda y.u] \mid t[x \setminus \mathbf{C}] \mid t[x \setminus\!\!\setminus \lambda y.\mathbf{C}] \\
 \text{(List Contexts)} \mathbf{L} & ::= \square \mid \mathbf{L}[x \setminus u] \mid \mathbf{L}[x \setminus\!\!\setminus \lambda y.u]
 \end{array}$$

The set of terms (resp. **pure terms**) is denoted by $A_{\mathbf{R}}$ (resp. A). We write $|t|$ for the **size** of t , *i.e.* for its number of constructors. We write \mathbf{I} for the identity function $\lambda x.x$. The construction $[x \setminus u]$ is an **explicit substitution (ES)**, and $[x \setminus\!\!\setminus \lambda y.u]$ an **explicit distributor**: the first one is used to copy arbitrary terms, while the second one is used specifically to duplicate abstractions. We write $[x \triangleleft u]$ to denote an **explicit cut** in general, which is either $[x \setminus u]$ or $[x \setminus\!\!\setminus \lambda y.u]$ when u is $\lambda y.u'$, typically to factorize some definitions and proofs where they behave similarly in both cases. When using the general notation $t[x \triangleleft u]$, we define $x(\triangleleft) = 1$ if the term is an ES, and $x(\triangleleft) = 0$ otherwise.

We use two notions of **contexts**. Term contexts \mathbf{C} extend those of the λ -calculus to explicit cuts. List contexts \mathbf{L} denote an arbitrary list of explicit cuts. They will be used to implement reduction *at a distance* in the operational semantics defined ahead.

Free/bound variables of terms are defined as usual, notably $\mathbf{fv}(t[x \triangleleft u]) := \mathbf{fv}(t) \setminus \{x\} \cup \mathbf{fv}(u)$. These notions are extended to contexts as expected, in particular $\mathbf{fv}(\square) := \emptyset$. The **domain** of a **list context** is given by $\mathbf{dlc}(\square) := \emptyset$ and $\mathbf{dlc}(\mathbf{L}[x \triangleleft u]) := \mathbf{dlc}(\mathbf{L}) \cup \{x\}$. α -conversion [13] is extended to $\lambda\mathbf{R}$ -terms as expected and used to avoid capture of free variables. We write $t\{x \setminus u\}$ for the meta-level (capture-free) substitution simultaneously replacing all the free occurrences of the variable x in t by the term u .

The **application of a context \mathbf{C} to a term t** , written $\mathbf{C}\langle t \rangle$, replaces the hole \square of \mathbf{C} by t . For instance, $\square\langle t \rangle = t$ and $(\lambda x.\square)\langle t \rangle = \lambda x.t$. This operation is not defined modulo α -conversion, so that capture of variables eventually happens. Thus, we also consider another kind of application of contexts to terms, denoted with double brackets, which is only defined if there is no capture of variables. For instance, $(\lambda y.\square)\langle\langle x \rangle\rangle = \lambda y.x$ while $(\lambda x.\square)\langle\langle x \rangle\rangle$ is undefined.

Operational semantics. ES may block some expected *meaningful* (i.e. non-structural) reductions. For instance, β -reduction is blocked in $(\lambda x.t)[y\backslash v]u$ because an ES lies between the function and its argument. This kind of stuck redexes do not happen in graphical representations (e.g. [28]), but it is typical in the sequential structure of *term* syntaxes.

There are at least two ways to handle this issue. The first one is based on *structural/permutation* rules, as in [33], where the substitution is first pushed outside the application node, as $(\lambda x.t)[y\backslash v]u \rightarrow ((\lambda x.t)u)[y\backslash v]$, so that β -reduction is finally unblocked. The second, less elementary, possibility is given by an operational semantics *at a distance* [6,4], where the β -rule can be fired by a rule like $L\langle\lambda x.t\rangle u \rightarrow L\langle t[x\backslash u] \rangle$, L being an arbitrary list context. The distance paradigm is therefore used to gather meaningful and permutation rules in only one reduction step. In $\lambda\mathbf{R}$, we combine these two technical tools. First, we consider the following permutation rules, all of them are constrained by the condition $x \notin \mathbf{fv}(t)$.

$$\begin{array}{lll} \lambda x.u[y\triangleleft t] \mapsto_{\pi} (\lambda x.u)[y\triangleleft t] & v[x\triangleleft u]t & \mapsto_{\pi} (vt)[x\triangleleft u] \\ tv[x\triangleleft u] & \mapsto_{\pi} (tv)[x\triangleleft u] & t[y\triangleleft v[x\triangleleft u]] \mapsto_{\pi} t[y\triangleleft v][x\triangleleft u] \end{array}$$

The reduction relation \rightarrow_{π} is defined as the closure of the rules \mapsto_{π} under *all* contexts. It does not hold any computational content, only a structural one that unblocks redexes by moving explicit cuts out.

In order to highlight the computational content of node replication we combine distance and permutations within the $\lambda\mathbf{R}$ -**calculus**, given by the closure of the following rules by all the contexts.

$$\begin{array}{lll} L\langle\lambda x.t\rangle u & \mapsto_{\mathbf{dB}} & L\langle t[x\backslash u] \rangle \\ t[x\backslash L\langle uv \rangle] & \mapsto_{\mathbf{app}} & L\langle t\{x\backslash yz\}[y\backslash u][z\backslash v] \rangle \text{ where } y \text{ and } z \text{ are fresh} \\ t[x\backslash L\langle \lambda y.u \rangle] & \mapsto_{\mathbf{dist}} & L\langle t[x\backslash \lambda y.z[z\backslash u]] \rangle \text{ where } z \text{ is fresh} \\ t[x\backslash \lambda y.u] & \mapsto_{\mathbf{abs}} & L\langle t\{x\backslash \lambda y.p\} \rangle \text{ where } u \rightarrow_{\pi}^* L\langle p \rangle \text{ and } y \notin \mathbf{fv}(L) \\ t[x\backslash L\langle y \rangle] & \mapsto_{\mathbf{var}} & L\langle t\{x\backslash y\} \rangle \end{array}$$

Notice in the five rules above that the (meta-level) substitution is *full* (it is performed simultaneously on all free occurrences of the variable x), and the list context L is always pushed outside the term t . We will highlight in green such list contexts in the forthcoming examples to improve readability. Apart from rule \mathbf{dB} used to fire β -reductions, there are four substitution rules used to copy abstractions, applications and variables, pushing outside all the cuts surrounding the node to be copied. Rule \mathbf{app} copies one application node, while rule \mathbf{var} copies one variable node. The case of abstractions is more involved as explained below.

The specificity in copying an abstraction $\lambda y.u$ is due to the (binding) relation between λy and all the free occurrences of y in its body u . Abstractions are thus copied in two stages. The first one is implemented by the rule **dist**, creating a distributor in which a potentially replaceable abstraction is placed, while moving its body inside a new ES. There are then two ways to replicate nodes of the body. Either they can be copied inside the distributor (where the binding relation between λy and the bound occurrences of y is kept intact), or they can be pushed outside the distributor, by means of the (non-deterministic) rule **abs**. In the second case, however, free occurrences of y cannot be pushed outside the abstraction (with binder y) to be duplicated, at the risk of breaking consistency: only shared components without y links can be then pushed outside. These components are gathered together into a list context L , which is pushed outside by using permutation rules, before performing the substitution of the pure body containing all the bound occurrences of y . Specifying this operation using only distance is hard, thus permutation rules are also used in our rule **abs**.

The **s**-substitution relation $\rightarrow_{\mathbf{s}}$ (resp. distant Beta relation $\rightarrow_{\mathbf{dB}}$) is defined as the closure of $\mapsto_{\mathbf{app}} \cup \mapsto_{\mathbf{dist}} \cup \mapsto_{\mathbf{abs}} \cup \mapsto_{\mathbf{var}}$ (resp. $\mapsto_{\mathbf{dB}}$) under *all* contexts, and the reduction relation $\rightarrow_{\mathbf{R}}$ is the union of $\rightarrow_{\mathbf{s}}$ and $\rightarrow_{\mathbf{dB}}$.

Example 1. Let $t_0 = (\lambda x_1.x_1)(\lambda y.Iy)$. In what follows, we underline the term where the reduction is performed:

$$\begin{aligned} t_0 &\rightarrow_{\mathbf{dB}} x_1[x_1 \backslash \lambda y.Iy] \rightarrow_{\mathbf{dist}} x_1[x_1 \backslash \lambda y.z[z \backslash Iy]] \rightarrow_{\mathbf{app}} x_1[x_1 \backslash \lambda y.(z_1 z_2)[z_1 \backslash I][z_2 \backslash y]] \\ &\rightarrow_{\mathbf{dist}} x_1[x_1 \backslash \lambda y.(z_1 z_2)[z_1 \backslash \lambda x_3.z_3[z_3 \backslash x_3]][z_2 \backslash y]] \\ &\rightarrow_{\mathbf{var}} x_1[x_1 \backslash \lambda y.(z_1 y) \underline{[z_1 \backslash \lambda x_3.z_3[z_3 \backslash x_3]]}] \rightarrow_{\mathbf{abs}} (\lambda y.z_1 y)[z_1 \backslash \lambda x_3.z_3[z_3 \backslash x_3]] \end{aligned}$$

Let \mathcal{R} be any reduction relation. We write $\rightarrow_{\mathcal{R}}^*$ for the reflexive-transitive closure of $\rightarrow_{\mathcal{R}}$. A term t is said to be **\mathcal{R} -confluent** iff $t \rightarrow_{\mathcal{R}}^* u$ and $t \rightarrow_{\mathcal{R}}^* s$ implies there is t' such that $u \rightarrow_{\mathcal{R}}^* t'$ and $s \rightarrow_{\mathcal{R}}^* t'$. The relation \mathcal{R} is **confluent** iff every term is \mathcal{R} -confluent. A term t is said to be in **\mathcal{R} -normal form** (written also \mathcal{R} -nf) iff there is no t' such that $t \rightarrow_{\mathcal{R}} t'$. A term t is said to be **\mathcal{R} -terminating** or **\mathcal{R} -normalizing** iff there is no infinite \mathcal{R} -sequence starting at t . The reduction \mathcal{R} is said to be **terminating** iff every term is \mathcal{R} -terminating.

Levels. The notion of level plays a key role in this work. Intuitively, the level of a variable in a term indicates the maximal depth of its free occurrences w.r.t. ES (and not w.r.t. explicit distributors). However, in order to keep soundness w.r.t. the permutation rules, levels are computed along *linked chains* of ES. For instance, the level of w in both $x[x \backslash y[y \backslash w]]$ and $x[x \backslash y][y \backslash w]$ is 2. Formally, the **level** of a variable z in a term t is defined by (structural) induction, while assuming by α -conversion that z is not a bound variable in t :

$$\begin{aligned} \mathbf{lv}_z(x) &:= 0 & \mathbf{lv}_z(t_1 t_2) &:= \max(\mathbf{lv}_z(t_1), \mathbf{lv}_z(t_2)) & \mathbf{lv}_z(\lambda y.t) &:= \mathbf{lv}_z(t) \\ \mathbf{lv}_z(t[x \triangleleft u]) &:= \begin{cases} \mathbf{lv}_z(t) & \text{if } z \notin \mathbf{fv}(u) \\ \max(\mathbf{lv}_z(t), \mathbf{lv}_x(t) + \mathbf{lv}_z(u) + x(\triangleleft)) & \text{otherwise} \end{cases} \end{aligned}$$

Notice that $\mathbf{lv}_w(t) = 0$ whenever $w \notin \mathbf{fv}(t)$ or t is pure. We illustrate the concept of level by an example. Consider $t = x[x \setminus z[y \setminus w]][w \setminus w']$, then $\mathbf{lv}_z(t) = 1$, $\mathbf{lv}_{w'}(t) = 3$ and $\mathbf{lv}_y(t) = 0$ because $y \notin \mathbf{fv}(t)$. This notion is also extended to contexts as expected, *i.e.* $\mathbf{lv}_\square(\mathbf{C}) = \mathbf{lv}_z(\mathbf{C}(\langle\langle z \rangle\rangle))$, where z is a fresh variable.

Lemma 2. *Let $t \in \Lambda_{\mathbf{R}}$. If $t_0 \rightarrow_{\pi, \mathbf{s}} t_1$, then $\mathbf{lv}_w(t_0) \geq \mathbf{lv}_w(t_1)$ for any $w \in \mathcal{X}$.*

It is worth noticing that there are two cases when the level of a variable in a term may decrease: using a permutation rule to push an explicit cut out of another cut when the first one is a void cut, or using rule $\mapsto_{\mathbf{var}}$.

Hence, levels alone are not enough to prove termination of $\rightarrow_{\mathbf{s}}$. We then define a decreasing measure for $\rightarrow_{\mathbf{s}}$ in which not only variables are indexed by a level, but also constructors. For instance, in $t[x \setminus \lambda y.yz]$, we can consider that the level of *all* the constructors of $\lambda y.yz$ have level $\mathbf{lv}_x(t)$. This will ensure that the level of an abstraction will decrease when applying rule **dist**, as well as the level of an application when applying rule **app**. This is what we do next.

3 Operational Properties

We now prove three key properties of the $\lambda\mathbf{R}$ -calculus: termination of the reduction system $\rightarrow_{\mathbf{s}}$, relation between $\lambda\mathbf{R}$ and the λ -calculus, and confluence of the reduction system $\rightarrow_{\lambda\mathbf{R}}$.

Termination of $\rightarrow_{\mathbf{s}}$. Some (rather informal) arguments are provided in [33] to justify termination of the substitution subrelation of their whole calculus. We expand these ideas into an alternative full formal proof adapted to our case, which is based on a measure being strictly decreasing w.r.t. $\rightarrow_{\mathbf{s}}$.

We consider a set \mathcal{O} of objects of the form $\mathbf{a}(k, n)$ or $\mathbf{b}(k)$ ($k, n \in \mathbb{N}$), which is equipped with the following ordering $>^{\mathcal{O}}$:

$$\begin{aligned} \mathbf{a}(k, n) >^{\mathcal{O}} \mathbf{a}(k', n) & \text{ if } k > k', \text{ or } (k = k' \text{ and } n > n') & \mathbf{b}(k) >^{\mathcal{O}} \mathbf{a}(k', n) & \text{ if } k \geq k' \\ \mathbf{a}(k, n) >^{\mathcal{O}} \mathbf{b}(k') & \text{ if } k > k' & \mathbf{b}(k) >^{\mathcal{O}} \mathbf{b}(k') & \text{ if } k > k' \end{aligned}$$

Lemma 3. *The order $>^{\mathcal{O}}$ on the set \mathcal{O} is well-founded.*

We write $>_{\text{MUL}}^{\mathcal{O}}$ for the multiset extension of the order $>^{\mathcal{O}}$ on \mathcal{O} , which turns out to be well-founded [8] by Lem. 3. We are now ready to (inductively) define our **cuts level** measure $\mathbf{C}(_)$ on terms, where the following operation on multisets is used $p \cdot M := [\mathbf{a}(p + k, n) \mid \mathbf{a}(k, n) \in M] \sqcup [\mathbf{b}(p + k) \mid \mathbf{b}(k) \in M]$, where \sqcup denotes multiset union.

$$\begin{aligned} \mathbf{C}(x) & := [] & \mathbf{C}(\lambda x.t) & := \mathbf{C}(t) & \mathbf{C}(tu) & := \mathbf{C}(t) \sqcup \mathbf{C}(u) \\ \mathbf{C}(t[x \setminus u]) & := \mathbf{C}(t) \sqcup (\mathbf{lv}_x(t) + 1) \cdot \mathbf{C}(u) \sqcup [\mathbf{a}(\mathbf{lv}_x(t) + 1, |u|)] \\ \mathbf{C}(t[x \setminus\setminus u]) & := \mathbf{C}(t) \sqcup \mathbf{lv}_x(t) \cdot \mathbf{C}(u) \sqcup [\mathbf{b}(\mathbf{lv}_x(t))] \end{aligned}$$

Intuitively, the integer k in $\mathbf{a}(k, n)$ and $\mathbf{b}(k)$ counts the level of variables bound by explicit cuts, while n counts the size of terms to be substituted by an ES. Remark that for every pure term p we have $\mathbf{C}(p) = []$. Moreover:

Lemma 4. *Let $t_0 \in \Lambda_{\mathbf{R}}$. Then $t_0 \rightarrow_{\pi} t_1$ (resp. $t_0 \rightarrow_{\mathbf{s}} t_1$) implies $\mathbf{C}(t_0) \geq_{\text{MUL}}^{\mathcal{O}} \mathbf{C}(t_1)$ (resp. $\mathbf{C}(t_0) >_{\text{MUL}}^{\mathcal{O}} \mathbf{C}(t_1)$).*

As an example, consider the following reduction sequence:

$$\begin{aligned} t_0 &= (yy)[y \setminus (\lambda z.x)w] && \rightarrow_{\text{app}} (y_1 y_2)(y_1 y_2)[y_1 \setminus \lambda z.x][y_2 \setminus w] = t_1 \rightarrow_{\text{var}} \\ t_2 &= (y_1 w)(y_1 w)[y_1 \setminus \lambda z.x] && \rightarrow_{\text{dist}} (y_1 w)(y_1 w)[y_1 \setminus \lambda z.r[r \setminus x]] = t_3 \end{aligned}$$

We have $\mathbf{C}(t_0) = [\mathbf{a}(1, 4)]$, $\mathbf{C}(t_1) = [\mathbf{a}(1, 1), \mathbf{a}(1, 2)]$, $\mathbf{C}(t_2) = [\mathbf{a}(1, 2)]$, $\mathbf{C}(t_3) = [\mathbf{a}(1, 1), \mathbf{b}(0)]$. So $\mathbf{C}(t_i) >_{\text{MUL}} \mathbf{C}(t_{i+1})$ for $i = 0, 1, 2, 3$.

Corollary 5. *The reduction relation $\rightarrow_{\mathbf{s}}$ is terminating.*

Simulations. We show the relation between $\lambda\mathbf{R}$ and the λ -calculus, as well as the atomic λ -calculus. For that, we introduce a projection from $\lambda\mathbf{R}$ -terms to λ -terms implementing the unfolding of all the explicit cuts: $x^{\downarrow} := x$, $(\lambda x.t)^{\downarrow} := \lambda x.t^{\downarrow}$, $(tu)^{\downarrow} := t^{\downarrow}u^{\downarrow}$, $(t[x \triangleleft u])^{\downarrow} := t^{\downarrow}\{x \setminus u^{\downarrow}\}$. Thus e.g. $x[x \setminus z][y \setminus w][w \setminus w']^{\downarrow} = z$.

Lemma 6. *Let $t_0 \in \Lambda_{\mathbf{R}}$. If $t_0 \rightarrow_{\mathbf{R}} t_1$, then $t_0^{\downarrow} \rightarrow_{\beta}^* t_1^{\downarrow}$. In particular, if either $t_0 \rightarrow_{\pi} t_1$ or $t_0 \rightarrow_{\mathbf{s}} t_1$, then $t_0^{\downarrow} = t_1^{\downarrow}$.*

The relation $\rightarrow_{\mathbf{s}}$ enjoys **full composition** on *pure* terms, namely, for any $p \in \Lambda$, $t[x \setminus p] \rightarrow_{\mathbf{s}}^+ t\{x \setminus p\}$. This property does not hold in general. Indeed, if $t = xx$, then $(xx)[x \setminus z[z \setminus w]]$ does not \mathbf{s} -reduce to $(z[z \setminus w])(z[z \setminus w])$, but to $(zz)[z \setminus w]$. However, full composition restricted to pure terms is sufficient to prove simulation of the λ -calculus.

Lemma 7 (Simulation of the λ -calculus). *Let $p_0 \in \Lambda$. If $p_0 \rightarrow_{\beta} p_1$, then $p_0 \rightarrow_{\text{AB}} \rightarrow_{\mathbf{s}}^+ p_1$.*

The previous results have an important consequence relating the original atomic λ -calculus and the $\lambda\mathbf{R}$ -calculus. Indeed, it can be shown that reduction in the atomic λ -calculus is captured by $\lambda\mathbf{R}$, and vice-versa. More precisely, the $\lambda\mathbf{R}$ -calculus can be simulated into the atomic λ -calculus by Lem. 6 and [33], while the converse holds by [33] and Lem. 7.

A more structural correspondence between $\lambda\mathbf{R}$ and the atomic λ -calculus could also be established. Indeed, $\lambda\mathbf{R}$ can be first refined into a (non-linear) calculus *without* distance, let say $\lambda\mathbf{R}'$, so that permutation rules are integrated in the intermediate calculus as independent rules. Then a structural relation can be established between $\lambda\mathbf{R}$ and $\lambda\mathbf{R}'$ on one side, and $\lambda\mathbf{R}'$ and the atomic λ -calculus on the other side (as for example done in [43] for the λ -calculus).

Confluence. By Cor. 5 the reduction relation $\rightarrow_{\mathbf{s}}$ is terminating. It is then not difficult to conclude confluence of $\rightarrow_{\mathbf{s}}$ by using the unfolding function \cdot^{\downarrow} . Therefore, by termination of $\rightarrow_{\mathbf{s}}$ any $t \in \Lambda_{\mathbf{R}}$ has an \mathbf{s} -nf, and by confluence this \mathbf{s} -nf is unique (and computed by the unfolding function). Using the interpretation method [35] together with Lem. 6, Cor. 5, and Lem. 7, one obtains:

Theorem 8. *The reduction relation $\rightarrow_{\mathbf{R}}$ is confluent.*

4 Encoding Evaluation Strategies

In the theory of programming languages [56], the notion of *calculus* is usually based on a non-deterministic rewriting relation, providing an equational system of calculation, while the deterministic notion of *strategy* is associated to a concrete machinery being able to implement a specific evaluation procedure. Typical evaluation strategies are call-by-name, call-by-value, call-by-need, etc.

Although the atomic λ -calculus was introduced as a technical tool to implement full laziness, only its (non-deterministic) equational theories was studied. In this paper we bridge the gap between the theoretical presentation of the atomic λ -calculus and concrete specifications of evaluation strategies. Indeed, we use the $\lambda\mathbb{R}$ -calculus to investigate two concrete cases: a call-by-name strategy implementing weak head reduction, based on full substitution, and the call-by-need fully lazy strategy, which uses linear substitution.

In both cases, explicit cuts can in principle be placed anywhere in the distributors, thus demanding to dive deep in such terms to deal with them. We then restrict the set of terms to a subset \mathbb{U} , which simplifies the formal reasoning of explicit cuts inside distributors. Indeed, distributors will all be of the shape $[x \backslash \lambda y.L\langle p \rangle]$, where p is a pure term (and L is a *commutative list* defined below). We argue that this restriction is natural in a weak implementation of the λ -calculus: it is true on pure terms and is preserved through evaluation. We consider the following grammars.

$$\begin{array}{ll}
 \text{(Linear Cut Values)} & \mathbb{T} ::= \lambda x.LL\langle p \rangle \text{ where } y \in \text{d1c}(LL) \implies |p|_y = 1 \\
 \text{(Commutative Lists)} & LL ::= \square \mid LL[x \backslash p] \mid LL[x \backslash \mathbb{T}] \text{ where } |LL|_x = 0 \\
 \text{(Values)} & v ::= \lambda x.p \\
 \text{(Restricted Terms)} & \mathbb{U} ::= x \mid v \mid \mathbb{U}\mathbb{U} \mid \mathbb{U}[x \backslash \mathbb{U}] \mid \mathbb{U}[x \backslash \mathbb{T}]
 \end{array}$$

A term t generated by any of the grammars G defined above is written $t \in G$. Thus *e.g.* $\lambda x.(yz)[y \backslash \mathbb{I}][z \backslash \mathbb{I}] \in \mathbb{T}$ but $\lambda x.(yy)[y \backslash \mathbb{I}] \notin \mathbb{T}$, $\square[x \backslash yz][x' \backslash \mathbb{I}] \in LL$ but $\square[x \backslash yz][y \backslash \mathbb{I}] \notin LL$, and $(yz)[y \backslash \mathbb{I}] \in \mathbb{U}$ but $(yz)[y \backslash \lambda x.(yy)[y \backslash \mathbb{I}]] \notin \mathbb{U}$.

The set \mathbb{T} is stable by the relation \rightarrow_s , but \mathbb{U} is clearly not stable under the whole $\rightarrow_{\mathbb{R}}$ relation, where dB -reductions may occur under abstractions. However, \mathbb{U} is stable under both weak strategies to be defined: call-by-name and call-by-need. We factorize the proofs by proving stability for a more general relation $\rightarrow_{\mathbb{R}'}$, defined as the relation $\rightarrow_{\mathbb{R}}$ with dB -reductions forbidden under abstractions and inside distributors.

Lemma 9 (Stability of the Grammar by $\rightarrow_s/\rightarrow_{\mathbb{R}'}$).

1. If $t \in \mathbb{T}$ and $t \rightarrow_s t'$, then $t' \in \mathbb{T}$.
2. If $t \in \mathbb{U}$ and $t \rightarrow_{\mathbb{R}'} t'$, then $t' \in \mathbb{U}$.

4.1 Call-by-name

The **call-by-name** (CBN) strategy $\rightarrow_{\text{name}}$ (Fig. 1) is defined on the set of terms \mathbb{U} as the union of the following relations \rightarrow_{ndb} and \rightarrow_{ns} . The strategy is *weak* as there is no reduction under abstractions. It is also worth noticing (as a particular case of Lem. 9) that $t \in \mathbb{U}$ and $t \rightarrow_{\text{name}} t'$ implies $t' \in \mathbb{U}$.

$\frac{t \mapsto_{\text{dB}} t'}{t \rightarrow_{\text{ndb}} t'} \quad (\text{dB})$	$\frac{t \rightarrow_{\text{ndb}} t'}{tu \rightarrow_{\text{ndb}} t'u} \quad (\text{app_dB})$	$\frac{t \rightarrow_{\text{ndb}} t'}{t[x \triangleleft u] \rightarrow_{\text{ndb}} t'[x \triangleleft u]} \quad (\text{sub_dB})$
$\frac{t \mapsto_{\text{s}} t'}{t \rightarrow_{\text{ns}} t'} \quad (\text{s})$	$\frac{t \rightarrow_{\text{ns}} t'}{tu \rightarrow_{\text{ns}} t'u} \quad (\text{app_s})$	$\frac{t \rightarrow_{\text{ns}} t'}{u[x \parallel \lambda y. t] \rightarrow_{\text{ns}} u[x \parallel \lambda y. t']} \quad (\text{sub_s})$

Fig. 1. Call-by-Name Strategy

Example 10. Let $t_0 = (\lambda x_1. \mathbb{I}(x_1 \mathbb{I}))(\lambda y. \mathbb{I}y)$. Then,

$$\begin{aligned}
t_0 &\rightarrow_{\text{dB}}^{(\text{dB})} (\mathbb{I}(x_1 \mathbb{I})) [x_1 \setminus \lambda y. \mathbb{I}y] \rightarrow_{\text{dist}}^{(\text{s})} (\mathbb{I}(x_1 \mathbb{I})) [x_1 \setminus \lambda y. z [z \setminus \mathbb{I}y]] \rightarrow_{\text{app}}^{(\text{sub_s})} \\
&(\mathbb{I}(x_1 \mathbb{I})) [x_1 \setminus \lambda y. (z_1 z_2) [z_1 \setminus \mathbb{I}] [z_2 \setminus y]] \rightarrow_{\text{var}}^{(\text{sub_s})} (\mathbb{I}(x_1 \mathbb{I})) [x_1 \setminus \lambda y. (z_1 y) [z_1 \setminus \mathbb{I}]] \rightarrow_{\text{abs}}^{(\text{s})} \\
&(\mathbb{I}((\lambda y. z_1 y) \mathbb{I})) [z_1 \setminus \mathbb{I}] \rightarrow_{\text{dB}}^{(\text{sub_dB})} x_2 [x_2 \setminus (\lambda y. z_1 y) \mathbb{I}] [z_1 \setminus \mathbb{I}]
\end{aligned}$$

Although the strategy $\rightarrow_{\text{name}}$ is not deterministic, it enjoys the remarkable *diamond* property, guaranteeing in particular that all reduction sequences starting from t and ending in a normal form have the same length.

It is worth noticing that simulation lemmas also hold between call-by-name in the λ -calculus, known as weak head reduction and denoted by \rightarrow_{whr} , and the $\lambda\mathbb{R}$ -calculus. Indeed, \rightarrow_{whr} is defined as the β -reduction rule closed by contexts $E ::= \square \mid Et$. Then, as a consequence of Lem. 7, we have that $p_0 \rightarrow_{\text{whr}} p_1$ implies $p_0 \rightarrow_{\mathbb{R}}^* p_1$, and as a consequence of Lem. 6, we have that $t_0 \rightarrow_{\text{name}} t_1$ implies $t_0^\downarrow \rightarrow_{\beta}^* t_1^\downarrow$. More importantly, call-by-name in the λ -calculus and call-by-name in the $\lambda\mathbb{R}$ -calculus are also related. Indeed,

Lemma 11 (Relating Call-by-Name Strategies).

- Let $p_0 \in \Lambda$. If $p_0 \rightarrow_{\text{whr}} p_1$ then $p_0 \rightarrow_{\text{name}}^+ p_1$.
- Let $t_0 \in \mathcal{U}$. If $t_0 \rightarrow_{\text{name}} t_1$ then $t_0^\downarrow \rightarrow_{\text{whr}}^* t_1^\downarrow$.

4.2 Call-by-need

We now specify a deterministic strategy `fneed` implementing demand-driven computations and only linearly replicating nodes of *values* (i.e. pure abstractions). Given a value $\lambda x.p$, only the piece of structure containing the paths between the binder λx and all the free occurrences of x in p , named *skeleton*, will be copied. All the other components of the abstraction will remain shared, thus avoiding some future duplications of redexes, as explained in the introduction. By copying only the smallest possible substructure of the abstraction, the strategy `fneed` implements an optimization of call-by-need called *fully lazy sharing* [60]. First, we formally define the key notions we are going to use.

A **free expression** [39,9] of a *pure* term p is a strict subterm q of p such that every free occurrence of a variable in q is also a free occurrence of the variable in p . A **free expression** of p is **maximal** if it is not a subterm of

another free expression of p . From now on, we will consider the multiset of all maximal free expressions (**MFE**) of a term. Thus *e.g.* the MFEs of $\lambda y.p$, where $p = (\mathbb{I}y)\mathbb{I}(\lambda z.zyw)$, is given by the multiset $[\mathbb{I}, \mathbb{I}, w]$.

An n -**ary context** ($n \geq 0$) is a term with n holes \square . A skeleton is an n -ary pure context where the maximal free expressions w.r.t. a variable set θ are replaced with holes. Formally, the θ -**skeleton** $\{\{p\}\}^\theta$ of a pure term p , where $\theta = \{x_1 \dots x_n\}$, is the n -ary pure context $\{\{p\}\}^\theta$ such that $\{\{p\}\}^\theta \langle q_1, \dots, q_n \rangle = p$, for $[q_1, \dots, q_n]$ the maximal free expressions of $\lambda x_1 \dots \lambda x_n.p$ ⁴. Thus, for the same p as before, $\lambda y.\{\{p\}\}^y = \lambda y.(\square y)\square(\lambda z.zy\square)$.

The Splitting Operation. Splitting a term into a skeleton and a multiset of MFEs is at the core of full laziness. This can naturally be implemented in the node replication model, as observed in [33]. Here, we define a (small-step) strategy \rightarrow_{st} on the set of terms \mathbf{T} to achieve it (Fig. 2), which is indeed a subset of the reduction relation $\lambda\mathbf{R}$ ⁵. The relation \rightarrow_{st} makes use of four basic rules which are parameterized by the variable y upon which the skeleton is built, written \mapsto^y . There are also two contextual (inductive) rules.

$\frac{}{t[x \setminus y] \mapsto_{\text{var}}^y t\{x \setminus y\}}$	$\frac{y \in \text{fv}(p_1 p_2)}{t[x \setminus p_1 p_2] \mapsto_{\text{app}}^y t\{x \setminus x_1 x_2\}[x_1 \setminus p_1][x_2 \setminus p_2]}$
$\frac{y \in \text{fv}(\lambda z.p)}{t[x \setminus \lambda z.p] \mapsto_{\text{dist}}^y t[x \setminus \lambda z.w[w \setminus p]]}$	$\frac{y \in \text{fv}(\lambda z.\text{LL}\langle p \rangle) \quad z \notin \text{fv}(\text{LL})}{t[x \setminus \lambda z.\text{LL}\langle p \rangle] \mapsto_{\text{abs}}^y \text{LL}\langle t\{x \setminus \lambda z.p\} \rangle}$
$\frac{t \mapsto^y t' \quad y \in \text{fv}(t) \quad y \notin \text{fv}(\text{LL})}{\lambda y.\text{LL}\langle t \rangle \rightarrow_{\text{st}} \lambda y.\text{LL}\langle t' \rangle} \text{ctx}_1$	$\frac{t \rightarrow_{\text{st}} t' \quad y \in \text{fv}(t) \quad y \notin \text{fv}(\text{LL})}{\lambda y.\text{LL}\langle u[x \setminus t] \rangle \rightarrow_{\text{st}} \lambda y.\text{LL}\langle u[x \setminus t'] \rangle} \text{ctx}_2$

Fig. 2. Relation \rightarrow_{st} : Splitting Skeleton and MFEs in Small-Step Semantics

Example 12. Let $y, z \notin \text{fv}(t)$, so that t is the MFE of $\lambda y.x[x \setminus \lambda z.(yt)z]$. Then,

$$\begin{aligned} & \lambda y.x[x \setminus \lambda z.(yt)z] \rightarrow_{\text{dist}}^y \lambda y.x[x \setminus \lambda z.w[w \setminus (yt)z]] \rightarrow_{\text{app}}^z \\ & \lambda y.x[x \setminus \lambda z.(w_1 w_2)[w_1 \setminus yt][w_2 \setminus z]] \rightarrow_{\text{var}}^z \lambda y.x[x \setminus \lambda z.(w_1 z) [w_1 \setminus yt]] \rightarrow_{\text{abs}}^y \\ & \lambda y.(\lambda z.w_1 z)[w_1 \setminus yt] \rightarrow_{\text{app}}^y \lambda y.(\lambda z.(x_1 x_2)z)[x_1 \setminus y][x_2 \setminus t] \rightarrow_{\text{var}}^y \lambda y.(\lambda z.(yx_2)z)[x_2 \setminus t] \end{aligned}$$

Notice that the focused variable changes from y to z , then back to y . This is because \rightarrow_{st} constructs the innermost skeletons first.

Lemma 13. *The reduction relation \rightarrow_{st} is confluent and terminating.*

Thus, from now on, we denote by \Downarrow_{st} the function relating a term of \mathbf{T} to its unique **st-nf**.

⁴ The order of variables in the set θ is indeed irrelevant.

⁵ Since \rightarrow_{st} acts only on terms in \mathbf{T} , it is handled by linear substitution.

Lemma 14 (Correctness of \rightarrow_{st}). *Let $p \in \Lambda$ and q_1, \dots, q_n be the MFEs of $\lambda y.p$. Then $\lambda y.z[z \setminus p] \Downarrow_{\text{st}} \lambda y.\{\{p\}\}^{\{y\}} \langle x_1, \dots, x_n \rangle [x_i \setminus q_i]_{i \leq n}$ where the variables x_1, \dots, x_n are fresh and pairwise distinct.*

Since the small-step semantics is contained in the $\lambda\mathbb{R}$ -calculus, we use it to build our call-by-need strategy of $\lambda\mathbb{R}$.

The strategy. The **call-by-need strategy** $\rightarrow_{\text{f1need}}$ (Fig. 3) is defined on the set of terms \mathbb{U} , by using closure under the *need contexts*, given by the grammar $\mathbb{N} ::= \square \mid \mathbb{N}t \mid \mathbb{N}[x \triangleleft t] \mid \mathbb{N}\langle\langle x \rangle\rangle[x \setminus \mathbb{N}]$, where $\mathbb{N}\langle\langle - \rangle\rangle$ denotes capture-free application of contexts (Sec. 2). As for call-by-name (Sec. 4.1), the call-by-need strategy is *weak*, because no *meaningful* reduction steps are performed under abstractions.

$$\begin{array}{l} \mathbb{L}\langle\lambda x.p\rangle u \quad \mapsto_{\text{dB}} \mathbb{L}\langle p[x \setminus u] \rangle \\ \mathbb{N}\langle\langle x \rangle\rangle[x \setminus \mathbb{L}\langle\lambda y.p\rangle] \mapsto_{\text{sp1}} \mathbb{L}\langle\mathbb{L}\langle\mathbb{N}\langle\langle x \rangle\rangle[x \setminus \lambda y.p']\rangle\rangle \quad \text{if } \lambda y.z[z \setminus p] \Downarrow_{\text{st}} \lambda y.\mathbb{L}\langle p' \rangle \\ \mathbb{N}\langle\langle x \rangle\rangle[x \setminus v] \quad \mapsto_{\text{sub}} \mathbb{N}\langle\langle v \rangle\rangle[x \setminus v] \end{array}$$

Fig. 3. Call-by-Need Strategy

Rule **dB** is the same one used to define **name**. Although rules **sp1** and **sub** could have been presented in a unique rule of the form $\mathbb{N}\langle\langle x \rangle\rangle[x \setminus \mathbb{L}\langle\lambda y.p\rangle] \mapsto \mathbb{L}\langle\mathbb{L}\langle\mathbb{N}\langle\langle \lambda y.p' \rangle\rangle[x \setminus \lambda y.p']\rangle\rangle$, we prefer to keep them separate since they represent different stages in the strategy. Indeed, rule **sp1** only uses node replication operations to compute the skeleton of the abstraction, while rule **sub** implements one-shot *linear* substitution.

Notice that as a particular case of Lem. 9, $t \in \mathbb{U}$ and $t \rightarrow_{\text{f1need}} t'$ implies $t' \in \mathbb{U}$. Another interesting property is that $t \rightarrow_{\text{sub}} t'$ implies $\mathbf{1}_{\mathbf{v}_z}(t) \geq \mathbf{1}_{\mathbf{v}_z}(t')$. Moreover, $\rightarrow_{\text{f1need}}$ is deterministic.

Example 15. Let $t_0 = (\lambda x.(\mathbb{I}(Ix)))\lambda y.y\mathbb{I}$. Needed variable occurrences are highlighted in **orange**.

$$\begin{array}{l} t_0 \rightarrow_{\text{dB}} (\mathbb{I}(Ix))[x \setminus \lambda y.y\mathbb{I}] \rightarrow_{\text{dB}} x_1[x_1 \setminus \mathbb{I}x][x \setminus \lambda y.y\mathbb{I}] \\ \rightarrow_{\text{dB}} x_1[x_1 \setminus x_2[x_2 \setminus x]][x \setminus \lambda y.y\mathbb{I}] \rightarrow_{\text{sp1}} x_1[x_1 \setminus x_2[x_2 \setminus x]][x \setminus \lambda y.yz_1][z_1 \setminus \mathbb{I}] \\ \rightarrow_{\text{sub}} x_1[x_1 \setminus x_2[x_2 \setminus \lambda y.yz_1]][x \setminus \lambda y.yz_1][z_1 \setminus \mathbb{I}] \\ \rightarrow_{\text{sp1}} x_1[x_1 \setminus x_2[x_2 \setminus \lambda y.yz_2][z_2 \setminus z_1]][x \setminus \lambda y.yz_1][z_1 \setminus \mathbb{I}] \\ \rightarrow_{\text{sub}} x_1[x_1 \setminus (\lambda y.yz_2) [x_2 \setminus \lambda y.yz_2][z_2 \setminus z_1]][x \setminus \lambda y.yz_1][z_1 \setminus \mathbb{I}] \\ \rightarrow_{\text{sp1}} x_1[x_1 \setminus \lambda y.yz_3][z_3 \setminus z_2][x_2 \setminus \lambda y.yz_2][z_2 \setminus z_1][x \setminus \lambda y.yz_1][z_1 \setminus \mathbb{I}] \\ \rightarrow_{\text{sub}} (\lambda y.yz_3)[x_1 \setminus \lambda y.yz_3][z_3 \setminus z_2][x_2 \setminus \lambda y.yz_2][z_2 \setminus z_1][x \setminus \lambda y.yz_1][z_1 \setminus \mathbb{I}] \end{array}$$

5 A Type System for the $\lambda\mathbb{R}$ -calculus

This section introduces a quantitative type system \mathcal{V} for the $\lambda\mathbb{R}$ -calculus. Non-idempotent intersection [26] has one main advantage over the idempotent model

[14]: it gives *quantitative* information about the length of reduction sequences to normal forms [21]. Indeed, not only typability and normalization can be proved to be equivalent, but a measure based on type derivations provides an *upper bound* to normalizing reduction sequences. This was extensively investigated in different logical/computational frameworks [5,18,20,25,42,47]. However, no quantitative result based on types exists in the literature for the node replication model, including the attempts done for deep inference [30]. The typing rules of our system are in themselves not surprising (see [46]), but they provide a handy quantitative characterization of fully lazy normalization (Sec. 6).

Types are built on the following grammar of types and multi-types, where α ranges over a set of base types and \mathbf{a} is a special type constant used to type terms reducing to normal abstractions.

$$\text{(Types)} \quad \sigma := \mathbf{a} \mid \alpha \mid \mathcal{M} \rightarrow \sigma \quad \text{(Multi-Types)} \quad \mathcal{M} := [\sigma_i]_{i \in I}$$

We write $|\mathcal{M}|$ to denote the **size of a multi-type** \mathcal{M} . **Typing contexts**, written Γ, Δ, Σ are functions from variables to multiset types, assigning the empty multiset to all but a finite set of variables. The domain of Γ is given by $\text{dom}(\Gamma) := \{x \mid \Gamma(x) \neq []\}$. The **union of contexts**, written $\Gamma + \Delta$, is defined by $(\Gamma + \Delta)(x) := \Gamma(x) \sqcup \Delta(x)$, where \sqcup denotes multiset union. An example is $(x : [\sigma], y : [\tau]) + (x : [\sigma], z : [\tau]) = (x : [\sigma, \sigma], y : [\tau], z : [\tau])$. This notion is extended to several contexts as expected, so that $+_{i \in I} \Gamma_i$ denotes a finite union of contexts, and the empty context when $I = \emptyset$. We write $\Gamma; \Delta$ for $\Gamma + \Delta$ when $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset$. **Type judgments** have the form $\Gamma \vdash t : \sigma$, where Γ is a typing context, t is a term and σ is a type.

$\frac{}{x : [\sigma] \vdash x : \sigma} \text{(ax)}$	$\frac{\Gamma; x : \mathcal{M} \vdash t : \sigma}{\Gamma \vdash \lambda x.t : \mathcal{M} \rightarrow \sigma} \text{(abs)}$	$\frac{\Gamma \vdash t : \mathcal{M} \rightarrow \sigma \quad \Delta \vdash u : \mathcal{M}}{\Gamma + \Delta \vdash t u : \sigma} \text{(app)}$
$\frac{}{\vdash \lambda x.t : \mathbf{a}} \text{(ans)}$	$\frac{(\Gamma_i \vdash t : \sigma_i)_{i \in I}}{+_{i \in I} \Gamma_i \vdash t : [\sigma_i]_{i \in I}} \text{(many)}$	$\frac{\Gamma; x : \mathcal{M} \vdash t : \sigma \quad \Delta \vdash u : \mathcal{M}}{\Gamma + \Delta \vdash t[x \triangleleft u] : \sigma} \text{(cut)}$

Fig. 4. Typing System \mathcal{V}

A **(typing) derivation** is a tree obtained by applying the (inductive) typing rules of system \mathcal{V} (Fig. 4), introduced in [46]. The notation $\Phi \triangleright \Gamma \vdash t : \sigma$ means there is a derivation named Φ of the judgment $\Gamma \vdash t : \sigma$ in system \mathcal{V} . A term t is typable in system \mathcal{V} , or \mathcal{V} -typable, iff there is a context Γ and a type σ such that $\Phi \triangleright \Gamma \vdash t : \sigma$. The **size of a type derivation** $\text{sz}(\Phi)$ is defined as the number of its **abs**, **app** and **ans** rules. The typing system is **relevant** in the sense that $\Phi \triangleright \Gamma \vdash t : \sigma$ implies $\text{dom}(\Gamma) \subseteq \text{fv}(t)$.

Type derivations can be measured by 3-tuples. We use a $+$ operation on 3-tuples as pointwise addition: $(a, b, c) + (e, f, g) = (a + e, b + f, c + g)$. These 3-tuples are computed by a **weighted derivation level** function defined on typing derivations as $\text{D}(\Phi) := \text{M}(\Phi, 1)$, where $\text{M}(-, -)$ is inductively defined below. In

the cases **(abs)**, **(app)** and **(cut)**, we let Φ_t (resp. Φ_u) be the subderivation of the type of t (resp. Φ_u) and in **(many)** we let Φ_t^i be the i -th derivation of the type of t for each $i \in I$.

- For **(ax)**, $M(\Phi_x, m) = (0, 0, 1)$,
- For **(abs)**, $M(\Phi_{\lambda x.t}, m) = M(\Phi_t, m) + (1, m, 0)$.
- For **(ans)**, $M(\Phi_{\lambda x.t}, m) = (1, m, 0)$.
- For **(app)**, $M(\Phi_{tu}, m) = M(\Phi_t, m) + M(\Phi_u, m) + (1, m, 0)$.
- For **(cut)**, $M(\Phi_{t[x\triangleleft u]}, m) = M(\Phi_t, m) + M(\Phi_u, m + 1\mathbf{v}_x(t) + x(\triangleleft))$.
- For **(many)**, $M(\Phi_t, m) = \sum_{i \in I} M(\Phi_t^i, m)$.

Notice that the first and the third components of any 3-tuple $M(\Phi, m)$ do not depend on m . Intuitively, the first (resp. third) component of the 3-tuple counts the number of application/abstraction (resp. **(ax)**) rules in the typing derivation. The second one takes into account the number of application/abstraction rules as well, but *weighted* by the level of the constructor. The 3-tuples are ordered lexicographically.

Example 16. Let $\sigma = [\tau] \rightarrow \tau$. Consider the following type derivation Φ :

$$\frac{\frac{\frac{x : [\tau] \vdash x : \tau \quad (\mathbf{ax})}{y : [\sigma] \vdash y : \sigma} \quad (\mathbf{ax}) \quad \frac{\frac{z : [\tau] \vdash z : \tau \quad (\mathbf{ax})}{z : [\tau] \vdash z : [\tau]} \quad (\mathbf{many})}{z : [\tau] \vdash z : [\tau]} \quad (\mathbf{app})}{y : [\sigma], z : [\tau] \vdash yz : \tau} \quad (\mathbf{many})}{y : [\sigma], z : [\tau] \vdash x[x \setminus yz] : \tau} \quad (\mathbf{cut})}{x : [\tau] \vdash x : \tau} \quad (\mathbf{ax})$$

This gives $D(\Phi) = (1, 2, 3)$. Moreover, for $x[x \setminus yz] \rightarrow_{\mathbf{app}} (x_1 x_2)[x_1 \setminus y][x_2 \setminus z]$ we have $\Phi' \triangleright y : [\sigma], z : [\tau] \vdash (x_1 x_2)[x_1 \setminus y][x_2 \setminus z] : \tau$ and $D(\Phi') = (1, 1, 4)$.

6 Observational Equivalence

The type system \mathcal{V} characterizes normalization of both **name** and **flneed** strategies as follows: every typable term normalizes and every normalisable term is typable. In this sense, system \mathcal{V} can be seen as a (quantitative) *model* [17] of our call-by-name and call-by-need strategies. We prove these results by studying the appropriate lemmas, notably weighted subject reduction and weighted subject expansion. We then deduce observational equivalence between the **name** and the **flneed** strategies from the fact that their associated normalization properties are both fully characterized by the same typing system.

Soundness. Soundness of system \mathcal{V} w.r.t. both $\rightarrow_{\mathbf{name}}$ and $\rightarrow_{\mathbf{flneed}}$ is investigated in this section. More precisely, we show that typable terms are normalizing for both strategies. In contrast to reducibility techniques needed to show this kind of result for simple types [34], soundness is achieved here by relatively simple combinatorial arguments based again on decreasing measures. We start by studying the interaction between system \mathcal{V} and linear as well as full substitution.

Lemma 17 (Partial Substitution). *Let $\Phi \triangleright \Gamma; x : \mathcal{M} \vdash \mathbb{C}\langle\langle x \rangle\rangle : \sigma$ and \sqsubseteq denote multiset inclusion. Then, there exists $\mathcal{N} \sqsubseteq \mathcal{M}$ such that for every $\Phi_u \triangleright \Delta \vdash u : \mathcal{N}$ we have $\Psi \triangleright \Gamma + \Delta; x : \mathcal{M} \setminus \mathcal{N} \vdash \mathbb{C}\langle\langle u \rangle\rangle : \sigma$ and, for every $m \in \mathbb{N}$, $\mathbb{M}(\Psi, m) = \mathbb{M}(\Phi, m) + \mathbb{M}(\Phi_u, m + 1\nu_{\square}(\mathbb{C})) - (0, 0, |\mathcal{N}|)$.*

Corollary 18 (Substitution). *If $\Phi_t \triangleright \Gamma; x : \mathcal{M} \vdash t : \sigma$ and $\Phi_u \triangleright \Delta \vdash u : \mathcal{M}$, then $\Phi \triangleright \Gamma + \Delta \vdash t\{x \setminus u\} : \sigma$, and for all $m \in \mathbb{N}$ we have $\mathbb{M}(\Phi, m) \leq \mathbb{M}(\Phi_t, m) + \mathbb{M}(\Phi_u, m + 1\nu_x(t))$. Moreover, $|\mathcal{M}| > 0$ iff the inequality is strict.*

The key idea to show soundness is that the measure $\mathbb{D}(\cdot)$ decreases w.r.t. the reduction relations $\rightarrow_{\text{name}}$ and $\rightarrow_{\text{flneed}}$:

Lemma 19 (Weighted Subject Reduction). *Let $\Phi_{t_0} \triangleright \Gamma \vdash t_0 : \sigma$.*

1. *If $t_0 \rightarrow_{\pi} t_1$, then there exists $\Phi_{t_1} \triangleright \Gamma \vdash t_1 : \sigma$ such that $\mathbb{D}(\Phi_{t_0}) = \mathbb{D}(\Phi_{t_1})$.*
2. *If $t_0 \rightarrow_{\mathfrak{s}} t_1$, then there exists $\Phi_{t_1} \triangleright \Gamma \vdash t_1 : \sigma$ such that $\mathbb{D}(\Phi_{t_0}) \geq \mathbb{D}(\Phi_{t_1})$.*
3. *If $t_0 \rightarrow_{\text{ndb}} t_1$, then there exists $\Phi_{t_1} \triangleright \Gamma \vdash t_1 : \sigma$ such that $\mathbb{D}(\Phi_{t_0}) > \mathbb{D}(\Phi_{t_1})$.*
4. *If $t_0 \rightarrow_{\text{flneed}} t_1$, then there exists $\Phi_{t_1} \triangleright \Gamma \vdash t_1 : \sigma$ such that $\mathbb{D}(\Phi_{t_0}) > \mathbb{D}(\Phi_{t_1})$.*

Proof. By induction on $\mathfrak{r} \in \{\pi, \mathfrak{s}, \text{ndb}, \text{flneed}\}$, using Lem. 17 and Cor. 18.

Theorem 20 (Typability implies name-Normalization). *Let $\Phi_t \triangleright \Gamma \vdash t : \sigma$. Then t is name-normalizing.*

Proof. Suppose t is not name-normalizing. Since $\rightarrow_{\mathfrak{s}}$ is terminating by Cor. 5, then every infinite $\rightarrow_{\text{name}}$ -reduction sequence starting at t must necessarily have an infinite number of dB-steps. Moreover, all terms in such an infinite sequence are typed by Lem 19. Therefore, Lem. 19:3 (resp. Lem. 19:2) guarantees that all dB (resp. \mathfrak{s}) reduction steps involved in such $\rightarrow_{\text{name}}$ -reduction sequence strictly decrease (resp. do not increase) the measure $\mathbb{D}(\cdot)$. This leads to a contradiction because the order $>$ on 3-tuples $\mathbb{D}(\cdot)$ is well-founded. Then t is necessarily name-normalizing.

Theorem 21 (Typability implies flneed-Normalization). *Let $\Phi_t \triangleright \Gamma \vdash t : \sigma$. Then t is flneed-normalizing. Moreover, $\mathbb{D}(\Phi_t)$ is an upper bound to the length of the flneed-reduction evaluation to flneed-nf.*

Proof. The property trivially holds by Lem. 19:4 since the lexicographic order on 3-tuples is well-founded.

Completeness. We address here completeness of system \mathcal{V} with respect to $\rightarrow_{\text{name}}$ and $\rightarrow_{\text{flneed}}$. More precisely, we show that normalizing terms in each strategy are typable. The basic property in showing that consists in guaranteeing that normal forms are typable.

The following lemma makes use of a notion of **needed variable**:
 $\text{nv}(x) := \{x\}$, $\text{nv}(tu) := \text{nv}(t)$, $\text{nv}(t[x \setminus u]) := \text{nv}(t)$, $\text{nv}(\lambda x.t) := \emptyset$, $\text{nv}(t[y \setminus u]) := (\text{nv}(t) \setminus \{y\}) \cup \text{nv}(u)$ if $y \in \text{nv}(t)$ and $\text{nv}(t[y \setminus u]) := \text{nv}(t)$ otherwise.

Lemma 22 (flneed-nfs are Typable). *Let t be in flneed-nf. Then there exists a derivation $\Phi \triangleright \Gamma \vdash t : \tau$ such that for any $x \notin \text{nv}(t)$, $\Gamma(x) = []$.*

Because **name-nfs** are also **flneed-nfs**, we infer the following corollary for free.

Corollary 23 (name-nfs are Typable). *Let t be in **name-nf**. Then there is a derivation $\Phi \triangleright \Gamma \vdash t : \tau$.*

Now we need lemmas stating the behavior of partial and full (anti-)substitution w.r.t. typing.

Lemma 24 (Partial Anti-Substitution). *Let $C\langle\langle x \rangle\rangle, u$ be terms s.t. $x \notin \text{fv}(u)$ and $\Phi \triangleright \Gamma \vdash C\langle\langle u \rangle\rangle : \sigma$. Then $\exists \Gamma', \exists \Delta, \exists \mathcal{M}, \exists \Phi', \exists \Phi_u$ s.t. $\Gamma = \Gamma' + \Delta, \Phi' \triangleright \Gamma' + x : \mathcal{M} \vdash C\langle\langle x \rangle\rangle : \sigma$ and $\Phi_u \triangleright \Delta \vdash u : \mathcal{M}$.*

Corollary 25 (Anti-Substitution). *Let u be a term s.t. $x \notin \text{fv}(u)$ and $\Phi \triangleright \Gamma \vdash t\{x \setminus u\} : \sigma$. Then $\exists \Gamma', \exists \Delta, \exists \mathcal{M}, \exists \Phi', \exists \Phi_u$ s.t. $\Gamma = \Gamma' + \Delta, \Phi' \triangleright \Gamma'; x : \mathcal{M} \vdash t : \sigma$ and $\Phi_u \triangleright \Delta \vdash u : \mathcal{M}$.*

To achieve completeness, we show that typing is preserved by anti-reduction. We decompose the property as follows:

Lemma 26 (Subject Expansion). *Let $\Phi_{t_1} \triangleright \Gamma \vdash t_1 : \sigma$. If $t_0 \rightarrow_{\mathbf{r}} t_1$, where $\mathbf{r} \in \{\pi, \mathbf{s}, \mathbf{ndb}, \mathbf{flneed}\}$, then there exists $\Phi_{t_0} \triangleright \Gamma \vdash t_0 : \sigma$.*

Proof. The proof is by induction on $\rightarrow_{\mathbf{r}}$ and uses Lem. 24 and Cor. 25.

Theorem 27 (name-Normalization implies Typability). *Let t be a term. If t is **name-normalizing**, then t is \mathcal{V} -typable.*

Proof. Let t be **name-normalizing**. Then $t \rightarrow_{\mathbf{name}}^n u$ and u is a **name-nf**. We reason by induction on n . If $n = 0$, then $t = u$ is typable by Cor. 23. Otherwise, we have $t \rightarrow_{\mathbf{name}} t' \rightarrow_{\mathbf{name}}^{n-1} u$. By the *i.h.* t' is typable and thus by Lem. 26 (because $\rightarrow_{\mathbf{ns}}$ is included in $\rightarrow_{\mathbf{s}}$), t turns out to be also typable.

Theorem 28 (flneed-Normalization implies Typability). *Let t be a term. If t is **flneed-normalizing**, then t is \mathcal{V} -typable.*

Proof. Similar to the previous proof but using Lem. 22 instead of Cor. 23.

Summing up, Thms. 20, 27, 21 and 28 give:

Theorem 29. *Let t be a $\lambda\mathbf{R}$ -term. t is **name-normalizing** iff t is **flneed-normalizing** iff t is \mathcal{V} -typable.*

All the technical tools are now available to conclude observational equivalence between our two evaluation strategies based on node replication. Let \mathcal{R} be any reduction notion on $A_{\mathbf{R}}$. Then, two terms $t, u \in A_{\mathbf{R}}$ are said to be **\mathcal{R} -observationally equivalent**, written $t \equiv u$, if for any context C , $C\langle t \rangle$ is \mathcal{R} -normalizing iff $C\langle u \rangle$ is \mathcal{R} -normalizing.

Theorem 30. *For all terms $t, u \in A_{\mathbf{R}}$, t and u are **name-observationally equivalent** iff t and u are **flneed-observationally equivalent**.*

Proof. By Thm. 29, $t \equiv_{\mathbf{name}} u$ means that $C\langle t \rangle$ is \mathcal{V} -typable iff $C\langle u \rangle$ is \mathcal{V} -typable, for all C . By the same theorem, this is also equivalent to say that $C\langle t \rangle$ is **flneed-normalizing** iff $C\langle u \rangle$ is **flneed-normalizing** for any C , *i.e.* $t \equiv_{\mathbf{flneed}} u$.

7 Related Works and Conclusion

Several calculi with ES bridge the gap between formal higher-order calculi and concrete implementations of programming languages (see a survey in [40]). The first of such calculi, *e.g.* [1,16], were all based on *structural* substitution, in the sense that the ES operator is syntactically propagated step-by-step through the term structure until a variable is reached, when the substitution finally takes place. The correspondence between ES and Linear Logic Proof-Nets [24] led to the more recent notion of calculi *at a distance* [6,4,2], enlightening a natural and new application of the Curry-Howard interpretation. These calculi implement linear/partial substitution *at a distance*, where the search of variable occurrences is abstracted out with context-based rewriting rules, and thus no ES propagation rules are necessary. A third model was introduced by the seminal work of Gundersen, Heijltjes, and Parigot [33,34], introducing the atomic λ -calculus to implement node replication.

Inspired by the last approach we introduced the $\lambda\mathbf{R}$ -calculus, capturing the essence of node replication. In contrast to [33], we work with an implicit (structural) mechanism of weakening and contraction, a design choice which aims at focusing and highlighting the node replication model, which is the core of our calculus, so that we obtain a rather simple and natural formalism used in particular to specify evaluation strategies. Indeed, besides the proof of the main operational meta-level properties of our calculus (confluence, termination of the substitution calculus, simulations), we use linear and non-linear versions of $\lambda\mathbf{R}$ to specify evaluation strategies based on node replication, namely call-by-name and call-by-need evaluation strategies.

The first description of call-by-need was given by Wadsworth [60], where reduction is performed on *graphs* instead of terms. Weak call-by-need on *terms* was then introduced by Ariola and Felleisen [7], and by Maraist, Odersky and Wadler [54,53]. Reformulations were introduced by Accattoli, Barenbaum and Mazza [3] and by Chang and Felleisen [22]. Our call-by-need strategy is inspired by the calculus in [3], which uses the distance paradigm [6] to gather together meaningful and permutation rules, by clearly separating *multiplicative* from *exponential* rules, in the sense of Linear Logic [27].

Full laziness has been formalized in different ways. Pointer graphs [60,59] are DAGs allowing for an elegant representation of sharing. Labeled calculi [15] implement pointer graphs by adding annotations to λ -terms, which makes the syntax more difficult to handle. Lambda-lifting [38,39] implements full laziness by resorting to translations from λ -terms to supercombinators. In contrast to all the previous formalisms, our calculus is defined on standard λ -terms with explicit cuts, without the use of any complementary syntactical tool. So is Ariola and Felleisen's call-by-need [7], however, their notion of full laziness relies on external (ad-hoc) meta-level operations used to extract the skeleton. Our specification of call-by-need enjoys fully lazy sharing, where the skeleton extraction operation is internally encoded in the term calculus operational semantics. Last but not least, our calculus has strong links with proof-theory, notably deep inference.

Balabonski [10,9] relates many formalisms of full laziness and shows that they are equivalent when considering the number of β -steps to a normal form. It would then be interesting to understand if his unified approach, (abstractly) stated by means of the theory of residuals [50,51], applies to our own strategy.

We have also studied the calculus from a semantical point of view, by means of intersection types. Indeed, the type system can be seen as a model of our implementations of call-by-name and call-by-need, in the sense that typability and normalization turn out to be equivalent.

Intersection types go back to [23] and have been used to provide characterizations of qualitative [14] as well as quantitative [21] models of the λ -calculus, where typability and normalization coincide. Quantitative models specified by means of non-idempotent types [26,48] were first applied to the λ -calculus (see a survey in [19]) and to several other formalisms ever since, such as call-by-value [25,20], call-by-need [42,5], call-by-push-value [31,18] and classical logic [47]. In the present work, we achieve for the first time a quantitative characterization of fully lazy normalization, which provides upper bounds for the length of reduction sequences to normal forms.

The characterizations provided by intersection type systems sometimes lead to observational equivalence results (*e.g.* [42]). In this work we succeed to prove observational equivalence related to a fully lazy implementation of weak call-by-need, a result which would be extremely involved to prove by means of syntactical tools of rewriting, as done for weak call-by-need in [7]. Moreover, our result implies that our node replication implementation of full laziness is observationally equivalent to standard call-by-name and to weak call-by-need (see [42]), as well as to the more semantical notion of neededness (see [45]).

A Curry-Howard interpretation of the logical *switch* rule of deep inference is given in [58,57] as an end-of-scope operator, thus introducing the *spinal atomic* λ -calculus. The calculus implements a refined optimization of call-by-need, where only the *spine* of the abstraction (tighter than the skeleton) is duplicated. It would be interesting to adapt the $\lambda\mathbf{R}$ -calculus to spine duplication by means of an appropriate end-of-scope operator, such as the one in [37]. Further optimizations might also be considered.

Finally, this paper only considers weak evaluation strategies, *i.e.* with reductions forbidden under abstractions, but it would be interesting to extend our notions to full (strong) evaluations too [29,12]. Extending full laziness to classical logic would be another interesting research direction, possibly taking preliminary ideas from [36]. We would also like to investigate (quantitative) *tight* types for our fully lazy strategy, as done for weak call-by-need in [5], which does not seem evident in our node replication framework.

References

1. Abadi, M., Cardelli, L., Curien, P., Lévy, J.: Explicit substitutions. In: POPL. pp. 31–46. ACM Press (1990)
2. Accattoli, B.: Proof nets and the linear substitution calculus. In: ICTAC. Lecture Notes in Computer Science, vol. 11187, pp. 37–61. Springer (2018)
3. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling abstract machines. In: ICFP. pp. 363–376. ACM (2014)
4. Accattoli, B., Bonelli, E., Kesner, D., Lombardi, C.: A nonstandard standardization theorem. In: POPL. pp. 659–670. ACM (2014)
5. Accattoli, B., Guerrieri, G., Leberle, M.: Types by need. In: ESOP. Lecture Notes in Computer Science, vol. 11423, pp. 410–439. Springer (2019)
6. Accattoli, B., Kesner, D.: The structural *lambda*-calculus. In: CSL. Lecture Notes in Computer Science, vol. 6247, pp. 381–395. Springer (2010)
7. Ariola, Z.M., Felleisen, M.: The call-by-need lambda calculus. *J. Funct. Program.* **7**(3), 265–301 (1997)
8. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press (1998)
9. Balabonski, T.: La plein paresse, une certain optimalité : partage de sous-termes et stratégies de réduction en réécriture d’ordre supérieur. Ph.D. thesis, Paris 7 (2012), <http://www.theses.fr/2012PA077198>
10. Balabonski, T.: A unified approach to fully lazy sharing. In: POPL. pp. 469–480. ACM (2012)
11. Balabonski, T.: Weak optimality, and the meaning of sharing. In: ICFP. pp. 263–274. ACM (2013)
12. Balabonski, T., Barenbaum, P., Bonelli, E., Kesner, D.: Foundations of strong call by need. *Proc. ACM Program. Lang.* **1**(ICFP), 20:1–20:29 (2017)
13. Barendregt, H.P.: The lambda calculus - its syntax and semantics, *Studies in logic and the foundations of mathematics*, vol. 103. North-Holland (1985)
14. Barendregt, H.P., Dekkers, W., Statman, R.: Lambda Calculus with Types. *Perspectives in logic*, Cambridge University Press (2013)
15. Blanc, T., Lévy, J., Maranget, L.: Sharing in the weak lambda-calculus. In: Processes, Terms and Cycles. Lecture Notes in Computer Science, vol. 3838, pp. 70–87. Springer (2005)
16. Bloo, R., Rose, K.: Preservation of strong normalization in named lambda calculi with explicit substitution and garbage collection. In: CSN. pp. 62–72. Netherlands Computer Science Research Foundation (1995)
17. Bucciarelli, A., Ehrhard, T.: On phase semantics and denotational semantics: the exponentials. *Ann. Pure Appl. Log.* **109**(3), 205–241 (2001)
18. Bucciarelli, A., Kesner, D., Ríos, A., Viso, A.: The bang calculus revisited. In: FLOPS. Lecture Notes in Computer Science, vol. 12073, pp. 13–32. Springer (2020)
19. Bucciarelli, A., Kesner, D., Ventura, D.: Non-idempotent intersection types for the lambda-calculus. *Log. J. IGPL* **25**(4), 431–464 (2017)
20. Carraro, A., Guerrieri, G.: A semantical and operational account of call-by-value solvability. In: FoSSaCS. Lecture Notes in Computer Science, vol. 8412, pp. 103–118. Springer (2014)
21. de Carvalho, D.: Sémantiques de la logique linéaire et temps de calcul. Ph.D. thesis, Université Aix-Marseille II (2007), <https://www.theses.fr/2007AIX22066>
22. Chang, S., Felleisen, M.: The call-by-need lambda calculus, revisited. In: ESOP. Lecture Notes in Computer Science, vol. 7211, pp. 128–147. Springer (2012)

23. Coppo, M., Dezani-Ciancaglini, M.: A new type assignment for λ -terms. *Arch. Math. Log.* **19**(1), 139–156 (1978)
24. Di Cosmo, R., Kesner, D., Polonowski, E.: Proof nets and explicit substitutions. *Math. Struct. Comput. Sci.* **13**(3), 409–450 (2003)
25. Ehrhard, T.: Collapsing non-idempotent intersection types. In: *CSL. LIPIcs*, vol. 16, pp. 259–273. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012)
26. Gardner, P.: Discovering needed reductions using type theory. In: *TACS. Lecture Notes in Computer Science*, vol. 789, pp. 555–574. Springer (1994)
27. Girard, J.: Linear logic. *Theor. Comput. Sci.* **50**, 1–102 (1987)
28. Girard, J.Y.: Proof-nets: The parallel syntax for proof-theory, *Lecture Notes in Pure Applied Mathematics*, vol. 180, p. 97124. Marcel Dekker (1996)
29. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: *ICFP*. pp. 235–246. ACM (2002)
30. Guerrieri, G., Heijltjes, W.B., Paulus, J.W.N.: A deep quantitative type system. In: *CSL. LIPIcs*, vol. 183, pp. 24:1–24:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
31. Guerrieri, G., Manzonetto, G.: The bang calculus and the two girard’s translations. In: *Linearity-TLLA@FLoC. EPTCS*, vol. 292, pp. 15–30 (2018)
32. Guglielmi, A., Gundersen, T., Parigot, M.: A proof calculus which reduces syntactic bureaucracy. In: *RTA. LIPIcs*, vol. 6, pp. 135–150. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2010)
33. Gundersen, T., Heijltjes, W., Parigot, M.: Atomic lambda calculus: A typed lambda-calculus with explicit sharing. In: *LICS*. pp. 311–320. IEEE Computer Society (2013)
34. Gundersen, T., Heijltjes, W., Parigot, M.: A proof of strong normalisation of the typed atomic lambda-calculus. In: *LPAR. Lecture Notes in Computer Science*, vol. 8312, pp. 340–354. Springer (2013)
35. Hardin, T.: Confluence results for the pure strong categorical logic CCL: lambda-calculi as subsystems of CCL. *Theor. Comput. Sci.* **65**(3), 291–342 (1989)
36. He, F.: The Atomic Lambda-Mu Calculus. Ph.D. thesis, University of Bath (Jan 2018)
37. Hendriks, D., van Oostrom, V.: λ . In: *CADE. Lecture Notes in Computer Science*, vol. 2741, pp. 136–150. Springer (2003)
38. Hughes, J.: The design and implementation of programming languages. Tech. Rep. PRG-40, Oubl (Jul 1983)
39. Jones, S.L.P.: *The Implementation of Functional Programming Languages*. Prentice-Hall (1987)
40. Kesner, D.: The theory of calculi with explicit substitutions revisited. In: *CSL. Lecture Notes in Computer Science*, vol. 4646, pp. 238–252. Springer (2007)
41. Kesner, D.: A theory of explicit substitutions with safe and full composition. *Log. Methods Comput. Sci.* **5**(3) (2009)
42. Kesner, D.: Reasoning about call-by-need by means of types. In: *FoSSaCS. Lecture Notes in Computer Science*, vol. 9634, pp. 424–441. Springer (2016)
43. Kesner, D., Lengrand, S.: Resource operators for lambda-calculus. *Inf. Comput.* **205**(4), 419–473 (2007)
44. Kesner, D., Renaud, F.: A prismoid framework for languages with resources. *Theor. Comput. Sci.* **412**(37), 4867–4892 (2011)
45. Kesner, D., Ríos, A., Viso, A.: Call-by-need, neededness and all that. In: *FoSSaCS. Lecture Notes in Computer Science*, vol. 10803, pp. 241–257. Springer (2018)

46. Kesner, D., Ventura, D.: Quantitative types for the linear substitution calculus. In: IFIP TCS. Lecture Notes in Computer Science, vol. 8705, pp. 296–310. Springer (2014)
47. Kesner, D., Vial, P.: Non-idempotent types for classical calculi in natural deduction style. *Log. Methods Comput. Sci.* **16**(1) (2020)
48. Kfoury, A.J.: A linearization of the lambda-calculus and consequences. *J. Log. Comput.* **10**(3), 411–436 (2000)
49. Lamping, J.: An algorithm for optimal lambda calculus reduction. In: POPL. pp. 16–30. ACM Press (1990)
50. Lévy, J.J.: Réductions correctes et optimales dans le lambda-calcul. Ph.D. thesis, Université Paris VII (1978)
51. Lévy, J.J.: Optimal reductions in the lambda-calculus. In: To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisms, pp. 159–191. Academic Press Inc (1980)
52. Lévy, J., Maranget, L.: Explicit substitutions and programming languages. In: FSTTCS. Lecture Notes in Computer Science, vol. 1738, pp. 181–200. Springer (1999)
53. Maraist, J., Odersky, M., Turner, D.N., Wadler, P.: Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theor. Comput. Sci.* **228**(1-2), 175–210 (1999)
54. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. *J. Funct. Program.* **8**(3), 275–317 (1998)
55. Paulus, J.W.N., Heijltjes, W.: Deep-inference intersection types. In: Int. Workshop: Twenty Years of Deep Inference. pp. 1–3 (2018), http://t-news.cn/Floc2018/FLoC2018-pages/proceedings_paper_652.pdf
56. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.* **1**(2), 125–159 (1975)
57. Sherratt, D., Heijltjes, W., Gundersen, T., Parigot, M.: Spinal atomic lambda-calculus. In: FoSSaCS. Lecture Notes in Computer Science, vol. 12077, pp. 582–601. Springer (2020)
58. Sherratt, D.R.: A lambda-calculus that achieves full laziness with spine duplication. Ph.D. thesis, University of Bath (Mar 2019)
59. Shivers, O., Wand, M.: Bottom-up beta-reduction: Uplinks and lambda-dags. *Fundam. Informaticae* **103**(1-4), 247–287 (2010)
60. Wadsworth, C.P.: Semantics and Pragmatics of the Lambda Calculus. Ph.D. thesis, Oxford University (1971)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

