AAECC manuscript No.
(will be inserted by the editor)

Lambda-Calculus with Director Strings

Maribel Fernández[1], Ian Mackie[1], François-Régis Sinot[2,⋆]

[1] Department of Computer Science, King's College London
Strand, London WC2R 2LS, UK
e-mail: {maribel,ian}@dcs.kcl.ac.uk
[2] LIX, École Polytechnique, 91128 Palaiseau, France
e-mail: frs@lix.polytechnique.fr

Received: date / Revised version: date

Abstract We present a name free λ-calculus with explicit substitutions, based on a generalised notion of director strings. Terms are annotated with information – directors – that indicate how substitutions should be propagated. We first present a calculus where we can simulate arbitrary β-reduction steps, and then simplify the rules to model the evaluation of functional programs (reduction to weak head normal form). We also show that we can define the closed reduction strategy. This is a weak strategy which, in contrast with standard weak strategies, allows certain reductions to take place inside λ-abstractions thus offering more sharing. Our experimental results confirm that, for large combinator-based terms, our weak evaluation strategies out-perform standard evaluators. Moreover, we derive two abstract machines for strong reduction which inherit the efficiency of the weak evaluators.

Key words λ-calculus – explicit substitutions – director strings – strategies

1 Introduction

In the λ-calculus, the operation of substitution used in the β-reduction rule is defined outside the system: it is a meta-operation (see [5]). In contrast, *explicit substitution calculi* define substitution with reduction rules at the same level as β-reduction. Over the last years a whole range of explicit substitution calculi have been proposed, starting with the work of de Bruijn [14] and the $\lambda\sigma$-calculus [1]. Although there are many different applications of such calculi, one of the main advantages that we see in describing the process of propagation of substitution at the same level as β-reduction is that it allows us to control the substitution process, with an emphasis on implementation.

⋆ Projet Logical, Pôle Commun de Recherche en Informatique du plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

There are different notations for substitution, or more precisely, for variables in the λ-calculus. Explicit substitution calculi can be classified as:

- *named*, when variables are denoted by names such as $x, y, \ldots$; and
- *unnamed*, for instance when numbers (also called *indices*) are used.

During the process of propagation of substitution, it may be necessary to perform α-conversion (i.e. variable renaming) to avoid variable capture or clash. Unnamed explicit substitution calculi are thus often preferred for implementation purposes (although there are some exceptions, see for instance [33, 17]). De Bruijn notation [13] has, without doubt, become the standard name-free syntax for explicit substitution calculi. The purpose of this paper is to define an alternative notation for unnamed explicit substitution calculi, based on *director strings*.

Director strings were introduced by Kennaway and Sleep [22] for combinator reduction, which translated to the λ-calculus gives a system where no reduction can be performed under abstractions. They were generalised in [17, 18] in order to define *closed reduction* for the λ-calculus (a weak reduction strategy which in contrast with standard weak strategies allows certain reductions to take place inside λ-abstractions, thus offering more sharing). A further generalisation of director strings was used in [35] to define strong reduction strategies for the λ-calculus, and to derive abstract machines suitable for reduction to weak head normal form and to full normal form. In this present paper we define a calculus of explicit substitutions with director strings in which any strategy of reduction in the λ-calculus can be simulated, and we explore the properties of this general director string calculus as a rewrite system and also as a means to express efficient (weak and strong) reduction strategies for the λ-calculus. We consider this important for several reasons:

- Director strings offer an alternative to de Bruijn notation [13] for unnamed calculi. However, as for de Bruijn notation, the syntax is not as readable as the corresponding named version. We will show that the general notation can be simplified in some cases, for instance, closed reduction turns out to be a natural restriction leading to a very simple rewrite system for weak reduction.
- Director strings are a natural notation for explicit substitutions from an operational point of view: terms are annotated to indicate what they should do with a substitution. Substitutions are only propagated to places where they are needed, thus these calculi preserve strong normalisation (i.e. if a λ-term is strongly normalisable, so is its compilation). Other calculi preserving strong normalisation are presented in [27, 12] (see [29, 9] for counterexamples in $\lambda\sigma$).
- We provide a generalisation of the director strings introduced by Kennaway and Sleep [22] for combinator reduction. With our generalised director strings we can simulate arbitrary β-reductions.

We thus see the calculi presented in this paper both as an alternative syntax for explicit substitutions and as a basis for more efficient implementations of the λ-calculus. We present three calculi based on director strings. The first one, which we call λ_o, is a general system where any β-reduction in the λ-calculus can be simulated. From a theoretical point of view, λ_o has the desired properties (it is confluent, preserves strong normalisation, fully simulates the λ-calculus), however, from an

implementation perspective, its generality is a drawback rather than an advantage. In the other two calculi, which we call λ_l and λ_c, reduction is restricted so that only some evaluation strategies (which are efficient) can be simulated. In this sense, λ_l and λ_c are weak, but not as weak as standard weak calculi. It is well-known that standard weak explicit substitution calculi avoid α-conversion by allowing neither reduction under abstraction nor propagation of substitution through an abstraction (see for instance [11]). In contrast, our weak calculi allow certain reductions under, and propagation of substitutions through, abstractions. In this way more reductions can be shared. Moreover we may use the explicit information given by directors to avoid copying a substitution which contains a free variable, avoiding the duplication of potential redexes.

We have implemented a family of abstract machines for weak and strong reduction based on the director strings calculi, and the benchmarks (given in Section 8) show that the level of sharing obtained is close to optimal reduction [23, 19, 3] with considerably less overhead in many cases. Immediate applications of this work include, on one hand λ-calculus/functional language evaluators (where weak reduction is needed), and on the other hand, partial evaluation (also called program specialisation) and proof assistants based on powerful type theories (where strong reduction is needed).

1.1 Related Work

Our work is clearly related to the general work on explicit substitutions, starting from de Bruijn's seminal $\lambda C\xi\phi$ [14] (see [7] for a modern presentation) and the $\lambda\sigma$-calculus [1]. However, it is much more in line with the use of explicit substitutions for controlling the substitution process in implementations of the λ-calculus [2, 36, 21, 24, 31]. Our calculi are closer to $\lambda\upsilon$ [26, 27] than to $\lambda\sigma$ [1] in the sense that we do not have a syntactic construct for concatenation (also sometimes referred to as composition) of independent substitutions. Nadathur's work [30, 31] is also concerned with efficiency and has some common points with ours (although in a quite different framework): for instance, a variant of his calculus has conditions of closedness on certain terms.

Efficiency and sharing in the λ-calculus have been important topics in the last twenty years. There is, in the literature, a wide range of mechanisms used for sharing: environments [36], sharing graphs [4], calculi with explicit addressing [8, 24]. We use director strings and the mechanism of explicit substitution itself for the purpose of sharing, i.e. we do not use any external machinery. While optimal sharing [28] means optimal number of β-reductions, its implementation relies on sharing graphs [23, 19, 3, 4] in which a wealth of costly book-keeping rules are necessary (see [25] for instance). Hence we will in this paper give to efficiency a rather algorithmic meaning, or more pragmatically, we will count the total number of reduction steps necessary to reach a normal form, provided these steps are elementary in some sense.

Director strings were introduced in [22] for combinatory reduction. A first generalisation was used in [18] for closed reduction, which was the starting point of [35]. This present work is a substantially revised and extended version of [35].

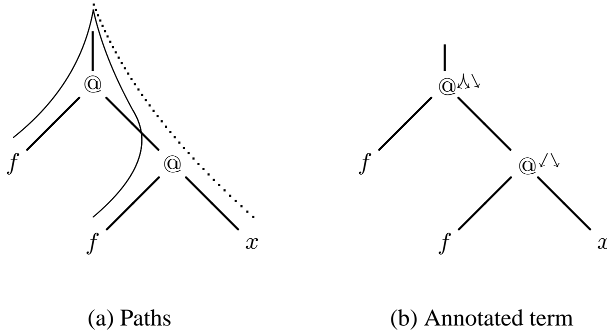
1.2 Overview

The rest of this paper is structured as follows. In the following section we provide the background material and define the syntax of director strings. In Section 3 we present a general calculus where we can simulate arbitrary β-reduction steps. Section 4 presents the simplified local open calculus, and the closed reduction system. A type system for these calculi is presented in Section 5. We then use these calculi to define several strategies: weak (Section 6) and strong ones (Section 7), which we experimentally compare (Section 8). Finally, we conclude the paper in Section 9.

2 Director Strings

2.1 Background

We briefly recall the basic ideas of director strings [22]. As a motivating example, consider a term with two free variables f, x and substitutions for both of them: $((f(fx))[F/f])[X/x]$. The best way to perform these substitutions is to propagate them *only* to the places in the syntactic tree where they are required. Figure 1(a) shows the *paths* which the substitutions should follow in the tree, where the solid line corresponds to the substitution for f, and the dotted line for x.



(a) Paths (b) Annotated term

Fig. 1 Substitution paths and director strings

A natural way to guide the substitutions to their correct destination is given in Figure 1(b) by director strings, which annotate each node in the graph with information about where the substitution must go (on both application nodes the first arrow-like symbol or director corresponds to f and the second to x). When the substitution for f passes the root of this term, a copy of F is sent to both subterms, and the $\bigwedge$ director is erased. The second substitution can then pass the root, where it is directed uniquely to the right branch by the director $\searrow$. Note that substitutions

are copied only when they need to be: if there is just one occurrence of a variable in a term, then no duplication is performed.

This simple idea works well when the substitution is closed (does not contain free variables). Otherwise, as each open substitution passes a given node we must add the additional directors for each free variable in the substitution.

We end this section by briefly recalling the analogy between director strings and combinator reduction, as presented in [22]. The reduction rules for the S, B, C, K combinators are the following:

$$
\begin{array}{ll}
\mathsf{S}\,x\,y\,z & \to x\,z\,(y\,z) \\
\mathsf{B}\,x\,y\,z & \to x\,(y\,z) \\
\mathsf{C}\,x\,y\,z & \to x\,z\,y \\
\mathsf{K}\,x\,y & \to x
\end{array}
$$

where $\mathsf{S}\,x\,y$ takes an argument and directs it to both x and y; $\mathsf{B}\,x\,y$ takes an argument and directs it just to y; $\mathsf{C}\,x\,y$ takes an argument and directs it just to x; and finally $\mathsf{K}\,x$ takes an argument and discards it. Thus we can annotate the application $x\,y$ with the combinators, which correspond exactly to the directors: S is $\curlywedge$, B is $\searrow$, C is $\swarrow$ and K is $-$ (see below).

2.2 Syntax

We assume the reader is familiar with the λ-calculus [5] and rewrite systems [15]. We recall that a reduction relation $\to$ is terminating (or strongly normalising) if all reduction sequences are finite. It is locally confluent if $t \to u$ and $t \to v$ implies that there exists w such that $u \to^* w$ and $v \to^* w$; we say that u and v are joinable in this case[1]. If u and v are joinable with one-step reductions, the relation is strongly confluent. It is confluent if $t \to^* u$ and $t \to^* v$ implies that there exists w such that $u \to^* w$ and $v \to^* w$.

We now introduce more formally the syntax of annotated terms. This syntax is common to the different rewrite systems that will be described later.

Definition 1 (λ-calculus with Director Strings) *We define four syntactic categories:*

Directors: We use five special symbols, called directors, ranged over by α,γ,δ:

1. *'$\searrow$' indicates that the substitution should be propagated only to the right branch of a binary construct (application or substitution, as given below).*
2. *'$\swarrow$' indicates that the substitution should be propagated only to the left branch of a binary construct.*
3. *'$\curlywedge$' indicates that the substitution should be propagated to both branches of a binary construct.*
4. *'$\downarrow$' indicates that the substitution should traverse a unary construct (abstraction and variables, see below).*

[1] $\to^*$ denotes the reflexive and transitive closure of $\to$.

5. '$-$' *indicates that the substitution should be discarded (when the variable concerned does not occur in a term).*

Strings: A director string is either empty, denoted by ϵ*, or built from the above symbols (so is of the form* $\alpha_1\alpha_2\ldots\alpha_n$ *where the* α_i*'s are directors). We use Greek letters such as* ρ,$\sigma\ldots$ *to range over strings. The length of a string* σ *is denoted by* $|\sigma|$*. If* α *is a director, then* α^n *denotes a string of* α*'s of length* n*. If* σ *is a director string of length* n *and* $1 \leq i \leq j \leq n$*,* σ_i *denotes the* i^{th} *director of* σ *and* $\sigma_{\setminus i} = \sigma_1\ldots\sigma_{i-1}\sigma_{i+1}\ldots\sigma_n$ *is* σ *where the* i^{th} *director has been removed.* $\sigma_{i..j} = \sigma_i\ldots\sigma_j$ *is our notation for substrings.* $|\sigma|_l$ *denotes the number of* $\swarrow$ *and* $\curlywedge$ *occurring in* σ*,* $|\sigma|_r$ *the number of* $\searrow$ *and* $\curlywedge$*, and* $|\sigma|_+$ *the number of directors that* are not $-$.

Preterms: Let σ *range over strings,* k *be a natural number and* $\mathbf{t}, \mathbf{u}$ *range over preterms, which are defined by the following grammar:*

$$\mathbf{t} ::= \Box^\sigma \mid (\lambda\mathbf{t})^\sigma \mid (\mathbf{t}\ \mathbf{u})^\sigma \mid (\mathbf{t}[k/\mathbf{u}])^\sigma$$

We denote preterms by $\mathbf{t}$*, or* t^σ *if we want to emphasise the director string* σ*.*

Terms: Well-formed terms are preterms that recursively satisfy the following conditions, where $\mathcal{U} = (\downarrow | -)^*$ *and* $\mathcal{B} = (\swarrow \mid \curlywedge \mid \searrow \mid -)^*$*:*

Name	*Term*	*Constraints*
Variable	$\Box^\sigma$	$\sigma \in \mathcal{U}, \lvert\sigma\rvert_+ = 1$
Abstraction	$(\lambda t^\rho)^\sigma$	$\sigma \in \mathcal{U},\ \lvert\rho\rvert = \lvert\sigma\rvert_+ + 1$
Application	$(t^\rho\ u^\nu)^\sigma$	$\sigma \in \mathcal{B},\ \lvert\rho\rvert = \lvert\sigma\rvert_l,\ \lvert\nu\rvert = \lvert\sigma\rvert_r$
Substitution	$(t^\rho[k/u^\nu])^\sigma$	$\sigma \in \mathcal{B},\ \lvert\rho\rvert = \lvert\sigma\rvert_l + 1,\ \lvert\nu\rvert = \lvert\sigma\rvert_r,\ 1 \leq k \leq \lvert\rho\rvert$

Remark 1 For terms we use the same convention as for preterms, writing $\mathbf{t}$ or t^σ depending on whether or not we need to mention specifically the director string of an annotated term, as in the definition above. To sum up the naming conventions in this paper, boldface lower-case letters designate preterms in general (hence also well-formed terms), while normal lower-case letters designate preterms (or terms) with their root director string removed, hence explicitly needing a director string to form a preterm or term. Upper-case letters are reserved for terms of the usual λ-calculus.

We have a variety of different term constructs:

- $\Box$ represents variables (a place holder),
- $(\lambda\mathbf{t})^\sigma$ is an abstraction,
- $(\mathbf{t}\ \mathbf{u})^\sigma$ is an application,
- finally $(\mathbf{t}[k/\mathbf{u}])^\sigma$ is our notation for explicit substitution, meaning that the variable corresponding to the kth director in $\mathbf{t}$'s string is to be replaced by $\mathbf{u}$. We will often write $(\mathbf{t}[\mathbf{u}])^\sigma$ instead of $(\mathbf{t}[1/\mathbf{u}])^\sigma$ when the substitution binds the first variable.

The name of the variable is of no interest since the director strings give the path that the substitution must follow through the term to ensure that it gets to the right place; all we need is a place holder.

In contrast with other explicit substitutions syntax, ours has explicit information about duplication ($\curlywedge$) and erasing ($-$). This is inspired by calculi for linear logic, and will allow us a finer control on substitutions: we may reduce a subterm more when encountering a $\curlywedge$, thus taking advantage of the mechanism of explicit substitution to share some reductions. There is an alternative presentation which combines the director '$-$' with abstraction, using the notation $(\lambda^{-}\mathtt{t})^{\sigma}$ to indicate that the bound variable does not occur in the term $\mathtt{t}$. The resulting syntax is simpler, and it allows us to erase terms as soon as possible. However, it does not allow to define β-reduction in full generality. We will discuss this choice again in Section 4.

As with most λ-calculi, we will adopt several syntactic conventions: we will drop parentheses whenever we can, and omit the empty string ϵ unless it is essential.

2.3 Compilation and Readback

We use λ-terms with director strings as an intermediate language. We thus need to provide a function to compile usual λ-terms into this syntax and another to read them back. Notice that, as usual, we consider terms of the λ-calculus modulo α-conversion (renaming of bound variables).

The following definition of a compilation from the usual λ-calculus into director strings syntax indicates precisely how the strings and terms are built. We use an auxiliary ordered list $[x_1, \ldots, x_n]$ in the compilation function to keep track of the variable names corresponding to each director in the strings. Each step of the compilation function goes down one node in the syntax tree of the term, and computes the corresponding string using auxiliary functions ξ and θ. We denote by $\mathsf{fv}(M)$ the set of free variables of the λ-term M. We use the standard notations for the empty list and the cons and append operations ($[\,]$, $x :: \ell$ and $\ell \cdot \ell'$ respectively) and abbreviate $x_1 :: \ldots :: x_n :: [\,]$ to $[x_1, \ldots, x_n]$ or $\vec{x}$. We denote ℓ_i the ith element of a list ℓ.

Definition 2 (Compilation) *Let M be a λ-term with $\mathsf{fv}(M) \subseteq \{x_1, \ldots, x_n\}$, its compilation $[\![M]\!]_{\vec{x}}$ is defined as follows:*

$$\begin{array}{lll} [\![x]\!]_{\vec{x}} & = \square^{\sigma} & \textit{where } ([x], \sigma) = \xi_x(\vec{x}) \\ [\![\lambda x.M]\!]_{\vec{x}} & = (\lambda\, [\![M]\!]_{\ell \cdot [x]})^{\sigma} & \textit{where } (\ell, \sigma) = \xi_{(\lambda x.M)}(\vec{x}) \\ [\![(M\ N)]\!]_{\vec{x}} & = ([\![M]\!]_{\ell}\, [\![N]\!]_{\ell'})^{\sigma} & \textit{where } (\ell, \ell', \sigma) = \theta_{M,N}(\vec{x}) \end{array}$$

$$\xi_M([\,]) = ([\,], \epsilon)$$

$$\xi_M(x :: \ell) = \left\{ \begin{array}{ll} (x :: \ell', \downarrow \sigma) & \textit{if } x \in \mathsf{fv}(M) \\ (\ell', -\sigma) & \textit{if } x \notin \mathsf{fv}(M) \end{array} \right\} \textit{where} (\ell', \sigma) = \xi_M(\ell)$$

$$\theta_{M,N}([\,]) = ([\,], [\,], \epsilon)$$

$$\theta_{M,N}(x :: \ell) = \left\{ \begin{array}{ll} (x :: \ell', \ell'', \swarrow \sigma) & \textit{if } x \in \mathsf{fv}(M) \setminus \mathsf{fv}(N) \\ (\ell', x :: \ell'', \searrow \sigma) & \textit{if } x \in \mathsf{fv}(N) \setminus \mathsf{fv}(M) \\ (x :: \ell', x :: \ell'', \curlywedge \sigma) & \textit{if } x \in \mathsf{fv}(M) \cap \mathsf{fv}(N) \\ (\ell', \ell'', -\sigma) & \textit{if } x \notin \mathsf{fv}(M) \cup \mathsf{fv}(N) \end{array} \right\} \begin{array}{l} \textit{where} \\ (\ell', \ell'', \sigma) = \theta_{M,N}(\ell) \end{array}$$

When there is no ambiguity, we use the notation $[\![M]\!]_{\vec{x}}$ *with the implicit assumption that* $\mathsf{fv}(M) \subseteq \vec{x}$*, and write* $[\![M]\!]$ *when the list is empty.*

Remark 2 (Order of directors) In an abstraction $(\lambda t^{\rho})^{\sigma}$, the *last* director in the string ρ corresponds to the bound variable. This is reminiscent of Crégut's *reversed de Bruijn's indexing* [10].

Example 1 We show the compilation of some λ-terms:

$$
\begin{array}{lll}
\mathsf{I} & = [\![\lambda x.x]\!] & = (\lambda \Box^{\downarrow})^{\epsilon} \\
\mathsf{K} & = [\![\lambda x.\lambda y.x]\!] & = (\lambda(\lambda \Box^{\downarrow -})^{\downarrow})^{\epsilon} \\
\mathsf{S} & = [\![\lambda x.\lambda y.\lambda z.(xz)(yz)]\!] & = (\lambda(\lambda(\lambda((\Box^{\downarrow}\Box^{\downarrow})^{\swarrow\searrow}(\Box^{\downarrow}\Box^{\downarrow})^{\swarrow\searrow})^{\swarrow\searrow\curlywedge})^{\downarrow\downarrow})^{\downarrow})^{\epsilon} \\
\mathsf{2} & = [\![\lambda f.\lambda x.f(fx)]\!] & = (\lambda(\lambda(\Box^{\downarrow}(\Box^{\downarrow}\Box^{\downarrow})^{\swarrow\searrow})^{\curlywedge\searrow})^{\downarrow})^{\epsilon}
\end{array}
$$

In order to show that the result of the compilation is a well-formed term, we need two auxiliary lemmas.

Lemma 1 *If* $(M\ N)$ *is an application with* $\mathsf{fv}(M\ N) \subseteq \{x_1, \ldots, x_n\}$*, then* $\theta_{M,N}([x_1, \ldots, x_n]) = (\ell_1, \ell_2, \sigma)$ *where* $\ell_1 = \mathsf{fv}(M)$*,* $\ell_2 = \mathsf{fv}(N)$*, and* $|\sigma| = n$*.*

Proof Straightforward induction on n. □

Lemma 2 (Length of Strings) *Let* M *be a* λ*-term and* $\mathsf{fv}(M) \subseteq \{x_1, \ldots, x_n\}$*, then:*

$$[\![M]\!]_{\vec{x}} = u^{\sigma} \text{ where } |\sigma| = n$$

In particular, if M *is closed then* $[\![M]\!]$ *has an empty director string (*ϵ*).*

Proof By induction on n.

For $n = 0$, M is either an abstraction, in which case the compiled term has director string $\downarrow^{0} = \epsilon$ as required, or an application and $[\![(M\ N)]\!] = ([\![M]\!]\ [\![N]\!])^{\epsilon}$ since $\theta_{M,N}([\,]) = ([\,], [\,], \epsilon)$ by definition.

For $n > 0$ we distinguish cases according to M. The cases of variable and abstraction are trivial. The interesting case is application. In this case, the property is a direct consequence of Lemma 1. □

Proposition 1 (Consistency of Compilation) *If* M *is a* λ*-term with* $\mathsf{fv}(M) \subseteq \vec{x}$ *then* $[\![M]\!]_{\vec{x}}$ *is a well-formed term.*

Proof By induction on the structure of λ-terms. If M is a variable then the result holds trivially. If M is an abstraction, then the result holds by induction. If M is an application, it is a consequence of Lemma 1, the induction hypothesis, and the construction of σ in the definition of θ. □

Since we prefer to think of this calculus as some form of intermediate language, we also provide a notion of readback, which simply puts names back in.

Definition 3 (Readback) *Let* $\mathbf{t} = t^\sigma$ *be a term where* $|\sigma| = n$*, and let* $x_1, \ldots, x_n$ *be* n *fresh variables. We define the readback of* $\mathbf{t}$ *as* $\llparenthesis \mathbf{t} \rrparenthesis_{[x_1,\ldots,x_n]}$*, where the readback function (which uses an auxiliary list* $\vec{M}$ *of* λ*-terms) is defined as follows:*

$$
\begin{array}{lll}
\llparenthesis \square^\sigma \rrparenthesis_{\vec{M}} & = M & \textit{where } [M] = \kappa_\sigma(\vec{M}) \\
\llparenthesis (\lambda \mathbf{t})^\sigma \rrparenthesis_{\vec{M}} & = \lambda x. \llparenthesis \mathbf{t} \rrparenthesis_{\kappa_\sigma(\vec{M}\cdot[x])} & \textit{where } x \textit{ is fresh} \\
\llparenthesis (\mathbf{t}\ \mathbf{u})^\sigma \rrparenthesis_{\vec{M}} & = \llparenthesis \mathbf{t} \rrparenthesis_{\ell}\ \llparenthesis \mathbf{u} \rrparenthesis_{\ell'} & \textit{where } (\ell, \ell') = \gamma_\sigma(\vec{M}) \\
\llparenthesis (\mathbf{t}[k/\mathbf{u}])^\sigma \rrparenthesis_{\vec{M}} & = \llparenthesis \mathbf{t} \rrparenthesis_{[\ell_1,\ldots,\ell_{k-1},\llparenthesis \mathbf{u} \rrparenthesis_{\ell'},\ell_k,\ldots,\ell_m]} & \textit{where } (\ell, \ell') = \gamma_\sigma(\vec{M})
\end{array}
$$

$$
\begin{array}{ll}
\kappa_\epsilon([\,]) & = [\,] \\
\kappa_{\downarrow\sigma}(M :: \ell) & = M :: \kappa_\sigma(\ell) \\
\kappa_{-\sigma}(M :: \ell) & = \kappa_\sigma(\ell)
\end{array}
$$

$$
\left.
\begin{array}{ll}
\gamma_\epsilon([\,]) & = ([\,],[\,]) \\
\gamma_{\swarrow\sigma}(M :: \ell) & = (M :: \ell', \ell'') \\
\gamma_{\searrow\sigma}(M :: \ell) & = (\ell', M :: \ell'') \\
\gamma_{\curlywedgedownarrow\sigma}(M :: \ell) & = (M :: \ell', M :: \ell'') \\
\gamma_{-\sigma}(M :: \ell) & = (\ell', \ell'')
\end{array}
\right\} \textit{where } (\ell', \ell'') = \gamma_\sigma(\ell)
$$

Notice that in the case of a variable $\square^\sigma$ we must have $|\sigma|_+ = 1$ since the term is well-formed, hence $\kappa_\sigma(\vec{M})$ is a singleton. Also note that the auxiliary list $\vec{M}$ may contain arbitrary λ-terms, and not necessarily only variables as in the compilation. This makes it possible to complete substitutions in a thorough and elegant way: we only put them in the list, and the terms will be guided to the right places, thanks to the directors. More precisely:

$$\llparenthesis \mathbf{t} \rrparenthesis_{[x_1,\ldots,x_n]}\{x_i \mapsto M_i\} = \llparenthesis \mathbf{t} \rrparenthesis_{[x_1,\ldots,x_{i-1},M_i,x_{i+1},\ldots,x_n]}$$

where $M\{x \mapsto N\}$ is our notation for implicit substitution (the meta-operation of substitution in the λ-calculus).

Example 2 We give two small examples of the readback procedure:

$$
\begin{array}{ll}
\llparenthesis (\lambda \square^{\downarrow})^\epsilon \rrparenthesis & = \lambda x. \llparenthesis \square^{\downarrow} \rrparenthesis_{[x]} = \lambda x.x \\
\llparenthesis (\lambda(\lambda(\square^{\downarrow}(\square^{\downarrow}\,\square^{\downarrow})^{\swarrow\searrow})^{\curlywedgedownarrow\searrow})^{\downarrow})^\epsilon \rrparenthesis & = \lambda x.\lambda y. \llparenthesis (\square^{\downarrow}(\square^{\downarrow}\,\square^{\downarrow})^{\swarrow\searrow})^{\curlywedgedownarrow\searrow} \rrparenthesis_{[x,y]} \\
& = \lambda x.\lambda y.x\,(x\,y)
\end{array}
$$

To prove that the readback function is well defined we use the following lemma:

Lemma 3 *If* $|\sigma| = n$ *and* $M_1, \ldots, M_n$ *are* λ*-terms then*

- $\mathrm{length}(\kappa_\sigma([M_1,\ldots,M_n])) = |\sigma|_+$ *if* $\sigma \in \mathcal{U}$*;*
- $\gamma_\sigma([M_1,\ldots,M_n]) = (\ell, \ell')$ *where* $\mathrm{length}(\ell) = |\sigma|_l$ *and* $\mathrm{length}(\ell') = |\sigma|_r$.

Proof Straightforward induction on n. □

Lemma 4 *During the readback of a well-formed term, the readback procedure is only called under the form* $\llparenthesis t^\rho \rrparenthesis_{[M_1,\ldots,M_n]}$ *with* $|\rho| = n$.

Proof By induction and Lemma 3. Each step of the procedure adjusts the length of the auxiliary list according to the constraints on well-formed terms (see Definitions 1 and 3). □

Proposition 2 (Consistency of Readback) *If t^σ is a well-formed term with $|\sigma| = n$ and $x_1, \dots, x_n$ are n fresh variables then $(\!| t^\sigma |\!)_{[x_1,\dots,x_n]}$ is a λ-term.*

Proof We prove the more general property:

If t^σ is a well-formed term, $|\sigma| = n$ and $M_1, \dots, M_n$ are λ-terms, then $(\!| t^\sigma |\!)_{[M_1,\dots,M_n]}$ is well-formed.

This is proved by induction on t, using Lemma 4. The case of a variable $\Box^\sigma$ is trivial since $|\sigma|_+ = 1$ by well-formedness. For an abstraction, application or substitution the result follows directly by induction and Lemma 4. □

The compilation and the readback function are inverses modulo α-conversion. To prove it we use the following auxiliary lemma:

Lemma 5 – *$\xi_M([x_1, \dots, x_n]) = (\ell, \sigma)$ implies $\kappa_\sigma([x_1, \dots, x_n]) = \ell$;*
– *$\theta_{M,N}([x_1, \dots, x_n]) = (\ell_1, \ell_2, \sigma)$ implies $\gamma_\sigma([x_1, \dots, x_n]) = (\ell_1, \ell_2)$.*

Proof Straightforward induction on n. □

Proposition 3 (Inverses) *If M is a λ-term with $\mathsf{fv}(M) \subseteq \vec{x}$, then $(\!| [\![M]\!]_{\vec{x}} |\!)_{\vec{x}} =_\alpha M$. In particular, if M is a closed λ-term, $(\!| [\![M]\!] |\!) =_\alpha M$.*

Proof By induction on M.

- Variable:
  $(\!| [\![x]\!]_{\vec{x}} |\!)_{\vec{x}} = x$ as required.
- Abstraction:
  $(\!| [\![(\lambda x.M)]\!]_{\vec{x}} |\!)_{\vec{x}} = (\!| (\lambda [\![M]\!]_{\ell\cdot[x]})^\sigma |\!)_{\vec{x}} = \lambda y.(\!| [\![M]\!]_{\ell\cdot[x]} |\!)_{\ell\cdot[y]}$, using Lemma 5, where $(\ell, \sigma) = \xi_{\lambda x.M}(\vec{x})$. By induction, this is α-equivalent to $\lambda x.M$.
- Application:
  $(\!| [\![M\ N]\!]_{\vec{x}} |\!)_{\vec{x}} = (\!| ([\![M]\!]_{\ell'}\ [\![N]\!]_{\ell''})^\sigma |\!)_{\vec{x}}$, where $(\ell', \ell'', \sigma) = \theta_{M,N}(\vec{x})$.
  By Lemma 5 and the definition of readback:
  $$(\!| ([\![M]\!]_{\ell'}\ [\![N]\!]_{\ell''})^\sigma |\!)_{\vec{x}} = ((\!| [\![M]\!]_{\ell'} |\!)_{\ell'}\ (\!| [\![N]\!]_{\ell''} |\!)_{\ell''}).$$
  The result then follows directly by induction. □

3 The Open Calculus

We will now give the reduction rules that will allow us to fully simulate the λ-calculus. This calculus is called *open* (λ_o) in contrast with the calculus for closed reduction (λ_c) defined in Section 4.3, which is simpler but does not fully simulate β-reduction.

3.1 The Beta *Rule*

We need a *Beta* rule to eliminate β-redexes and introduce an explicit substitution instead. In a compiled term $(\lambda t^{\nu})^{\rho}$ the variable bound by the abstraction is determined by the *last* director in ν (see Remark 2), and $|\nu| = |\rho|_{+} + 1$ according to the constraints in Definition 1. Since ρ may contain erasing directors $(-)$ which we have to preserve, we move them up to the director string of the substitution:

Beta	$((\lambda \mathbf{t})^{\rho}\ \mathbf{u})^{\sigma} \rightsquigarrow_o (\mathbf{t}[\lvert\rho\rvert_{+}{+}1 \,/\, \mathbf{u}])^{\tau}$ where $\tau = \psi_b(\sigma, \rho)$

with:

$$\begin{aligned}
\psi_b(\ \epsilon,\ \ \epsilon) &= \ \epsilon \\
\psi_b(\searrow \sigma,\ \ \rho) &= \searrow \psi_b(\sigma, \rho) \\
\psi_b(\swarrow \sigma,\ \downarrow \rho) &= \swarrow \psi_b(\sigma, \rho) \\
\psi_b(\curlywedge \sigma,\ \downarrow \rho) &= \curlywedge \psi_b(\sigma, \rho) \\
\psi_b(- \sigma,\ \ \rho) &= - \psi_b(\sigma, \rho) \\
\psi_b(\swarrow \sigma,\ - \rho) &= - \psi_b(\sigma, \rho) \\
\psi_b(\curlywedge \sigma,\ - \rho) &= \searrow \psi_b(\sigma, \rho)
\end{aligned}$$

Remark 3 If the function is closed (i.e. has an empty director string), then the substitution binds the first (and only) variable of $\mathbf{t}$, and we simply have:

$$((\lambda \mathbf{t})^{\epsilon}\ \mathbf{u})^{\sigma} \rightsquigarrow_o (\mathbf{t}[\mathbf{u}])^{\sigma}.$$

Based on this idea we will define a simplified system for the evaluation of closed terms in Section 4.

3.2 Propagation Rules

We need rules to propagate the substitutions created by the *Beta* rule defined above in Section 3.1. The directors indicate the path that the substitution should follow: we will need a rule per term construct and possible director.

To understand how the rules for the propagation of substitutions are defined, consider a simple case: an application $(\mathbf{t}\ \mathbf{u})^{\swarrow\rho}$ with a substitution concerning the first variable, i.e. we have $((\mathbf{t}\ \mathbf{u})^{\swarrow\rho}[\mathbf{v}])^{\sigma}$. The substitution should be propagated to the left branch of the application node as the director $\swarrow$ indicates. Therefore we need a rule of the form:

$$(\mathit{App}_1) \qquad ((\mathbf{t}\ \mathbf{u})^{\swarrow\rho}[\mathbf{v}])^{\sigma} \rightsquigarrow ((\mathbf{t}[\mathbf{v}])^{\rho'}\ \mathbf{u})^{\sigma'}$$

Let's try to find ρ' and σ'. Suppose that a (closed) substitution is applied to the left and right hand-sides of the above equation. If, for example, $\sigma = \rho = \swarrow$, then the substitution is for $\mathbf{t}$ on the left, so we must have $\sigma' = \rho' = \swarrow$ to ensure that the substitution is also guided towards $\mathbf{t}$ on the right. If $\sigma = \swarrow$ and $\rho = \searrow$, then it is to be directed towards $\mathbf{u}$, and we must have $\sigma' = \searrow$ and ρ' is not concerned (say $\rho' = \epsilon$ here). Finally, if $\sigma = \searrow$, the substitution is for $\mathbf{v}$, and we write $\sigma' = \swarrow$ and $\rho' = \searrow$.

Name	*Reduction*	*Cond.*
Var	$(\Box^{\rho}[i/v^{\nu}])^{\sigma} \quad \rightsquigarrow_o \quad v^{\tau} \quad$ where $\tau = \pi'(\sigma, \nu)$	$\rho_i = \downarrow$
App_1	$((\mathbf{t}\ \mathbf{u})^{\rho}[i/\mathbf{v}])^{\sigma} \quad \rightsquigarrow_o \quad ((\mathbf{t}[j/\mathbf{v}])^{\upsilon}\ \mathbf{u})^{\tau}$	$\rho_i = \swarrow$
	where $\upsilon = \phi_l(\sigma, \rho_{\backslash i}), \tau = \psi_1(\sigma, \rho_{\backslash i}), j = \lvert\rho_{1..i}\rvert_l$	
App_2	$((\mathbf{t}\ \mathbf{u})^{\rho}[i/\mathbf{v}])^{\sigma} \quad \rightsquigarrow_o \quad (\mathbf{t}\ (\mathbf{u}[k/\mathbf{v}])^{\omega})^{\tau}$	$\rho_i = \searrow$
	where $\omega = \phi_r(\sigma, \rho_{\backslash i}), \tau = \psi_2(\sigma, \rho_{\backslash i}), k = \lvert\rho_{1..i}\rvert_r$	
App_3	$((\mathbf{t}\ \mathbf{u})^{\rho}[i/\mathbf{v}])^{\sigma} \quad \rightsquigarrow_o \quad ((\mathbf{t}[j/\mathbf{v}])^{\upsilon}\ (\mathbf{u}[k/\mathbf{v}])^{\omega})^{\tau}$	$\rho_i = \curlywedge$
where $\upsilon = \phi_l(\sigma, \rho_{\backslash i}), \omega = \phi_r(\sigma, \rho_{\backslash i}), \tau = \psi_3(\sigma, \rho_{\backslash i}), j = \lvert\rho_{1..i}\rvert_l, k = \lvert\rho_{1..i}\rvert_r$		
Lam	$((\lambda\mathbf{t})^{\rho}[i/\mathbf{v}])^{\sigma} \quad \rightsquigarrow_o \quad \left(\lambda(\mathbf{t}[i/\mathbf{v}])^{\upsilon\cdot\swarrow}\right)^{\tau}$	$\rho_i = \downarrow$
	where $\upsilon = \phi_d(\sigma, \rho_{\backslash i}), \tau = \psi_d(\sigma, \rho_{\backslash i})$	
Comp	$((\mathbf{t}[j/\mathbf{u}])^{\rho}[i/\mathbf{v}])^{\sigma} \quad \rightsquigarrow_o \quad (\mathbf{t}[j/(\mathbf{u}[k/\mathbf{v}])^{\omega}])^{\tau}$	$\rho_i = \searrow$
	where $\omega = \phi_r(\sigma, \rho_{\backslash i}), \tau = \psi_2(\sigma, \rho_{\backslash i}), k = \lvert\rho_{1..i}\rvert_r$	
Erase	$(t^{\rho}[i/\mathbf{v}])^{\sigma} \quad \rightsquigarrow_o \quad t^{\tau} \quad$ where $\tau = \pi(\sigma, \rho_{\backslash i})$	$\rho_i = -$

Fig. 2 Propagation Rules: $\mathcal{P}$

σ_1	ρ_1	ϕ_l	ϕ_r	ψ_1	ψ_2	ψ_3	π
$\searrow$	ϵ	$\searrow$	$\searrow$	$\swarrow$	$\searrow$	$\curlywedge$	$-$
$\swarrow$	$\searrow$	ϵ	$\swarrow$	$\searrow$	$\searrow$	$\searrow$	$\searrow$
$\swarrow$	$\swarrow$	$\swarrow$	ϵ	$\swarrow$	$\swarrow$	$\swarrow$	$\swarrow$
$\swarrow$	$\curlywedge$	$\swarrow$	$\swarrow$	$\curlywedge$	$\curlywedge$	$\curlywedge$	$\curlywedge$
$\curlywedge$	$\searrow$	$\searrow$	$\curlywedge$	$\curlywedge$	$\searrow$	$\curlywedge$	$\searrow$
$\curlywedge$	$\swarrow$	$\curlywedge$	$\searrow$	$\swarrow$	$\curlywedge$	$\curlywedge$	$\swarrow$
$\curlywedge$	$\curlywedge$	$\curlywedge$	$\curlywedge$	$\curlywedge$	$\curlywedge$	$\curlywedge$	$\curlywedge$
$-$	ϵ	ϵ	ϵ	$-$	$-$	$-$	$-$
$\swarrow$	$-$	ϵ	ϵ	$-$	$-$	$-$	$-$
$\curlywedge$	$-$	$\searrow$	$\searrow$	$\swarrow$	$\searrow$	$\curlywedge$	$-$

σ_1	ρ_1	ψ_d	ϕ_d	ψ_b
$\searrow$	ϵ	$\downarrow$	$\searrow$	$\searrow$
$\swarrow$	$\downarrow$	$\downarrow$	$\swarrow$	$\swarrow$
$\curlywedge$	$\downarrow$	$\downarrow$	$\curlywedge$	$\curlywedge$
$-$	ϵ	$-$	ϵ	$-$
$\swarrow$	$-$	$-$	ϵ	$-$
$\curlywedge$	$-$	$\downarrow$	$\searrow$	$\searrow$

Fig. 3 Functions used in the Propagation Rules

We obtain most of the propagation rules in the same way. Notice that not every combination of directors is to be considered, as some of them do not correspond to well-formed terms. Figure 2 shows the set $\mathcal{P}$ of propagation rules.

The various functions ϕ and ψ used in the propagation rules just compute the *ad hoc* director strings. They are generated recursively in the same way as above from the tables in Figure 3.

For example:

$$\begin{aligned} \phi_l(\ \epsilon,\ \ \epsilon) &= \ \epsilon \\ \phi_l(\searrow \sigma,\ \ \rho) &= \searrow \phi_l(\sigma, \rho) \\ \phi_l(\swarrow \sigma, \searrow \rho) &= \ \phi_l(\sigma, \rho) \\ \phi_l(\swarrow \sigma, \swarrow \rho) &= \swarrow \phi_l(\sigma, \rho), \text{ etc.} \end{aligned}$$

The function π used only in the *Erase* rule adds a new director $-$ for every free variable of an erased term, and can be paraphrased from Figure 3 in the following way (α is any director):

$$\begin{aligned}
\pi(\ \epsilon, \ \ \epsilon) &= \ \epsilon \\
\pi(\searrow \sigma, \ \ \rho) &= -\,\pi(\sigma,\rho) \\
\pi(\swarrow \sigma, \alpha\,\rho) &= \alpha\,\pi(\sigma,\rho) \\
\pi(\curlywedge \sigma, \alpha\,\rho) &= \alpha\,\pi(\sigma,\rho) \\
\pi(-\,\sigma, \ \ \rho) &= -\,\pi(\sigma,\rho)
\end{aligned}$$

The function π' used for the *Var* rule performs a similar job (π' is not in Figure 3 because it does not follow exactly the same pattern):

$$\begin{aligned}
\pi'(\ \epsilon, \ \ \epsilon) &= \ \epsilon \\
\pi'(\searrow \sigma, \alpha\,\nu) &= \alpha\,\pi'(\sigma,\nu) \\
\pi'(\swarrow \sigma, \ \ \nu) &= -\,\pi'(\sigma,\nu) \\
\pi'(\curlywedge \sigma, \alpha\,\nu) &= \alpha\,\pi'(\sigma,\nu) \\
\pi'(-\,\sigma, \ \ \nu) &= -\,\pi'(\sigma,\nu)
\end{aligned}$$

These rules deserve some explanations:

- The *Var* rule is the simplest. When the substitution reaches such a place holder with a corresponding director $\downarrow$, we know that it is indeed the right variable (because the substitution has been guided there earlier and is not erased). We know that $\rho_{\setminus i} = -^n$ if the term is well-formed so that both '$-$' and $\swarrow$ in σ should result in a '$-$' in τ to ensure that the erased variables are preserved. Moreover we do not need to inspect ρ in the computation of τ.
- The rules for application are the main rules here. Depending on ρ_i, the substitution is guided to the left or right, or copied in *App*$_3$ only when there is more than one occurrence of the given variable. The new director strings are computed by *ad hoc* functions from σ and ρ (omitting the ith director of the last one).
- Surprisingly, the rule that allows an open substitution to pass through an abstraction (*Lam*) is simple. This is quite remarkable, as this is especially difficult in usual calculi. For example, it requires α-conversion in a calculus with names.
- The *Comp* rule is the counterpart of *App*$_2$ in terms of substitution instead of application. We could have written composition rules for substitutions similar to *App*$_1$ and *App*$_3$, but the substitutions would then be allowed to *overtake* (i.e. their order would not be preserved), which means that the system would trivially fail to preserve strong normalisation.
- The *Erase* rule applies to a substitution in any term construct, provided the director corresponding to the substitution is '$-$'. Then the substitution is simply discarded and new '$-$' directors are added to take into account possible free variables of the discarded term.

We have a small number of propagation rules in comparison with standard explicit substitution calculi. However, our rules require non-trivial syntactical computations on director strings when we consider arbitrary substitutions. The system can be drastically simplified if we impose some restrictions on the substitutions, as we will see in the next section. Note that the condition on ρ_i in the rules has been externalised only to improve readability and is of course a simple pattern-matching.

Example 3 We show a reduction sequence using this calculus. Consider the λ-term $\lambda x.(\lambda y.y)x$ which contains a single redex:

$$[\![\lambda x.(\lambda y.y)x]\!] = (\lambda((\lambda\Box^{\downarrow})^{\epsilon}\Box^{\downarrow})^{\searrow})^{\epsilon} \rightsquigarrow_o (\lambda(\Box^{\downarrow}[\Box^{\downarrow}])^{\searrow})^{\epsilon} \rightsquigarrow_o (\lambda\Box^{\downarrow})^{\epsilon} = [\![\lambda x.x]\!]$$

Note that an encoding into combinators, using director strings as presented in [22], would not allow this redex to be contracted, and thus if used as an argument could potentially be duplicated. In this sense, our calculus offers a generalisation of the director strings of [22].

3.3 Properties

Lemma 6 (Preservation of Well-Formedness) *If t^{σ} is a well-formed term and $t^{\sigma} \rightsquigarrow_o u^{\tau}$, then u^{τ} is a well-formed term, moreover $|\sigma| = |\tau|$.*

Proof We have to prove that each rule produces a well-formed term with a director string of the same size. The proof is laborious but not difficult, and we omit the details. $\Box$

The process of propagating a substitution to the corresponding leaves in the tree associated to the term, using the rules in $\mathcal{P}$, is terminating: each rule either erases the substitution or moves it down towards the leaves. Formally, we prove the termination of $\mathcal{P}$ by using an interpretation function. This function computes the lengths of the paths to be traversed by a substitution to reach the corresponding leaves. We call it the *distance* of a substitution.

Definition 4 (Distance) *The distance associated to the substitution $[i/\mathbf{v}]$ in the term $(\mathbf{t}[i/\mathbf{v}])^{\rho}$ is computed by the function $|\mathbf{t}[i/\mathbf{v}]|$ defined by induction on $\mathbf{t}$ as follows:*

$$\begin{array}{lll}
|\Box^{\sigma}[i/\mathbf{v}]| & = 1 & \\
|(\lambda\mathbf{t})^{\sigma}[i/\mathbf{v}]| & = 1 & \textit{(if } \sigma_i = -\textit{)} \\
|(\lambda\mathbf{t})^{\sigma}[i/\mathbf{v}]| & = 1 + |\mathbf{t}[i/\mathbf{v}]| & \textit{(if } \sigma_i = \downarrow\textit{)} \\
|(\mathbf{t}\,\mathbf{u})^{\sigma}[i/\mathbf{v}]| & = |(\mathbf{t}[j/\mathbf{u}])^{\sigma}[i/\mathbf{v}]| = 1 + |\mathbf{t}[k/\mathbf{v}]| & \textit{(if } \sigma_i = \swarrow, \\
& & k \textit{ computed as in Fig. 2)} \\
|(\mathbf{t}\,\mathbf{u})^{\sigma}[i/\mathbf{v}]| & = |(\mathbf{t}[j/\mathbf{u}])^{\sigma}[i/\mathbf{v}]| = 1 + |\mathbf{u}[k/\mathbf{v}]| & \textit{(if } \sigma_i = \searrow, \\
& & k \textit{ computed as in Fig. 2)} \\
|(\mathbf{t}\,\mathbf{u})^{\sigma}[i/\mathbf{v}]| & = |(\mathbf{t}[j/\mathbf{u}])^{\sigma}[i/\mathbf{v}]| = 1 + |\mathbf{t}[k/\mathbf{v}]| + |\mathbf{u}[k'/\mathbf{v}]| & \textit{(if } \sigma_i = \swarrow\!\!\searrow, \\
& & k, k' \textit{ computed as in Fig. 2)} \\
|(\mathbf{t}\,\mathbf{u})^{\sigma}[i/\mathbf{v}]| & = |(\mathbf{t}[j/\mathbf{u}])^{\sigma}[i/\mathbf{v}]| = 1 & \textit{(if } \sigma_i = -\textit{)}
\end{array}$$

Proposition 4 (Termination of Propagation) *The set $\mathcal{P}$ of propagation rules is terminating.*

Proof We define an interpretation that associates to each term $\mathbf{t}$ a multiset with an element $|\mathbf{u}[i/\mathbf{v}]|$ for each subterm $(\mathbf{u}[i/\mathbf{v}])^{\rho}$ occurring in $\mathbf{t}$. Each application of a propagation rule decreases the interpretation of the term: rules *Var* and *Erase* erase one element of the multiset, and in the other rules one element is replaced by one or two strictly smaller elements. $\Box$

It is easy to see that using the propagation rules we eventually reach a pure term (i.e. a term without substitutions):

Proposition 5 (Completion of Substitutions) *Every term has a $\mathcal{P}$-normal form which is a pure term.*

Proof We prove by contradiction that any term of the form $(\mathbf{u}[i/\mathbf{v}])^\rho$ can be reduced by a rule in $\mathcal{P}$. Assume there is a counterexample, and take one of minimal size: $(\mathbf{t}[i/\mathbf{w}])^\sigma$. Consider all the cases for $\mathbf{t}$. If $\mathbf{t}$ is a variable, application or abstraction obviously we can apply a rule from $\mathcal{P}$. If it is a substitution $(\mathbf{t}'[j/\mathbf{u}'])^\upsilon$, then it is reducible (by the minimality of the counterexample). $\square$

Lemma 7 (Local Confluence of Propagation) *The set $\mathcal{P}$ of propagation rules is locally confluent.*

Proof $\mathcal{P}$ has seven critical pairs, all with *Comp*. In the version of the calculus without the director for erasing (Section 4), they all easily converge. But here, the pairs reduce to terms where the erasing may occur at two different levels, and we need to go one step further with the corresponding rule to flatten these two strings to one, and obtain syntactically equal terms. Of course, this makes the proof rather tiresome, due to the definition by cases of the functions involved in the rules, so we only give the details for the *Var*/*Comp* pair, as the other critical pairs can be dealt with in a similar way.

$$
\begin{array}{ccc}
((\Box^\chi[j/u^\nu])^\rho[i/\mathbf{v}])^\sigma & \xrightarrow{\ \textit{Comp}\ } & (\Box^\chi[j/(u^\nu[k/\mathbf{v}])^\omega])^\tau \\
\downarrow{\scriptstyle \textit{Var}} & & \downarrow{\scriptstyle \textit{Var}} \\
(u^\gamma[i/\mathbf{v}])^\sigma & & (u^\nu[k/\mathbf{v}])^\mu \\
& \searrow \quad ? \quad \swarrow &
\end{array}
$$

with the conditions $\chi_j = \downarrow, \rho_i = \searrow$ and the intermediate results:
$k = |\rho_{1..i}|_r, \gamma = \pi'(\rho, \nu), \tau = \psi_2(\sigma, \rho_{\setminus i}), \omega = \phi_r(\sigma, \rho_{\setminus i}), \mu = \pi'(\tau, \omega)$.

The two terms may be syntactically different, the only difference being that directors '$-$' may have been added in γ in the left term and in μ in the right term, so at different levels. Fortunately, these terms are not in $\mathcal{P}$-normal form (thanks to Proposition 5), and we know that another $\mathcal{P}$-rule may be applied at the root of both terms. (There is one case where we can't, but then we have $u^\nu \rightsquigarrow_o^* u'^{\nu'}$ where we can reduce at the root and $u^\gamma = u^{\pi'(\rho,\nu)} \rightsquigarrow_o^* u'^{\pi'(\rho,\nu')}$, and the situation is similar). It is indeed the same rule for both terms, because the unlabelled term to the left of the substitution is the same and $\gamma_i = \nu_k$, which comes from the following observations:

(i) $\pi'(\rho, \nu)$ is well-defined if and only if $|\nu| = |\rho|_r$;
(ii) if $|\nu| = |\rho|_r$, $1 \leq i \leq |\rho|$ and $k = |\rho_{1..i}|_r$, $\pi'(\rho_{1..i}, \nu_{1..k})$ is a prefix of $\pi'(\rho, \nu)$;
(iii) if $|\nu| = |\rho|_r$, $|\pi'(\rho, \nu)| = |\rho|$;

(iv) if $|\nu| = |\rho|_r$, $1 \leq i \leq |\rho|$ and $k = |\rho_{1..i}|_r$, $(\pi'(\rho, \nu))_{1..i} = \pi'(\rho_{1..i}, \nu_{1..k})$.

(i), (ii), (iii) are clear from the definition of π' and (iv) is a direct consequence of (ii) and (iii). As a corollary, we have indeed from (iv), back to our case, that $\gamma_i = \nu_k$ and $(\pi'(\rho, \nu))_{\setminus i} = \pi'(\rho_{\setminus i}, \nu_{\setminus k})$. $\quad (*)$

Now, to prove that the terms are equal after this (unknown) reduction, we just need to show that $f(\sigma, \gamma_{\setminus i}) = f(\mu, \nu_{\setminus k})$ for $f \in \{\phi_l, \phi_r, \phi_d, \psi_1, \psi_2, \psi_3, \psi_d, \pi, \pi'\}$ and $|\gamma_{1..i}|_r = |\nu_{1..k}|_r$, $|\gamma_{1..i}|_l = |\nu_{1..k}|_l$.

This job is of course awfully tiresome and we will omit most of it. We will however deal completely with one case, say ψ_1, which means that our unknown rule was in fact *App*$_1$, and we want to compare the outermost director strings of both terms. We thus have to compare $\psi_1(\sigma, \gamma_{\setminus i})$ and $\psi_1(\mu, \nu_{\setminus k})$ for every valid case, i.e. for every case that corresponds to initially well-formed terms:

σ	$\rho_{\setminus i}$	$\nu_{\setminus k}$	$\gamma_{\setminus i}$	$\psi_1(\sigma, \gamma_{\setminus i})$	τ	ω	μ	$\psi_1(\mu, \nu_{\setminus k})$
$\searrow$	ϵ	ϵ	ϵ	$\swarrow$	$\searrow$	$\searrow$	$\searrow$	$\swarrow$
$\swarrow$	$\searrow$	$\searrow$	$\searrow$	$\searrow$	$\searrow$	$\swarrow$	$\swarrow$	$\searrow$
$\swarrow$	$\searrow$	$\swarrow$	$\swarrow$	$\swarrow$	$\searrow$	$\swarrow$	$\swarrow$	$\swarrow$
$\swarrow$	$\searrow$	$-$	$-$	$-$	$\searrow$	$\swarrow$	$\swarrow$	$-$
$\swarrow$	$\swarrow$	ϵ	$-$	$-$	$\swarrow$	ϵ	$-$	$-$
$\swarrow$	$-$	ϵ	$-$	$-$	$\swarrow$	ϵ	$-$	$-$
$-$	ϵ	ϵ	ϵ	$-$	$-$	ϵ	$-$	$-$

Note that we compute $\gamma_{\setminus i}$ from $\rho_{\setminus i}$ and $\nu_{\setminus k}$ thanks to $(*)$.

The table above shows that if σ, ρ, ν are as defined in a well-formed term, $\psi_1(\sigma, \gamma_{\setminus i}) = \psi_1(\mu, \nu_{\setminus k})$ (cases with $\curlywedge$ have been omitted but are similar), hence, after the application of an *App*$_1$ rule to both terms of the critical pair, their outermost director strings are the same. A similar proof with ϕ_l would allow to conclude that these terms are indeed the same, which concludes this case. All the other cases are similar. $\square$

Local confluence and termination imply confluence, using Newman's Lemma [32].

Corollary 1 (Confluence of Propagation) *The set $\mathcal{P}$ of propagation rules is confluent on λ_o-terms.*

As a consequence of confluence (Corollary 1) and termination (Proposition 4), the set $\mathcal{P}$ of propagation rules defines a function from a term $\mathbf{t}$ to its unique normal form, denoted $\mathcal{P}(\mathbf{t})$. The following lemma shows that $\mathcal{P}$ implements the meta-operation of substitution in the λ-calculus: to compute $M\{x \mapsto N\}$ we compile M, compile N, create an explicit substitution, and normalise it with $\mathcal{P}$.

Lemma 8 (Substitution) *Let M and N be λ-terms. Let $[\![M]\!]_{\vec{x}} = t^\rho$, $[\![N]\!]_{\vec{y}} = \mathbf{u}$. Let $\rho' = \rho-$ and $i = |\rho| + 1$ if $x \notin \vec{x}$, otherwise $\rho' = \rho$ and i is the position of x in $\vec{x}$. Then $(\!|\mathcal{P}((t^{\rho'}[i/\mathbf{u}])^\sigma)|\!)_{\vec{z}} = M\{x \mapsto N\}$ (where $\vec{x}, \vec{y}, \vec{z}, \sigma$ are adequately chosen).*

Proof By induction on M. If M is a variable we distinguish two cases: if $M = y \neq x$ we obtain the required result using the rule *Erase*, otherwise we use *Var*. The case of an application follows directly by induction using rules App_1, App_2, App_3 or *Erase*. For an abstraction we use the induction hypothesis and rules *Lam* or *Erase*. □

Reduction in λ_o is sound with respect to the λ-calculus, in the following sense.

Proposition 6 (Soundness of λ_o) *If* $\mathbf{t} \rightsquigarrow_o \mathbf{u}$ *then* $(\!|\mathbf{t}|\!)_{\vec{x}} \to_\beta^* (\!|\mathbf{u}|\!)_{\vec{x}}$. *In particular:*

1. *if* $\mathbf{t} \rightsquigarrow_{\mathcal{P}} \mathbf{u}$ *then* $(\!|\mathbf{t}|\!)_{\vec{x}} = (\!|\mathbf{u}|\!)_{\vec{x}}$*, and*
2. *if* $\mathbf{t}$ *is a* $\mathcal{P}$*-normal form then* $(\!|\mathbf{t}|\!)_{\vec{x}} \to_\beta (\!|\mathbf{u}|\!)_{\vec{x}}$.

Proof The result is trivial if the reduction step uses a rule in $\mathcal{P}$, since the readback function performs the substitution. Note that

$$(\!|\mathbf{t}|\!)_{[x_1,\ldots,x_{i-1},M_i,x_{i+1},\ldots,x_n]} = (\!|\mathbf{t}|\!)_{[x_1,\ldots,x_n]}\{x_i \mapsto M_i\}.$$

Assume we use the *Beta* rule. We proceed by induction on $\mathbf{t}$. The only interesting case is when the reduction takes place at the root of $\mathbf{t}$. In this case $\mathbf{t} = ((\lambda\mathbf{w})^\rho\,\mathbf{v})^\sigma$ and $\mathbf{u} = (\mathbf{w}[|\rho|_+ + 1/\mathbf{v}])^\tau$. By definition of readback, $(\!|\mathbf{t}|\!)_{\vec{x}} = (\lambda x.(\!|\mathbf{w}|\!)_{\vec{y}})(\!|\mathbf{v}|\!)_{\vec{z}}$, where $\vec{x}, \vec{y}, \vec{z}$ are the corresponding lists of free variables. There is therefore a β-reduction step: $(\!|\mathbf{t}|\!)_{\vec{x}} \to_\beta (\!|\mathbf{w}|\!)_{\vec{y}}\{x \mapsto (\!|\mathbf{v}|\!)_{\vec{z}}\} = (\!|\mathbf{u}|\!)_{\vec{x}}$ by definition of readback. Note that in the general case this *Beta*-redex might occur in a substitution and it might be copied or erased by the readback function, thus $(\!|\mathbf{t}|\!)_{\vec{x}} \to_\beta^* (\!|\mathbf{u}|\!)_{\vec{x}}$. In particular, if $\mathbf{t}$ is a $\mathcal{P}$-normal form then $(\!|\mathbf{t}|\!)_{\vec{x}} \to_\beta (\!|\mathbf{u}|\!)_{\vec{x}}$ since $\mathbf{t}$ does not contain substitutions. □

We also have a Simulation Theorem which shows the completeness of λ_o with respect to β-reduction.

Theorem 1 (Simulation) *Let* M *be a* λ*-term with* $\mathsf{fv}(M) \subseteq \vec{x}$. *If* $M \to_\beta N$*, then there exists* $\mathbf{u}$ *such that* $[\![M]\!]_{\vec{x}} \rightsquigarrow_o^* \mathbf{u}$ *and* $(\!|\mathbf{u}|\!)_{\vec{x}} = N$.

$$\begin{array}{ccc} M & \xrightarrow{\ \to_\beta\ } & N \\ {\scriptstyle [\![\cdot]\!]}\downarrow & & \uparrow{\scriptstyle (\!|\cdot|\!)} \\ \mathbf{t} & \xrightarrow[\ \rightsquigarrow_o^*\ ]{} & \mathbf{u} \end{array}$$

Proof Since the compilation function (Definition 2) transforms applications into applications and abstractions into abstractions, it is clear that if M has a β-redex then $[\![M]\!]_{\vec{x}}$ has a *Beta*-redex. We proceed by induction on M. The only interesting case is when the β-reduction takes place at the root of M. In this case, $M = (\lambda x.M')\,N'$ and $N = M'\{x \mapsto N'\}$. Using the compilation function, we get:

– $[\![\lambda x.M']\!]_{\vec{z}} = (\lambda\mathbf{r})^\nu$ where $\mathbf{r} = [\![M']\!]_{\vec{z}'x}$
– $[\![N']\!]_{\vec{y}} = \mathbf{w}$
– $[\![M]\!]_{\vec{x}} \rightsquigarrow_{Beta} (\mathbf{r}[i/\mathbf{w}])^\sigma$

Using the Substitution Lemma (Lemma 8) we obtain the required result:

$$(\mathbf{r}[i/\mathbf{w}])^\sigma \rightsquigarrow_o^* \mathbf{u} \text{ with } (\!|\mathbf{u}|\!)_{\vec{x}} = N.$$

The reduction steps in the latter reduction are all in $\mathcal{P}$. □

Remark 4 In general, we do not have $\mathbf{u} = [\![N]\!]_{\vec{x}}$ (which would be stronger). For instance, $[\![\lambda xy.((\lambda z.y)\ x)]\!] = (\lambda(\lambda((\lambda\Box^{\downarrow -})^{\downarrow}\Box^{\downarrow})^{\searrow\swarrow})^{\downarrow})^{\epsilon} \rightsquigarrow_o^* (\lambda(\lambda\Box^{-\downarrow})^{\downarrow})^{\epsilon}$, and $(\!|(\lambda(\lambda\Box^{-\downarrow})^{\downarrow})^{\epsilon}|\!) = \lambda xy.y$, while $[\![\lambda xy.y]\!] = (\lambda(\lambda\Box^{\downarrow})^{-})^{\epsilon}$. Intuitively, there are two different levels where we may say that x should be erased. The compilation function makes the canonical choice to put this information at the highest possible level, i.e. right below the corresponding λ, but it may also happen that a variable disappears "at runtime", as in the example above. Then we could of course add a rule to make the erasing director go up until it reaches its canonical position, so that we would indeed have $\mathbf{u} = [\![N]\!]_{\vec{x}}$. But this would be useless, and this is exactly what the simulation theorem says: we may have different internal representations corresponding to the same term in the course of reduction, but we do not care as long as we can still correctly read back a λ-term at the end of the reduction. Hence although seemingly weaker than one could expect, Theorem 1 is exactly the right property for an intermediate language that cleanly separates the readback from the reduction rules.

In order to prove confluence of λ_o, we need some extra lemmas. Let's define $\mathbf{t} \to_B \mathbf{u}$ if $\mathbf{t}, \mathbf{u}$ are in $\mathcal{P}$-normal form and $\mathbf{t} \rightsquigarrow_{Beta} \rightsquigarrow_{\mathcal{P}}^* \mathbf{u}$.

Lemma 9 $\mathbf{t} \rightsquigarrow_o \mathbf{u} \Rightarrow \mathcal{P}(\mathbf{t}) \to_B^* \mathcal{P}(\mathbf{u})$.

Proof Obvious if $\mathbf{t} \rightsquigarrow_{\mathcal{P}} \mathbf{u}$. If $\mathbf{t} \rightsquigarrow_{Beta} \mathbf{u}$, the interesting case is when reduction is at the root. Then $((\lambda\mathbf{t})^\rho\, \mathbf{v})^\sigma \rightsquigarrow_{Beta} (\mathbf{t}[|\rho|_+{+}1/\mathbf{v}])^\tau$ and $((\lambda\mathcal{P}(\mathbf{t}))^\rho\, \mathcal{P}(\mathbf{v}))^\sigma \to_B \mathcal{P}((\mathcal{P}(\mathbf{t})[|\rho|_+{+}1/\mathcal{P}(\mathbf{v})])^\tau) = \mathcal{P}(\mathbf{u})$. □

Lemma 10 *If* $\mathbf{t} \to_B \mathbf{u}$ *then* $(\!|\mathbf{t}|\!)_{\vec{x}} \to_\beta (\!|\mathbf{u}|\!)_{\vec{x}}$ *(one step).*

Proof Direct consequence of Proposition 6. □

According to the relation $\to_B$ on $\mathcal{P}$-normal forms of λ_o-terms, we may define the notions of residuals and developments as in the λ-calculus (see [5]). Then clearly by Lemma 10, to a residual of $\mathbf{t}$ according to $\to_B$ corresponds a residual of $(\!|\mathbf{t}|\!)_{\vec{x}}$ using $\to_\beta$. We hence have a Finite Developments Theorem for $\to_B$:

Lemma 11 (FD for $\to_B$) *If* $\mathbf{t}$ *is a* $\mathcal{P}$*-normal form, all developments of* $\mathbf{t}$ *according to the reduction* $\to_B$ *are finite.*

Proof Assume an infinite development of $(\mathbf{t}, \mathcal{F})$ where $\mathbf{t}$ is a λ_o-term in $\mathcal{P}$-normal form and $\mathcal{F}$ a set of $\to_B$-redex occurrences of $\mathbf{t}$:

$$\mathbf{t} \to_B \mathbf{t}_1 \to_B \cdots \to_B \mathbf{t}_n \to_B \cdots$$

Then we have by Lemma 10:

$$(\!|\mathbf{t}|\!)_{\vec{x}} \to_\beta (\!|\mathbf{t}_1|\!)_{\vec{x}} \to_\beta \cdots \to_\beta (\!|\mathbf{t}_n|\!)_{\vec{x}} \to_\beta \cdots$$

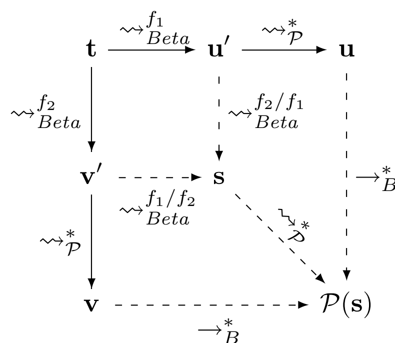
where every step is a residual of a redex corresponding to $\mathcal{F}$. Since the λ-calculus satisfies the Finite Developments Theorem [5], this is impossible. □

A complete development of $(\mathbf{t}, \mathcal{F})$ is a development of $(\mathbf{t}, \mathcal{F})$ such that there is no residual of $\mathcal{F}$ after it.

Lemma 12 *All complete developments of* $(\mathbf{t}, \mathcal{F})$ *with* $\rightarrow_B$ *end with the same term.*

Proof Assume that $\mathbf{t} \rightarrow_B \mathbf{u}$ and $\mathbf{t} \rightarrow_B \mathbf{v}$ are the first steps in two complete developments of $(\mathbf{t}, \mathcal{F})$, obtained by reducing redexes f_1 and f_2 respectively. Then, by definition of $\rightarrow_B$, $\mathbf{u}$ and $\mathbf{v}$ are $\mathcal{P}$-normal forms, and there are terms $\mathbf{u}'$ and $\mathbf{v}'$ such that $\mathbf{t} \rightsquigarrow_{Beta} \mathbf{u}' \rightsquigarrow^*_{\mathcal{P}} \mathbf{u} \equiv \mathcal{P}(\mathbf{u}')$ and $\mathbf{t} \rightsquigarrow_{Beta} \mathbf{v}' \rightsquigarrow^*_{\mathcal{P}} \mathbf{v} \equiv \mathcal{P}(\mathbf{v}')$.

If the $Beta$-redex reduced is the same in both cases, then obviously $\mathbf{u} \equiv \mathbf{v}$ because $\rightsquigarrow_{\mathcal{P}}$ is confluent. Let's assume that $f_1, f_2 \in \mathcal{F}$ are different $Beta$-redexes. Then $\mathbf{u}$ and $\mathbf{v}$ can be joined using Lemma 9, as shown in the following diagram, where $\rightsquigarrow^{f_i/f_j}_{Beta}$ denotes the $Beta$-reduction of the residuals of f_i relative to f_j (see [5]):



Note that the rule $Beta$ is left-linear and only superposes trivially with itself, which guarantees the existence of the terms $\mathbf{s}$ and $\mathcal{P}(\mathbf{s})$ obtained by developing $(\mathbf{t}, \{f_1, f_2\})$.

Lemma 11 and Newman's Lemma [32] complete the proof. □

Lemma 13 $\rightarrow_B$ *is confluent.*

Proof Define $\mathbf{t} \rightarrowtail_B \mathbf{u}$ if $\mathbf{u}$ is a complete development of $(\mathbf{t}, \mathcal{F})$ according to $\rightarrow_B$ for some $\mathcal{F}$. Then:

- $\rightarrow^*_B = \rightarrowtail^*_B$
- $\rightarrowtail_B$ is strongly confluent: if $\mathbf{t} \rightarrowtail_B \mathbf{t}_1$ (with $\mathcal{F}_1$) and $\mathbf{t} \rightarrowtail_B \mathbf{t}_2$ (with $\mathcal{F}_2$), then let $\mathbf{t}_3$ be the complete development of $\mathbf{t}$ with $\mathcal{F}_1 \cup \mathcal{F}_2$. $\mathbf{t} \rightarrowtail_B \mathbf{t}_1$ results from a partial development of $(\mathbf{t}, \mathcal{F}_1 \cup \mathcal{F}_2)$, hence by completely developing the residuals of $\mathcal{F}_1 \cup \mathcal{F}_2$ in $\mathbf{t}_1$, one obtains by Lemma 12 $\mathbf{t}_1 \rightarrowtail_B \mathbf{t}_3$. Similarly $\mathbf{t}_2 \rightarrowtail_B \mathbf{t}_3$.

□

Theorem 2 (Confluence) *The calculus* λ_o *is confluent.*

Proof The situation is the following:

$$\begin{array}{ccc} \mathbf{t} & \xrightarrow{\rightsquigarrow_o^*} & \mathbf{u} \\ {\scriptstyle \rightsquigarrow_o^*}\downarrow & & \downarrow{\scriptstyle \rightsquigarrow_o^*} \\ \mathbf{v} & \xrightarrow[\rightsquigarrow_o^*]{} & \mathbf{w} \end{array}$$

We use the interpretation method. By Lemma 9, $\mathcal{P}(\mathbf{t}) \to_B^* \mathcal{P}(\mathbf{u})$ and $\mathcal{P}(\mathbf{t}) \to_B^* \mathcal{P}(\mathbf{v})$. By Lemma 13, there exists $\mathbf{w}$ such that $\mathcal{P}(\mathbf{u}) \to_B^* \mathbf{w}$ and $\mathcal{P}(\mathbf{v}) \to_B^* \mathbf{w}$. And since $\to_B^* \subset \rightsquigarrow_o^*$, the derivations $\mathbf{u} \rightsquigarrow_o^* \mathcal{P}(\mathbf{u}) \rightsquigarrow_o^* \mathbf{w}$ and $\mathbf{v} \rightsquigarrow_o^* \mathcal{P}(\mathbf{v}) \rightsquigarrow_o^* \mathbf{w}$ conclude the proof. □

The confluence property above is sometimes called *ground confluence* in the terminology of explicit substitution calculi, since it applies to terms in λ_o rather than terms with metavariables of the rewrite system. Another property that has been extensively studied for explicit substitution calculi is the preservation of strong normalisation (PSN): a calculus of explicit substitutions preserves strong normalisation if the compilation of a strongly normalisable λ-term is strongly normalisable (see [29] for a counterexample for $\lambda\sigma$, and [27,12] for calculi satisfying PSN). Thanks to our limited form of composition (the rule *Comp* only allows to move a substitution inside another where the substituted variable does occur) λ_o preserves strong normalisation. Our proof of PSN is inspired from that of λ_υ [27]. We first define a notion of minimal infinite derivation and external reduction. Intuitively, a derivation is minimal if we always reduce a lowest possible redex to keep non termination, and a reduction step is external if it does not take place inside a substitution. We denote by $\rightsquigarrow_{Beta,p}$ a *Beta* reduction at position p, and by $\rightsquigarrow_{o,p}$ an arbitrary λ_o reduction at position p.

Definition 5 (Minimal Derivation) *An infinite λ_o sequence of reductions*

$$\mathbf{t}_1 \rightsquigarrow_{Beta,p_1} \mathbf{t}'_1 \rightsquigarrow_{\mathcal{P}}^* \cdots \mathbf{t}_i \rightsquigarrow_{Beta,p_i} \mathbf{t}'_i \rightsquigarrow_{\mathcal{P}}^* \ldots$$

is minimal *if for any other infinite derivation*

$$\mathbf{t}_1 \rightsquigarrow_{Beta,p_1} \mathbf{t}'_1 \rightsquigarrow_{\mathcal{P}}^* \cdots \mathbf{t}_i \rightsquigarrow_{Beta,q} \mathbf{u} \rightsquigarrow_{\mathcal{P}}^* \cdots$$

we have $q \neq p_i p'$ for every p'.

In other words, in any other infinite derivation, p_i and q are disjoint or q is above p_i, which means that the *Beta*-redex we reduce is a lowest one.

Definition 6 (External Reduction) *The set $Ext(\mathbf{t})$ of external positions in the term $\mathbf{t}$ is defined as follows, where Λ denotes the root position and $x\,X$ denotes $\{x \cdot y \mid y \in X\}$ if X is a set of positions:*

$$\begin{array}{ll} Ext(\Box^\sigma) & = \{\Lambda\} \\ Ext((\lambda\mathbf{t})^\sigma) & = 1Ext(\mathbf{t}) \cup \{\Lambda\} \\ Ext((\mathbf{t}\ \mathbf{u})^\sigma) & = 1Ext(\mathbf{t}) \cup 2Ext(\mathbf{u}) \cup \{\Lambda\} \\ Ext((\mathbf{t}[k/\mathbf{u}])^\sigma) & = 1Ext(\mathbf{t}) \cup \{\Lambda\} \end{array}$$

A reduction step $\mathbf{t} \rightsquigarrow_{o,p} \mathbf{u}$ *is external if* $p \in Ext(\mathbf{t})$*, otherwise it is internal. We write* $\mathbf{t} \rightsquigarrow_o^{ext} \mathbf{u}$ *(resp.* $\mathbf{t} \rightsquigarrow_o^{int} \mathbf{u}$*) to emphasise that a rewrite is external (resp. internal).*

Lemma 14 *If* $\mathbf{t} \rightsquigarrow_{Beta,p} \mathbf{u}$ *is an external reduction step then* $(\!|\mathbf{t}|\!)_{\vec{x}} \rightarrow_\beta^+ (\!|\mathbf{u}|\!)_{\vec{x}}$*. In particular* $(\!|\mathbf{t}|\!)_{\vec{x}} \neq (\!|\mathbf{u}|\!)_{\vec{x}}$ *if* $(\!|\mathbf{t}|\!)_{\vec{x}}$ *is strongly normalisable.*

Proof The term $\mathbf{t}$ contains a *Beta* redex, and since it is external it is obviously not erased in the readback (see Proposition 6). Hence $(\!|\mathbf{t}|\!)_{\vec{x}} \rightarrow_\beta^+ (\!|\mathbf{u}|\!)_{\vec{x}}$. □

Lemma 15 *If there is an infinite derivation*

$$\mathbf{t} \rightsquigarrow_o \mathbf{t}_1 \rightsquigarrow_o \mathbf{t}_2 \rightsquigarrow_o \mathbf{t}_3 \rightsquigarrow_o \mathbf{t}_4 \cdots$$

in which all the Beta *steps are internal, then there exists another infinite derivation*

$$\mathbf{t} \rightsquigarrow_{\mathcal{P}}^* \mathbf{u}_1 \rightsquigarrow_o \mathbf{u}_2 \rightsquigarrow_o \mathbf{u}_3 \rightsquigarrow_o \mathbf{u}_4 \cdots$$

in which the rewrites from $\mathbf{t}$ *to* $\mathbf{u}_1$ *are the only external ones (i.e.* all *steps are internal from* $\mathbf{u}_1$*). Moreover, the transformation preserves minimality.*

Proof It is sufficient to prove that if $\mathbf{u} \rightsquigarrow_o^{int} \mathbf{v} \rightsquigarrow_o^{ext} \mathbf{w}$ in a minimal derivation then $\mathbf{u} \rightsquigarrow_o^{ext} \mathbf{v}' \rightsquigarrow_o^{int*} \mathbf{w}$ preserving minimality. The proof is by induction on the structure of $\mathbf{u}$. If the external rewrite does not take place at the root position then the result follows by induction. If the external rewrite takes place at the root of $\mathbf{u}$, then the rule applied is in $\mathcal{P}$ by assumption, and therefore $\mathbf{u}$ is a term of the form $(\mathbf{t}[i/\mathbf{v}])^\sigma$. We distinguish cases according to the rule applied. If the rule is *Var*, *App*$_1$, *App*$_2$, *Lam*, or *Comp* the rewrite steps commute. If the rule is *App*$_3$ the internal rewrite is replaced by two internal rewrites. If the rule is *Erase* the internal rewrite is not needed.

The termination of $\mathcal{P}$ (Proposition 4) ensures that there are a finite number of external steps $\mathbf{t} \rightsquigarrow_{\mathcal{P}}^* \mathbf{u}_1$, completing the proof. □

We will prove a result which is slightly more general than preservation of strong normalisation and will also be used to prove the termination of typeable terms in Section 5.

Proposition 7 *If* $(\!|\mathbf{t}|\!)_{\vec{x}}$ *is a strongly normalisable* λ*-term then* $\mathbf{t}$ *is strongly normalisable in* λ_o*.*

Proof For a contradiction, assume that there is an infinite reduction sequence starting from $\mathbf{t}$. We show that there is an infinite derivation for $(\!|\mathbf{t}|\!)_{\vec{x}}$. For this we consider an infinite *minimal* reduction sequence, which contains an infinite number of applications of *Beta* since the propagation rules are terminating:

$$\mathbf{t} \rightsquigarrow_{\mathcal{P}}^* \mathbf{t}_1 \rightsquigarrow_{Beta} \mathbf{t}_2 \rightsquigarrow_{\mathcal{P}}^* \mathbf{t}_3 \rightsquigarrow_{Beta} \mathbf{t}_4 \cdots$$

By Proposition 6:

$$(\!|\mathbf{t}|\!)_{\vec{x}} = (\!|\mathbf{t}_1|\!)_{\vec{x}} \rightarrow_\beta^* (\!|\mathbf{t}_2|\!)_{\vec{x}} = (\!|\mathbf{t}_3|\!)_{\vec{x}} \rightarrow_\beta^* (\!|\mathbf{t}_4|\!)_{\vec{x}} \cdots$$

and since $(\!|\mathbf{t}|\!)_{\vec{x}}$ is strongly normalisable by assumption, there exists k such that for all $i \geq k$, $(\!|\mathbf{t}_i|\!)_{\vec{x}} = (\!|\mathbf{t}_{i+1}|\!)_{\vec{x}}$. Hence, by Lemma 14, all the *Beta* reduction steps after that point are internal, and by Lemma 15 we can assume that *all* the reduction steps after $\mathbf{t}_k$ are internal. There is therefore a subterm $\mathbf{u}[i/\mathbf{v}]$ of $\mathbf{t}_k$ with an infinite reduction sequence out of $\mathbf{v}$. But this substitution was created by a previous *Beta* step, which contradicts the minimality assumption. □

Theorem 3 (PSN) *If M is a strongly normalisable λ-term with $\mathsf{fv}(M) \subseteq \vec{x}$ then $[\![M]\!]_{\vec{x}}$ is strongly normalisable in λ_o.*

Proof Direct consequence of Propositions 7 and 3. □

4 Simplified Calculi

We now have a general framework to simulate the λ-calculus with a director strings notation. However, our aim is to search for new efficient reduction strategies, so we may give up completeness if we can gain some efficiency and simplicity, provided that we are still able at least to reduce closed terms to weak head normal form[2], which is the widely accepted minimal requirement for a λ-evaluator, such as found in functional compilers and interpreters.

We will thus introduce two new calculi λ_l and λ_c (with reduction $\rightsquigarrow_l$ and $\rightsquigarrow_c$ respectively) on the same terms, which will be simplifications (specialisations) of λ_o. By definition, the following relation will hold:

$$\rightsquigarrow_c \subset \rightsquigarrow_l \subset \rightsquigarrow_o^*$$

so that, for instance, λ_l and λ_c will inherit properties of termination of λ_o.

From an algorithmic point of view, the rewrite rules of λ_o cannot be considered as constant time operations, because we have to access directors at arbitrary positions in strings, and the computation of the new director strings *a priori* seems to require a time linear in the size of the original ones. In contrast with λ_o, the calculus λ_l performs only *local* reduction steps. In other words, λ_l is a restriction of λ_o where the rewrite rules make only local modifications in the director strings. It is the most general sub-calculus of λ_o with such a property.

The calculus λ_c is a restriction of λ_l which implements a *closed reduction strategy*. This is the most interesting calculus from an implementation perspective, as our benchmarks show (see Section 8).

We will first define the common syntax for λ_l and λ_c, then we will give the two sets of rewrite rules defining these calculi, focusing on the properties of λ_c (the proofs for λ_l are similar).

[2] A λ-term is in weak head normal form if it is either an abstraction or a nested application ultimately beginning by a variable, i.e. any term not beginning by a redex.

4.1 Syntax

Definition 7 (Simplified Terms) *In this section, the syntax is slightly different:*

- *The directors are $\swarrow$, $\searrow$, $\curlywedge$ and $\downarrow$ only ($-$ is no longer a director).*
- *Preterms are defined according to the following grammar:*

$$\mathbf{t} ::= \Box \mid (\lambda \mathbf{t})^\sigma \mid (\lambda^- \mathbf{t})^\sigma \mid (\mathbf{t}\ \mathbf{u})^\sigma \mid (\mathbf{t}[\mathbf{u}])^\sigma$$

  where $(\lambda^- \mathbf{t})^\sigma$ is an abstraction where the bound variable does not occur in $\mathbf{t}$ and $(\mathbf{t}[\mathbf{u}])^\sigma$ is a substitution for the first variable of $\mathbf{t}$ only.
- *Terms are preterms satisfying the constraints given in Definition 1 for λ_o, taking $\mathcal{U} = \downarrow^*$ and $\mathcal{B} = (\swarrow \mid \curlywedge \mid \searrow)^*$. Note that now variables do not have a director string. Moreover, abstractions have the following additional constraint:*

$$(\lambda^- t^\rho)^\sigma \textit{ is well-formed if } \sigma \in \mathcal{U} \textit{ and } |\rho| = |\sigma|$$

The new set of terms can be seen as a subset of λ_o-terms using some abbreviations:

Proposition 8 (Syntactic Equivalence) *The following mapping from simplified terms to λ_o-terms is injective.*

- $\gimel(\Box) = \Box^\downarrow$
- $\gimel((\lambda \mathbf{t})^\sigma) = (\lambda \gimel(\mathbf{t}))^\sigma$
- $\gimel((\lambda^- t^\rho)^\sigma) = (\lambda \gimel(t^{\rho-}))^\sigma$
- $\gimel((\mathbf{t}\ \mathbf{u})^\sigma) = (\gimel(\mathbf{t})\ \gimel(\mathbf{u}))^\sigma$
- $\gimel((\mathbf{t}[\mathbf{u}])^\sigma) = (\gimel(\mathbf{t})[1/\gimel(\mathbf{u})])^\sigma$

We will hence sometimes feel free to identify simplified terms with λ_o-terms (i.e. to omit $\gimel$).

4.2 The Local Open Calculus

Taking a closer look at the *Beta* rule, we notice that for a redex where the function is closed, we generate a substitution for the first (and only) variable of the function body (which was bound by the abstraction). Moreover, we know from [17] that restricting β-reduction to closed functions still allows to reach weak head normal form, for closed terms. In this section we will thus describe the calculus resulting from this restriction which greatly simplifies the rules. This calculus will be called *local open* (λ_l) because it still allows open substitutions to propagate, even inside abstractions (in contrast with the system of closed reduction [17]). This does not need global rewrite steps if we restrict the syntax to allow substitutions for the first director only.

Definition 8 (**λ_l-calculus**) *Below are shown the reduction rules for the local open calculus.*

Name	*Reduction*		
Beta	$((\lambda \mathbf{t})^{\epsilon}\ \mathbf{u})^{\sigma}$	$\rightsquigarrow_l$	$(\mathbf{t}[\mathbf{u}])^{\sigma}$
BetaE	$((\lambda^{-} \mathbf{t})^{\epsilon}\ u^{\epsilon})^{\epsilon}$	$\rightsquigarrow_l$	$\mathbf{t}$
Var	$(\square[\mathbf{v}])^{\sigma}$	$\rightsquigarrow_l$	$\mathbf{v}$
App$_1$	$((\mathbf{t}\ \mathbf{u})^{\swarrow\rho}[\mathbf{v}])^{\searrow^m \cdot \swarrow^n}$	$\rightsquigarrow_l$	$((\mathbf{t}[\mathbf{v}])^{\searrow^m \cdot \swarrow^{\vert\rho\vert_l}}\ \mathbf{u})^{\swarrow^m \cdot \rho}$
App$_2$	$((\mathbf{t}\ \mathbf{u})^{\searrow\rho}[\mathbf{v}])^{\searrow^m \cdot \swarrow^n}$	$\rightsquigarrow_l$	$(\mathbf{t}\ (\mathbf{u}[\mathbf{v}])^{\searrow^m \cdot \swarrow^{\vert\rho\vert_r}})^{\searrow^m \cdot \rho}$
App$_3$	$((\mathbf{t}\ \mathbf{u})^{\curlywedge\rho}[\mathbf{v}])^{\searrow^m \cdot \swarrow^n}$	$\rightsquigarrow_l$	$((\mathbf{t}[\mathbf{v}])^{\searrow^m \cdot \swarrow^{\vert\rho\vert_l}}\ (\mathbf{u}[\mathbf{v}])^{\searrow^m \cdot \swarrow^{\vert\rho\vert_r}})^{\curlywedge^m \cdot \rho}$
Lam	$((\lambda \mathbf{t})^{\downarrow\rho}[\mathbf{v}])^{\sigma}$	$\rightsquigarrow_l$	$(\lambda (\mathbf{t}[\mathbf{v}])^{\sigma \cdot \swarrow})^{\downarrow^{\vert\sigma\vert}}$
LamE	$((\lambda^{-} \mathbf{t})^{\downarrow\rho}[\mathbf{v}])^{\sigma}$	$\rightsquigarrow_l$	$(\lambda^{-} (\mathbf{t}[\mathbf{v}])^{\sigma})^{\downarrow^{\vert\sigma\vert}}$
Comp	$((\mathbf{t}[\mathbf{w}])^{\searrow^{n+1} \cdot \swarrow^m}[\mathbf{v}])^{\searrow^p \cdot \swarrow^q}$	$\rightsquigarrow_l$	$(\mathbf{t}[(\mathbf{w}[\mathbf{v}])^{\searrow^p \cdot \swarrow^n}])^{\searrow^{p+n} \cdot \swarrow^m}$

Even though the rules in this system apply to terms with director strings of a particular pattern, the system is still suitable to reduce general terms, thanks to the following property.

Lemma 16 (Completeness of the Reduction) *In any reduct of a closed compiled term, any subterm of the form $(\mathbf{t}[\mathbf{v}])^{\sigma}$ has a director string $\sigma = \searrow^m \cdot \swarrow^n$ for some natural numbers m, n.*

Proof It is sufficient to notice that the rules for propagation only generate substitutions of this form, and that the *Beta* rule does as well: if the term $((\lambda \mathbf{t})^{\epsilon}\ \mathbf{u})^{\sigma}$ is well-formed (and it is by induction) then σ is of the form $\searrow^n$. □

We remark that, in this calculus, we allow even open substitutions to pass through abstractions, without any global reduction step. This is of course one of the greatest strengths of this calculus, compared to those based on names or de Bruijn indices.

Lemma 17 (Preservation of Well-Formedness) *If $\mathbf{t}$ is a well-formed term in λ_l and $\mathbf{t} \rightsquigarrow_l \mathbf{u}$ then $\mathbf{u}$ is a well-formed term in λ_l.*

Proof Straightforward inspection of the rewrite rules. □

Proposition 9 $\rightsquigarrow_l \subset \rightsquigarrow_o^*$

Proof We can easily check that all the rules given in Definition 8, except *BetaE*, are particular cases of the rules for λ_o taking into account the simplified syntax of terms. The rule *BetaE* can be simulated in λ_o with two reductions: *Beta* followed by *Erase*. □

Theorem 4 below summarises the properties of λ_l. The calculus λ_l is confluent and preserves strong normalisation. However, it does not fully simulate β-reduction. Instead we can show that it can be used to evaluate closed λ-terms. This is a consequence of the fact that λ_c, which will be shown to be a restriction of λ_l, can compute weak head normal forms of closed terms (see next section).

Theorem 4 (Properties of λ_l)

Confluence: λ_l is confluent.
PSN: λ_l preserves strong normalisation.
Adequacy: λ_l can evaluate closed terms.

Proof Confluence is shown by easily adapting the proof of confluence of λ_c, which is given in the next section. PSN comes from Theorem 3 (PSN for λ_o) and Proposition 9. Adequacy will be obvious from Theorems 7 and 8 (adequacy for λ_c) and Proposition 10 (see next section). $\square$

4.3 The Closed Calculus

The notion of closed reduction was introduced in [17] using a calculus of explicit substitutions with names where β-reductions are performed when the function part is closed (as above) and substitutions are propagated through abstractions only if they are closed, which is crucial to avoid α-conversion in a named setting. These restrictions were expressed by rewrite rules with external conditions on free variables, and the internalisation of these conditions was the main motivation for the introduction of director strings in [18].

We can easily derive a calculus for closed reduction (λ_c) by adding restrictions to the rules of λ_l (see Definition 8) as follows: *Beta*, *BetaE*, *Var* are the same; in every other rule we force the substitution $\mathbf{v}$ to be closed, that is, to have an empty string (ϵ). Moreover, in *App*$_{1,2,3}$, $m = 0$, and in *Comp*, $m = p = 0$ and $n = q$. This thus leads to the very simple following rewrite system.

Definition 9 (λ_c-calculus) *The reduction rules for the closed calculus are:*

Name	*Reduction*		
Beta	$((\lambda\mathbf{t})^{\epsilon}\ \mathbf{u})^{\searrow^n}$	$\rightsquigarrow_c$	$(\mathbf{t}[\mathbf{u}])^{\searrow^n}$
BetaE	$((\lambda^{-}\mathbf{t})^{\epsilon}\ u^{\epsilon})^{\epsilon}$	$\rightsquigarrow_c$	$\mathbf{t}$
Var	$(\square[\mathbf{v}])^{\searrow^n}$	$\rightsquigarrow_c$	$\mathbf{v}$
App$_1$	$((\mathbf{t}\ \mathbf{u})^{\swarrow\rho}[v^{\epsilon}])^{\swarrow^n}$	$\rightsquigarrow_c$	$((\mathbf{t}[v^{\epsilon}])^{\swarrow^{\lvert\rho\rvert_l}}\ \mathbf{u})^{\rho}$
App$_2$	$((\mathbf{t}\ \mathbf{u})^{\searrow\rho}[v^{\epsilon}])^{\swarrow^n}$	$\rightsquigarrow_c$	$(\mathbf{t}\ (\mathbf{u}[v^{\epsilon}])^{\swarrow^{\lvert\rho\rvert_r}})^{\rho}$
App$_3$	$((\mathbf{t}\ \mathbf{u})^{\swarrow\!\!\!\searrow\rho}[v^{\epsilon}])^{\swarrow^n}$	$\rightsquigarrow_c$	$((\mathbf{t}[v^{\epsilon}])^{\swarrow^{\lvert\rho\rvert_l}}\ (\mathbf{u}[v^{\epsilon}])^{\swarrow^{\lvert\rho\rvert_r}})^{\rho}$
Lam	$((\lambda\mathbf{t})^{\downarrow\rho}[v^{\epsilon}])^{\swarrow^n}$	$\rightsquigarrow_c$	$(\lambda(\mathbf{t}[v^{\epsilon}])^{\swarrow^{n+1}})^{\downarrow^n}$
LamE	$((\lambda^{-}\mathbf{t})^{\downarrow\rho}[v^{\epsilon}])^{\swarrow^n}$	$\rightsquigarrow_c$	$(\lambda^{-}(\mathbf{t}[v^{\epsilon}])^{\swarrow^n})^{\downarrow^n}$
Comp	$((\mathbf{t}[\mathbf{w}])^{\searrow^{n+1}}[v^{\epsilon}])^{\swarrow^n}$	$\rightsquigarrow_c$	$(\mathbf{t}[(\mathbf{w}[v^{\epsilon}])^{\swarrow^n}])^{\searrow^n}$

This system is complete because a property similar to Lemma 16 still holds here:

Lemma 18 (Completeness of the Reduction) *In any reduct of a closed compiled term, if a subterm is of the form $(\mathbf{t}[\mathbf{v}])^{\sigma}$ then either $\sigma = \swarrow^n$ or $\sigma = \searrow^n$ for some natural number n.*

Proof *Beta* creates a substitution annotated by $\searrow^n$, the other rules only generate substitutions with a $\swarrow^n$ string. $\Box$

Note that the choice between $\swarrow^n$ and $\searrow^n$ is always obvious from the context (by well-formedness). For instance, in rule *Comp*, we reduce a term of the form $((\mathbf{t}[\mathbf{w}])^{\searrow^{n+1}}[v^\epsilon])^{\swarrow^m}$, then it is easy to see that $m = n$.

Lemma 19 (Preservation of Well-Formedness) *If t^σ is a well-formed term in λ_c and $t^\sigma \leadsto_c u^\tau$ then u^τ is a well-formed term in λ_c. Moreover, $|\sigma| = |\tau|$.*

Proof Straightforward inspection of the rewrite rules. $\Box$

Proposition 10 $\leadsto_c \subset \leadsto_l \subset \leadsto_o^*$

Proof Since the rewrite rules in Definition 9 are particular instances of the rules of λ_l (Definition 8), the rewrite relation $\leadsto_c$ is included in $\leadsto_l$. The latter inclusion was established in Proposition 9. $\Box$

We will show that $\leadsto_c$ is confluent on λ_c-terms, preserves strong normalisation, and implements call-by-value and call-by-name evaluation strategies for closed λ-terms. For this, we first show that the propagation rules (i.e. all the rules except *Beta* and *BetaE*) are terminating and confluent, and are sufficient to implement the metaoperation of substitution in the λ-calculus, provided substitutions are closed. The set of propagation rules will be denoted $\mathcal{P}_c$.

Proposition 11 (Termination and Confluence of $\mathcal{P}_c$) *All the reduction sequences on λ_c using only rules in $\mathcal{P}_c$ are finite. Moreover, $\mathcal{P}_c$ is locally confluent (hence confluent).*

Proof Termination is a direct consequence of Proposition 4 since $\leadsto^*_{\mathcal{P}_c} \subset \leadsto^*_{\mathcal{P}}$.

Local confluence is proved by showing that the critical pairs are joinable. There is only one, between *Comp* and *Var* (the conditions on the director strings prevent superpositions between *Comp* and the other propagation rules).

$$\begin{array}{ccc}
((\Box[\mathbf{v}])^{\searrow^{n+1}}[u^\epsilon])^{\swarrow^n} & \xrightarrow{\textit{Comp}} & (\Box[(\mathbf{v}[u^\epsilon])^{\swarrow^n}])^{\searrow^n} \\
\big\downarrow \textit{Var} & & \big\downarrow \textit{Var} \\
(\mathbf{v}[u^\epsilon])^{\swarrow^n} & \equiv & (\mathbf{v}[u^\epsilon])^{\swarrow^n}
\end{array}$$

Since the propagation rules are terminating, the system is confluent by Newman's Lemma [32]. $\Box$

Since $\mathcal{P}_c$ is terminating and confluent, it defines a function associating to each λ_c-term $\mathbf{t}$ its unique normal form, denoted $\mathcal{P}_c(\mathbf{t})$.

Lemma 20 (Closed Substitutions) *Let M and N be λ-terms such that $x \in \mathsf{fv}(M) \subseteq \vec{x}$ and $\mathsf{fv}(N) = \emptyset$. Let $[\![M]\!]_{\vec{x}} = \mathbf{t}$, $[\![N]\!] = u^\epsilon$, $[\![M\{x \mapsto N\}]\!]_{\vec{y}} = v^\sigma$, where $\vec{y} = \vec{x} \backslash \{x\}$. Then $\mathcal{P}_c((\mathbf{t}[u^\epsilon])^\sigma) = v^\sigma$.*

Proof By induction on M. If M is a variable (in this case it must be x) then its compilation is simply $\Box$ and using the rule *Var*, $\Box[u^\epsilon] \rightsquigarrow_c u^\epsilon = [\![M\{x \mapsto N\}]\!]_{\vec{y}}$ as required. If M is an application, then one of the rules App_1, App_2, App_3 applies, and the result follows directly by induction. If M is an abstraction then either *Lam* or *LamE* applies and again the result follows by induction. $\Box$

As a consequence, we deduce that even with the restrictions imposed by λ_c rules, closed substitutions do not remain blocked: if a closed substitution is created, it will be fully propagated.

The following lemma shows that the rules *Beta* and *BetaE* also generate a confluent relation, which we denote $\rightsquigarrow_B$. In the proof we use an auxiliary relation, $\Rightarrow$, which makes $\rightsquigarrow_B$ steps in parallel. It is defined by induction.

Definition 10 (Parallel Beta) *Let $\Rightarrow$ be the rewrite relation on λ_c-terms inductively defined as follows.*

1. $\mathbf{t} \Rightarrow \mathbf{t}$,
2. $((\lambda\mathbf{t})^\epsilon\, \mathbf{u})^{\searrow^n} \Rightarrow (\mathbf{t}'[\mathbf{u}'])^{\searrow^n}$ *if* $\mathbf{t} \Rightarrow \mathbf{t}'$ *and* $\mathbf{u} \Rightarrow \mathbf{u}'$,
3. $((\lambda^-\mathbf{t})^\epsilon\, u^\epsilon)^\epsilon \Rightarrow \mathbf{t}'$ *if* $\mathbf{t} \Rightarrow \mathbf{t}'$,
4. $(\lambda\mathbf{t})^\sigma \Rightarrow (\lambda\mathbf{t}')^\sigma$ *if* $t \Rightarrow t'$,
5. $(\mathbf{t}\ \mathbf{u})^\sigma \Rightarrow (\mathbf{t}'\ \mathbf{u}')^\sigma$ *if* $\mathbf{t} \Rightarrow \mathbf{t}'$ *and* $\mathbf{u} \Rightarrow \mathbf{u}'$,
6. $(\mathbf{t}[\mathbf{v}])^\sigma \Rightarrow (\mathbf{t}'[\mathbf{v}'])^\sigma$ *if* $\mathbf{t} \Rightarrow \mathbf{t}'$ *and* $\mathbf{v} \Rightarrow \mathbf{v}'$.

Lemma 21 (Confluence of Beta reductions) $\rightsquigarrow_B$ *is confluent.*

Proof We show that $\Rightarrow$ is strongly confluent (i.e. has the diamond property), and since obviously $\rightsquigarrow_B \subseteq \Rightarrow$ and $\Rightarrow^* = \rightsquigarrow_B^*$, we obtain confluence of $\rightsquigarrow_B$.

Assume $\mathbf{t} \Rightarrow \mathbf{u}$ and $\mathbf{t} \Rightarrow \mathbf{v}$ ($\mathbf{u} \neq \mathbf{v}$). We show that $\exists\mathbf{w}$ such that $\mathbf{u} \Rightarrow \mathbf{w}$ and $\mathbf{v} \Rightarrow \mathbf{w}$ by induction on $\mathbf{t} \Rightarrow \mathbf{u}$. The proof is standard. The case $\mathbf{t} \Rightarrow \mathbf{t}$ is trivial, taking $\mathbf{w} \equiv \mathbf{v}$. We give the details for $((\lambda\mathbf{t})^\epsilon\, \mathbf{u})^{\searrow^n} \Rightarrow (\mathbf{t}'[\mathbf{u}'])^{\searrow^n}$, where $\mathbf{t} \Rightarrow \mathbf{t}'$ and $\mathbf{u} \Rightarrow \mathbf{u}'$. In this case, the only alternatives are:

- $\mathbf{v} \equiv ((\lambda\mathbf{t})^\epsilon\, \mathbf{u})^{\searrow^n}$, in which case we take $\mathbf{w} \equiv (\mathbf{t}'[\mathbf{u}'])^{\searrow^n}$, or
- $\mathbf{v} \equiv ((\lambda\mathbf{t}'')^\epsilon\, \mathbf{u}'')^{\searrow^n}$, where $\mathbf{t} \Rightarrow \mathbf{t}''$ and $\mathbf{u} \Rightarrow \mathbf{u}''$. By induction, there exists $\mathbf{w}'$ and $\mathbf{w}''$ such that $\mathbf{t}' \Rightarrow \mathbf{w}'$, $\mathbf{t}'' \Rightarrow \mathbf{w}'$, $\mathbf{u}' \Rightarrow \mathbf{w}''$, $\mathbf{u}'' \Rightarrow \mathbf{w}''$. Hence we take $\mathbf{w} \equiv (\mathbf{w}'[\mathbf{w}''])^{\searrow^n}$.

The case $((\lambda^-\mathbf{t})^\epsilon\, u^\epsilon)^\epsilon \Rightarrow \mathbf{t}'$ where $\mathbf{t} \Rightarrow \mathbf{t}'$ is similar. All the other cases follow directly by induction. $\Box$

Next we prove the commutation of $\rightsquigarrow_B$ and $\rightsquigarrow_{\mathcal{P}_c}$. Rosen's Lemma [34] states that if two confluent rewrite relations are independent (commute) then their union is confluent. Since $\rightsquigarrow_c = \rightsquigarrow_B \cup \rightsquigarrow_{\mathcal{P}_c}$ and we have just shown that both relations are confluent, commutation implies confluence of $\rightsquigarrow_c$.

Lemma 22 (Commutation) *If* $\mathbf{t} \rightsquigarrow_{\mathcal{P}_c}^* \mathbf{a}$ *and* $\mathbf{t} \rightsquigarrow_B^* \mathbf{b}$*, then there exists* $\mathbf{c}$ *such that* $\mathbf{a} \rightsquigarrow_B^* \mathbf{c}$ *and* $\mathbf{b} \rightsquigarrow_{\mathcal{P}_c}^* \mathbf{c}$.

Proof Again we use the parallel reduction relation $\Rightarrow$ (Definition 10). Since $\Rightarrow^*$ coincides with $\rightsquigarrow_B^*$, it is sufficient to prove that if $\mathbf{b} \Leftarrow \mathbf{t} \rightsquigarrow_{\mathcal{P}_c} \mathbf{a}$ then there exists a term $\mathbf{c}$ such that $\mathbf{b} \rightsquigarrow_{\mathcal{P}_c} \mathbf{c} \Leftarrow \mathbf{a}$. In this way we can close the diagram: $\mathbf{t} \rightsquigarrow_{\mathcal{P}_c}^* \mathbf{a}$ and $\mathbf{t} \rightsquigarrow_B^* \mathbf{b}$ (which is equivalent to $\mathbf{b} \;{}^*\!\!\Leftarrow \mathbf{t} \rightsquigarrow_{\mathcal{P}_c}^* \mathbf{a}$), by induction on the length of the derivation $\mathbf{t} \Rightarrow^* \mathbf{b}$.

We proceed by induction on $\mathbf{t} \Rightarrow \mathbf{b}$. We distinguish the following cases:

1. $\mathbf{t} \equiv \mathbf{b}$: Then we take $\mathbf{c} \equiv \mathbf{a}$.
2. $((\lambda\mathbf{t})^\epsilon\,\mathbf{u})^{\searrow^n} \Rightarrow \mathbf{b} \equiv (\mathbf{t}'[\mathbf{u}'])^{\searrow^n}$ where $\mathbf{t} \Rightarrow \mathbf{t}'$ and $\mathbf{u} \Rightarrow \mathbf{u}'$: Since no rule from $\mathcal{P}_c$ applies at the root, it must be either $\mathbf{t} \rightsquigarrow_{\mathcal{P}_c} \mathbf{t}''$ or $\mathbf{u} \rightsquigarrow_{\mathcal{P}_c} \mathbf{u}''$. In the first case, by induction, there exists a term $\mathbf{t}'''$ such that $\mathbf{t}' \rightsquigarrow_{\mathcal{P}_c} \mathbf{t}'''$ and $\mathbf{t}'' \Rightarrow \mathbf{t}'''$, therefore we can take $\mathbf{c} \equiv (\mathbf{t}'''[\mathbf{u}'])^{\searrow^n}$. In the second case, by induction, there exists $\mathbf{u}'''$ such that $\mathbf{u}' \rightsquigarrow_{\mathcal{P}_c} \mathbf{u}'''$ and $\mathbf{u}'' \Rightarrow \mathbf{u}'''$, therefore we can take $\mathbf{c} \equiv (\mathbf{t}'[\mathbf{u}'''])^{\searrow^n}$.
3. The case $((\lambda^-\mathbf{t})^\epsilon\, u^\epsilon)^\epsilon \Rightarrow \mathbf{b} \equiv \mathbf{t}'$ where $\mathbf{t} \Rightarrow \mathbf{t}'$ is similar to the above.
4. $(\mathbf{t}[\mathbf{v}])^\sigma \Rightarrow (\mathbf{t}'[\mathbf{v}'])^\sigma$ where $\mathbf{t} \Rightarrow \mathbf{t}'$ and $\mathbf{v} \Rightarrow \mathbf{v}'$: We distinguish two cases according to the position of the $\mathcal{P}_c$-reduction step $(\mathbf{t}[\mathbf{v}])^\sigma \rightsquigarrow_{\mathcal{P}_c} \mathbf{a}$.
   - If it does not apply at the root then the property holds by induction.
   - If it applies at the root: then there is a substitution θ and a rule $l \rightarrow r$ in $\mathcal{P}_c$ such that $(\mathbf{t}[\mathbf{v}])^\sigma = l\theta$ and $\mathbf{a} \equiv r\theta$.
     - If all the $\rightsquigarrow_B$-redexes that are contracted in the reduction $(\mathbf{t}[\mathbf{v}])^\sigma \Rightarrow (\mathbf{t}'[\mathbf{v}'])^\sigma \equiv \mathbf{b}$ are under variables in l (that is, they are in θ) then these variables are uniquely identified (since l is left-linear) and we can therefore define a substitution θ' such that $\theta \Rightarrow \theta'$ and the diagram commutes: $(\mathbf{t}[\mathbf{v}])^\sigma \equiv l\theta \rightsquigarrow_{\mathcal{P}_c} r\theta \equiv \mathbf{a} \Rightarrow r\theta' \equiv \mathbf{c}$ and $(\mathbf{t}[\mathbf{v}])^\sigma \equiv l\theta \Rightarrow l\theta' \equiv \mathbf{b} \rightsquigarrow_{\mathcal{P}_c} r\theta' \equiv \mathbf{c}$.
     - If there is a $\rightsquigarrow_B$-redex at the root of $\mathbf{t}$ in $(\mathbf{t}[\mathbf{v}])^\sigma$ then we have a critical pair, between the $\mathcal{P}_c$-rule applied at the root of $(\mathbf{t}[\mathbf{v}])^\sigma$ and the *Beta* or *BetaE* rule applied at the root of $\mathbf{t}$. In that case the $\mathcal{P}_c$-rule applied must be *App*$_2$ (it cannot be *App*$_1$ or *App*$_3$ because of the restrictions: the function has a director ϵ). Then the diagram commutes as follows: we have

$$(\mathbf{t}[\mathbf{v}])^\sigma \equiv (((\lambda\mathbf{w})^\epsilon\,\mathbf{u})^{\searrow^{n+1}}[v^\epsilon])^{\swarrow^n} \rightsquigarrow_{App_2} ((\lambda\mathbf{w})^\epsilon\,(\mathbf{u}[v^\epsilon])^{\swarrow^n})^{\searrow^n} \equiv \mathbf{a}$$

and

$$(\mathbf{t}[\mathbf{v}])^\sigma \equiv (((\lambda\mathbf{w})^\epsilon\,\mathbf{u})^{\searrow^{n+1}}[v^\epsilon])^{\swarrow^n} \Rightarrow ((\mathbf{w}'[\mathbf{u}'])^{\searrow^{n+1}}[\mathbf{v}'])^{\swarrow^n} \equiv \mathbf{b}$$

where $\mathbf{w} \Rightarrow \mathbf{w}', \mathbf{u} \Rightarrow \mathbf{u}', \mathbf{v} \Rightarrow \mathbf{v}'$, hence

$$\mathbf{a} \Rightarrow (\mathbf{w}'[(\mathbf{u}'[\mathbf{v}'])^{\swarrow^n}])^{\searrow^n} \equiv \mathbf{c}$$

and

$$\mathbf{b} \rightsquigarrow_{Comp} (\mathbf{w}'[(\mathbf{u}'[\mathbf{v}'])^{\swarrow^n}])^{\searrow^n} \equiv \mathbf{c}.$$

5. In the other cases the property follows directly by induction.

This concludes the proof. □

Theorem 5 (Confluence) *λ_c is confluent.*

Proof Consequence of Proposition 11, Lemma 21, and Lemma 22, using Rosen's Lemma [34]. □

Theorem 6 (PSN) *λ_c preserves strong normalisation.*

Proof Consequence of Theorem 3 (PSN for λ_o) and Proposition 10. □

The restrictions imposed on the rewrite rules of λ_c still allow us to simulate the usual evaluation strategies, for closed λ-terms. We will show that call-by-value and call-by-name reductions to weak head normal form can be implemented in λ_c.

Call-by-name. We recall the evaluation rules for weak reduction in the call-by-name λ-calculus:

$$\frac{}{x \Downarrow x} \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M} \qquad \frac{M \Downarrow \lambda x.M' \quad M'\{x \mapsto N\} \Downarrow V}{(MN) \Downarrow V}$$

where $M \Downarrow V$ means that M evaluates to the (principal) weak head normal form V using the call-by-name strategy.

Theorem 7 (Call-by-name Evaluation) *If M is a closed λ-term and $M \Downarrow V$ by the call-by-name strategy, then $[\![M]\!] \leadsto_c^* [\![V]\!]$.*

Proof By induction on the derivation of $M \Downarrow V$.

If M is a weak head normal form then the theorem holds trivially. Otherwise, it is an application $(M\,N)$ and $[\![(M\,N)]\!] = ([\![M]\!]\,[\![N]\!])^\epsilon$ (since $\mathsf{fv}(M\,N) = \emptyset$). By induction, since $M \Downarrow \lambda x.M'$, we know that $[\![M]\!] \leadsto_c^* [\![\lambda x.M']\!]$. Now there are two alternatives.

- $[\![\lambda x.M']\!] = (\lambda [\![M']\!]_{[x]})^\epsilon$ if $x \in \mathsf{fv}(M')$.
  Then $((\lambda [\![M']\!]_{[x]})^\epsilon [\![N]\!])^\epsilon \leadsto_c ([\![M']\!]_{[x]}[[\![N]\!]])^\epsilon$. By Lemma 20, $([\![M']\!]_{[x]}[[\![N]\!]])^\epsilon$ $\leadsto_c^* [\![M'\{x \mapsto N\}]\!]$ and by induction the latter reduces to $[\![V]\!]$ as required.
- $[\![\lambda x.M']\!] = (\lambda^- [\![M']\!])^\epsilon$ if $x \notin \mathsf{fv}(M')$.
  Then $((\lambda^- [\![M']\!])^\epsilon [\![N]\!])^\epsilon \leadsto_c [\![M']\!] = [\![M'\{x \mapsto N\}]\!]$ and by induction the latter reduces to $[\![V]\!]$ as required. □

Call-by-value. We recall the evaluation rules for weak reduction in the call-by-value λ-calculus:

$$\frac{}{x \Downarrow x} \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M} \qquad \frac{M \Downarrow \lambda x.M' \quad N \Downarrow V' \quad M'\{x \mapsto V'\} \Downarrow V}{(MN) \Downarrow V}$$

where V' is a weak head normal form.

Theorem 8 (Call-by-value Evaluation) *If M is a closed λ-term and $M \Downarrow V$ by the call-by-value strategy, then $[\![M]\!] \leadsto_c^* [\![V]\!]$.*

Proof Similar to Theorem 7. In the case of an application $(M\,N)$ where $M \Downarrow \lambda x.M'$ and $N \Downarrow V'$, we get: $[\![(M\,N)]\!] = ([\![M]\!]\,[\![N]\!])^{\epsilon}$ (since $\mathsf{fv}(M\,N) = \emptyset$), and by induction, $[\![M]\!] \rightsquigarrow_c^* [\![\lambda x.M']\!]$, $[\![N]\!] \rightsquigarrow_c^* [\![V']\!]$. Again there are two cases to consider, which are identical to those in the proof of Theorem 7 above. □

This system has several advantages as a basis for an implementation of a λ-calculus evaluator:

- closed substitutions can be propagated through abstractions, which permits more sharing of work than in standard weak calculi,
- it forbids copying open terms, which ensures that we never duplicate a potential redex.

Moreover, the shape of the rewrite rules shows that in most cases we do not need to represent director strings as complex structures. For example, a director string of a substitution can be represented as a simple relative integer, because of Lemma 18. In the same spirit, the information on $|\rho|_l$ and $|\rho|_r$ can be maintained at each step. The system can thus be implemented in a realistic and efficient way, giving to each rewrite step a constant algorithmic cost. We will come back to this point in Section 6.

5 A Type System for Director Strings Calculi

For completeness, we present in this section a type system common to the above defined calculi and which enjoys the same kind of properties as the usual simply typed λ-calculus. We present the system with the syntax of λ_o, but it is of course also valid for λ_l and λ_c thanks to Proposition 8. The types are the simple types of the λ-calculus: type variables $A, B \ldots$ and function types $A \to B$. We also add a generic constant $\star^{\sigma}$ of type $\top$ (well-formed iff $\sigma = -^n$ for some n). Typing judgements will be of the form $\Gamma \vdash \mathbf{t} : A$ where the context Γ is an ordered list of types.

Definition 11 (Typed Terms) *Each term* $\mathbf{t}$ *is assigned a type in a given context:* $\Gamma \vdash \mathbf{t} : A$*, as given by the following set of rules. A term is typeable if there exists a context such that it is typeable in that context. The typing rules make use of two auxiliary relations defined below.*

$$\frac{\Gamma \overset{\sigma}{\sim} \emptyset}{\Gamma \vdash \star^{\sigma} : \top}\,(Cst) \qquad \frac{\Gamma \overset{\sigma}{\sim} A}{\Gamma \vdash \Box^{\sigma} : A}\,(Ax)$$

$$\frac{\Gamma', A \vdash \mathbf{t} : B \quad \Gamma \overset{\sigma}{\sim} \Gamma'}{\Gamma \vdash (\lambda \mathbf{t})^{\sigma} : A \to B}\,(Abs)$$

$$\frac{\Gamma_1 \vdash \mathbf{t} : A \to B \qquad \Gamma_2 \vdash \mathbf{u} : A \qquad \Gamma \overset{\sigma}{\prec} \begin{cases} \Gamma_1 \\ \Gamma_2 \end{cases}}{\Gamma \vdash (\mathbf{t}\ \mathbf{u})^{\sigma} : B}\,(App)$$

$$\frac{\Gamma_1\langle i/A\rangle \vdash \mathbf{t} : B \qquad \Gamma_2 \vdash \mathbf{u} : A \qquad \Gamma \xrightarrow{\sigma}\!\begin{cases}\Gamma_1\\ \Gamma_2\end{cases}}{\Gamma \vdash (\mathbf{t}[i/\mathbf{u}])^{\sigma} : B}\ (Sub)$$

where $\Gamma\langle i/A\rangle = A_1, \ldots, A_{i-1}, A, A_i, \ldots, A_n$ *if* $\Gamma = A_1, \ldots, A_n$.

Note that there are no structural rules, as the type of a given variable is always at a known location in the context.

In the typing rules we used two auxiliary relations. The first one ($\Gamma \stackrel{\sigma}{\sim} \Gamma'$) ensures that the right types are erased from Γ depending on σ. The second one ensures that a context adequately splits according to a director string, which is shown as $\Gamma \xrightarrow{\sigma}\!\begin{cases}\Gamma_1\\ \Gamma_2\end{cases}$.

$$\frac{}{\emptyset \stackrel{\epsilon}{\sim} \emptyset} \qquad \frac{\Gamma \stackrel{\sigma}{\sim} \Gamma'}{A, \Gamma \stackrel{\downarrow\sigma}{\sim} A, \Gamma'} \qquad \frac{\Gamma \stackrel{\sigma}{\sim} \Gamma'}{A, \Gamma \stackrel{-\sigma}{\sim} \Gamma'}$$

$$\frac{}{\emptyset \xrightarrow{\epsilon}\!\begin{cases}\emptyset\\ \emptyset\end{cases}} \qquad \frac{\Gamma \xrightarrow{\sigma}\!\begin{cases}\Gamma_1\\ \Gamma_2\end{cases}}{A, \Gamma \xrightarrow{\swarrow\!\searrow\sigma}\!\begin{cases}A, \Gamma_1\\ A, \Gamma_2\end{cases}} \qquad \frac{\Gamma \xrightarrow{\sigma}\!\begin{cases}\Gamma_1\\ \Gamma_2\end{cases}}{A, \Gamma \xrightarrow{-\sigma}\!\begin{cases}\Gamma_1\\ \Gamma_2\end{cases}}$$

$$\frac{\Gamma \xrightarrow{\sigma}\!\begin{cases}\Gamma_1\\ \Gamma_2\end{cases}}{A, \Gamma \xrightarrow{\searrow\sigma}\!\begin{cases}\Gamma_1\\ A, \Gamma_2\end{cases}} \qquad \frac{\Gamma \xrightarrow{\sigma}\!\begin{cases}\Gamma_1\\ \Gamma_2\end{cases}}{A, \Gamma \xrightarrow{\swarrow\sigma}\!\begin{cases}A, \Gamma_1\\ \Gamma_2\end{cases}}$$

Remark 5 If a term $\mathbf{t}$ is typeable in a context Γ, then the length of Γ is exactly the length of $\mathbf{t}$'s string. In particular, a term with empty director string (e.g. the compilation of a closed λ-term) is typeable if and only if it is typeable in the empty context.

Example 4 We give two small examples of type derivations in this system.

1. $(\lambda\Box^{\downarrow})^{\epsilon} : A \to A$

$$\frac{\overline{A \vdash \Box^{\downarrow} : A}}{\emptyset \vdash (\lambda\Box^{\downarrow})^{\epsilon} : A \to A}$$

2. $(\lambda(\lambda\Box^{\downarrow-})^{\downarrow})^{\epsilon} : A \to B \to A$

$$\frac{\dfrac{\overline{A, B \vdash \Box^{\downarrow-} : A}}{A \vdash (\lambda\Box^{\downarrow-})^{\downarrow} : B \to A}}{\emptyset \vdash (\lambda(\lambda\Box^{\downarrow-})^{\downarrow})^{\epsilon} : A \to B \to A}$$

A first useful property is the following:

Proposition 12 (Well-Formedness) *Typeable preterms are well-formed terms.*

Proof The functions $\dot{\sim}$ and $\dot{\prec}$ clearly enforce the constraints on well-formed terms (Definition 1). □

This type system corresponds to the λ-calculus simple type system (see for instance [6]), whose judgements we denote $\Gamma \vdash_\lambda M : A$, in the following sense:

Lemma 23

1. $x_1 : A_1, \ldots, x_n : A_n \vdash_\lambda M : A \Longrightarrow A_1, \ldots, A_n \vdash [\![M]\!]_{[x_1,\ldots,x_n]} : A$
2. $A_1, \ldots, A_n \vdash \mathbf{t} : A \Longrightarrow x_1 : A_1, \ldots, x_n : A_n \vdash_\lambda (\!|\mathbf{t}|\!)_{[x_1,\ldots,x_n]} : A$

Proof By straightforward induction on the type derivation. □

We also have the basic results expected from a simply typed calculus:

Theorem 9 (Subject Reduction) *If* $\Gamma \vdash \mathbf{t} : A$ *and* $\mathbf{t} \rightsquigarrow \mathbf{u}$ *then* $\Gamma \vdash \mathbf{u} : A$ *(for* $\rightsquigarrow \in \{\rightsquigarrow_o, \rightsquigarrow_l, \rightsquigarrow_c\}$*).*

Proof By Proposition 10, the cases for λ_l and λ_c follow from the general case of λ_o, which is proved by checking that each rule preserves type. We will only illustrate the proof on the case of *App*$_1$, the others being similar.

The situation is the following, where $\rho_i = \swarrow$, $j = |\rho_{1..i}|_l$, $\upsilon = \phi_l(\sigma, \rho_{\backslash i})$, $\tau = \psi_1(\sigma, \rho_{\backslash i})$ and $\Gamma = A_1, \ldots, A_n$:

$$\frac{\dfrac{\Delta_1 \vdash \mathbf{t} : A \to B \quad \Delta_2 \vdash \mathbf{u} : A \quad \Delta \overset{\rho}{\prec} \begin{cases}\Delta_1\\ \Delta_2\end{cases}}{\Delta = \Gamma_1\langle i/C\rangle \vdash (\mathbf{t}\ \mathbf{u})^\rho : B} \quad \Gamma_2 \vdash \mathbf{v} : C \quad \Gamma \overset{\sigma}{\prec} \begin{cases}\Gamma_1\\ \Gamma_2\end{cases}}{\Gamma \vdash ((\mathbf{t}\ \mathbf{u})^\rho[i/\mathbf{v}])^\sigma : B}$$

$$\downarrow$$

$$\frac{\dfrac{\Omega_1\langle j/C\rangle \vdash \mathbf{t} : A \to B \quad \Omega_2 \vdash \mathbf{v} : C \quad \Theta_1 \overset{\upsilon}{\prec} \begin{cases}\Omega_1\\ \Omega_2\end{cases}}{\Theta_1 \vdash (\mathbf{t}[j/\mathbf{v}])^\upsilon : A \to B} \quad \Theta_2 \vdash \mathbf{u} : A \quad \Gamma \overset{\tau}{\prec} \begin{cases}\Theta_1\\ \Theta_2\end{cases}}{\Gamma \vdash ((\mathbf{t}[j/\mathbf{v}])^\upsilon\ \mathbf{u})^\tau : B}$$

Then subject reduction is verified for this rule, provided that

$$\begin{cases} \Omega_1\langle j/C\rangle = \Delta_1 \\ \Omega_2 = \Gamma_2 \\ \Theta_2 = \Delta_2 \end{cases}$$

Let's write $L_\sigma = \{i \mid \sigma_i = \swarrow \text{ or } \swarrow\!\!\searrow\}$, $R_\sigma = \{i \mid \sigma_i = \searrow \text{ or } \swarrow\!\!\searrow\}$ and $[A]_I = A_{i_1}, \ldots, A_{i_m}$ if $I = \{i_1, \ldots, i_m\}$ and $(i_j)_j$ is increasing, so that $\Gamma = [A]_{\{1\ldots n\}}$. Then we have:

$$\begin{aligned} &\Gamma_1 = [A]_{L_\sigma} \\ &\Gamma_2 = [A]_{R_\sigma} \\ &\Delta_1 = [A]_{L_\sigma \cap L_\rho}\langle j/C\rangle \ \text{ since } \rho_i = \swarrow \text{ and } j = |\rho_{1..i}|_l \\ &\Delta_2 = [A]_{L_\sigma \cap R_\rho} \end{aligned}$$

and on the other hand:

$$\begin{array}{ll} \Theta_1 = [A]_{L_\tau} & \\ \Theta_2 = [A]_{R_\tau} = [A]_{L_\sigma \cap R_\rho} & \text{by definition of } \psi_1 \\ \Omega_1 = [A]_{L_\tau \cap L_\upsilon} = [A]_{L_\sigma \cap L_\rho} & \text{by definition of } \phi_l \\ \Omega_2 = [A]_{L_\tau \cap R_\upsilon} = [A]_{R_\sigma} & \text{again by definition of } \phi_l \end{array}$$

Hence subject reduction holds for *App*$_1$. The other cases are similar. □

Theorem 10 (Termination) *If $\Gamma \vdash \mathbf{t} : A$ then $\mathbf{t}$ is strongly normalisable (in λ_o, λ_l, λ_c).*

Proof Since typeable λ-terms are strongly normalisable, this is a direct consequence of Lemma 23 and PSN for the corresponding calculus. □

6 An Abstract Machine for Evaluation

In this section we will exhibit a strategy which makes use of the explicit information carried by director strings to efficiently reduce closed terms to weak head normal form. Efficiency is measured with respect to the total number of rewriting steps (not just β-steps) and we will give experimental comparisons in Section 8.

Notice that the syntax of director strings allows us to identify the moment when we have to copy a term, and we can reduce it before copying. In particular, we may want to use the most general rules, in order to be able to reduce a term to be copied to its full normal form, thus avoiding copying any redex. However, if we do so, then open substitutions are allowed in *App*$_3$ as well, which means that terms with free variables, i.e. potential redexes, might be copied. Our experimental tests confirmed that restricting just that rule to the closed case, we obtain a strategy very similar to closed reduction. This is because the propagation of an open substitution is very likely to be blocked by this restriction. Thus, the best strategy we found is based on the closed calculus, which is good news since it is also the simplest.

We cannot expect to reduce to full normal form with the closed rules, but some open terms can still be reduced. Thus, our strategy to compute the weak head normal form of a term $\mathbf{t}$ can be summarised as follows: we use the closed calculus, which allows some extra reductions under abstractions, but we stop the reduction as soon as we reach a weak head normal form of $\mathbf{t}$. The extra reductions are done only when we reduce a subterm to be copied, to share more work than in usual strategies.

To formally specify this strategy we will interleave one strategy which reduces under λ's and one which does not. We thus define three mutually recursive relations: $\to_w$, $\to_f$ and $\to$. The last one will be the strategy we want to exhibit. The relation $\to_w$ is defined in Figure 4, which, together with the other two relations below define the big-step operational semantics which the closed abstract machine implements.

The reduction relation $\to_w$ is used as a tool to define the other two and should not be interpreted on its own, as it does not treat the case of an abstraction. Notice

$$\frac{\mathbf{t} \to (\lambda \mathbf{r})^{\epsilon} \quad (\mathbf{r}[\mathbf{u}])^{\sigma} \to \mathbf{v}}{(\mathbf{t}\ \mathbf{u})^{\sigma} \to_w \mathbf{v}}\,(\mathit{Beta}) \qquad \frac{\mathbf{t} \to (\lambda^{-} \mathbf{v})^{\epsilon}}{(\mathbf{t}\ \mathbf{u})^{\epsilon} \to_w \mathbf{v}}\,(\mathit{BetaE})$$

$$\frac{}{\Box \to_w \Box}\,(\mathit{Ax}) \qquad \frac{\mathbf{t} \to \mathbf{v} \quad \mathbf{v} \neq (\lambda \mathbf{r})^{\epsilon} \wedge (\mathbf{v} \neq (\lambda^{-} \mathbf{r})^{\epsilon} \vee \rho \neq \epsilon)}{(\mathbf{t}\ u^{\rho})^{\sigma} \to_w (\mathbf{v}\ u^{\rho})^{\sigma}}\,(\mathit{Arg})$$

$$\frac{((\mathbf{t}[v^{\epsilon}])^{\swarrow |\rho|_l}\ \mathbf{u})^{\rho} \to \mathbf{w}}{((\mathbf{t}\ \mathbf{u})^{\swarrow \rho}[v^{\epsilon}])^{\sigma} \to_w \mathbf{w}}\,(\mathit{App1}) \qquad \frac{(\mathbf{t}\ (\mathbf{u}[v^{\epsilon}])^{\swarrow |\rho|_r})^{\rho} \to \mathbf{w}}{((\mathbf{t}\ \mathbf{u})^{\searrow \rho}[v^{\epsilon}])^{\sigma} \to_w \mathbf{w}}\,(\mathit{App2})$$

$$\frac{v^{\epsilon} \to_f \mathbf{v}' \quad ((\mathbf{t}[\mathbf{v}'])^{\swarrow |\rho|_l}\ (\mathbf{u}[\mathbf{v}'])^{\swarrow |\rho|_r})^{\rho} \to \mathbf{w}}{((\mathbf{t}\ \mathbf{u})^{\swarrow\!\!\searrow \rho}[v^{\epsilon}])^{\sigma} \to_w \mathbf{w}}\,(\mathit{App3})$$

$$\frac{(\lambda(\mathbf{t}[u^{\epsilon}])^{\swarrow |\rho|+1})^{\rho} \to \mathbf{v}}{((\lambda \mathbf{t})^{\downarrow \rho}[u^{\epsilon}])^{\sigma} \to_w \mathbf{v}}\,(\mathit{Lam}) \qquad \frac{(\lambda^{-}(\mathbf{t}[u^{\epsilon}])^{\swarrow |\rho|})^{\rho} \to \mathbf{v}}{((\lambda^{-} \mathbf{t})^{\downarrow \rho}[u^{\epsilon}])^{\sigma} \to_w \mathbf{v}}\,(\mathit{LamE})$$

$$\frac{\mathbf{v} \to \mathbf{w}}{(\Box[\mathbf{v}])^{\sigma} \to_w \mathbf{w}}\,(\mathit{Var}) \qquad \frac{(\mathbf{t}[(\mathbf{u}[v^{\epsilon}])^{\swarrow |\rho|}])^{\rho} \to \mathbf{w}}{((\mathbf{t}[\mathbf{u}])^{\searrow \rho}[v^{\epsilon}])^{\sigma} \to_w \mathbf{w}}\,(\mathit{Comp})$$

$$\frac{\mathbf{t} \to \mathbf{u} \quad (\mathbf{u}[v^{\rho}])^{\sigma} \to \mathbf{w} \quad \rho \neq \epsilon}{(\mathbf{t}[v^{\rho}])^{\sigma} \to_w \mathbf{w}}\,(\mathit{Subst})$$

Fig. 4 The relation $\to_w$

that the *(App$_3$)* rule calls the stronger reduction $\to_f$, which is defined by:

$$\frac{\mathbf{t} \to_w \mathbf{v}}{\mathbf{t} \to_f \mathbf{v}} \qquad \frac{\mathbf{t} \to_f \mathbf{v}}{(\lambda \mathbf{t})^{\sigma} \to_f (\lambda \mathbf{v})^{\sigma}} \qquad \frac{\mathbf{t} \to_f \mathbf{v}}{(\lambda^{-} \mathbf{t})^{\sigma} \to_f (\lambda^{-} \mathbf{v})^{\sigma}}$$

$\to_f$ is the relation which reduces under λ's (but not to full normal form).

Finally, $\to$ is the combination of the two other relations: we reduce to weak head normal form, but we reduce more the subterms that will be copied.

$$\frac{\mathbf{t} \to_w \mathbf{v}}{\mathbf{t} \to \mathbf{v}} \qquad \frac{}{(\lambda \mathbf{t})^{\sigma} \to (\lambda \mathbf{t})^{\sigma}} \qquad \frac{}{(\lambda^{-} \mathbf{t})^{\sigma} \to (\lambda^{-} \mathbf{t})^{\sigma}}$$

It may seem that the machine returns terms which are not weak head normal forms (cf. *(Arg)* rule). In fact, the theory ensures that this is not the case: starting from a closed term, the closed rules allow us to reach a weak head normal form (see Theorem 7). Nevertheless, the *(Arg)* rule may be applied in a reduction of a term to be copied, so it is indispensable.

The *(Subst)* and *(Comp)* rules call for a comment: the restriction on *(Subst)* (v^{ρ} open) forces *(Comp)* to be used as much as possible before reducing to the left of a closed substitution. Both intuition and experimentation confirm that this is indeed the right choice.

Notice that each rule either corresponds to a closed rule (see Definition 9) or focuses reduction on a subterm (corresponding to stack manipulations) and can indeed be implemented in constant time.

7 Reduction to Full Normal Form

We have presented so far a rather complex system to fully simulate β-reduction and simpler systems to reach only weak head normal form. If we were however interested in computing full normal forms, which is the case for many applications (e.g. partial evaluation or proof assistants), then we could of course use the general setting. But this is not really satisfactory (it does not provide any guidance towards an efficient strategy). On the other hand, we have an efficient strategy to reduce closed terms to weak head normal form. The idea then naturally arises to use our efficient weak evaluator to reach full normal form, in a way similar to [10].

The idea is to reduce a closed term to weak head normal form, then to distinguish the variable bound by the outermost abstraction in some way (to "freeze" it), so that we can still consider the term under the λ as closed, and recursively apply the same process to this subterm. There are several ways to distinguish those variables in the syntax. Below we present two natural alternatives.

7.1 With Names

If we choose to represent the frozen variables with names, we can avoid any complex manipulation of the director strings to keep track of the paths to these variables. As a result, we obtain a rather simple system because we can use the usual rules (for example the closed ones), where the frozen variables are just considered as constants and do not need any extra rule. Moreover readback into named λ-calculus is then performed at the same time.

Formally, we extend the syntax of terms in the following way, where σ ranges over strings, and x ranges over variable names:

$$\mathbf{t}, \mathbf{u} ::= \square \mid (\lambda \mathbf{t})^\sigma \mid (\lambda^- \mathbf{t})^\sigma \mid (\mathbf{t}\ \mathbf{u})^\sigma \mid (\mathbf{t}[\mathbf{u}])^\sigma \mid x \mid \lambda' x.\mathbf{t}$$

that is, we add named variables, whose implicit director string is ϵ, and named abstraction binding written as $\lambda' x.\mathbf{t}$. We do not write any director string for this abstraction, since we will always consider closed terms of this form.

Using a weak evaluation relation $\Downarrow_w$, we can then define reduction to full normal form $\Downarrow_f$.

$$\frac{\mathbf{t} \Downarrow_w (\lambda \mathbf{t}')^\epsilon \qquad (\mathbf{t}'[x])^\epsilon \Downarrow_f \mathbf{t}'' \qquad x \text{ fresh}}{\mathbf{t} \Downarrow_f \lambda' x.\mathbf{t}''}$$

$$\frac{\mathbf{t} \Downarrow_w (\lambda^- \mathbf{t}')^\epsilon \qquad \mathbf{t}' \Downarrow_f \mathbf{t}'' \qquad x \text{ fresh}}{\mathbf{t} \Downarrow_f \lambda' x.\mathbf{t}''}$$

$$\frac{\mathbf{t} \Downarrow_w x \qquad (x \text{ variable})}{\mathbf{t} \Downarrow_f x}$$

$$\frac{\mathbf{t} \Downarrow_w (\mathbf{u}\ \mathbf{v})^\epsilon \qquad \mathbf{u} \Downarrow_f \mathbf{u}' \qquad \mathbf{v} \Downarrow_f \mathbf{v}'}{\mathbf{t} \Downarrow_f (\mathbf{u}'\ \mathbf{v}')^\epsilon}$$

Notice that the last rule is used since we are now in a calculus with constants (the named variables), and the weak head normal form of a term may be an application (e.g. $(x\ \mathbf{t})^\epsilon$ where x is a named variable).

Proposition 13 (Correctness) *Let's write* $\tilde{\mathbf{u}}$ *for the readback of a reduced term* $\mathbf{u}$*, so* $\tilde{\cdot}$ *just transforms* λ' *to* λ *and erases the remaining director strings. Then, if* M *is a closed* λ*-term:*

$$[\![M]\!] \Downarrow_f \mathbf{u} \iff M \to_\beta^* \tilde{\mathbf{u}} \text{ irreducible.}$$

Proof Assuming correctness of $\Downarrow_w$, the proof is easily adapted from [10] or the more recent [20]. □

If we want to reduce an open term $\mathbf{t} = t^\sigma$ with $|\sigma| = n$, we first take n fresh variable names $x_1, \ldots, x_n$ and start the reduction from:

$$((\ldots((\mathbf{t}[x_1])^{\surd^{n-1}}[x_2])^{\surd^{n-2}} \ldots [x_{n-1}])^{\surd}[x_n])^\epsilon$$

The reduction to normal form follows exactly the same strategy as the corresponding weak reduction. Thus, for terms for which full and weak head normal forms are the same, the two processes need the same numbers of β and total steps. In particular, this strategy is much more efficient than the usual naive one.

Although we now have to deal with names and fresh variables during reduction (which was not the case for reduction to weak head normal form), we still do not have to deal with name capture and α-conversion. Also, the readback is now simplified.

7.2 With Directors

The previous strategy performs readback at the same time as computation of the normal form, which may, or may not, be wished. We can however implement a similar idea using only directors, obtaining a result in this syntax. We just need a way to distinguish between usual variables, and frozen ones, which correspond to an abstraction outside of the term we actually want to reduce to weak head normal form. This can be done in a quite obvious way: by introducing a new kind of directors corresponding to these frozen variables. However, the frozen part of director strings may be of any form, so we need to use the general rules on this part. From an algorithmic point of view, this means that the cost of a reduction step may be at most linear in the depth of λ-abstractions in the resulting normal form, which still seems reasonable. We do not go further on this point as the interesting ideas have already been exposed in the previous section.

8 Experimental Results

One of the main motivations for this work is the desire to find more efficient implementations of the λ-calculus. This interest is not simply to find minor tweaks

(or optimisations) of existing systems, but rather to attempt a fresh start to uncover new strategies, and new machinery, which give asymptotic improvements over standard techniques in use in implementations of functional languages. Consequently it is essential that our strategies are implemented so that the ideas of this paper can be backed up with some experimental evidence and compared to existing evaluators.

It is difficult, if not impossible, to find a relevant measure to compare different implementations. This point is even more pertinent when we are comparing *prototype* implementations. Nevertheless, what we are able to do is to identify atomic reduction steps for several evaluators. Although the algorithmic cost of a single step may vary for each evaluator, the respective speed-up can still be examined. The following benchmarks suggest that the strategies developed in the paper behave better than standard reduction strategies, and moreover offer some surprising statistics with respect to some of the very best evaluators known to date.

We have two set of examples: one that belongs to $\lambda\mathsf{I}$ in order to avoid the problem of erasing, and a small set of $\lambda\mathsf{K}$ terms to illustrate some specific behaviours. The Church numerals are an excellent means to produce a panel of large λ-terms. We recall that Church numerals are of the form $n = \lambda f.\lambda x.f^n\,x$ and that application corresponds to exponentiation: $n\,m \equiv m^n$. We apply Church numerals in our examples to $\mathsf{I}\,\mathsf{I}$, where $\mathsf{I} = \lambda x.x$, which is sufficient to force reduction to full normal form, and allows us to compare weak and full reducers. We also use the combinators $\mathsf{K} = \lambda x.\lambda y.x$ and $\mathsf{M} = \lambda x.\lambda y.(\mathsf{K}\,\mathsf{I}\,x)(\mathsf{K}\,\mathsf{I}\,x\,y)$.

We compare our machines with standard call-by-value and call-by-name evaluators, and in addition to the optimal interpreter of Asperti et al. (BOHM [3]). The latter result provides a comparison with the best known evaluator for such terms. We show the total number of steps of these evaluators (including stack manipulations). We show the number of β-reductions between round brackets, thus the number shown for BOHM is the minimum number of β-reductions possible. The results for the machines that reduce to full normal form are not shown, as they are the same as those of the underlying weak strategies on these examples.

Term	closed	CBV	CBN	BOHM
2 2 I I	61(9)	78(11)	76(12)	40(9)
2 2 2 I I	140(19)	362(42)	471(60)	93(16)
5 5 I I	217(33)	29 723(3 913)	31 250(4 689)	208(33)
5 2 2 I I	832(109)	-	-	847(31)
2 2 2 2 2 I I	1 507 714(196 655)	-	-	1 074 037 060(61)
M (5 5 I I) I	266(42)	41(8)	31(8)	22(8)
K I (5 5 I I)	7(2)	29731(3915)	7(2)	4(2)

To put some of these results into perspective, we remark that the actual time taken to compute, for example, 5 2 2 I I using OCaml is around 5 minutes, and around 3 minutes using Standard ML (both implementing a variant of call-by-value). The results for both closed reduction and BOHM are essentially instantaneous.

The main point that we want to make with the above table is that closed reduction, a simple implementation of the λ-calculus, clearly out performs traditional strategies, such as call-by-value, and moreover is a serious competitor to highly sophisticated implementations, such as BOHM. The comparison with call-by-value and call-by-name shows that allowing reductions under abstractions, which is especially easy with director strings compared to usual calculi, is crucial for both sharing and efficiency.

The interesting point is the comparison on large terms. The results show that our machine is able to reduce larger terms than the other machines, and the larger the term, the better is our machine compared to the others. This hints that it allows for a high degree of sharing (because the larger the term, the more possible sharing). The last example of the first set shows that our machine eventually explodes in terms of number of β-reductions compared to the optimal one but outperforms BOHM in total number of steps, which is our notion of efficiency.

Besides this classical benchmark, we give two $\lambda\mathsf{K}$ terms to illustrate the following points:

- The closed strategy may perform more work than necessary, when every copy of a term will be erased. The occurrence of such terms in practical situations is however questionable. One could also combine the closed strategy with a call-by-need strategy similarly to [16] to avoid this problem.
- Except for the previous situation, we may avoid doing useless work, as opposed to call-by-value for instance, i.e. we also have some of the advantages of call-by-name.

9 Conclusion

We have presented a name-free syntax to represent terms of the λ-calculus with explicit substitutions, in a way that follows the usual intuitions about the operational semantics of the propagation of substitutions. We have given a general calculus on director strings which can fully simulate the λ-calculus, with rather complicated rules. We then described an intermediate calculus, the local open calculus, with very simple rules and still allowing open substitutions to traverse abstractions without global rewriting. Finally, we derived the closed reduction calculus of [18], which internalises the conditions on the original system [17].

These calculi were used as a basis to describe and implement abstract machines for weak and strong reduction (the latter was an open problem for director strings). Efficiency was our main motivation, and we found in practice that these machines are quite efficient on large terms and allow for a high degree of sharing. In particular, they quite favourably compare to standard evaluators, which suggests that more efficient implementations of functional languages and λ-calculus based proof assistants are still possible.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

2. Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 233–246. ACM Press, 1995.
3. A. Asperti, C. Giovanetti, and A. Naletto. The Bologna optimal higher-order machine. *Journal of Functional Programming*, 6(6):763–810, 1996.
4. A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.
5. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.
6. H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, 1992.
7. Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. Lambda-upsilon, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, 1996.
8. Z. Benaissa, K. H. Rose, and P. Lescanne. Modeling sharing and recursion for weak reduction strategies using explicit substitution. In H. Kuchen and D. Swierstra, editors, *8th PLILP—Symposium on Programming Language Implementation and Logic Programming*, pages 393–407, Aachen, Germany, 1996.
9. R. Bloo and H. Geuvers. Explicit substitution: on the edge of strong normalization. *Theoretical Computer Science*, 211(1):375–395, 1999.
10. P. Crégut. An abstract machine for lambda-terms normalization. In *Lisp and Functional Programming 1990*, pages 333–340. ACM Press, 1990.
11. P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.
12. R. David and B. Guillaume. A λ-calculus with explicit weakening and explicit substitution. *Mathematical Structures in Computer Science*, 11(1):169–206, 2001.
13. N. G. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.
14. N. G. de Bruijn. A namefree lambda calculus with facilities for internal definition of expressions and segments. Technical Report T.H.-Report 78-WSK-03, Department of Mathematics, Eindhoven University of Technology, 1978.
15. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Formal Methods and Semantics*, volume B. North-Holland, 1989.
16. R. Ennals and S. P. Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In C. Norris and J. James B. Fenwick, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP-03)*, volume 38, 9 of *ACM SIGPLAN Notices*, pages 287–298. ACM Press, 2003.
17. M. Fernández and I. Mackie. Closed reductions in the λ-calculus. In J. Flum and M. Rodríguez-Artalejo, editors, *Proceedings of Computer Science Logic (CSL'99)*, number 1683 in Lecture Notes in Computer Sciences, pages 220–234. Springer-Verlag, 1999.
18. M. Fernández and I. Mackie. Director strings and explicit substitutions. WESTAPP'01, Utrecht, 2001.
19. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–26, Albequerque, New Mexico, 1992.
20. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings of ICFP'02, Pittsburgh, Pennsylvania, USA*, 2002.

21. T. Hardin, L. Maranget, and B. Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–176, 1998.
22. J. Kennaway and M. Sleep. Director strings as combinators. *ACM Transactions on Programming Languages and Systems*, 10(4):602–626, 1988.
23. J. Lamping. An algorithm for optimal lambda calculus reductions. In *Proceedings 17 th ACM Symposium on Principles of Programming Languages*, pages 16–30, 1990.
24. F. Lang. *Modèles de la β-réduction pour les implantations*. PhD thesis, École Normale Supérieure de Lyon, 1998.
25. J. L. Lawall and H. G. Mairson. Optimality and inefficiency: What isn't a cost model of the lambda calculus? In *International Conference on Functional Programming*, pages 92–101, 1996.
26. P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$: a journey through calculi of explicit substitutions. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages (POPL'94)*. ACM Press, 1994.
27. P. Lescanne and J. Rouyer-Degli. The calculus of explicit substitutions lambda-upsilon. Technical Report RR-2222, INRIA, 1995.
28. J.-J. Lévy. Optimal reductions in the lambda-calculus. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.
29. P.-A. Melliès. Typed lambda-calculi with explicit substitutions may not terminate. In *Proceedings of the 2nd International Conference on Typed Lambda Calculi and Applications*, number 902 in Lecture Notes in Computer Science, pages 328–334. Springer-Verlag, 1995.
30. G. Nadathur. A fine-grained notation for lambda terms and its use in intensional operations. *Journal of Functional and Logic Programming*, 1999(2), 1999.
31. G. Nadathur. The suspension notation for lambda terms and its use in metalanguage implementations. In R. de Queiroz, L. C. Pereira, and E. H. Haeusler, editors, *Electronic Notes in Theoretical Computer Science*, volume 67. Elsevier, 2002.
32. M. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.
33. K. Rose. Explicit substitution - tutorial and survey, 1996. Lecture Series LS-96-3, BRICS, Dept. of Computer Science, University of Aarhus, Denmark.
34. B. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, 1973.
35. F.-R. Sinot, M. Fernández, and I. Mackie. Efficient reductions with director strings. In R. Nieuwenhuis, editor, *Proceedings of Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.
36. N. Yoshida. Optimal reduction in weak lambda-calculus with shared environments. *Journal of Computer Software*, 11(6):3–18, 1994.