Closed reduction: explicit substitutions without \$\alpha\$-conversion

M. FERN&Aacute;NDEZ, I. MACKIE and F-R. SINOT

Mathematical Structures in Computer Science / Volume 15 / Issue 02 / April 2005, pp 343 - 381
DOI: 10.1017/S0960129504004633, Published online: 14 March 2005

Link to this article: http://journals.cambridge.org/abstract_S0960129504004633

How to cite this article:
M. FERNÁNDEZ, I. MACKIE and F-R. SINOT (2005). Closed reduction: explicit substitutions without $\alpha$-conversion. Mathematical Structures in Computer Science, 15, pp 343-381 doi:10.1017/S0960129504004633

Math. Struct. in Comp. Science (2005), *vol.* 15, *pp.* 343–381. © 2005 Cambridge University Press
doi:10.1017/S0960129504004633 Printed in the United Kingdom

Closed reduction: explicit substitutions without α-conversion

M. FERNÁNDEZ[†], I. MACKIE[†] and F-R. SINOT[†‡]

[†]*Department of Computer Science, King's College London*
Strand, London, WC2R 2LS U.K.
Email: maribel,ian,francois@dcs.kcl.ac.uk

[‡]*LIX, École Polytechnique, 91128 Palaiseau cedex, France.*

Received 26 June 2003; revised 9 July 2004

Starting from the λ-calculus with names, we develop a family of calculi with explicit substitutions that overcome the usual syntactical problems of substitution. The key idea is that only closed substitutions can be moved through certain constructs. This gives a weak form of reduction, called *closed reduction*, which is rich enough to capture both the call-by-value and call-by-name evaluation strategies in the λ-calculus. Moreover, since substitutions can move through abstractions and reductions are allowed under abstractions (if certain conditions hold), closed reduction naturally provides an efficient notion of reduction with a high degree of sharing and low overheads. We present a family of abstract machines for closed reduction. Our benchmarks show that closed reduction performs better than all standard weak strategies, and its low overheads make it more efficient than optimal reduction in many cases.

1. Introduction

The λ-calculus is the foundational model of functional programming languages. The β-rule describes the application of a function (abstraction) to its argument, substituting the formal parameter of the function by the actual argument: $(\lambda x.t)u \to_\beta t[x := u]$. The notation $t[x := u]$ means t where all free occurrences of the variable x are replaced by u. One of the key problems of studying β-reduction is that this substitution operation is defined outside the rewrite system. We identify two important points:

— substitution must preserve the meaning of the term, which consequently implies that variable renamings (α-conversions) are required to avoid variable capture, introducing *additional* substitutions;

— the actual computational work in performing a β-reduction step is in the substitution process: terms may be copied and erased during substitution propagation, and, moreover, different evaluation strategies are possible for this propagation.

To obtain a better understanding of these issues, λ-calculi with explicit substitutions have been proposed, which place the meta-operation of substitution at the same level as β-reduction. In recent years there has been a proliferation of variants of the λ-calculus with explicit substitutions, starting with the $\lambda\sigma$-calculus (Abadi *et al.* 1991). In most studies of explicit substitutions the following properties are considered foremost:

— simulation of β-reduction in full generality;

— preservation of strong normalisation;

— confluence on open or on ground terms.

Such properties are clearly desirable for a complete study of β-reduction, and are also essential for some applications, see, for instance, Dowek *et al.* (2001). However, if we are only interested in using the λ-calculus for *implementation* based studies, β-reduction does not need to be simulated in full generality, and, in particular, poor strategies should be eliminated. Additionally, confluence on open terms is no longer an essential property. However, preservation of strong normalisation does, of course, remain a useful property. In this paper we study λ-calculi with explicit substitutions specifically for the implementation of the λ-calculus in an efficient way: substitutions are needed to express computational aspects of β-reduction such as copying and erasing of terms.

The λ-calculus with *names* is often considered an inconvenient starting point for the study of explicit substitutions because variables require renaming during reduction, and consequently new names need to be generated dynamically (α-conversion is costly). It is because of this 'defect' that most explicit substitution calculi use de Bruijn notation (de Bruijn 1972). Thus, explicit substitution calculi have become synonymous with de Bruijn notation (although there are exceptions, see, for instance, Rose (1996)). The main advantage of this notation is that it avoids generating new variable names, and has also been claimed to be closer to an implementation. However, new problems are introduced:

— Shifting operations (incrementing numbers) are needed, which is essentially encoding the generation of new names.

— Substitutions are often represented as *lists*, which implies that new operations are needed to manipulate this structure.

— Properties such as confluence and preservation of strong normalisation cannot be easily obtained (see, for instance, David and Guillaume (2001)).

It is well known that the problems above can be avoided simply by not allowing substitution through abstraction. Almost all evaluators for the λ-calculus are based on weak reduction, which is characterised precisely by rejecting reduction under an abstraction. As a consequence, the most expensive and delicate part of the substitution process is removed from the system. However, this is achieved at a price because terms with blocked substitutions (closures) may be copied, which can cause redexes (and potential redexes) to be duplicated. As Çağman and Hindley (Çağman and Hindley 1998) point out, α-conversion can also be avoided if β-redexes are closed (that is, $(\lambda x.t)u$ does not contain free variables). However, this is also a strong restriction, and has similar drawbacks.

Building on these ideas, in this paper we propose a family of calculi with *names* that provide a basis for implementations of functional programming languages and overcome the problems mentioned above. A key advantage of using names is that our calculi are closer to our intuitions about *what* happens during the propagation of substitutions, and we do not introduce any of the problems that are generally associated with alternative notations (for instance manipulating indexes). The main features of the calculi that we will define are:

— the absence of α-conversion;
— a tight control over the substitution process to avoid useless computations;
— a strategy of reduction with a good degree of computation sharing.

Our point of departure is a rewrite system that simply adds to the β-rule the meta-level definition of substitution as a collection of conditional rewrite rules. The usual definition of substitution is, however, very naive, at least from a computing perspective: substitutions are pushed from the root of the term to *all* the leaves. But why would one push substitutions into branches of the term where there are no occurrences of the free variable to replace? Indeed, this is the essence of the well-known counter-example to the preservation of strong normalisation in $\lambda\sigma$ (Melliès 1995). We will avoid this problem by also making explicit the copying and erasing phases of substitution, for which we were inspired by various calculi for linear logic (see, for instance, Abramsky (1993)). This will also allow us to control (and avoid) the issues of duplicating and erasing free variables in the substitution process. We will eliminate variable clash and capture problems by permitting certain reductions to take place only when a subterm is closed. This removes the necessity of generating fresh variable names during reduction, which overcomes the main objections to explicit substitution calculi with names.

To summarise, the key aspect of the reduction strategy used in this paper, which we call *closed reduction*, is that it allows the following to be carried out easily:

— reduction under, and substitution through, an abstraction;
— avoidance of duplication of free variables;
— clean garbage collection.

There are several ways of obtaining a closed reduction strategy, yielding different calculi of explicit substitutions with explicit resource management. The first, which we call λ_{ca}, is characterised by the fact that in the β-rule the argument must be closed. Since this is the only rule that creates a substitution, this implies that all the substitutions are closed. Although restrictive, this calculus is adequate for the implementation of functional programs, and enjoys properties such as confluence, termination of the substitution rules, and preservation of strong normalisation.

From the point of view of the sharing of computations, λ_{ca} is better than standard weak strategies but is not yet completely satisfactory. We will define a second calculus, called λ_{cf}, where the key idea is that we can create open substitutions but they must be closed when they pass through an abstraction. Like λ_{ca}, this calculus is presented as a rewrite system working on ground terms (we do not rewrite terms with metavariables). Nevertheless, this calculus gives partial answers to the problems raised by Muñoz (1996), and is adequate for the evaluation of functional programs. We show that λ_{cf} is confluent, preserves strong normalisation, is terminating on typed terms, and can simulate the usual evaluation strategies: *call-by-value* and *call-by-name*. We also compare λ_{ca}, λ_{cf}, and variants such as λ_{cl}, which requires closed redexes in the β-rule as in Çağman and Hindley (1998), and λ_{caf}, which generalises both λ_{ca} and λ_{cf}.

We use this explicit substitution syntax to define a family of abstract machines for closed reduction using a structured operational semantics for the calculus. These machines have been implemented, and the benchmarks indicate that the level of sharing obtained is close to optimal reduction, with considerably less overhead.

Related work

Our work builds on the use of explicit substitutions for controlling the substitution process in the λ-calculus, with an emphasis on implementation, for instance, the call-by-need λ-calculus of Ariola *et al.* (1995) and calculi with shared environments (Yoshida 1994). Also oriented towards implementation are Hardin *et al.* (1998), Rose (1996) and Nadathur (1999). In the latter, information about closedness is used to define conditional reductions, although with different motivations and in a framework quite different from ours (with a *suspension notation* using de Bruijn indices). Our notion of closed reduction is inspired by a strategy for cut-elimination in linear logic, used in a proof of soundness of the geometry of interaction by Girard (Girard 1989). This paper extends and generalises Fernández and Mackie (1999).

Overview

In the next section we motivate our work by looking at the problematic issues in explicit substitution calculi. In Section 3.1 we introduce λ_{c}-terms, and in Sections 3.2 and 3.3 we present two calculi corresponding to two strategies for closed reduction: λ_{ca} (for *closed argument*) and λ_{cf} (for *closed function*). The relation to β-reduction and strategies in the λ-calculus is given in Section 3.4. In Section 3.5 we present two other variants of closed reduction, and discuss our choices in Section 3.6. In Section 4 we give a type system and show subject reduction and termination. Implementations are discussed in Section 5, where we define and compare abstract machines. We conclude the paper in Section 6.

2. Background and motivation

Our starting point is the λ-calculus with β-reduction, where the usual meta-operation of substitution is replaced by a collection of conditional rewrite rules inspired by the original definition of substitution given by Church. We use explicit substitution calculi with names throughout, but most of what we have to say can be formulated in de Bruijn notation. We assume knowledge of the untyped λ-calculus (Barendregt 1984). We use $t[x := u]$ to denote the usual implicit substitution. Λ denotes the set of λ-terms. Note that we do not consider λ-terms modulo α-conversion, we will deal with that issue explicitly.

Definition 2.1 (λ-calculus with explicit substitutions). Let x range over variables, $\star$ denote an arbitrary constant, and t, u range over terms (that is, labelled trees) defined by

$$t, u \ ::= x \mid \star \mid \lambda x.t \mid t\,u \mid t[u/x]$$

We assume application associates to the left

$$tuv = (tu)v,$$

Table 1. *λ-calculus with explicit substitutions.*

Name	Reduction			Condition
Beta	$(\lambda x.t)v$	$\to$	$t[v/x]$	
Cons	$\star[v/x]$	$\to$	$\star$	
Var1	$x[v/x]$	$\to$	v	
Var2	$y[v/x]$	$\to$	y	$x \neq y$
App	$(tu)[v/x]$	$\to$	$(t[v/x])(u[v/x])$	
Lam1	$(\lambda x.t)[v/x]$	$\to$	$(\lambda x.t)$	
Lam2	$(\lambda y.t)[v/x]$	$\to$	$\lambda y.t[v/x]$	$x \notin \mathsf{fv}(t) \vee y \notin \mathsf{fv}(v), x \neq y$
Lam3	$(\lambda y.t)[v/x]$	$\to$	$\lambda z.t[z/y][v/x]$	$x \in \mathsf{fv}(t), y \in \mathsf{fv}(v), x \neq y, z$ fresh
Comp	$t[u/x][v/y]$	$\to$	$t[v/y][u[v/y]/x]$	$x \notin \mathsf{fv}(v)$

and adopt a similar convention for substitutions

$$t[u/x][v/y] = (t[u/x])[v/y].$$

We also abbreviate $\lambda x.\lambda y.t$ to $\lambda xy.t$. The notion of free variables of a term t (notation $\mathsf{fv}(t)$) is the usual one, where $\mathsf{fv}(t[u/x]) = \mathsf{fv}((\lambda x.t)u)$. Table 1 gives the conditional rewrite rules for this calculus. The condition 'z fresh' means that z is a variable not occurring in the left-hand side.

Table 1 in fact defines a set of reduction schemes involving metavariables t, u, v ranging over terms. x, y, z may be viewed as metavariables ranging over λ-calculus variables, or as constants in a rewrite system with infinite rules (for example, rule *Var1* applies to any λ-calculus variable with a substitution). By instantiation, these rule schemes define a rewrite relation over λ-terms, which is closed under context. Reduction can take place in *any* context, in particular, under abstractions and within substitutions:

$$\frac{t \to t'}{tu \to t'u} \qquad \frac{u \to u'}{tu \to tu'} \qquad \frac{t \to t'}{\lambda x.t \to \lambda x.t'} \qquad \frac{t \to t'}{t[u/x] \to t'[u/x]} \qquad \frac{u \to u'}{t[u/x] \to t[u'/x]}$$

There are variants of the above calculus, for instance, if we start with a term with all bound variables chosen to be different, we can alternatively perform α-conversion in the *App* and *Comp* rules, when substitutions are copied[†]. However, we prefer to take the above, more familiar, definition as a starting point.

Analysing this calculus from an *implementation* perspective, we can identify some defects:

— Substitutions are propagated exhaustively. In the rules *App* and *Comp* the substitution v is copied, even if the variable x does not occur free in either of the subterms.

[†] Consequently, in the linear λ-calculus where erasing and, particularly, copying are not allowed, there is no need for α-conversion if we start with an initial term with all bound variables different.

— Following on from the previous point, *Var2* and *Lam1* erase the term v when it is known that the substitution is not needed.

— α-conversion is necessary in the *Lam3* rule, which consequently creates the additional substitution $t[z/y]$, which will also be propagated exhaustively. Moreover, a fresh variable must be generated by some kind of external process.

— The *Comp* rule can be applied *ad infinitum*, thus there is no hope of proving general termination results for this calculus.

We now look at ways of reducing these overheads, first by focusing on eliminating α-conversion, then on substitution propagation.

2.1. *α-conversion*

There are several heavy-handed remedies that can be prescribed to avoid α-conversion. For closed terms, it is well known that α-conversion is not required if we forbid reduction under abstraction (which is sufficient to perform reduction to weak head normal form, WHNF). In explicit substitution calculi, this form of *weak reduction* also includes forbidding substitution propagation through an abstraction (thus the *Lam* rules above are removed from the system). However, we observe that this restriction is too strong. For instance, there would be no problems if we were to reduce the head redex in $t = \lambda x.(\lambda y.y)x$, but this redex will be duplicated when reducing the following application weakly:

$$(\lambda x.xx)t \to^* (\lambda x.(\lambda y.y)x)(\lambda x.(\lambda y.y)x).$$

Even worse, not just actual redexes but potential redexes may also be copied: for example, in the previous application take $t = (\lambda z.xyz)[\lambda x.x/x][\lambda y.y/y]$. Again, no α-conversion is necessary in this case.

This motivates our first improvement. To avoid such duplications, we will allow reduction under abstractions, and propagation of substitutions through abstractions, but only when this does not introduce α-conversion. There are a number of possible solutions that exist, which are less heavy-handed than weak reduction to WHNF, for instance:

— Closed redexes (Çağman and Hindley 1998). If the *Beta* rule $(\lambda x.t)u \to t[u/x]$ is restricted to apply only when $\mathsf{fv}((\lambda x.t)u) = \emptyset$, there is no need for α-conversion: all substitutions created by this rule will be closed. This calculus allows more reductions than standard weak reduction on closed terms, and has many useful properties.

— Eliminate rule *Lam3*. As this is the only rule that requires α-conversion, we can remove it from the system. Although this is the least restrictive system, it requires two explicit membership tests (in rule *Lam2*).

The question we address in this paper is whether we can improve on the above solutions by providing a middle ground between them. The two calculi we will present can be simply understood as finding a compromise that offers as much reduction as possible, without the cost of α-conversion.

2.2. *Propagation of substitutions*

Pushing a substitution $[v/x]$ through an application naively copies the term v (and thus also any redexes occurring in v). This could be regarded as the key source of inefficiency at the syntactic level. Consider a term such as $(xt_1t_2\cdots t_n)[v/x]$, where the only occurrence of x is the one shown. For the substitution process to complete, with the rules given, we require $n+1$ copies of the substitution to be made, plus any additional copies depending on the structure of the terms t_i. Then all but one of the copies made will be discarded. However, there is one unique occurrence of x in this term, so no duplication (or sharing even) needs to be done at all. This suggests our next refinement of the calculus, by replacing the rule for substitution through an application with the set of rules

$$\begin{array}{lcll} (tu)[v/x] & \to & (t[v/x])u & (x \in \mathsf{fv}(t), x \notin \mathsf{fv}(u)) \\ (tu)[v/x] & \to & t(u[v/x]) & (x \notin \mathsf{fv}(t), x \in \mathsf{fv}(u)) \end{array}$$

which correspond to the cases where x occurs linearly in the term. We then need two additional rules to capture the crucial cases when x does not occur at all, and several times respectively[†]:

$$\begin{array}{lcll} (tu)[v/x] & \to & tu & (x \notin \mathsf{fv}(tu)) \\ (tu)[v/x] & \to & (t[v/x])(u[v/x]) & (x \in \mathsf{fv}(t), x \in \mathsf{fv}(u)) \end{array}$$

Factoring out the linear and the non-linear occurrences of variables in terms can be done at the syntactic level, avoiding some of the conditions on the rules. We thus extend the calculus with the following two constructs and reduction rules:

— **Erase** $(\epsilon_x.t)$. If $x \notin \mathsf{fv}(t)$, it can be made to appear explicitly using the erase construct, where x is a fresh variable. The associated reduction rule preserves this notation:

$$(\epsilon_x.t)[v/x] \to \epsilon_{x_1}.\epsilon_{x_2}.\cdots\epsilon_{x_n}.t$$

where $\{x_1,\ldots,x_n\} = \mathsf{fv}(v)$. Note, however, that if v is closed, the rule becomes simply

$$(\epsilon_x.t)[v/x] \to t$$

By imposing the condition that $\mathsf{fv}(v) = \varnothing$, erasing (garbage collection) becomes a simpler, all-in-one operation.

— **Copy** $(\delta_x^{y,z}.u)$. If x occurs twice in a term, we rename one occurrence to y and the other to z (y and z fresh) and use the copy construct, with the following rule:

$$(\delta_x^{y,z}.t)[v/x] \to \delta_{\vec{x}}^{\vec{y},\vec{z}}.t[v[\vec{y}/\vec{x}]/y][v[\vec{z}/\vec{x}]/z]$$

where $\vec{x} = \mathsf{fv}(v)$, and $\vec{y}$ and $\vec{z}$ are assumed to be fresh. Now we have clearly introduced

[†] These four alternatives for pushing a substitution through an application can also be understood by appealing to the combinators **S**, **B**, **C**, **K**.

a wealth of α-conversions into this calculus, but again, note that if v is closed, the rule becomes simply

$$(\delta_x^{y,z}.t)[v/x] \to t[v/y][v/z]$$

Not only are the above reduction rules much simpler in the case where the substitutions are closed, but it is also a key to efficiency. If terms that contain free variables are copied, potential redexes might also be duplicated. Thus our design decision is to only copy terms that do not contain free variables and do not need any α-conversion.

Note that we now no longer need the rules *Var2* and *Lam1* since no substitution will ever reach a term unless it contains the free variable being substituted. Substitutions are now guided through the term by the copying construct and the *App* rule to the correct destination(s).

With the addition of the explicit constructs for erasing and copying substitutions, all variables occur exactly once in a term. Thus the rule for *Comp* also splits into two:

$$\begin{array}{rcll} t[u/x][v/y] & \to & t[v/y][u/x] & (y \in \mathsf{fv}(t)) \\ t[u/x][v/y] & \to & t[u[v/y]/x] & (y \in \mathsf{fv}(u)) \end{array}$$

The first is a direct cause of non-termination of the rewrite system, and thus we exclude it. The second is useful, since it allows substitutions to move towards completion. Indeed, it is more than useful, as its inclusion can offer shorter reduction paths. A term of the form $(tu)[v/x][w/y]$, where $y \in \mathsf{fv}(v), x \in \mathsf{fv}(t)$ has the following reductions available:

$$\begin{tikzcd}
& (tu)[v/x][w/y] \arrow[dl] \arrow[dr] & \\
((t[v/x])u)[w/y] \arrow[d] & & (tu)[v[w/y]/x] \arrow[d] \\
(t[v/x][w/y])u \arrow[rr] & & (t[v[w/y]/x])u
\end{tikzcd}$$

Clearly, allowing composition of substitutions to take place before pushing substitutions gives a shorter reduction path. This situation arises whenever a substitution is pushed through a construct, not just applications. The following result clarifies the extent to which this observation is important.

Proposition 2.2 (Order of substitutions). Let $t = t_0[t_1/x_0][t_2/x_1]\cdots[t_n/x_{n-1}]$ where $\mathsf{fv}(t_i) = \{x_i\}, i < n$, and $\mathsf{fv}(t_n) = \emptyset$. A reduction sequence $t \to^* u$ to propagate the substitutions will be $O(n)$ best case and $O(n^2)$ worst case.

Proof. There is a choice between innermost and outermost composition of substitutions. The following diagram illustrates the consequences. To simplify the diagram, we will drop the terms, and just keep the brackets, thus the example term is $[\,][\,]\cdots[\,]$, and the rule for composition of substitutions is $[\,][\,] \to [[\,]]$. We indicate on the arrow the number of times we apply this rule:

$$\begin{tikzcd}
 & {[\,][\,]\cdots[\,][\,]} \arrow[dl, "n"'] \arrow[dr, "1"] & \\
{[[\,]\cdots[\,]]} \arrow[d, "n-1"'] & & {[\,][\,]\cdots[[\,]]} \arrow[d, "1"] \\
{[[[\,]\cdots[\,]]]} \arrow[d, dashed, "\sum_{i=0}^{n-2}(n-i)"'] & & {[\,]\cdots[[[\,]]]} \arrow[d, dashed, "n-2"] \\
{[\cdots[\,]\cdots]} & & {[\cdots[\,]\cdots]}
\end{tikzcd}$$

Thus the innermost sequence required $\sum_{i=0}^{n}(n-i) = \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$, which is $O(n^2)$, whereas the outermost sequence is $O(n)$. □

The order in which we perform the composition of substitutions has a dramatic effect on the efficiency of the rewriting system. One natural way to get an outermost (efficient) strategy is to require that the outer substitution be closed.

This completes our basic observations about α-conversion and substitution propagation. Based on this, in the following sections we present a family of calculi, for which we first define a common syntax.

3. Calculi with names

3.1. *Syntax:* λ_c

We begin by defining λ_c-terms, which are λ-terms with explicit constructs for substitutions, copying and erasing. We will compile λ-terms into λ_c-terms.

Definition 3.1 (λ_c-terms). We use x, y, z to denote variables, t, u, v to denote terms, and $\star$ for an arbitrary constant. Table 2 shows the term constructions, together with the variable constraints that must be satisfied for each construction, and the associated free variables. Λ_c denotes the set of λ_c-terms.

The variable constraints imply that each variable occurs free in a term exactly once (in this calculus we could have the η-rule $(\lambda x.tx) \to t$ without the side condition $x \notin \mathsf{fv}(t)$).

These terms should be understood as an intermediate language: λ-terms are compiled into it. For the rest of this paper, all λ_c-terms will be assumed to be generated by a compilation function $[\![\cdot]\!] : \Lambda \to \Lambda_\mathsf{c}$, or be reducts of those terms.

Table 2. λ_{c}-*terms.*

Name	Term	Variable Constraint	Free Variables
Constant	$\star$	$-$	$\varnothing$
Variable	x	$-$	$\{x\}$
Abstraction	$\lambda x.t$	$x \in \mathsf{fv}(t)$	$\mathsf{fv}(t) - \{x\}$
Application	tu	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
Erase	$\epsilon_x.t$	$x \notin \mathsf{fv}(t)$	$\mathsf{fv}(t) \cup \{x\}$
Copy	$\delta_x^{y,z}.t$	$x \notin \mathsf{fv}(t), y \neq z, \{y,z\} \subseteq \mathsf{fv}(t)$	$(\mathsf{fv}(t) - \{y,z\}) \cup \{x\}$
Substitution	$t[u/x]$	$x \in \mathsf{fv}(t), (\mathsf{fv}(t) - \{x\}) \cap \mathsf{fv}(u) = \varnothing$	$(\mathsf{fv}(t) - \{x\}) \cup \mathsf{fv}(u)$

Definition 3.2 (Compilation). Let t be a λ-term. Its compilation $[\![t]\!]$ into λ_{c} is defined as: $[x_1]\ldots[x_n]\langle t\rangle$ where $\mathsf{fv}(t) = \{x_1,\ldots,x_n\}$, $n \geqslant 0$, we assume without loss of generality that the variables are processed in lexicographic order, and $\langle\cdot\rangle$ is defined by

$$\begin{array}{rcll}
\langle\star\rangle & = & \star & \\
\langle x\rangle & = & x & \\
\langle tu\rangle & = & \langle t\rangle\langle u\rangle & \\
\langle\lambda x.t\rangle & = & \lambda x.[x]\langle t\rangle & \text{if } x \in \mathsf{fv}(t) \\
 & = & \lambda x.\epsilon_x.\langle t\rangle & \text{otherwise.}
\end{array}$$

We define $[\cdot]\cdot$ by

$$\begin{array}{lcll}
[x]x & = & x & \\
[x](\lambda y.t) & = & \lambda y.[x]t & \\
[x](tu) & = & \delta_x^{x',x''}.[x'](t[x := x'])[x''](u[x := x'']) & x \in \mathsf{fv}(t), x \in \mathsf{fv}(u) \\
 & = & ([x]t)u & x \in \mathsf{fv}(t), x \notin \mathsf{fv}(u) \\
 & = & t([x]u) & x \in \mathsf{fv}(u), x \notin \mathsf{fv}(t) \\
[x](\epsilon_y.t) & = & \epsilon_y.[x]t & \\
[x](\delta_y^{y',y''}.t) & = & \delta_y^{y',y''}.[x]t &
\end{array}$$

where the substitution $t[x := u]$ is the usual (implicit) notion, and the variables x' and x'' above are assumed to be fresh.

Note that the free variables are processed in lexicographic order. For example,

$$[\![(xy)(xy)]\!] = [x][y]\langle(xy)(xy)\rangle = \delta_y^{y',y''}.\delta_x^{x',x''}.(x'y')(x''y'') \neq [y][x]\langle(xy)(xy)\rangle$$

We could of course have chosen any ordering in the definition.

Terms built from the compilation are *pure* terms: they do not contain substitutions. The essence of compilation is to perform variable counting, and to place the erase and copy constructs where they are needed. The particular compilation that we have chosen has the property that erase is placed outermost, and copy is placed innermost, which intuitively are the most effective places (erase as quickly as possible, but postpone the duplication until necessary during substitution propagation). Consequently, erase constructs only occur immediately after abstractions: for example, $\lambda x.\epsilon_x.t$. We could therefore introduce a new abstraction $\lambda\_.t$ that combines these. However, for most of this paper we prefer to keep them separate.

We now investigate some general properties of λ_c and the compilation function.

Proposition 3.3. If t is a λ-term then:

1 $\mathsf{fv}([\![t]\!]) = \mathsf{fv}(t)$.

2 $[\![t]\!]$ is a valid λ_c-term (satisfying the variable constraints in Table 2).

Proof.

1 The translation function does not erase variables, and only introduces bound ones.

2 We proceed by induction on the structure of t. If $t \equiv \star$ or $t \equiv x$, the result holds trivially (both translate to terms that do not have variable constraints). Assume $\mathsf{fv}(t) = \{x_1, \dots, x_n\}$. If $t \equiv \lambda x.u$, there are two cases to consider:

(a) If $x \in \mathsf{fv}(u)$,

$$[x_1]\cdots[x_n]\langle\lambda x.u\rangle = [x_1]\cdots[x_n]\lambda x.[x]\langle u\rangle = \lambda x.[x_1]\cdots[x_n][x]\langle u\rangle$$

By hypothesis, $[x_1]\cdots[x_n][x]\langle u\rangle$ is a valid λ_c-term, and by Part 1 we know that $x \in \mathsf{fv}([x_1]\cdots[x_n][x]\langle u\rangle)$, thus $\lambda x.[x_1]\cdots[x_n][x]\langle u\rangle$ is also valid.

(b) If $x \notin \mathsf{fv}(u)$,

$$[x_1]\cdots[x_n]\langle\lambda x.u\rangle = [x_1]\cdots[x_n]\lambda x.\epsilon_x.\langle u\rangle = \lambda x.\epsilon_x.[x_1]\cdots[x_n]\langle u\rangle$$

By hypothesis, $[x_1]\cdots[x_n]\langle u\rangle$ is a valid λ_c-term, and $x \notin \mathsf{fv}([x_1]\cdots[x_n]\langle u\rangle)$ by Part 1, thus $\epsilon_x.[x_1]\cdots[x_n]\langle u\rangle$ is valid, and $\lambda x.\epsilon_x.[x_1]\cdots[x_n]\langle u\rangle$ is also valid.

Finally, if $t \equiv uv$, then $[x_1]\cdots[x_n]\langle uv\rangle = [x_1]\cdots[x_n](\langle u\rangle\langle v\rangle)$. Assume also that $\mathsf{fv}(u)\cap\mathsf{fv}(v) = \{x_{i_1}, \dots, x_{i_p}\}$ for some $p \geqslant 0$. Now if we let u' be the term $u[x_{i_1} := x'_{i_1}, \dots, x_{i_p} := x'_{i_p}]$ and $v' = v[x_{i_1} := x''_{i_1}, \dots, x_{i_p} := x''_{i_p}]$, and let $y_{j_1}, \dots, y_{j_q}$ and $z_{k_1}, \dots, z_{k_r}$ be the lists of free variables of u' and v', respectively, in lexicographic ordering, we get

$$[x_1]\cdots[x_n](\langle u\rangle\langle v\rangle) = \delta_{x_{i_p}}^{x'_{i_p},x''_{i_p}}.\dots.\delta_{x_{i_1}}^{x'_{i_1},x''_{i_1}}.[y_{j_1}]\dots[y_{j_q}]\langle u'\rangle[z_{k_1}]\dots[z_{k_r}]\langle v'\rangle$$

By Part 1 and the induction hypothesis, $[y_{j_1}]\dots[y_{j_q}]\langle u'\rangle$ and $[z_{k_1}]\dots[z_{k_r}]\langle v'\rangle$ are both valid λ_c-terms. They have no variables in common (since the common variables have been renamed), therefore the application is a valid λ_c-term. Next, $x_{i_1}, \dots, x_{i_p}$ are not free in the application, $x'_{i_j} \neq x''_{i_j}$ and x'_{i_j}, x''_{i_j} are free variables of the application (for all $1 \leqslant j \leqslant p$), thus the resulting term is valid. □

We can easily recover a λ-term from a λ_c-term by simply erasing the additional constructs and completing the substitutions. For this we give a readback function.

Definition 3.4 (Readback). The function $(\cdot)^* : \Lambda_\mathsf{c} \to \Lambda$ is defined inductively as follows:

$$\begin{array}{lll lll}
\star^* & = & \star & x^* & = & x \\
(tu)^* & = & t^*u^* & (\lambda x.t)^* & = & \lambda x.t^* \\
(\epsilon_x.t)^* & = & t^* & (\delta_x^{y,z}.t)^* & = & t^*[y := x][z := x] \\
(t[u/x])^* & = & t^*[x := u^*] & & &
\end{array}$$

A useful property of readback is given by the following proposition.

Proposition 3.5. If t is a λ-term, $[\![t]\!]^* = t$.

Proof. The proof is by straightforward induction over the structure of the λ-term t. □

Example 3.6 (Terms arising from compilation). We give several example terms in this calculus, which are obtained using the compilation function

$$\begin{array}{lcll}
\mathbf{I} & = & [\![\lambda x.x]\!] & = \lambda x.x \\
\mathbf{K} & = & [\![\lambda x.\lambda y.x]\!] & = \lambda x.\lambda y.\epsilon_y.x \\
\mathbf{S} & = & [\![\lambda x.\lambda y.\lambda z.xz(yz)]\!] & = \lambda x.\lambda y.\lambda z.\delta_z^{z',z''}.(xz')(yz'') \\
\mathbf{2} & = & [\![\lambda f.\lambda x.f(fx)]\!] & = \lambda f.\lambda x.\delta_f^{g,h}.g(hx) \\
\mathbf{Y} & = & [\![\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))]\!] & = \lambda f.\delta_f^{g,h}.(\lambda x.g(\delta_x^{x',x''}.x'x'')) \\
 & & & \qquad (\lambda x.h(\delta_x^{x',x''}.x'x''))
\end{array}$$

As a non-example, consider the term $\lambda x.\lambda y.\epsilon_x.y$. This is a valid λ_{c}-term (*cf.* Table 2), but does not arise from the compilation function. Note, however, that there is a standard way to transform it into an equivalent, compiled term: $[\![(\lambda x.\lambda y.\epsilon_x.y)^*]\!] = \lambda x.\epsilon_x.\lambda y.y$.

This completes the definition and basic properties of the syntax of our calculus. In the following sections we define several reduction relations for λ_{c}-terms, with the key property that α-conversion is not required.

3.2. *Closed arguments:* λ_{ca}

Following on from Section 2, we observe that all the problems of α-conversion can be eradicated if all substitutions are closed. Since the only rule that creates a substitution is *Beta*: $(\lambda x.t)v \to t[v/x]$, we start by requiring that the argument v be closed in this rule. There are a number of immediate consequences of this constraint: in particular, we no longer need the *Comp* rule. In this section we formalise this calculus, called λ_{ca}, and study its properties. This strategy for implementing the λ-calculus is clearly a weak one, but we will show that it is adequate for the evaluation of programs.

Definition 3.7 (λ_{ca}-calculus). The set Λ_{ca} of *valid* λ_{ca}-*terms* contains all the λ_{c}-terms that are the image of the compilation function and their reducts under the reduction relation $\to_{\mathsf{ca}}$ generated by the conditional rewrite system in Table 3.

As usual, we write $\to_{\mathsf{ca}}^*$ for the transitive reflexive closure of $\to_{\mathsf{ca}}$. Rewriting can be applied in every context; in particular, within substitutions and under abstractions. Normal forms of this calculus are the irreducible terms. Following the tradition started with $\lambda\sigma$ (Abadi *et al.* 1991) we call σ the subset of rules for substitution, that is, all the rules except *Beta*. A σ-normal form is a term that cannot be reduced using the rules in σ.

Remark 3.8.

- By the condition on the *Beta* rule, all substitutions generated are *closed*, and thus there is no risk of variable clash or variable capture in the rules *Lam*, *Copy2* and *Erase2*. Thus no conditions are required if we assume valid λ_{ca}-terms.

Table 3. λ_{ca}*-reduction.*

Name	Reduction			Condition
Beta	$(\lambda x.t)v$	$\to_{\mathsf{ca}}$	$t[v/x]$	$\mathsf{fv}(v) = \varnothing$
Var	$x[v/x]$	$\to_{\mathsf{ca}}$	v	
App1	$(tu)[v/x]$	$\to_{\mathsf{ca}}$	$(t[v/x])u$	$x \in \mathsf{fv}(t)$
App2	$(tu)[v/x]$	$\to_{\mathsf{ca}}$	$t(u[v/x])$	$x \in \mathsf{fv}(u)$
Lam	$(\lambda y.t)[v/x]$	$\to_{\mathsf{ca}}$	$\lambda y.t[v/x]$	
Copy1	$(\delta_x^{y,z}.t)[v/x]$	$\to_{\mathsf{ca}}$	$t[v/y][v/z]$	
Copy2	$(\delta_{x'}^{y,z}.t)[v/x]$	$\to_{\mathsf{ca}}$	$\delta_{x'}^{y,z}.t[v/x]$	
Erase1	$(\epsilon_x.t)[v/x]$	$\to_{\mathsf{ca}}$	t	
Erase2	$(\epsilon_{x'}.t)[v/x]$	$\to_{\mathsf{ca}}$	$\epsilon_{x'}.t[v/x]$	

— There are variants of the reduction rules that do not change the basic results of this paper. For instance, in a simply typed framework, the *Copy1* rule can reduce the substituted term v to normal form before copying, thus avoiding the duplication of redexes. We will study this variant in Section 5.

In this section we will only work with valid λ_{ca}-terms (that is, terms derived from the compilation of λ-terms, and their reducts, which are also valid by Proposition 3.10 below).

Example 3.9. We will now give several examples to provide a flavour of the reduction relation. The first example indicates that reduction is stronger than the weak reduction of combinatory logic (note that **KI** reduces to a pure normal form):

$$\mathbf{KI} = (\lambda x.\lambda y.\epsilon_y.x)(\lambda x.x) \to_{\mathsf{ca}} (\lambda y.\epsilon_y.x)[\lambda x.x/x] \to^*_{\mathsf{ca}} \lambda y.\epsilon_y.\lambda x.x$$

To show that the calculus is weak, consider

$$\mathbf{22} = (\lambda f.\lambda x.\delta_f^{g,h}.g(hx))\mathbf{2} \to_{\mathsf{ca}} (\lambda x.\delta_f^{g,h}.g(hx))[\mathbf{2}/f] \to^*_{\mathsf{ca}} \lambda x.\mathbf{2}(\mathbf{2}x)$$

Now it is impossible to duplicate the variable x in this (weak) normal form – we must wait for a closing substitution before continuing. If we apply this to two closed terms, reduction can continue to full normal form:

$$\begin{aligned}
\mathbf{22II} &\to^*_{\mathsf{ca}} \mathbf{2(2I)I} \to^*_{\mathsf{ca}} (\lambda x.\delta_f^{g,h}.g(hx))[\mathbf{2I}/f]\mathbf{I}\\
&\to_{\mathsf{ca}} (\lambda x.\delta_f^{g,h}.g(hx))[(\lambda x.\delta_f^{g,h}.g(hx))[\mathbf{I}/f]/f]\mathbf{I}\\
&\to^*_{\mathsf{ca}} (\lambda x.\delta_f^{g,h}.g(hx))[(\lambda x.\mathbf{I}(\mathbf{I}x))/f]\mathbf{I}\\
&\to^*_{\mathsf{ca}} (\lambda x.(\lambda x.\mathbf{I}(\mathbf{I}x))((\lambda x.\mathbf{I}(\mathbf{I}x))x))\mathbf{I}\\
&\to^*_{\mathsf{ca}} (\lambda x.\mathbf{I}(\mathbf{I}x))((\lambda x.\mathbf{I}(\mathbf{I}x))\mathbf{I})\\
&\to^*_{\mathsf{ca}} (\lambda x.\mathbf{I}(\mathbf{I}x))(\mathbf{I}(\mathbf{II}))\\
&\to^*_{\mathsf{ca}} (\lambda x.\mathbf{I}(\mathbf{I}x))\mathbf{I} \to^*_{\mathsf{ca}} \mathbf{I}(\mathbf{II}) \to^*_{\mathsf{ca}} \mathbf{I}
\end{aligned}$$

If one studies this example in detail, there are several possible evaluation strategies. The best (shortest) route to take is always to push closed substitutions through an application, and reduce terms to normal form before copying. However, the restriction imposed in the *Beta* rule does not allow us to reduce $\lambda x.\mathbf{I}(\mathbf{I}x)$ before copying it, so we cannot share its computation. Although λ_{ca} permits more sharing than standard weak reduction to WHNF (Proposition 3.42 below), it is not entirely satisfactory from this point of view. To ameliorate this defect, we present a refinement of the strategy in Section 3.3.

As a last example, consider the term **Y**, which in the λ-calculus has a weak head normal form, but no head normal form. However, even though we reduce under abstractions there is no infinite sequence of reductions out of **Y** in λ_{ca} since the only redex (shown underlined) has a free variable h in the argument:

$$\lambda f.\delta_f^{g,h}.\underline{(\lambda x.g(\delta_x^{x',x''}.x'x''))(\lambda x.h(\delta_x^{x',x''}.x'x''))}$$

It is easy to see, however, that the term **YI** does generate a non-terminating sequence, as the argument is a closed term.

Note that we never need α-conversions in the reductions above, although the same variable name may occur several times in the term being reduced.

Proposition 3.10 (Correctness of $\rightarrow_{\mathsf{ca}}$). Let t be a valid λ_{ca}-term, and $t \rightarrow_{\mathsf{ca}} u$. Then:

1 $\mathsf{fv}(t) = \mathsf{fv}(u)$.
2 u is a valid λ_{c}-term.

Proof.

1 Only closed terms are erased, and no new variables are created during reduction.
2 It is sufficient to prove that the relation $\rightarrow_{\mathsf{ca}}$ preserves the variable constraints given in Table 2. We distinguish cases:

Beta: $(\lambda x.t)u \rightarrow_{\mathsf{ca}} t[u/x]$ if $\mathsf{fv}(u) = \varnothing$.
By assumption, $x \in \mathsf{fv}(t)$, and clearly $(\mathsf{fv}(t) - \{x\}) \cap \mathsf{fv}(u) = \varnothing$ because $\mathsf{fv}(u) = \varnothing$. Thus $t[u/x]$ is a valid λ_{c}-term.

Var: $x[v/x] \rightarrow_{\mathsf{ca}} v$.
This case is trivial.

App1 : $(tu)[v/x] \rightarrow_{\mathsf{ca}} (t[v/x])u$ if $x \in \mathsf{fv}(t)$.
The term $t[v/x]$ is valid, since $x \in \mathsf{fv}(t)$ and $\mathsf{fv}(v) = \varnothing$. By assumption, $\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$, thus $(\mathsf{fv}(t) - \{x\}) \cap \mathsf{fv}(u) = \varnothing$, so $(t[v/x])u$ is a valid λ_{c}-term.

App2: $(tu)[v/x] \rightarrow_{\mathsf{ca}} t(u[v/x])$ if $x \in \mathsf{fv}(u)$.
This case follows the same reasoning as the case for *App1*.

Lam: $(\lambda y.t)[v/x] \rightarrow_{\mathsf{ca}} \lambda y.t[v/x]$.
The term $t[v/x]$ is valid since $x \in \mathsf{fv}(t)$ and $\mathsf{fv}(v) = \varnothing$. Since $y \in \mathsf{fv}(t)$, we have $\lambda y.t[v/x]$ is a valid λ_{c}-term.

Copy1 : $(\delta_x^{y,z}.t)[v/x] \rightarrow_{\mathsf{ca}} t[v/y][v/z]$.
The term $t[v/y]$ is valid since $y \in \mathsf{fv}(t)$ and $\mathsf{fv}(v) = \varnothing$. Similarly, $t[v/y][v/z]$ is also a valid λ_{c}-term because $z \in \mathsf{fv}(t[v/y])$.

Copy2: $(\delta_{x'}^{y,z}.t)[v/x] \rightarrow_{\mathsf{ca}} \delta_{x'}^{y,z}.t[v/x]$.
The term $t[v/x]$ is valid for the same reasons as above. Now $\delta_{x'}^{y,z}.t[v/x]$ is valid since y and z have not become bound by the substitution.

Erase1 : $(\epsilon_x.t)[v/x] \rightarrow_{\mathsf{ca}} t$.
This case is trivial.

Erase2 : $(\epsilon_{x'}.t)[v/x] \rightarrow_{\mathsf{ca}} \epsilon_{x'}.t[v/x]$.
Using the same reasoning as above, the term $t[v/x]$ is valid, and since $x' \notin \mathsf{fv}(t[v/x])$, the term $\epsilon_{x'}.t[v/x]$ is also valid. □

Corollary 3.11. $\Lambda_{\mathsf{ca}} \subset \Lambda_{\mathsf{c}}$.

Proof. By Proposition 3.10, Part 2, all λ_{ca}-terms are valid λ_{c}-terms. The inclusion is strict because $\lambda z.(\lambda x.xy)[z/y]$ is a valid λ_{c}-term, but not a valid λ_{ca}-term (the substitution is open). □

In order to show that substitutions are well-defined in this system, we will prove that σ is terminating and confluent, and that closed substitutions cannot remain blocked. We also show that the commutation of substitutions can be derived from the system, which justifies the exclusion of the *Comp* rule. To prove the termination of the rules for substitution we use the following definition.

Definition 3.12 (Distance of a variable). Let $|t|_x$ be the *distance* from the root of the term t to the occurrence of the free variable x, which is defined as:

$$\begin{array}{llllll}
|x|_x & = |\epsilon_x.t|_x & = 1 & & & \\
|\lambda y.t|_x & = |\delta^{y,z}_{x'}.t|_x & = |\epsilon_y.t|_x & = 1+|t|_x & & \\
|tu|_x & = |t[u/y]|_x & = 1+|t|_x & & & (\text{if } x \in \mathsf{fv}(t)) \\
|tu|_x & = |t[u/y]|_x & = 1+|u|_x & & & (\text{if } x \in \mathsf{fv}(u)) \\
|\delta^{y,z}_{x}.t|_x & = 1+|t|_y+|t|_z & & & &
\end{array}$$

Proposition 3.13 (Termination of substitutions). There are no infinite reduction sequences in λ_{ca} using only the rules in σ.

Proof. We define an interpretation that associates to each term t a multiset with one element for each subterm $w[s/x]$ occurring in t, which is $|\ w\ |_x$. Each application of a substitution rule decreases the interpretation of the term, since we always apply a substitution to a subterm of t or erase it, and the distance is strictly reduced. □

Proposition 3.14 (Confluence of substitutions). If $t =_\sigma u$, there exists s such that $t \rightarrow^*_\sigma s$ and $u \rightarrow^*_\sigma s$.

Proof. Since there are no critical pairs in σ, the system is locally confluent, and since it is terminating, it is also confluent by Newman's Lemma (Newman 1942). Note also that the substitution rules are left-linear, so we can deduce confluence directly since the system is orthogonal (Klop 1980). □

We now look at the problem of completion of substitutions inside Λ_{ca}, but maintaining the philosophy of only considering terms that are derived from the compilation of λ-terms. In particular, this implies that all the substitutions are closed.

Lemma 3.15. If v is a closed term, $t[v/x]$ is not a normal form in λ_{ca}.

Proof. We use induction on the structure of t. Since $t[v/x]$ is a λ_{ca}-term, $x \in \mathsf{fv}(t)$, thus t cannot be $\star$. If t is a variable, application, abstraction, erase or copy, we can apply one of the rules for substitution. If $t = u[w/y]$, then $x \in \mathsf{fv}(u)$ (it cannot occur free in w since the *Beta* rule that created the substitution could not be applied with an open argument), therefore we can apply the induction hypothesis in $u[w/y]$. □

As a consequence, substitutions do not remain blocked.

Proposition 3.16 (Completion of closed substitutions). Every term in Λ_{ca} has a σ-normal form, which is a pure term (it does not contain substitutions).

Proof. All substitutions in λ_{ca} are closed by definition. Thus, by Lemma 3.15, any subterm $t[v/x]$ can be reduced, and since, by Proposition 3.13, this process terminates, we will eventually reach a term without substitutions. □

We now show that the commutation of closed substitutions can be derived.

Lemma 3.17 (Commutation of substitutions). In Λ_{ca}, for any terms t, u, v, we have

$$t[u/x][v/y] \; =_\sigma \; t[v/y][u/x]$$

Proof. Let $N(t, x, y)$ be the number of positions of variables in t affected by the substitutions $[u/x]$, $[v/y]$ (recall that u, v are closed so the substitutions do not introduce variables). We proceed by induction on $(N(t, x, y), t)$ using the ordering $(>_N, \rhd)_{lex}$ where $>_N$ is the usual ordering on natural numbers, $\lhd$ denotes the subterm relation (so $\rhd$ means superterm), and *lex* indicates lexicographic composition.

Without loss of generality, we assume that t is a σ normal form (see Proposition 3.13), and consider all possible cases for t.

— t cannot be a variable or $\star$ since $x, y \in \mathsf{fv}(t)$ by definition of λ_{ca}.

— If $t = \lambda z.s$, the property holds by induction.

— If $t = (sw)$, we distinguish two cases: if $x, y \in \mathsf{fv}(s)$ or $x, y \in \mathsf{fv}(w)$, the result follows by induction. Otherwise, assume $x \in \mathsf{fv}(s)$ and $y \in \mathsf{fv}(w)$ (the symmetric case is similar). Then $t[u/x][v/y] \to^*_{\mathsf{ca}} (s[u/x])(w[v/y])$ and $t[v/y][u/x] \to^*_{\mathsf{ca}} (s[u/x])(w[v/y])$.

— If $t = \epsilon_z.s$, the result follows by induction. If $t = \epsilon_x.s$ (the case $t = \epsilon_y.s$ is analogous), we have $t[u/x][v/y] \to_{\mathsf{ca}} s[v/y]$ and $t[v/y][u/x] \to_{\mathsf{ca}} (\epsilon_x.s[v/y])[u/x] \to_{\mathsf{ca}} s[v/y]$.

— If $t = \delta_z^{z',z''}.s$, the result follows by induction. If $t = \delta_x^{x',x''}.s$, then $t[u/x][v/y] \to_{\mathsf{ca}} s[u/x'][u/x''][v/y]$, and $t[v/y][u/x] \to_{\mathsf{ca}} (\delta_x^{x',x''}.s[v/y])[u/x] \to_{\mathsf{ca}} s[v/y][u/x'][u/x'']$. The case $t = \delta_y^{y',y''}.s$ is analogous.
Let $s[u/x'] \to^*_{\mathsf{ca}} t'$ in σ normal form. Since $N(t', x'', y) < N(t, x, y)$, by induction, $t'[u/x''][v/y] =_\sigma t'[v/y][u/x'']$. Therefore $s[u/x'][u/x''][v/y] =_\sigma s[u/x'][v/y][u/x'']$.
Also by induction, $s[u/x'][v/y] =_\sigma s[v/y][u/x']$. Therefore $s[u/x'][u/x''][v/y] =_\sigma s[v/y][u/x'][u/x'']$. We then deduce $t[u/x][v/y] =_\sigma t[v/y][u/x]$.

— The case $t = s[w/z]$ is not possible since, by assumption, t is a σ normal form (see Proposition 3.16). □

Local confluence is an easy consequence of the previous lemma.

Proposition 3.18 (Local confluence). Let $t \in \Lambda_{\mathsf{ca}}$. If $t \to_{\mathsf{ca}} u$ and $t \to_{\mathsf{ca}} v$, there is a term $s \in \Lambda_{\mathsf{ca}}$ such that $u \to^*_{\mathsf{ca}} s$ and $v \to^*_{\mathsf{ca}} s$.

Proof. There is only one critical pair, generated by *Beta* and *App1* (the other potential superpositions are eliminated from the system because of the conditions in the rules). If $y \in \mathsf{fv}(\lambda x.t)$, and $\mathsf{fv}(u) = \mathsf{fv}(v) = \emptyset$,

$$\begin{tikzcd}
((\lambda x.t)u)[v/y] \arrow[r] \arrow[d] & ((\lambda x.t)[v/y])u \arrow[r] & (\lambda x.t[v/y])u \arrow[d] \\
t[u/x][v/y] & =_\sigma & t[v/y][u/x]
\end{tikzcd}$$

Lemma 3.17 and Proposition 3.14 show that this critical pair can be joined using the rules in σ. Note that if $y \in \mathsf{fv}(u)$, the critical pair is eliminated, since the *Beta*-reduction step is blocked until the substitution is made. □

Although λ_{ca} enjoys other useful properties, such as the preservation of the strong normalisation property of λ-terms and confluence, we will not study it any further since we will present an improved version of the calculus in the next section, for which all these properties will be shown.

3.3. *Closed functions:* λ_{cf}

The condition on the *Beta* rule of λ_{ca} restricts sharing, as Example 3.9 shows. To avoid α-conversion, it is sufficient to require a closed substitution in the rules that involve a binder. This is the idea underlying the system λ_{cf}, which provides an alternative set of reduction rules for λ_{c}-terms. The *Comp* rule (which allows the composition of substitutions) will no longer be derived, but will be added in a controlled way to avoid non-termination. Recall from Section 2 that the *Comp* rule with the linear syntax divides into two rules:

$$\begin{array}{lcll} t[u/x][v/y] & \to & t[u[v/y]/x] & (y \in \mathsf{fv}(u)) \\ t[u/x][v/y] & \to & t[v/y][u/x] & (y \in \mathsf{fv}(t)) \end{array}$$

For λ_{ca}, neither rule was needed, but we were able to derive the inter-convertibility relation corresponding to the second one. We will now add the first one since it allows the substitution process to move towards completion. This rule still leaves some freedom in the choice of strategy for the application of substitutions, with surprising consequences.

However, when leaving λ_{ca}, that is, when allowing the generation of open substitutions, we may have a confluence problem in the system due to the *Beta*/*Lam* critical pair. Consider the term $((\lambda x.t)u)[v/y]$, where $y \in \mathsf{fv}(t)$, $\mathsf{fv}(u) \neq \emptyset$, and $\mathsf{fv}(v) = \emptyset$.

$$\begin{tikzcd}
((\lambda x.t)u)[v/y] \arrow[r] \arrow[d] & ((\lambda x.t)[v/y])u \arrow[r] & (\lambda x.t[v/y])u \arrow[d] \\
t[u/x][v/y] & \overset{?}{\longleftrightarrow} & t[v/y][u/x]
\end{tikzcd}$$

Since we have eliminated one of the cases for the composition of substitutions, we cannot close this diagram (Lemma 3.17 cannot be applied with an open substitution). However, we can avoid this confluence problem by only allowing *Beta*-reduction to take place when the function is closed:

$$(\lambda x.t)u \to t[u/x] \quad \text{if } \mathsf{fv}(\lambda x.t) = \emptyset$$

Table 4. λ_{cf}-*reduction.*

Name		*Reduction*		Condition
Beta	$(\lambda x.t)u$	$\to_{\mathsf{cf}}$	$t[u/x]$	$\mathsf{fv}(\lambda x.t) = \varnothing$
Var	$x[v/x]$	$\to_{\mathsf{cf}}$	v	
App1	$(tu)[v/x]$	$\to_{\mathsf{cf}}$	$(t[v/x])u$	$x \in \mathsf{fv}(t)$
App2	$(tu)[v/x]$	$\to_{\mathsf{cf}}$	$t(u[v/x])$	$x \in \mathsf{fv}(u)$
Lam	$(\lambda y.t)[v/x]$	$\to_{\mathsf{cf}}$	$\lambda y.t[v/x]$	$\mathsf{fv}(v) = \varnothing$
Copy1	$(\delta_x^{y,z}.t)[v/x]$	$\to_{\mathsf{cf}}$	$t[v/y][v/z]$	$\mathsf{fv}(v) = \varnothing$
Copy2	$(\delta_{x'}^{y,z}.t)[v/x]$	$\to_{\mathsf{cf}}$	$\delta_{x'}^{y,z}.t[v/x]$	
Erase1	$(\epsilon_x.t)[v/x]$	$\to_{\mathsf{cf}}$	t	$\mathsf{fv}(v) = \varnothing$
Erase2	$(\epsilon_{x'}.t)[v/x]$	$\to_{\mathsf{cf}}$	$\epsilon_{x'}.t[v/x]$	
Comp	$t[w/y][v/x]$	$\to_{\mathsf{cf}}$	$t[w[v/x]/y]$	$x \in \mathsf{fv}(w)$

This resolves the above problem by cutting out the left-hand branch of the diagram. It is interesting to note that in other work on explicit substitutions it is the right-hand branch that is cut out: for instance, see Muñoz (1996).

With these observations, we obtain a new calculus of closed reductions, which is based on closed functions, λ_{cf}.

Definition 3.19 (λ_{cf}-calculus). The set Λ_{cf} of *valid* λ_{cf}-*terms* contains all the λ_{c}-terms that are the image of the compilation function and their reducts under the reduction relation $\to_{\mathsf{cf}}$ generated by the conditional rewrite system in Table 4. Again, we call σ the set of rules for substitution, that is, all the rules except *Beta*. As usual, we write $\to_{\mathsf{cf}}^*$ for the transitive reflexive closure of $\to_{\mathsf{cf}}$. Note that reduction rules can be applied in every context: in particular, reductions can take place within substitutions, and under abstractions.

Remark 3.20. There are a number of possible variants of the reduction rules:

- The *Erase2* rule does not have any condition. It is easily seen from the variable constraints in Table 2 that if $(\epsilon_{x'}.t)[v/x]$ is a well-formed term, then so is $\epsilon_{x'}.t[v/x]$, as x' cannot be free in v. Similarly, in *Copy2*, we know x' cannot be free in v, and y, z are fresh variables introduced in the compilation.
- We could also require the *App1*, *App2* and *Comp* rules to be closed. This gives a weaker calculus, with a more directed strategy, but, on the other hand, it would force the shortest reduction path, as well as avoiding some critical pairs in the system. However, we prefer to allow these rules to be used with open terms since this calculus gives the minimal requirements of closed reduction having reasonable properties.
- We could also add a number of 'optimisation' rules, for instance, $t[x/x] \to t$, but we prefer not to include these in the basic theory. This one corresponds to $t[id] \to t$ in some other explicit substitution calculi.

Since the syntax of terms remains the same as in λ_{ca}, we use the same compilation and read-back functions to translate λ-terms. The set Λ_{cf} of *valid terms* is, however, different from Λ_{ca} (for example, the term $\lambda z.(\lambda x.xy)[z/y]$ is in Λ_{cf} but not in Λ_{ca}).

Example 3.21. The term $\lambda x.\lambda y.(xz)[y/z]$ is not a valid λ_{cf}-term, since the *Beta* rule must have been applied when the function $\lambda z.(xz)$ was not closed (not a valid reduction). Also, note that $\lambda x.\lambda y.\epsilon_x.y$ is not valid, since it does not come from the compilation function (ϵ_x is not in the correct place) and no reduction in λ_{cf} will produce it.

We now give several examples of reductions in λ_{cf}.

Example 3.22. Consider again the terms from Example 3.6. The term **KI** still reduces to a pure normal form:

$$\mathbf{KI} = (\lambda x.\lambda y.\epsilon_y.x)(\lambda x.x) \to_{\mathsf{cf}} (\lambda y.\epsilon_y.x)[\lambda x.x/x] \to^*_{\mathsf{cf}} \lambda y.\epsilon_y.\lambda x.x$$

Although this calculus is not as weak as λ_{ca}, the reduction of **22** does not reach a full normal form.

$$\begin{array}{rl}
\mathbf{22} \;=& (\lambda f x.\delta_f^{g,h}.g(hx))\mathbf{2} \\
\to_{\mathsf{cf}} & (\lambda x.\delta_f^{g,h}.g(hx))[\mathbf{2}/f] \\
\to^*_{\mathsf{cf}} & \lambda x.\mathbf{2}(\mathbf{2}x) \\
\to_{\mathsf{cf}} & \lambda x.(\lambda x.\delta_f^{g,h}.g(hx))[\mathbf{2}x/f] \\
\to_{\mathsf{cf}} & \lambda x.(\lambda x.\delta_f^{g,h}.g(hx))[(\lambda x.\delta_f^{g,h}.g(hx))[x/f]/f]
\end{array}$$

To continue, it is sufficient to apply this to a closed term, for instance **I** (recall that we needed two arguments in λ_{ca} in Example 3.9).

$$\begin{array}{rl}
\mathbf{22I} \to^*_{\mathsf{cf}} & ((\lambda x.\delta_f^{g,h}.g(hx))[(\lambda x.\delta_f^{g,h}.g(hx))[x/f]/f])[\mathbf{I}/x] \\
\to^*_{\mathsf{cf}} & (\lambda x.\delta_f^{g,h}.g(hx))[(\lambda x.\delta_f^{g,h}.g(hx))[\mathbf{I}/f]/f] \\
\to^*_{\mathsf{cf}} & (\lambda x.\delta_f^{g,h}.g(hx))[(\lambda x.\mathbf{I}(\mathbf{I}x))/f] \\
\to^*_{\mathsf{cf}} & (\lambda x.\delta_f^{g,h}.g(hx))[\mathbf{I}/f] \\
\to^*_{\mathsf{cf}} & (\lambda x.\mathbf{I}(\mathbf{I}x)) \\
\to^*_{\mathsf{cf}} & \lambda x.x
\end{array}$$

Again, there is no infinite sequence of reduction out of **Y** even though we reduce under abstractions, since the only redex (shown underlined) has a free variable g in the function:

$$\lambda f.\delta_f^{g,h}.\underline{(\lambda x.g(\delta_x^{x',x''}.x'x''))(\lambda x.h(\delta_x^{x',x''}.x'x''))}$$

The term **YI** does generate a non-terminating sequence: $\mathbf{YI} \to^*_{\mathsf{cf}} \mathbf{I}(\mathbf{YI}) \to^*_{\mathsf{cf}} \mathbf{YI} \to^*_{\mathsf{cf}} \cdots$

Proposition 3.23 (Correctness of $\to_{\mathsf{cf}}$). Let t be a λ_{cf}-term, and $t \to_{\mathsf{cf}} u$. Then:

1 $\mathsf{fv}(t) = \mathsf{fv}(u)$.

2 u is a valid λ_{c}-term, that is, $\to_{\mathsf{cf}}$ preserves the variable constraints in Table 2.

Proof. Both parts follow the same reasoning as spelled out for λ_{ca} in Proposition 3.10. □

Corollary 3.24. $\Lambda_{\mathsf{cf}} \subset \Lambda_{\mathsf{c}}$.

Proof. By Proposition 3.23, Part 2, all λ_{cf}-terms are valid λ_{c}-terms. The inclusion is strict because $\lambda x.\lambda y.(xz)[y/z]$ is a valid λ_{c}-term, but not a valid λ_{cf}-term. □

We can now prove termination of the substitution rules using the same interpretation technique as in λ_{ca}.

Proposition 3.25 (Termination of substitutions). There are no infinite reduction sequences in λ_{cf} using only the rules in σ.

Another important property of the system is that closed substitutions do not remain blocked.

Lemma 3.26. If v is a closed term, then $t[v/x]$ is not a normal form in λ_{cf}.

Proof. We use induction on the structure of t. Since $t[v/x]$ is a λ_{cf}-term, $x \in \mathsf{fv}(t)$, thus t cannot be $\star$. If t is a variable, application, abstraction, erase or copy, we can apply one of the rules for substitution. If $t = u[w/y]$ then there are two cases:

1 If $x \in \mathsf{fv}(w)$, we can apply the *Comp* rule.

2 If $x \in \mathsf{fv}(u)$, then w must be closed otherwise the *Beta* rule could not be applied to create $[v/x]$. Therefore we can apply the induction hypothesis in $u[w/y]$. □

For λ_{cf}, we do not have an analogous result to Proposition 3.16 for λ_{ca} (not all terms can be reduced to pure terms in λ_{cf}, though this is true for closed terms).

We now look at the problem of preservation of strong normalisation (a calculus of explicit substitutions preserves strong normalisation if the compilation of a strongly normalisable λ-term is strongly normalisable). Unlike many systems of explicit substitutions, λ_{cf} and λ_{ca} preserve strong normalisation. Our proof is inspired by the proof given for λ_{υ} (Lescanne and Rouyer-Degli 1995); we will give the proof for λ_{cf}, but exactly the same applies to λ_{ca}. We first define a notion of minimal infinite derivation. Intuitively, a derivation is minimal if we always reduce a lowest possible redex to keep non-termination. We use $\rightarrow_{Beta,p}$ to denote a *Beta* reduction at position p.

Definition 3.27 (Minimal derivation). An infinite λ_{cf} derivation

$$t_1 \rightarrow_{Beta,p_1} t_1' \rightarrow_{\sigma}^{*} \cdots t_i \rightarrow_{Beta,p_i} t_i' \rightarrow_{\sigma}^{*} \ldots$$

is *minimal* if for any other infinite derivation

$$t_1 \rightarrow_{Beta,p_1} t_1' \rightarrow_{\sigma}^{*} \cdots t_i \rightarrow_{Beta,q} u \rightarrow_{\sigma}^{*} \cdots$$

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.

Lemma 3.28.

1 If $t \rightarrow_{\sigma}^{*} t'$, then $t^{*} = t'^{*}$.

2 If $t \rightarrow_{Beta} t'$ is a step in a minimal derivation starting from the compilation of a λ-term, then $t^{*} \rightarrow_{\beta}^{+} t'^{*}$.

Proof.

1 This part follows by a straightforward inspection of the rules for substitution.

2 The term t contains a *Beta* redex, and the minimality assumption for the derivation ensures that in the readback t^{*} this redex is not erased (but, of course, it can be copied). Hence $t^{*} \rightarrow_{\beta}^{+} t'^{*}$. □

We will prove a result that is slightly more general than preservation of strong normalisation and will also help us to prove the termination of typeable λ_{cf}-terms in Section 4.1.

Proposition 3.29. If t^* is a strongly normalisable λ-term, t is strongly normalisable.

Proof. To generate a contradiction, assume that there is an infinite reduction sequence starting from t in λ_{cf}. In particular, there is an infinite minimal reduction sequence

$$t \to_{\sigma}^{*} t_1 \to_{Beta} t_2 \to_{\sigma}^{*} t_3 \to_{Beta} t_4 \cdots$$

Since the rules for substitution are terminating, each infinite derivation contains an infinite number of applications of *Beta*. By Lemma 3.28, we obtain an infinite derivation for t^* (which gives a contradiction):

$$t^* = t_1^* \to_{\beta}^{+} t_2^* = t_3^* \to_{\beta}^{+} t_4^* \cdots \qquad \square$$

Proposition 3.30 (Preservation of strong normalisation). If t is the translation of a strongly normalisable λ-term s, then t is strongly normalisable.

Proof. The statement is a direct consequence of Proposition 3.29 since $t^* = s$ by Proposition 3.5. □

We now turn our attention to the confluence problem for λ_{cf}[†]. It is easy to see that the system is locally confluent.

Proposition 3.31 (Local confluence). If $t \to_{\mathsf{cf}} u$ and $t \to_{\mathsf{cf}} v$, there is a term s such that $u \to_{\mathsf{cf}}^{*} s$ and $v \to_{\mathsf{cf}}^{*} s$.

Proof. There are seven critical pairs to consider, all of which converge as shown below. All the other potential superpositions are eliminated from the system because of the free variable constraints.

1 $((\lambda y.t)u)[v/x]$ where $\mathsf{fv}(\lambda y.t) = \varnothing$ and $x \in \mathsf{fv}(u)$:

$$\begin{tikzcd}
 & ((\lambda y.t)u)[v/x] \arrow[dl] \arrow[dr] & \\
t[u/y][v/x] \arrow[d] & & (\lambda y.t)(u[v/x]) \arrow[d] \\
t[u[v/x]/y] & \equiv & t[u[v/x]/y]
\end{tikzcd}$$

Note that if $x \in \mathsf{fv}(t)$, this critical pair is eliminated, since the *Beta*-reduction step is blocked until the substitution is made.

[†] Since the rewrite relation is only defined on ground terms, this is a ground confluence problem.

2 $y[w/y][v/x]$ where $x \in \mathsf{fv}(w)$:

$$\begin{tikzcd}
y[w/y][v/x] \arrow[r] \arrow[d] & y[w[v/x]/y] \arrow[d] \\
w[v/x] \arrow[r, phantom, "\equiv"] & w[v/x]
\end{tikzcd}$$

3 $(tu)[w/y][v/x]$ where $y \in \mathsf{fv}(t)$ and $x \in \mathsf{fv}(w)$:

$$\begin{tikzcd}
 & (tu)[w/y][v/x] \arrow[dl] \arrow[dr] & \\
((t[w/y])u)[v/x] \arrow[d] & & (tu)[w[v/x]/y] \arrow[dd] \\
(t[w/y][v/x])u \arrow[d] & & \\
(t[w[v/x]/y])u \arrow[rr, phantom, "\equiv"] & & (t[w[v/x]/y])u
\end{tikzcd}$$

4 The case when $y \in \mathsf{fv}(u)$ is almost identical to the previous case.

5 $(\epsilon_{x'}.t)[w/y][v/x]$ where $x \in \mathsf{fv}(w)$. This case is similar to the cases for application above.

6 $(\delta_z^{z',z''}.t)[w/y][v/x]$ where $x \in \mathsf{fv}(w)$. Again, this case is similar to the application and erase cases above.

7 $t[w/y][v/x][u/z]$ where $z \in \mathsf{fv}(v)$ and $x \in \mathsf{fv}(w)$:

$$\begin{tikzcd}
 & t[w/y][v/x][u/z] \arrow[dl] \arrow[dr] & \\
t[w[v/x]/y][u/z] \arrow[d] & & t[w/y][v[u/z]/x] \arrow[d] \\
t[w[v/x][u/z]/y] \arrow[rr] & & t[w[v[u/z]/x]/y]
\end{tikzcd}$$

□

Remark 3.32. If we restrict all the rules for λ_{cf} to have closed substitutions only, there is only one critical pair that needs analysing for local confluence (the first one), and in this case we get strong confluence directly, rather than local confluence (but we can do fewer reductions of course).

Corollary 3.33 (Confluence of $\to_\sigma$). If $t =_\sigma u$, there exists s such that $t \to^*_{\mathsf{cf}} s$ and $u \to^*_{\mathsf{cf}} s$.

Proof. The statement follows by Newman's Lemma (Newman 1942), since the substitution rules are terminating by Proposition 3.25. □

Using Newman's Lemma, we can also deduce confluence of terminating λ_{cf}-terms. But we can prove the confluence property without assuming termination. For this we first prove the commutation of *Beta*- and σ-reduction steps.

Lemma 3.34 (Commutation of *Beta* and σ in λ_{cf}). If $b \ {}_{Beta}{}^{*}\!\leftarrow s \to^*_\sigma a$, there exists c such that $b \to^*_\sigma c \ {}_{Beta}{}^{*}\!\leftarrow a$.

Proof. We use an auxiliary relation $\Rightarrow$, which makes parallel *Beta*-reductions in one step. It is defined by induction:

1 $t \Rightarrow t$,
2 $(\lambda x.t)u \Rightarrow t'[u'/x]$ if $\mathsf{fv}(\lambda x.t) = \varnothing$, $t \Rightarrow t'$ and $u \Rightarrow u'$,
3 $\lambda x.t \Rightarrow \lambda x.t'$ if $t \Rightarrow t'$,
4 $tu \Rightarrow t'u'$ if $t \Rightarrow t'$ and $u \Rightarrow u'$,
5 $t[v/x] \Rightarrow t'[v'/x]$ if $t \Rightarrow t'$ and $v \Rightarrow v'$,
6 $\delta^{y,z}_x.t \Rightarrow \delta^{y,z}_x.t'$ if $t \Rightarrow t'$,
7 $\epsilon_x.t \Rightarrow \epsilon_x.t'$ if $t \Rightarrow t'$.

First note that $\Rightarrow^*$ coincides with $\to^*_{Beta}$: indeed, $\to_{Beta} \subseteq \Rightarrow$ by definition, and $\Rightarrow \subseteq \to^*_{Beta}$ by an easy induction.

To prove the lemma, it is sufficient to show that if $b \Leftarrow s \to_\sigma a$, there exists a term c such that $b \to_\sigma c \Leftarrow a$. Then we can close the diagram $b \ {}_{Beta}{}^{*} \leftarrow s \to^*_\sigma a$ (which is equivalent to $b \ {}^{*}\!\Leftarrow s \to^*_\sigma a$), by induction on the length of the derivation $s \Rightarrow^* b$.

We proceed by induction on the definition of $s \Rightarrow b$. We distinguish the following cases:

1 $s \equiv b$. Then we take $c \equiv a$.
2 $s \equiv (\lambda x.t)u \Rightarrow b \equiv t'[u'/x]$, $\mathsf{fv}(\lambda x.t) = \varnothing$, $t \Rightarrow t'$ and $u \Rightarrow u'$. Since no rule from σ applies at the root of s, it must be either $t \to_\sigma t''$ (and $\mathsf{fv}(\lambda x.t'') = \varnothing$ by Proposition 3.23), or $u \to_\sigma u''$. In the first case, by induction, there exists a term t''' such that $t' \to_\sigma t'''$ and $t'' \Rightarrow t'''$, therefore we can take $c \equiv t'''[u'/x]$. In the second case, by induction, there exists u''' such that $u' \to_\sigma u'''$ and $u'' \Rightarrow u'''$, therefore we can take $c \equiv t'[u'''/x]$.
3 In the cases $s \equiv \lambda x.t \Rightarrow b \equiv \lambda x.t'$ where $t \Rightarrow t'$, $s \equiv tu \Rightarrow b \equiv t'u'$ where $t \Rightarrow t'$ and $u \Rightarrow u'$, $s \equiv \delta^{y,z}_x.t \Rightarrow b \equiv \delta^{y,z}_x.t'$ where $t \Rightarrow t'$, and $s \equiv \epsilon_x.t \Rightarrow b \equiv \epsilon_x.t'$ where $t \Rightarrow t'$, the property follows directly by induction.
4 $s \equiv t[v/x] \Rightarrow b \equiv t'[v'/x]$, $t \Rightarrow t'$ and $v \Rightarrow v'$. We distinguish two cases according to the position of the σ-reduction step $s \to_\sigma a$.

— If it does not apply at the root of s, the property holds by induction.

— If it applies at the root, there is a substitution θ and a rule $l \to r$ in σ such that $s \equiv l\theta$ and $a \equiv r\theta$.

– If all the *Beta*-redexes that are contracted in the reduction $s \Rightarrow b$ are under variables in l (that is, they are in θ), then these variables are uniquely identified (since l is left-linear) so we can define a substitution θ' such that $\theta \Rightarrow \theta'$ and the diagram commutes: $s \equiv l\theta \rightarrow_{\sigma} r\theta \equiv a \Rightarrow r\theta' \equiv c$ and $s \equiv l\theta \Rightarrow l\theta' \equiv b \rightarrow_{\sigma} r\theta' \equiv c$.

– If there is a *Beta*-reduction step at the root of t, we have a critical pair between the σ-rule applied at the root of s and the *Beta* rule applied at the root of t. In that case, the σ-rule applied must be App_2 (it cannot be App_1 because of the variable restrictions). Then the diagram commutes as follows: $s \equiv ((\lambda y.w)u)[v/x] \rightarrow_{App_2} (\lambda y.w)(u[v/x]) \equiv a \Rightarrow w'[u'[v'/x]/y] \equiv c$ and $s \equiv ((\lambda y.w)u)[v/x] \Rightarrow b \equiv w'[u'/y][v'/x] \rightarrow_{Comp} w'[u'[v'/x]/y] \equiv c$.

This concludes the proof. □

Proposition 3.35 (Confluence). Let t be a term in Λ_{cf}. If $t \rightarrow^*_{\mathsf{cf}} u$ and $t \rightarrow^*_{\mathsf{cf}} v$, there is a term s such that $u \rightarrow^*_{\mathsf{cf}} s$ and $v \rightarrow^*_{\mathsf{cf}} s$.

Proof. We have already shown the confluence of the σ rules (Corollary 3.33) and it is easy to see that the *Beta* rule alone is confluent. We deduce confluence of Λ_{cf} from Lemma 3.34 using Rosen's Lemma (Rosen 1973). □

3.4. *Relation to β-reduction*

Closed reduction (both λ_{ca} and λ_{cf}) is a strictly smaller theory than that induced by the usual β-reduction: not all β-redexes can be reduced, and, moreover, the substitution might not complete. The purpose of this section is to show that it is powerful enough to compute the usual evaluation strategies for the λ-calculus, which demonstrates that we have not lost any expressive power by adding severe constraints on the reduction system, at least when dealing with reduction to weak head normal form (WHNF) of closed terms.

There are three main strategies of reduction to WHNF: call-by-name, call-by-value and call-by-need (call-by-name with sharing). We will show that we can simulate call-by-name and call-by-value, in both λ_{ca} and λ_{cf} (we only give the details of the proofs for λ_{cf}). Since call-by-need is not formally defined on λ-terms but rather on an extended λ-calculus syntax, we will not discuss this strategy here.

The following lemma will be useful to establish the connection with β-reduction in the λ-calculus.

Lemma 3.36 (Substitution). Let t and w be λ-terms such that $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$, $n \geqslant 1$, and $\mathsf{fv}(w) = \varnothing$. Then $([x_1] \ldots [x_n]\langle t \rangle)[\langle w \rangle / x_1] \rightarrow^*_{\mathsf{cf}} [x_2] \ldots [x_n]\langle t[x_1 := w] \rangle$.

Proof. By Proposition 3.3 (Part 1), $\mathsf{fv}([\![w]\!]) = \varnothing$, and $x_1 \in \mathsf{fv}([x_1] \ldots [x_n]\langle t \rangle)$. We proceed by induction on t.

$t \equiv x$. $([x]\langle x\rangle)[\langle w\rangle/x] = x[\langle w\rangle/x] \to_{\mathsf{cf}} \langle w\rangle = \langle x[x:=w]\rangle$.

$t \equiv \lambda y.u$, $y \in \mathsf{fv}(u)$. Assume for simplicity that x is the only free variable of t.

$$\begin{aligned}
([x]\langle \lambda y.u\rangle)[\langle w\rangle/x] &= ([x](\lambda y.[y]\langle u\rangle))[\langle w\rangle/x] \\
&= (\lambda y.[x][y]\langle u\rangle)[\langle w\rangle/x] \\
&\to_{\mathsf{cf}} \lambda y.([x][y]\langle u\rangle)[\langle w\rangle/x] \\
&= \lambda y.[y]\langle u[x:=w]\rangle \qquad \text{(Induction)} \\
&= \langle \lambda y.u[x:=w]\rangle \\
&= \langle (\lambda y.u)[x:=w]\rangle
\end{aligned}$$

$t \equiv \lambda y.u$, $y \notin \mathsf{fv}(u)$. This follows in a similar way to the above, with $\langle \lambda y.u\rangle = \lambda y.\epsilon_y.\langle u\rangle$.

$t \equiv uv$, $x \in \mathsf{fv}(u)$. Assume for simplicity that x is the only free variable of t.

$$\begin{aligned}
([x]\langle uv\rangle)[\langle w\rangle/x] &= ([x](\langle u\rangle\langle v\rangle))[\langle w\rangle/x] \\
&= (([x]\langle u\rangle)\langle v\rangle)[\langle w\rangle/x] \\
&\to_{\mathsf{cf}} (([x]\langle u\rangle)[\langle w\rangle/x])\langle v\rangle \\
&= \langle u[x:=w]\rangle\langle v\rangle \qquad \text{(Induction)} \\
&= \langle (u[x:=w])v\rangle \\
&= \langle (uv)[x:=w]\rangle
\end{aligned}$$

$t \equiv uv$, $x \in \mathsf{fv}(v)$. This is similar to the above.

$t \equiv uv$, $x \in \mathsf{fv}(u) \wedge x \in \mathsf{fv}(v)$.

$$\begin{aligned}
([x]\langle uv\rangle)[\langle w\rangle/x] &= ([x](\langle u\rangle\langle v\rangle))[\langle w\rangle/x] \\
&= (\delta_x^{x',x''}.([x'](\langle u\rangle[x:=x']))([x''](\langle v\rangle[x:=x'']))[\langle w\rangle/x] \\
&\to_{\mathsf{cf}} (([x']\langle u\rangle[x:=x'])[\langle w\rangle/x'])(([x'']\langle v\rangle[x:=x''])[\langle w\rangle/x'']) \\
&= \langle u[x:=x'][x':=w]\rangle\langle v[x:=x''][x'':=w]\rangle \qquad \text{(Induction)} \\
&= \langle u[x:=w]\rangle\langle v[x:=w]\rangle \\
&= \langle (u[x:=w])(v[x:=w])\rangle \\
&= \langle (uv)[x:=w]\rangle
\end{aligned}$$

□

Note that the above lemma holds equally for λ_{ca} (indeed the proof is identical). We now look at the two strategies in turn.

Call-by-name. The evaluation rules for weak reduction in the call-by-name λ-calculus are as follows:

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.t' \quad t'[x:=u] \Downarrow v}{tu \Downarrow v}$$

where $t \Downarrow v$ means that a closed term t evaluates to (the principal) WHNF v.

Proposition 3.37 (Simulation of call-by-name in λ_{cf}). Let t be a closed λ-term. If $t \Downarrow v$ by the call-by-name strategy, there is a sequence of reductions $\llbracket t\rrbracket \to_{\mathsf{cf}}^{*} \llbracket v\rrbracket$.

Proof. We use induction on the height of the derivation for $t \Downarrow v$. The case when t is a weak head normal form is obvious. Otherwise it is an application. Using the translation and the hypothesis, we obtain:

$$\llbracket tu\rrbracket = \llbracket t\rrbracket\llbracket u\rrbracket \to_{\mathsf{cf}}^{*} \llbracket \lambda x.t'\rrbracket\llbracket u\rrbracket$$

There are now two cases to consider:

— If $x \in \mathsf{fv}(t')$, by using Lemma 3.36 and the hypothesis again, we obtain

$$[\![\lambda x.t']\!][\![u]\!] = (\lambda x.[x]\langle t'\rangle)\langle u\rangle \rightarrow_{\mathsf{cf}} ([x]\langle t'\rangle)[\langle u\rangle/x] \rightarrow^*_{\mathsf{cf}} [\![t'[x := u]]\!] \rightarrow^*_{\mathsf{cf}} [\![v]\!]$$

— Otherwise,

$$[\![\lambda x.t']\!][\![u]\!] = (\lambda x.\epsilon_x.\langle t'\rangle)\langle u\rangle \rightarrow_{\mathsf{cf}} (\epsilon_x.\langle t'\rangle)[\langle u\rangle/x] \rightarrow_{\mathsf{cf}} \langle t'\rangle = [\![t'[x := u]]\!] \rightarrow^*_{\mathsf{cf}} [\![v]\!]$$

which completes the proof. □

We also observe that the same reasoning holds for the λ_{ca}-calculus.

Call-by-value. The evaluation rules for weak reduction in the call-by-value λ-calculus are as follows:

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.t' \quad u \Downarrow v' \quad t'[x := v'] \Downarrow v}{tu \Downarrow v}$$

where v' is a WHNF.

Proposition 3.38 (Simulation of call-by-value in λ_{cf}). Let t be a closed λ-term. If $t \Downarrow v$ by the call-by-value strategy, there is a sequence of reductions $[\![t]\!] \rightarrow^*_{\mathsf{cf}} [\![v]\!]$.

Proof. The proof follows the same structure as used above for the simulation of call-by-name. Using the translation and the hypothesis twice, we obtain

$$[\![tu]\!] = [\![t]\!][\![u]\!] \rightarrow^*_{\mathsf{cf}} [\![\lambda x.t']\!][\![u]\!] \rightarrow^*_{\mathsf{cf}} [\![\lambda x.t']\!][\![v']\!]$$

There are two cases to consider, which are identical to those given for the proof of call-by-name simulation above. □

Again, observe that the same reasoning also holds for the λ_{ca}-calculus.

Remark 3.39. Note that even for reduction of closed terms to WHNF we do need composition of substitutions in λ_{cf}, unless we impose an outermost strategy. For instance, assume c is closed and $t = \lambda z.\epsilon_z.x$, then $(\lambda y.(\lambda x.t)\,y)\,c \rightarrow_{\mathsf{cf}} (\lambda y.t[y/x])\,c \rightarrow_{\mathsf{cf}} t[y/x][c/y]$ where the substitution for x cannot be propagated (because y is not a closed term), so the only applicable rule is a composition. Moreover, having this rule in the system allows more computation sharing.

Remark 3.40. Although we can reduce open terms (for example, $(\lambda x.x)y \rightarrow^*_{\mathsf{cf}} y$), we cannot, in general, simulate reduction to WHNF on open terms: for instance, $(\lambda x.xy)(\lambda x.x)$ is irreducible. We cannot, in general, simulate reduction to head normal form (HNF), even for closed terms: for example, $\lambda x.(\lambda y.\epsilon_y.x)t \not\rightarrow_{\mathsf{cf}}$. However, the calculus is stronger than standard weak reduction, since we can reduce under abstractions:

$$\lambda x.(\lambda y.y)x \rightarrow_{\mathsf{cf}} \lambda x.y[x/y] \rightarrow_{\mathsf{cf}} \lambda x.x$$

3.5. *Closed arguments and/or functions:* λ_{cl}, λ_{caf}

In previous sections we have defined two different calculi that enjoy the good properties of confluence and preservation of strong normalisation, and are both based on a restricted form of the *Beta* rule. As these restrictions are independent, it is natural to ask whether combinations of these calculi are possible and interesting.

If we take the 'intersection' of the two calculi, that is, if we restrict the *Beta* rule to the case where both the function and the argument are closed, we obtain the reduction of Combinatory Logic as shown in Çağman and Hindley (1998). We call this system λ_{cl}.

However, instead of restricting the set of possible reductions even more, we would rather take the 'union' of the two calculi, that is, allow the rule *Beta* to take place when either the function or the argument is closed, keeping all the σ rules of λ_{cf}. We will call this new calculus λ_{caf}. But in this case confluence cannot be taken for granted. Recall the diagram of the (possible) confluence problem from Section 3.3:

$$\begin{tikzcd}
((\lambda x.t)u)[v/y] \arrow[r] \arrow[d] & ((\lambda x.t)[v/y])u \arrow[r] & (\lambda x.t[v/y])u \arrow[d] \\
t[u/x][v/y] \arrow[rr, leftrightarrow, "?"] & & t[v/y][u/x]
\end{tikzcd}$$

where $x, y \in \mathsf{fv}(t)$ and $\mathsf{fv}(v) = \emptyset$.

The left-hand branch is an application of the *Beta* rule. However, $y \in \mathsf{fv}(t)$ (and if this is not the case, there is no confluence problem), so it cannot be the *Beta* rule of λ_{cf}. Therefore it is the closed-argument *Beta* rule, and u is closed. In the right-hand branch, we apply a *Lam* rule since v is closed too. Hence we are in exactly the same case as in Lemma 3.17, and we have $t[u/x][v/y] =_{\sigma} t[v/y][u/x]$ (the proof is similar to Lemma 3.17).

Since λ_{caf} differs from λ_{cf} only in the *Beta* rule, the previous discussion justifies the following property.

Proposition 3.41. The calculus λ_{caf} is locally confluent.

Proof. The proof is similar to that given for Proposition 3.31, using the previous argument for case 1, and the other cases without any changes. □

The other properties proved for λ_{ca} and λ_{cf} are still valid for λ_{caf}, and the proofs can be easily adapted. We now compare the four systems.

Proposition 3.42. For closed terms,

— λ_{ca}, λ_{cf} allow for more sharing than standard weak reduction (either *call-by-name* or *call-by-value*).

— λ_{ca}, λ_{cf} allow for more sharing than λ_{cl}.

— λ_{caf} allows for more sharing than λ_{ca} and λ_{cf}.

Proof. The first point is a trivial consequence of the propositions in Section 3.4, and the other two points are immediate from inspection of the systems. □

3.6. *Canonicity*

So far we have introduced several α-conversion free calculi obtained by allowing certain rules to take place only when some terms are closed. These choices may look a bit *ad hoc*. However, now that things are well in order we will see that they are not so arbitrary, and, indeed, there are not many choices left.

Adopting the syntax of λ_{c}, we specify our objectives:

— We require a *Beta* rule $(\lambda x.t)u \to t[u/x]$ with eventual restrictions, and rules to propagate substitutions with the obvious intended meaning (that is, propagate the substitution towards the occurrences of the corresponding free variable).

— We do not want to deal with α-conversion. Thus, we have to restrict not just the *Lam* rule but the *Copy1* rule also. The most general *Lam* rule without α-conversion is

$$(\lambda y.t)[v/x] \to \lambda y.t[v/x] \quad x \neq y, y \notin \mathsf{fv}(v)$$

But then the reduction depends on the name of a bound variable (that is, two α-equivalent terms do not have the same redexes), which might lead to confluence problems. A safer approach is to broaden the condition to $\mathsf{fv}(v) = \varnothing$. For the other rule, α-conversion is actually needed as soon as the substitution is open. Thus, we impose

$$(\delta_x^{y,z}.t)[v/x] \to t[v/y][v/z] \quad \mathsf{fv}(v) = \varnothing$$

— We want our calculus to allow reduction (at least) to weak head normal form. The WHNF is actually what is needed when implementing a functional language, and is the minimum requirement for a λ-calculus to be of any practical use.

— We want to preserve strong normalisation, which also implies that we want the σ-rules to be strongly normalising. The reason for this is that we adopt an implementation perspective, and we want to be free to choose any strategy to reduce strongly normalisable λ-terms.

— Finally, we require our calculus to be confluent. Again, this seems to be a minimal requirement for a deterministic implementation of the λ-calculus and if we are to be free to choose any strategy. This last requirement leads to some restrictions in the rules. Indeed, we have the familiar *Beta/App1* critical pair, which we will write down again for the last time:

$$\begin{tikzcd}
((\lambda x.t)u)[v/y] \arrow[r] \arrow[d] & ((\lambda x.t)[v/y])u \arrow[r] & (\lambda x.t[v/y])u \arrow[d] \\
t[u/x][v/y] \arrow[rr, phantom, "\overset{?}{\longleftrightarrow}"] & & t[v/y][u/x]
\end{tikzcd}$$

where $x, y \in \mathsf{fv}(t)$.

There are three ways to solve this potential problem:

1 Eliminate the critical pair by forbidding one of the reductions in the diagram.

— Cutting the right-hand branch is a possibility, which has been studied by Muñoz (Muñoz 1996). His idea is to forbid the distribution of the substitution in a stack of applications beginning with a function. However, this cannot be a first-order rule with the standard syntax, and therefore every application is annotated with special marks to distinguish stacks of applications beginning with a variable or a function. The resulting calculus is fully confluent and preserves strong normalisation, but does not fit with our 'specifications', since its syntax is more complex (and, as a matter of fact, uses de Bruijn indices).

— Now if we want to cut the left-hand branch of the critical pair, which is left as an open problem by Muñoz, we have to restrict the *Beta* rule exactly to the case where no substitution can be applied to the function, that is, to the case where the function is closed: so we are in the heart of λ_{cf}. But we can still allow some propagation rules to take place when the substitution is open, so we obtain a slightly larger set of reductions than just λ_{cf}. However, some propagation rules have to be restricted to avoid α-conversion (*Lam,Copy1*), or to ensure normalisation of the σ-rules (*Comp*, if wanted), so that in most cases a generated substitution will not be allowed to get through the whole term to the variables, since it will have to pass by one of the restricted rules. So the choices made in λ_{cf} of closing more than necessary do not in reality reduce the number of effective strategies available in the calculus, and lead to a much more homogeneous rewriting system.

2 Require the bottom line of the diagram to be derivable. In order to derive

$$t[u/x][v/y] =_\sigma t[v/y][u/x]$$

(with restrictions) in the system, without adding an *ad hoc* rule, we have to propagate the substitutions until they are not nested. The problem is that some of the propagation rules need the substitution to be closed (see above), so that to ensure that we can actually propagate the substitutions, we have to require them always to be closed. This is exactly the same as saying that we may only generate closed substitutions, that is, we have exactly λ_{ca}.

3 Add a rule to swap substitutions, ensuring that the bottom line holds. Of course, if we coarsely add a rule

$$t[u/x][v/y] \to t[v/y][u/x] \quad \text{for all } u,v$$

we lose the preservation of strong normalisation, which is not what we want. But we could, for instance, have a system with the closed propagation rules, and a 'closed' swapping rule like

$$t[u/x][v/y] \to t[v/y][u/x] \quad x, y \in \mathsf{fv}(t), \mathsf{fv}(u) \neq \varnothing, \mathsf{fv}(v) = \varnothing$$

Then we could allow a *Beta* rule without restrictions. This solution seems to preserve strong normalisation, to solve the stated confluence problem, and is quite meaningful: it actually allows closed substitutions to propagate more (that is, inside terms with open substitutions). However, this introduces a wealth of new needless possible strategies. Instead of generating two substitutions, the outermost being closed, and eventually swapping them, we prefer to generate the closed substitution first, propagate it until behind the right λ and then generate the second one so that they are already in the right order. This is morally equivalent but simpler. However, there may be alternative interesting possibilities for such a swapping rule, which we do not pursue here.

$$\frac{\Gamma, x : A, y : B, \Delta \vdash t : C}{\Gamma, y : B, x : A, \Delta \vdash t : C}\ (\text{Exchange})$$

$$\frac{}{\vdash \star : I}\ (\text{Unit}) \qquad \frac{}{x : A \vdash x : A}\ (\text{Var})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\ (\text{Abs}) \qquad \frac{\Gamma \vdash t : A \to B \qquad \Delta \vdash u : A}{\Gamma, \Delta \vdash tu : B}\ (\text{App})$$

$$\frac{\Gamma \vdash t : B}{\Gamma, x : A \vdash \epsilon_x.t : B}\ (\text{Erase}) \qquad \frac{\Gamma, x : A, y : A \vdash t : B}{\Gamma, z : A \vdash \delta_z^{x,y}.t : B}\ (\text{Copy})$$

$$\frac{\Gamma, x : A \vdash t : B \qquad \Delta \vdash v : A}{\Gamma, \Delta \vdash t[v/x] : B}\ (\text{Sub})$$

Fig. 1. Type assignment for λ_{c}.

To conclude, the λ_{ca} and λ_{cf} calculi do, in fact, arise quite naturally as α-conversion free calculi with the given specification.

4. Extending λ_{c}

In this section we will extend λ_{c} to obtain a minimalistic functional programming language. We start by defining a notion of a *program*, for which we introduce types.

4.1. *A type system for* λ_{c}

We consider a typed variant of λ_{c}, and investigate several of the usual properties associated with such calculi. The type system we give is common to both λ_{ca} and λ_{cf}, and for the properties we shall focus on λ_{cf}. However, all the results of this section also hold for λ_{ca} (and, moreover, are generally easier to obtain for λ_{ca}).

Definition 4.1. In Figure 1 we give the typing rules, where $\Gamma \vdash t : A$ indicates that the λ_{cf}-term t has type A in the context Γ, and I is a type constant. The context is treated as an ordered sequence $x_1 : A_1, \ldots, x_n : A_n$, and we give an explicit structural rule that allows elements of the sequence to be permuted. Note that this is the only structural rule, since copying and erasing elements of the sequence are done explicitly by the typing rules for *Copy* and *Erase*, respectively.

In this system, many of the syntactical constraints of the syntax are now captured by the rules. Moreover, the constraint on the axiom forces Γ to contain only the free variables of t, and no other. The typing rules are directly inspired by Curry's type system (see, for instance, Barendregt (1992)) for the λ-calculus, whose judgements are written $\Gamma \vdash_{\lambda} t : A$.

The *Copy* and *Erase* rules would normally be classified as structural rules. The following result indicates how the systems are related.

Lemma 4.2.

1 Let t be a λ-term such that $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$, and $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$. Then,

$$\Gamma \vdash_\lambda t : A \iff \Gamma \vdash [x_1] \cdots [x_n]\langle t \rangle : A$$

2 Let t be a valid λ_c-term such that $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$, and $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$. Then,

$$\Gamma \vdash t : A \iff \Gamma \vdash_\lambda t^* : A$$

Proof. The proof is by straightforward induction on the type derivation. □

We now give some standard results for this calculus.

Proposition 4.3 (Subject Reduction). If $\Gamma \vdash t : A$ and $t \to_\mathsf{cf} u$, then $\Gamma \vdash u : A$.

Proof. We show that each rule preserves types in Figure 2. The case for *App2* follows in a similar way to *App1*. □

We now look at the problem of termination of typeable terms. Since typeable λ-terms are terminating, the termination of typeable λ_cf-terms is a direct consequence of Proposition 3.29 and Lemma 4.2.

Proposition 4.4 (Termination). If $\Gamma \vdash t : A$ in λ_cf, then t is strongly normalisable.

Definition 4.5 (Programs). A *program* is a closed term of type I.

We can show that in our calculus programs can be reduced to values that are pure terms. This can be understood as meaning that for all closed terms of type I there are enough closed substitutions so that they can all complete.

Proposition 4.6 (Adequacy). If t is a program, then $t \to^*_\mathsf{cf} \star$.

Proof. By Subject Reduction, the type of the term is preserved under reduction. Assume, to produce a contradiction, that the program t is in normal form, and it is not $\star$. Since t is closed, it cannot be a variable, an erasing construct or a copying construct. Since it is of type I, it cannot be an abstraction either. If $t = u[v/x]$, then v is closed (since t is closed), and therefore one of the rules for substitution would apply by Lemma 3.26. Hence t is an application.

Let $t = u_1 u_2 \ldots u_n$, $n \geqslant 2$, such that u_1 is not an application. Since t is closed, so are $u_1, \ldots, u_n$. Hence u_1 is not a variable, a copying or an erasing. Since t is a normal form, u_1 cannot be an abstraction either (the *Beta* rule would apply). Therefore u_1 is a term of the form $s[s'/x]$ where s' is closed and x is the only free variable of s. But then u_1 is not a normal form (Lemma 3.26), which contradicts our assumption. □

4.2. λ_c *as a programming language*

The type system and the notion of a program above can be regarded as a simplification of a minimalistic functional programming language, such as PCF (Plotkin 1977), where we

$$\frac{\dfrac{x : A \vdash t : B}{\vdash \lambda x.t : A \to B} \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x.t)u : B} \;\to\; \frac{x : A \vdash t : B \quad \Gamma \vdash u : A}{\Gamma \vdash t[u/x] : B}$$

$$\frac{\dfrac{}{x : A \vdash x : A} \qquad \Gamma \vdash v : A}{\Gamma \vdash x[v/x] : A} \;\to\; \frac{}{\Gamma \vdash v : A}$$

$$\frac{\dfrac{\Gamma, x : C \vdash t : A \to B \quad \Delta \vdash u : A}{\Gamma, \Delta, x : C \vdash tu : B} \qquad \Theta \vdash v : C}{\Gamma, \Delta, \Theta \vdash (tu)[v/x] : B} \;\to\; \frac{\dfrac{\Gamma, x : C \vdash t : A \to B \quad \Theta \vdash v : C}{\Gamma, \Theta \vdash t[v/x] : A \to B} \qquad \Delta \vdash u : A}{\Gamma, \Delta, \Theta \vdash (t[v/x])u : B}$$

$$\frac{\dfrac{\Gamma, x : C, y : A \vdash t : B}{\Gamma, x : C \vdash \lambda y.t : A \to B} \qquad \vdash v : C}{\Gamma \vdash (\lambda y.t)[v/x] : A \to B} \;\to\; \frac{\dfrac{\Gamma, y : A, x : C \vdash t : B \quad \vdash v : C}{\Gamma, y : A \vdash t[v/x] : B}}{\Gamma \vdash \lambda y.t[v/x] : A \to B}$$

$$\frac{\dfrac{\Gamma, y : A, z : A \vdash t : B}{\Gamma, x : A \vdash \delta^{y,z}_{x}.t : B} \qquad \vdash v : A}{\Gamma \vdash (\delta^{y,z}_{x}.t)[v/x] : B} \;\to\; \frac{\dfrac{\Gamma, z : A, y : A \vdash t : B \quad \vdash v : A}{\Gamma, z : A \vdash t[v/y] : B} \qquad \vdash v : A}{\Gamma \vdash (t[v/y])[v/z] : B}$$

$$\frac{\dfrac{\Gamma, x : A, y : C, z : C \vdash t : B}{\Gamma, x' : C, x : A \vdash \delta^{y,z}_{x'}.t : B} \qquad \Delta \vdash v : A}{\Gamma, \Delta, x' : C \vdash (\delta^{y,z}_{x'}.t)[v/x] : B} \;\to\; \frac{\dfrac{\Gamma, x : A, y : C, z : C \vdash t : B \quad \Delta \vdash v : A}{\Gamma, \Delta, y : C, z : C \vdash t[v/x] : B}}{\Gamma, \Delta, x' : C \vdash \delta^{y,z}_{x'}.t[v/x] : B}$$

$$\frac{\dfrac{\Gamma \vdash t : B}{\Gamma, x : A \vdash \epsilon_{x}.t : B} \qquad \vdash v : A}{\Gamma \vdash (\epsilon_{x}.t)[v/x] : B} \;\to\; \frac{}{\Gamma \vdash t : B}$$

$$\frac{\dfrac{\Gamma, x : A \vdash t : B}{\Gamma, x' : C, x : A \vdash \epsilon_{x'}.t : B} \qquad \Delta \vdash v : A}{\Gamma, \Delta, x' : C \vdash (\epsilon_{x'}.t)[v/x] : B} \;\to\; \frac{\dfrac{\Gamma, x : A \vdash t : B \quad \Delta \vdash v : A}{\Gamma, \Delta \vdash t[v/x] : B}}{\Gamma, \Delta, x' : C \vdash \epsilon_{x'}.t[v/x] : B}$$

$$\frac{\dfrac{\Gamma, y : B \vdash t : C \quad \Delta, x : A \vdash w : B}{\Gamma, \Delta, x : A \vdash t[w/y] : C} \qquad \Theta \vdash v : A}{\Gamma, \Delta, \Theta \vdash t[w/y][v/x] : C} \;\to\; \frac{\Gamma, y : B \vdash t : C \qquad \dfrac{\Delta, x : A \vdash w : B \quad \Theta \vdash v : A}{\Delta, \Theta \vdash w[v/x] : B}}{\Gamma, \Delta, \Theta \vdash t[w[v/x]/y] : C}$$

Fig. 2. Subject reduction.

$$\frac{}{v \Downarrow v} \qquad \frac{t \Downarrow \lambda x.t' \quad t'[x := u] \Downarrow v'}{tu \Downarrow v'}$$

$$\frac{t(\mathsf{fix}(t)) \Downarrow v}{\mathsf{fix}(t) \Downarrow v} \qquad \frac{t \Downarrow n}{\mathsf{suc}(t) \Downarrow n+1} \qquad \frac{t \Downarrow n+1}{\mathsf{pred}(t) \Downarrow n} \qquad \frac{t \Downarrow 0}{\mathsf{pred}(t) \Downarrow 0}$$

$$\frac{t \Downarrow 0}{\mathsf{zer}(t) \Downarrow \mathsf{tt}} \qquad \frac{t \Downarrow n+1}{\mathsf{zer}(t) \Downarrow \mathsf{ff}} \qquad \frac{b \Downarrow \mathsf{tt} \quad t \Downarrow v}{\mathsf{if}(b,t,u) \Downarrow v} \qquad \frac{b \Downarrow \mathsf{ff} \quad u \Downarrow v}{\mathsf{if}(b,t,u) \Downarrow v}$$

Fig. 3. PCF evaluation.

Table 5. *Syntax: Extensions.*

Term	Type Constraint	Variable Constraint	Free Variables
$n : \mathsf{nat}$	–	–	$\varnothing$
$\mathsf{tt}, \mathsf{ff} : \mathsf{bool}$	–	–	$\varnothing$
$\mathsf{suc}(t) : \mathsf{nat}$	$t : \mathsf{nat}$	–	$\mathsf{fv}(t)$
$\mathsf{pred}(t) : \mathsf{nat}$	$t : \mathsf{nat}$	–	$\mathsf{fv}(t)$
$\mathsf{zer}(t) : \mathsf{bool}$	$t : \mathsf{nat}$	–	$\mathsf{fv}(t)$
$\mathsf{if}(b,t,u) : A$	$b : \mathsf{bool}, t : A, u : A$	$\mathsf{fv}(t) = \mathsf{fv}(u), \mathsf{fv}(t) \cap \mathsf{fv}(b) = \varnothing$	$\mathsf{fv}(b) \cup \mathsf{fv}(t)$
$\mathsf{fix}(t) : A$	$t : A \to A$	–	$\mathsf{fv}(t)$

have just one constant, and no arithmetic functions or recursion. There are no difficulties in extending this calculus to the full language of PCF, and indeed beyond, as we will now sketch.

In Figure 3 we recall the syntax and operational semantics for evaluation of PCF terms, where we write $t \Downarrow v$ to indicate that the closed term t evaluates to the value v. In these rules, the terms are typed λ-terms, extended with the constant functions of PCF, and substitution $t[x := u]$ is the usual meta-operation. Values are natural numbers, booleans (tt, ff) and abstractions.

Table 5 gives the extensions to λ_{c} so that we can capture the same syntax as PCF. Note that we have included both the type and variable constraints, from which one can easily write down the corresponding typing rules. We call the resulting language $\lambda_{\mathsf{cf}}^{\mathsf{pcf}}$.

It is straightforward to extend the compilation (Definition 3.2) to capture PCF. We just show the additional compilation for the constants.

Definition 4.7 (Compilation of PCF). Let t be a PCF term, $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$, $n \geqslant 0$. Its compilation into $\lambda_{\mathsf{cf}}^{\mathsf{pcf}}$ is defined as $[x_1] \ldots [x_n]\langle t \rangle$, with $\langle \cdot \rangle$ defined as the extension of the function given in Definition 3.2 where $\star \in \{\mathsf{tt}, \mathsf{ff}\}$ or $\star = n$ and

$$\begin{array}{lcll} \langle \mathsf{f}(t) \rangle & = & \mathsf{f}(\langle t \rangle) & (\mathsf{f} \in \{\mathsf{suc}, \mathsf{pred}, \mathsf{zer}, \mathsf{fix}\}) \\ \langle \mathsf{if}(b,t,u) \rangle & = & \mathsf{if}(\langle b \rangle, \epsilon_{y_1} \ldots . \epsilon_{y_p}.\langle t \rangle, \epsilon_{z_1} \ldots . \epsilon_{z_q}.\langle u \rangle) & (\mathsf{fv}(u) - \mathsf{fv}(t) = \{y_1, \ldots, y_p\}, \\ & & & \mathsf{fv}(t) - \mathsf{fv}(u) = \{z_1, \ldots, z_q\}, p \geqslant 0, q \geqslant 0) \end{array}$$

Table 6. $\lambda_{\mathsf{cf}}^{\mathsf{pcf}}$*-reduction.*

Name	Reduction			Condition
Succ	$\mathsf{suc}(n)$	$\to$	$n+1$	–
Pred1	$\mathsf{pred}(0)$	$\to$	0	–
Pred2	$\mathsf{pred}(n+1)$	$\to$	n	–
Iszero1	$\mathsf{zer}(0)$	$\to$	tt	–
Iszero2	$\mathsf{zer}(n+1)$	$\to$	ff	–
Cond1	$\mathsf{if}(\mathsf{tt}, t, u)$	$\to$	t	–
Cond2	$\mathsf{if}(\mathsf{ff}, t, u)$	$\to$	u	–
Rec	$\mathsf{fix}(t)$	$\to$	$t(\mathsf{fix}(t))$	$\mathsf{fv}(t) = \varnothing$
Sub1	$\mathsf{f}(t)[v/x]$	$\to$	$\mathsf{f}(t[v/x])$	–
Sub2	$\mathsf{if}(b, t, u)[v/x]$	$\to$	$\mathsf{if}(b[v/x], t, u)$	$x \in \mathsf{fv}(b)$
Sub3	$\mathsf{if}(b, t, u)[v/x]$	$\to$	$\mathsf{if}(b, t[v/x], u[v/x])$	$x \in \mathsf{fv}(t), \mathsf{fv}(v) = \varnothing$

We also add the following cases to the definition of $[\cdot]\cdot$:

$$\begin{array}{lll}
[x]\mathsf{f}(t) & = \mathsf{f}([x]t) & (\mathsf{f} \in \{\mathsf{suc}, \mathsf{pred}, \mathsf{zer}, \mathsf{fix}\}) \\
[x]\mathsf{if}(b, t, u) & = \mathsf{if}([x]b, t, u) & (x \in \mathsf{fv}(b), x \notin \mathsf{fv}(tu)) \\
 & = \mathsf{if}(b, [x]t, [x]u) & (x \notin \mathsf{fv}(b), x \in \mathsf{fv}(tu)) \\
 & = \delta_x^{y,z}.\mathsf{if}([y](b[x := y]), [z](t[x := z]), [z](u[x := z])) & (x \in \mathsf{fv}(b), \mathsf{fv}(tu)) \\
[x]\epsilon_x.t & = \epsilon_x.t &
\end{array}$$

Example 4.8. The following two examples of compiled PCF terms were designed specifically to bring out the details for the conditional:

$$\begin{array}{lll}
[\![\lambda x.\mathsf{if}(x, x, x)]\!] & = & \lambda x.\delta_x^{y,z}.\mathsf{if}(y, z, z) \\
[\![\lambda x.\lambda y.\lambda z.\mathsf{if}(x, y, z)]\!] & = & \lambda x.\lambda y.\lambda z.\mathsf{if}(x, \epsilon_z.y, \epsilon_y.z)
\end{array}$$

Definition 4.9 **($\lambda_{\mathsf{cf}}^{\mathsf{pcf}}$-reduction).** The set of rules are the same as in λ_{cf} with the additions given in Table 6, where $\mathsf{f} \in \{\mathsf{suc}, \mathsf{pred}, \mathsf{zer}, \mathsf{fix}\}$ in *Sub1*.

We can now summarise the main properties of this calculus, most of which are easy extensions of the results for typed λ_{cf}. Recall that PCF programs are closed terms of type nat or bool.

Proposition 4.10 (Properties of $\lambda_{\mathsf{cf}}^{\mathsf{pcf}}$).

1 **Subject reduction**. If $\Gamma \vdash t : A$ and $t \to u$, then $\Gamma \vdash u : A$.

2 **Adequacy**. If $t \Downarrow v$, then $[\![t]\!] \to^* [\![v]\!]$, and, consequently, if t is a program, we have one of:

— $t \to^* \mathsf{tt}$, if $t : \mathsf{bool}$;

— $t \to^* \mathsf{ff}$, if $t : \mathsf{bool}$;

— $t \to^* n$, if $t : \mathsf{nat}$;

— t diverges.

Proof. The first part is a straightforward extension of Lemma 4.3 for the typed calculus. For the second part, we first need to establish a property about substitution for PCF, analogous to Lemma 3.36. Let t and w be PCF terms such that $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$, $n \geqslant 1$, and $\mathsf{fv}(w) = \emptyset$. Then $([x_1] \ldots [x_n]\langle t\rangle)[\langle w\rangle/x_1] \to^* [x_2] \ldots [x_n]\langle t[x_1 := w]\rangle$. The proof of which follows the same reasoning as in the proof of Lemma 3.36.

Using this, it is then straightforward to establish a simulation of PCF reduction: if $t \Downarrow v$, there exists a sequence of reductions $[\![t]\!] \to^* [\![v]\!]$. □

5. Implementation: abstract machines

The aim of this section is to legitimate our work, theoretically and practically. We use the previous calculi to define strategies, which are implemented and compared experimentally. Here Proposition 3.31 (Local confluence) and Proposition 2.2 (Order of substitutions) can be put together to help us understand the best strategy for propagating substitutions, that is, they indicate which choices give the shortest reduction paths.

5.1. *Closed strategies*

The calculi defined earlier are more restrictive than the λ-calculus, hence not every strategy can be defined in our formalism. However, there are still many possibilities. We will informally justify some of the choices we have made with considerations of efficiency, and then show two particular strategies that in our experiments compare favourably with the usual ones. These will be strategies to reduce closed terms (programs) to weak-head normal form (value). Before the formal definition of the strategies, we will give some hints to explain how they work and why we made these choices.

- First, we want to take advantage of our ability to reduce under abstractions, but not at the top-level, since we want to stop on a weak-head normal form. In particular, it is only useful to perform these extra reductions on a term that will be copied, in order to share these reductions. Hence our formal definition will interleave a weak strategy with a stronger one, called only before an application of the *Copy1* rule.
- Before copying, the more we reduce the term to be copied, the more work is shared. Hence, we want to use our most general calculus, namely λ_{caf}. We will, however, give a strategy based on λ_{cf} as well (λ_{ca} alone is too restrictive).
- When we are in a position to apply a *Comp* rule, Proposition 2.2 shows that it is better to do so, rather than reduce the innermost substitution first. Our strategy will thus give a higher priority to *Comp*.
- A final remark is that the rules of λ_{caf} force the substitution to be closed in the rule *Copy1*, that is, we never copy a term with free variables. This is one of the key features of this strategy with respect to efficiency.

In Figure 4 we give the operational semantics of the closed strategy $\Downarrow_w$, using an auxiliary (stronger) relation $\Downarrow_f$. Note that the (*Copy1*) rule calls the stronger reduction $\Downarrow_f$, which is defined in exactly the same way, by just replacing the (*Axiom*) by:

$$\frac{t \Downarrow_f v}{\lambda x.t \Downarrow_f \lambda x.v} \qquad \frac{t \Downarrow_f v}{\epsilon_x.t \Downarrow_f \epsilon_x.v} \qquad \frac{t \Downarrow_f v}{\delta_x^{y,z}.t \Downarrow_f \delta_x^{y,z}.v} \qquad \frac{t \not\Downarrow_f \text{ by another rule}}{t \Downarrow_f t}$$

$$\frac{v \Downarrow_w w}{x[v/x] \Downarrow_w w}\ \text{(Var)} \qquad \frac{t \Downarrow_w \lambda x.r \quad r[u/x] \Downarrow_w v \quad \mathsf{fv}(t) = \varnothing \vee \mathsf{fv}(u) = \varnothing}{(t\,u) \Downarrow_w v}\ \text{(Beta)}$$

$$\frac{(\lambda y.t[u/x]) \Downarrow_w v \quad \mathsf{fv}(u) = \varnothing}{(\lambda y.t)[u/x] \Downarrow_w v}\ \text{(Lam)} \qquad \frac{t \Downarrow_w v \quad v \neq \lambda x.r \vee (\mathsf{fv}(t) \neq \varnothing \wedge \mathsf{fv}(u) \neq \varnothing)}{(t\,u) \Downarrow_w (v\,u)}\ \text{(Arg)}$$

$$\frac{(t[v/x])\,u \Downarrow_w w \quad x \in \mathsf{fv}(t)}{(t\,u)[v/x] \Downarrow_w w}\ \text{(App1)} \qquad \frac{t[u[v/x]/y] \Downarrow_w w \quad x \in \mathsf{fv}(u)}{t[u/y][v/x] \Downarrow_w w}\ \text{(Comp)}$$

$$\frac{t\,(u[v/x]) \Downarrow_w w \quad x \in \mathsf{fv}(u)}{(t\,u)[v/x] \Downarrow_w w}\ \text{(App2)} \qquad \frac{t \Downarrow_w u \quad u[v/x] \Downarrow_w w \quad \mathsf{fv}(v) \neq \varnothing}{t[v/x] \Downarrow_w w}\ \text{(Subst)}$$

$$\frac{t \Downarrow_w w \quad \mathsf{fv}(v) = \varnothing}{(\epsilon_x.t)[v/x] \Downarrow_w w}\ \text{(Erase1)} \qquad \frac{v \Downarrow_f v' \quad t[v'/y][v'/z] \Downarrow_w w \quad \mathsf{fv}(v) = \varnothing}{(\delta_x^{y,z}.t)[v/x] \Downarrow_w w}\ \text{(Copy1)}$$

$$\frac{\epsilon_{x'}.(t[v/x]) \Downarrow_w w \quad x \neq x'}{(\epsilon_{x'}.t)[v/x] \Downarrow_w w}\ \text{(Erase2)} \qquad \frac{\delta_{x'}^{y,z}.(t[v/x]) \Downarrow_w w \quad x \neq x'}{(\delta_{x'}^{y,z}.t)[v/x] \Downarrow_w w}\ \text{(Copy2)}$$

$$\frac{t \not\Downarrow_w \text{ by any other rule}}{t \Downarrow_w t}\ \text{(Axiom)}$$

Fig. 4. Operational semantics of the closed strategy.

$\Downarrow_w$ is indeed what we want: we reduce to weak head normal form, but we perform reduction under abstractions in subterms that will be copied. Note that the above strategy belongs to λ_{caf}. To define a closed strategy in λ_{cf}, which has some advantages from an implementation perspective (Sinot *et al.* 2003), it is sufficient to restrict the rule (*Beta*) to closed functions.

5.2. *Efficiency*

Our calculi live in the same framework as call-by-name and call-by-value λ-calculus, namely they are calculi on terms (as opposed to graphs). Every strategy in this framework has to decide somewhat arbitrarily what to do before a β-reduction (and also in our case, copying): should the argument be reduced first and how much? Call-by-name does no reduction, call-by-value reduces until the argument reaches a weak head normal form, and our closed strategy does additional reduction. These choices have strong consequences in terms of termination and efficiency with certain categories of λ-terms.

As with call-by-value, and in contrast with call-by-name, we choose to do more work while we can still share it. The consequence for call-by-value is that it is more efficient than call-by-name on a class of λ-terms where the argument can be reduced and is used several times in the body of the function. The same holds for closed reduction.

Note that this is not, however, a trivial consequence of the fact that we do ‘more work’ before copying (or almost equivalently before a β-reduction in the implicit sense). For instance, consider the following strategy: take call-by-value, except that before a β-reduction, the argument should be reduced to its full normal form. Then this strategy is (experimentally) less efficient than ours because some unsharing has to be done to reach full normal form. For example, free variables may be copied. Closed strategies avoid that problem.

It is difficult, if not impossible, to find a relevant measure to compare machines implemented in different frameworks. It is as difficult to find a set of terms agreed to be

Table 7. *Benchmarks.*

Term	CR (λ_{caf})	CR (λ_{cf})	CBV	CBN	CBVNF	BOHM
22II	61(9)	61(9)	78(11)	76(12)	98(11)	40(9)
222II	140(19)	140(19)	362(42)	471(60)	342(36)	93(16)
55II	217(33)	217(33)	29723(3913)	31250(4689)	22089(3153)	208(33)
522II	832(109)	832(109)	–	–	–	847(31)
55AI	237(35)	75892(9387)	89098(10163)	84375(10939)	22100(3155)	211(35)
M(55II)I	266(42)	266(42)	41(8)	31(8)	34621(3161)	22(8)
KI(55II)	7(2)	7(2)	29731(3915)	7(2)	34585(3155)	4(2)

relevant for comparison of efficiency. However, we give a small set of experimental results to illustrate our point. The terms chosen for comparisons are built from the classical combinators $\mathbf{I} = \lambda x.x$, $\mathbf{K} = \lambda x.\lambda y.x$, $\mathbf{n} = \lambda f.\lambda x.f^n\, x$ and the somewhat *ad hoc* combinators $\mathbf{A} = \lambda x.(\lambda y.yx)\mathbf{I}$ and $\mathbf{M} = \lambda x.\lambda y.(\mathbf{KI}x)(\mathbf{KI}xy)$.

Table 7 gives a comparison between the closed strategies of λ_{caf} and λ_{cf} and standard call-by-value and call-by-name evaluators. We also give a comparison with CBVNF, a call-by-value where the argument is reduced to full normal form before β-reduction. Note that this last strategy requires α-conversion. We show the total number of steps of these evaluators (including stack manipulations), and the number of β-reductions between round brackets. The closed strategies have been implemented using *director strings*, which internalise the information needed about free variables. More precisely, this information becomes part of the pattern, and thus each step has a constant algorithmic cost. We refer the reader to Sinot *et al.* (2003) for these implementation details. In the final column of the table we give the results for BOHM (Asperti *et al.* 1996), which remains the most efficient evaluator to date, and also indicates the optimal number of β-reductions. An entry '–' indicates that we were unable to compute the result.

The first series of examples is intended to show the asymptotic behaviour of the compared strategies when the size of the term increases. Both closed strategies defined in previous section are equivalent on these terms because every redex present or generated has a closed function part. Our point is that the better sharing allowed by closed reduction dramatically improves the efficiency on large terms. In fact, our machine is quite comparable to the best known evaluator for such terms, namely BOHM (Asperti *et al.* 1996), though it relies on a much simpler framework.

Besides this classical benchmark, we give three admittedly rather contrived terms to illustrate different points:

— The strategy based on λ_{caf} offers, of course, a better sharing than λ_{cf} in the presence of λ_{ca}-only redexes.

— Our strategies may perform more work than necessary, when every copy of a term will be erased. However, the occurrence of such terms in practical situations is questionable.

— Apart from the previous situation, we may avoid doing useless work, as opposed to call-by-value for instance, that is, we also have some of the advantages of call-by-name.

6. Conclusions

In this paper we have proposed a family of calculi with explicit substitutions and without α-conversion, and in which the essential principle is that the reduction process should be simple and capture the shortest reduction paths. There are no lists of substitutions, or operations to manipulate them. Both erasing and copying are explicit, in order to guide the substitution to the places where it is required and erase it as soon as possible when it is not needed, following the natural intuitions for the propagation of substitutions.

We have developed a series of abstract machines for closed reduction based on these calculi, and the experimental results suggest that closed reduction behaves better than the usual weak strategies. Moreover, these results indicate that they compare well with state-of-the-art evaluators based on considerably more complicated concepts.

Acknowledgements

We thank the referees for their valuable comments, and Gopalan Nadathur for pointing out some relevant references.

References

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

Abramsky, S. (1993) Computational Interpretations of Linear Logic. *Theoretical Computer Science* **111** 3–57.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. and Wadler, P. (1995) A call-by-need lambda calculus. In: *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, ACM Press 233–246.

Asperti, A., Giovannetti, C. and Naletto, A. (1996) The Bologna Optimal Higher-order Machine. *Journal of Functional Programming* **6** (6) 763–810.

Barendregt, H. P. (1984) The Lambda Calculus: Its Syntax and Semantics. *Studies in Logic and the Foundations of Mathematics* **103**, North-Holland Publishing Company, second, revised edition.

Barendregt, H. P. (1992) Lambda calculi with types. In: Abramsky, S., Gabbay, D. and Maibaum, T. S. E. (eds.) *Handbook of Logic in Computer Science*, Oxford University Press **2** (2) 117–309.

Çağman, N. and Hindley, J. R. (1998) Combinatory weak reduction in lambda calculus. *Theoretical Computer Science* **198** (1–2) 239–249.

David, R. and Guillaume, B. (2001) A λ-calculus with explicit weakening and explicit substitution. *Mathematical Structure in Computer Science* **11** (1) 169–206.

de Bruijn, N. G. (1972) Lambda calculus notation with nameless dummies. *Indagationes Mathematicae* **34** 381–392.

Dowek, G., Hardin, T. and Kirchner, C. (2001) HOL-$\lambda\sigma$: an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science* **11** (1) 21–45.

Fernández, M. and Mackie, I. (1999) Closed reduction in the λ-calculus. In: Flum, J. and Rodríguez-Artalejo, M. (eds.) Proceedings of Computer Science Logic (CSL'99). *Springer-Verlag Lecture Notes in Computer Science* **1683** 220–234.

Girard, J.-Y. (1989) Geometry of interaction 1: Interpretation of System F. In: Ferro, R., Bonotto, C., Valentini, S. and Zanardo, A. (eds.) Logic Colloquium 88. *Studies in Logic and the Foundations of Mathematics*, North Holland **127** 221–260.

Hardin, T., Maranget, L. and Pagano, B. (1998) Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming* **8** (2) 131–176.

Klop, J.-W. (1980) Combinatory Reduction Systems. *Mathematical Centre Tracts* **127**, Mathematischen Centrum, 413 Kruislaan, Amsterdam.

Lescanne, P. and Rouyer-Degli, J. (1995) The calculus of explicit substitutions lambda-upsilon. Technical Report RR-2222, INRIA.

Melliès, P.-A. (1995) Typed lambda-calculi with explicit substitutions may not terminate. In: Proceedings of the Second International Conference on Typed Lambda Calculi and Applications. *Springer-Verlag Lecture Notes in Computer Science* **902** 328–334.

Muñoz, C. (1996) Confluence and preservation of strong normalisation in an explicit substitutions calculus (extended abstract). In: *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, IEEE Computer Society Press.

Nadathur, G. (1999) A fine-grained notation for lambda terms and its use in intensional operations. *Journal of Functional and Logic Programming* **1999** (2).

Newman, M. (1942) On theories with a combinatorial definition of "equivalence". *Annals of Mathematics* **43** (2) 223–243.

Plotkin, G. (1977) LCF considered as a programming language. *Theoretical Computer Science* **5** (3) 223–256.

Rose, K. (1996) Explicit substitution – tutorial and survey. Lecture Series LS-96-3, BRICS, Dept. of Computer Science, University of Aarhus, Denmark.

Rosen, B. (1973) Tree-manipulating systems and Church–Rosser theorems. *Journal of the ACM* **20** (1) 160–187.

Sinot, F.-R., Fernández, M. and Mackie, I. (2003) Efficient reductions with director strings. In: Nieuwenhuis, R. (ed.) Proceedings of Rewriting Techniques and Applications (RTA'03). *Springer-Verlag Lecture Notes in Computer Science* **2706** 46–60.

Yoshida, N. (1994) Optimal reduction in weak lambda-calculus with shared environments. *Journal of Computer Software* **11** (6) 3–18.