Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 204 (2008) 163–179
www.elsevier.com/locate/entcs

Minimality in a Linear Calculus with Iteration

Sandra Alves[a] Mário Florido[a] Ian Mackie[b]
François-Régis Sinot[a]

[a] *Universidade do Porto, DCC/LIACC, Rua do Campo Alegre 1021–1051, Porto, Portugal*
[b] *LIX, CNRS UMR 7161, École Polytechnique, 91128 Palaiseau, France*

Abstract

System $\mathcal{L}$ is a linear version of Gödel's System $\mathcal{T}$, where the λ-calculus is replaced with a linear calculus; or alternatively a linear λ-calculus enriched with some constructs including an iterator. There is thus at the same time in this system a lot of freedom in reduction and a lot of information about resources, which makes it an ideal framework to start a fresh attempt at studying reduction strategies in λ-calculi. In particular, we show that call-by-need, the standard strategy of functional languages, can be defined directly and effectively in System $\mathcal{L}$, and can be shown minimal among weak strategies.

Keywords: linearity, iteration, System T, strategies, efficiency, minimality, optimality

1 Introduction

Gödel's System $\mathcal{T}$ is an extremely powerful calculus: essentially anything that we want to compute can be expressed [14]. A *linear* variant of this well-known calculus, called System $\mathcal{L}$, was introduced in [1], and shown to be every bit as expressive as System $\mathcal{T}$. The novelty of System $\mathcal{L}$ is that it is based on the linear λ-calculus, and all duplication and erasing can be done through an encoding using the iterator.

There are many well-known, and well-understood, strategies for reduction in the (pure) λ-calculus. When investigating deeper into the structure of terms, we get a deeper understanding of reduction (and *vice versa*). For instance, calculi with explicit resource management or explicit substitution enjoy a more fine-grained reduction. In a similar way, System $\mathcal{L}$ splits the usual λ in two different constructs: a binder, able to generate a substitution, and an iterator able to erase or copy its argument. This entails a finer control of these fundamentally different issues, which are intertwined in the λ-calculus. Having a calculus which offers at the same time a lot of freedom in reduction and a lot of information about resources makes it an ideal framework to start a fresh attempt at studying reduction strategies in λ-calculi.

doi:10.1016/j.entcs.2008.03.060

This paper is a first step towards a thorough study of reduction strategies for System $\mathcal{L}$. The main contributions of this paper are:

(i) we present, and compare, different ways of writing the reduction rules associated to iterators;

(ii) we define a weak reduction relation for System $\mathcal{L}$ (we call this new system *weak System* $\mathcal{L}$) similar to weak reduction used in the implementation of functional programming languages, where reduction is forbidden inside abstractions;

(iii) we present reduction strategies for the weak reduction relation: call-by-name, call-by-value, and call-by-need (emphasising this last one), proving that they are indeed strategies in a technical sense. Since neededness is usually undecidable, extra features (like sharing graphs, environments, explicit substitutions) are generally added to actually implement call-by-need. In contrast, for System $\mathcal{L}$, we can define call-by-need *within the calculus* in an *effective* way.

(iv) we give a proof of minimality of the call-by-need strategy. It is well-known that there exists no computable minimal strategy for the λ-calculus [5]. One of the main contributions in this paper is a computable (and effective) minimal strategy for weak System $\mathcal{L}$.

The rest of this paper is structured as follows. In the next section we present some related work. In Section 3, we recall some background on rewriting and System $\mathcal{L}$. In Section 4 we discuss the issues about strategies and choices. In Section 5 we define weak reduction in System $\mathcal{L}$, and study different weak strategies in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

In [4] (see also [5, Chapter 13]), a notion of *L-1-optimal* (or *minimal* [1], in this paper) strategy for the λ-calculus is defined as a normalising strategy, minimal with respect to the length of paths in the terms reduction graph. It was shown that there exists no computable optimal L-1-strategy for the λ-calculus. One of the main contributions in this paper is a set of computable minimal strategies for weak System $\mathcal{L}$ (a version of System $\mathcal{L}$ where reduction is forbidden inside abstractions).

Weak System $\mathcal{L}$ is very much inspired in weak λ-calculi [20,21,3,6], weak reductions for functional programming languages [23] and lazy evaluation models [18,27]. In all these works reduction is forbidden inside abstractions and lazy evaluation is achieved by enlarging the calculus with extra syntax (graph reductions [27], explicit let bindings [3,20] or explicit heap [18]) to express sharing of subterm evaluation. In this paper we present a set of minimal strategies for weak System $\mathcal{L}$ with the same features as lazy evaluation: there is no loss of sharing except inside abstractions, and we only reduce terms that are actually used, but we insist that our definition is effective, within the calculus. This is possible due to the finer control of linear substitution and copying using the iterator, as opposed to the λ-calculus where these

[1] In [15], the notion of minimality is different.

different issues are mixed up.

The notion of reduction in System $\mathcal{L}$, called closed reduction, is already weak in the sense that it imposes strong constraints on the application of reduction rules (see [11]). In this paper we define a weaker form of reduction: closed reduction without reduction inside abstractions. The main motivation for this further constraint is to define a simple computable minimal strategy for weak System $\mathcal{L}$.

Some previous works studied the relation between recursion and iteration in System $\mathcal{T}$ [22,8,24] showing that, in many cases, recursion is more efficient than iteration. We choose to use iteration in System $\mathcal{L}$ because it avoids the duplication of a variable, and it is then more suitable within a linear setting, such as System $\mathcal{L}$. Thus, in this paper, efficiency should be understood in this setting: a linear calculus with iteration. Another reason why we insist on using a linear discipline (and thus an iterator instead of a recursor) is that efficiency in a linear calculus, such as System $\mathcal{L}$, can be measured by the number of steps to normalise terms, because each reduction either decreases the size of the term (by linear β-reduction) or increases it by adding the size of the iterated function (applying the iteration reduction rule). This is no longer true in a non-linear setting, such as System $\mathcal{T}$, where the number of reduction steps cannot be used as a measure of efficiency (see [9] for a detailed discussion about the problems of naively using reduction steps as a measure of efficiency in a non-linear calculus).

In general, call-by-need (and even minimal strategies) may copy expressions in some situations (for example inside abstractions). Sharing of subterms across different instantiations of bound variables is addressed by optimal reduction strategies [19,17,12,21,28]. Although this line of research applied to System $\mathcal{L}$ is a promising one, optimal reduction in this sense is not an issue in this paper: here we follow the weak reduction approach, as is standard in the implementation of functional languages [23].

3 Background

3.1 Rewriting

We briefly recall some definitions, and refer the reader to [25] for more details.

Definition 3.1 An *abstract reduction system* (ARS) is a directed graph $(A, \to)$. We write $t \to u$ if there is an edge in $\to$ from t to u. The reflexive transitive closure of $\to$ is $\to^*$, and $\leftarrow$ is the inverse relation of $\to$. A normal form is a $t \in A$ such that there exists no u such that $t \to u$. We also write $t \to^n u$ if $t \underbrace{\to \cdots \to}_{n} u$.

Definition 3.2 $\to$ is said to have the *diamond property* [2] if, whenever $u_1 \leftarrow t \to u_2$ with $u_1 \neq u_2$, there exists a v such that $u_1 \to v \leftarrow u_2$. $\to$ is said to be *confluent* if $\to^*$ has the diamond property. $\to$ is said to be *strongly normalising* if there is no object admitting an infinite $\to$-reduction path.

[2] This property is called CR[1] in [25, Ex. 1.3.18], where "diamond property" means something else.

Definition 3.3 A *strategy* for an ARS $(A, \to)$ is a sub-ARS $(A, \to_\triangleright)$ of $(A, \to)$ (i.e. such that $\to_\triangleright \subseteq \to$) with the same normal forms.

Note that this definition is more liberal than others (e.g. [5]), in the sense that a strategy is not required to be deterministic.

Definition 3.4 A $\to$-strategy $\to_\triangleright$ is *normalising* if all $\to_\triangleright$-reduction paths starting from an object, which admits a finite $\to$-reduction path to normal form, are finite. It is *minimal* [3] if the length of any $\to_\triangleright$-reduction from an object a to a normal form b is minimal among all possible $\to$-reductions from a to b.

We do not want to recall too much about higher-order rewriting. The systems defined in this paper will fit the framework of *context-sensitive conditional expression reduction systems* (CERS) [16]. In particular, the notion of *residuals* make sense in these systems [7].

Definition 3.5 A redex is *needed* if some residual of it must be fired in any reduction to normal form.

In [26], van Oostrom gives a method to reduce the global problem of proving that a strategy is minimal (or maximal), to a verification of certain properties of local reduction diagrams. To avoid recalling all that work here, we combine some parts of Theorems 1 and 2 of [26], as the following theorem, which will be used to show the minimality of *call-by-need* among weak strategies (Theorem 6.8).

Theorem 3.6 *Let $\to_\triangleright$ be a $\to$-strategy. If, whenever $s \triangleleft\!\!- t \to u$, either u admits an infinite $\to_\triangleright$-reduction or there exists an r such that $s \to^n r \triangleleft\!\!-^m u$ with $n \leq m$, then $\to_\triangleright$ is normalising and minimal.*

3.2 System $\mathcal{L}$

In this section we recall the syntax and reduction rules of System $\mathcal{L}$ [1]. Table 1 gives the syntax of System $\mathcal{L}$. The set of linear λ-terms is built from: variables $x, y, \ldots$; linear abstraction $\lambda x.t$, where $x \in \mathsf{fv}(t)$; and linear application $t\,u$, where $\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$. Here $\mathsf{fv}(t)$ denotes the set of free variables of t. These conditions ensure that terms are syntactically linear (variables occur exactly once in each term).

Since we are in a linear calculus, we cannot have the usual notion of pairs and projections; instead, we have pairs and splitters which use both projections, as shown in Table 1. A simple example is the swapping function (see below).

Finally, we have booleans true and false, with a linear conditional; and numbers (built from $\mathsf{0}$ and S), with a linear iterator. $\mathsf{S}^n\mathsf{0}$ denotes n applications of S to $\mathsf{0}$.

The dynamics of the system is given by the set of conditional reduction rules in Table 2. The system fits in the framework of *context-sensitive conditional expression reduction systems* (CERS) [16]. The conditions on the rewrite rules ensure that *Beta* only applies to redexes where the argument is a closed term (which implies that α-conversion is not needed to implement substitution), and only closed functions are

[3] Minimality is called *L-1-optimality* in [5] and simply *optimality* in [28].

iterated. Table 2 gives the reduction rules for System $\mathcal{L}$, substitution is a meta-operation defined as usual. Reductions can take place in any context where the conditions are satisfied.

Construction	**Variable Constraint**	**Free Variables (fv)**
$0, \mathsf{true}, \mathsf{false}$	–	$\varnothing$
$\mathsf{S}\ t$	–	$\mathsf{fv}(t)$
$\mathsf{iter}\ t\ u\ w$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \mathsf{fv}(u) \cap \mathsf{fv}(w) = \varnothing$ $\mathsf{fv}(t) \cap \mathsf{fv}(w) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u) \cup \mathsf{fv}(w)$
x	–	$\{x\}$
tu	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
$\lambda x.t$	$x \in \mathsf{fv}(t)$	$\mathsf{fv}(t) \setminus \{x\}$
$\langle t, u\rangle$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
$\mathsf{let}\ \langle x, y\rangle = t\ \mathsf{in}\ u$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing; x, y \in \mathsf{fv}(u); x \neq y$	$\mathsf{fv}(t) \cup (\mathsf{fv}(u) \setminus \{x, y\})$
$\mathsf{cond}\ t\ u\ w$	$\mathsf{fv}(u) = \mathsf{fv}(w); \mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$

Table 1
Terms

Name	Reduction		Condition
Beta	$(\lambda x.t)\, u$	$\longrightarrow t[u/x]$	$\mathsf{fv}(u) = \varnothing$
Let	$\mathsf{let}\ \langle x, y\rangle = \langle t, u\rangle\ \mathsf{in}\ w$	$\longrightarrow (w[t/x])[u/y]$	$\mathsf{fv}(t) = \mathsf{fv}(u) = \varnothing$
Cond	$\mathsf{cond}\ \mathsf{true}\ u\ w$	$\longrightarrow u$	
Cond	$\mathsf{cond}\ \mathsf{false}\ u\ w$	$\longrightarrow w$	
Iter	$\mathsf{iter}\ 0\ u\ w$	$\longrightarrow u$	$\mathsf{fv}(w) = \varnothing$
Iter	$\mathsf{iter}\ (\mathsf{S}\ t)\ u\ w$	$\longrightarrow w(\mathsf{iter}\ t\ u\ w)$	$\mathsf{fv}(t) = \mathsf{fv}(w) = \varnothing$

Table 2
Closed reduction

We give some examples to illustrate the system:

- Swapping: $\mathsf{swap} = \lambda x.\mathsf{let}\ \langle y, z\rangle = x\ \mathsf{in}\ \langle z, y\rangle$.
- Erasing numbers[4]: although we are in a linear system, we can erase (more precisely: consume) numbers by using them in iterators.

$$\begin{aligned} \mathsf{fst} &= \lambda x.\mathsf{let}\ \langle t, u\rangle = x\ \mathsf{in}\ \mathsf{iter}\ u\ t\ (\lambda z.z) \\ \mathsf{snd} &= \lambda x.\mathsf{let}\ \langle t, u\rangle = x\ \mathsf{in}\ \mathsf{iter}\ t\ u\ (\lambda z.z) \end{aligned}$$

- Copying numbers: $C = \lambda x.\mathsf{iter}\ x\ \langle 0, 0\rangle\ (\lambda x.\mathsf{let}\ \langle a, b\rangle = x\ \mathsf{in}\ \langle \mathsf{S}\ a, \mathsf{S}\ b\rangle)$ takes a number n and returns a pair $\langle n, n\rangle$.
- Addition: $\mathsf{add} = \lambda mn.\mathsf{iter}\ m\ n\ (\lambda x.\mathsf{S}\ x)$
- Multiplication: $\lambda mn.\mathsf{iter}\ m\ 0\ (\mathsf{add}\ n)$
- Predecessor: $\lambda n.\mathsf{fst}(\mathsf{iter}\ n\ \langle 0, 0\rangle\ (\lambda x.\mathsf{let}\ \langle t, u\rangle = C(\mathsf{snd}\ x)\ \mathsf{in}\ \langle t, \mathsf{S}\ u\rangle))$

[4] Some terms t can be erased with $\mathsf{iter}\ 0\ t\ u$, but only those such that the construction is well-typed.

- Ackermann: $ack(m,n) = (\mathsf{iter}\ m\ (\lambda x.\mathsf{S}\ x)\ (\lambda gu.\mathsf{iter}\ (\mathsf{S}\ u)\ (\mathsf{S}\ \mathsf{0})\ g))\ n$

System $\mathcal{L}$ is essentially a typed calculus (this further restriction still fits in the framework of CERSs), and most of the properties stated in the remainder of this paper rely on this in a crucial way, although some properties are also valid in the untyped calculus (this will always be stated explicitly). We write $\Gamma \vdash_{\mathcal{L}} t : A$ if the term t has type A in the environment Γ, where A is a *linear type*: $A, B ::= \mathsf{Nat} \mid \mathsf{Bool} \mid A \multimap B \mid A \otimes B$ where Nat and Bool are the types of numbers and booleans. The full details of the type system are not essential for the remainder of this paper and are thus omitted. The type system, and further details, including a type reconstruction algorithm, can be found in [2].

Lemma 3.7 *System $\mathcal{L}$ is an* orthogonal *[16] CERS.*

Proof. Apart from easy syntactic verifications, we have to notice that all descendants of a redex are redexes. This comes from the preservation of linearity constraints, subject reduction [1], and the fact that a closed term cannot become open during reduction. □

As a corollary, this reproves confluence of System $\mathcal{L}$ [1]. We also recall from [1] that typable terms are strongly normalising and:

Theorem 3.8 (Adequacy) *If t is closed and typable, then one of the following holds:*

- $\vdash_{\mathcal{L}} t : \mathsf{Nat}$ *and* $t \to^* \mathsf{S}^n\,\mathsf{0}$ *for some integer n;*
- $\vdash_{\mathcal{L}} t : \mathsf{Bool}$ *and either* $t \to^* \mathsf{true}$ *or* $t \to^* \mathsf{false}$*;*
- $\vdash_{\mathcal{L}} t : A \multimap B$ *and* $t \to^* \lambda x.u$ *for some term u;*
- $\vdash_{\mathcal{L}} t : A \otimes B$ *and* $t \to^* \langle u, w\rangle$ *for some terms u, w.*

(For a proof, we refer the reader to the proof of Theorem 4 in [1].)

4 Intuitions and Choices

Here we emphasise what the exact choices are when defining reduction strategies in System $\mathcal{L}$, in particular, from an efficiency point of view.

Efficiency.
Semantically, we have: $\mathsf{iter}\ (\mathsf{S}^n\mathsf{0})\ u\ w = w^n(u)$ (i.e. n copies of w applied to u). However as shown in the given rewrite rules, we actually make use of $n+1$ occurrences of w, and then throw one away. To circumvent this defect we change the definition in order to stop at $\mathsf{S}\,\mathsf{0}$ rather than $\mathsf{0}$:

$$\begin{array}{lll} \mathsf{iter}\ \mathsf{0}\ u\ w & \to u & \mathsf{fv}(w) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}\,\mathsf{0})\ u\ w & \to w\,u & \mathsf{fv}(w) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ w & \to w(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w) & \mathsf{fv}(t) = \mathsf{fv}(w) = \varnothing \end{array}$$

There is no strong motivation behind the condition on the second rule, except to ensure the conservativity of the new rules with respect to System $\mathcal{L}$. It is clear that the last two rules split the previous one, and because there are only two cases to consider in the pattern matching (S and $\mathsf{0}$) then this will not have any consequences on any of the results of System $\mathcal{L}$, which can be stated as follows:

Proposition 4.1 *Let us call* $\xrightarrow{\mathsf{old}}$ *the reduction relation defined in Section 3.2 and* $\xrightarrow{\mathsf{new}}$ *the reduction relation with the modified rules for* iter *above. Then:*

(i) *if* $t \xrightarrow{\mathsf{new}} u$, *then* $t \xrightarrow{\mathsf{old}}{}^{n} u$ *with* $n = 1$ *or* $n = 2$;

(ii) *if* $t \xrightarrow{\mathsf{old}} u$, *then there exists a* w *such that* $t \xrightarrow{\mathsf{old}}{}^{*} \xrightarrow{\mathsf{new}} w$ *and* $u \xrightarrow{\mathsf{old}}{}^{*} w$;

(iii) v *is a* $\xrightarrow{\mathsf{new}}$*-normal form if and only if* v *is a* $\xrightarrow{\mathsf{old}}$*-normal form;*

(iv) $\xrightarrow{\mathsf{new}}$ *is strongly normalising;*

(v) *if* v *is a normal form (for* $\xrightarrow{\mathsf{old}}$ *and* $\xrightarrow{\mathsf{new}}$*), then* $t \xrightarrow{\mathsf{new}}{}^{*} v$ *if and only if* $t \xrightarrow{\mathsf{old}}{}^{*} v$;

(vi) $\xrightarrow{\mathsf{new}}$ *is confluent.*

Proof.

(i) Straightforward.

(ii) The problem in this case is that a $\xrightarrow{\mathsf{old}}$-redex is not necessarily a $\xrightarrow{\mathsf{new}}$-redex: if we have $\mathsf{iter}\ (\mathsf{S}t)\ u\ w \xrightarrow{\mathsf{old}} w(\mathsf{iter}\ t\ u\ w)$, we know that t is closed, and by adequacy (Theorem 3.8), $t \xrightarrow{\mathsf{old}}{}^{*} \mathsf{S}^n\,\mathsf{0}$ for some $n \geq 0$, so that, in the case $n \geq 1$, $\mathsf{iter}\ (\mathsf{S}t)\ u\ w \xrightarrow{\mathsf{old}}{}^{*} \mathsf{iter}\ (\mathsf{S}^{n+1}\,\mathsf{0})\ u\ w \xrightarrow{\mathsf{new}} w(\mathsf{iter}\ (\mathsf{S}^n\,\mathsf{0})\ u\ w)$ and $w(\mathsf{iter}\ t\ u\ w) \xrightarrow{\mathsf{old}}{}^{*} w(\mathsf{iter}\ (\mathsf{S}^n\,\mathsf{0})\ u\ w)$, and similarly in the case $n = 0$.

(iii) Consequence of Points i and ii.

(iv) Consequence of Point i and strong normalisation: suppose we have an infinite $\xrightarrow{\mathsf{new}}$-reduction, then we obtain an infinite $\xrightarrow{\mathsf{old}}$-reduction.

(v) The "only if" part is a consequence of Point i. For the "if" part, assume $t \xrightarrow{\mathsf{old}}{}^{*} v$ with v a normal form. Using Point iv, consider w such that $t \xrightarrow{\mathsf{new}}{}^{*} w$ and w is a normal form. By the "only if" part of this point, we know that $t \xrightarrow{\mathsf{old}}{}^{*} w$, thus $v = w$ by the unicity of normal forms in System $\mathcal{L}$ (consequence of the confluence of System $\mathcal{L}$).

(vi) Assume $t \xrightarrow{\mathsf{new}}{}^{*} u_1$ and $t \xrightarrow{\mathsf{new}}{}^{*} u_2$. By Point i, we also have $t \xrightarrow{\mathsf{old}}{}^{*} u_1$ and $t \xrightarrow{\mathsf{old}}{}^{*} u_2$. By confluence, there is a w such that $u_1 \xrightarrow{\mathsf{old}}{}^{*} w$ and $u_2 \xrightarrow{\mathsf{old}}{}^{*} w$. Using strong normalisation, let v be the $\xrightarrow{\mathsf{old}}$-normal form of w. Then, using Point v, $u_1 \xrightarrow{\mathsf{new}}{}^{*} v$ and $u_2 \xrightarrow{\mathsf{new}}{}^{*} v$.

□

Of course, if we are considering an untyped calculus, where non-terminating computations can be represented, then $\xrightarrow{\mathsf{old}}$ and $\xrightarrow{\mathsf{new}}$ are not equivalent as we now require to force more evaluation to complete the pattern matching: let Δ be the term $(\lambda x.\mathsf{iter}\ (\mathsf{S}^2\mathsf{0})\ (\lambda x_1 x_2.x_1 x_2)\ (\lambda z.zx))$ and Ω be the non-terminating (untyped) term $\Delta\Delta$. Then $\mathsf{iter}\ (\mathsf{S}\ \Omega)\ I\ (\lambda x.\mathsf{iter}\ \mathsf{0}\ I\ x)$ will terminate with the old system but

not with the new. In other words, iter is now more strict in its first argument. We use this version, because efficiency is now an issue. From now on, $\rightarrow$ means $\xrightarrow{\mathsf{new}}$.

Alternative iteration.
There are two ways of writing the rules for an iterator. The one given above (both $\xrightarrow{\mathsf{old}}$ and $\xrightarrow{\mathsf{new}}$) which we shall call *outer-iter* (and denote $\rightarrow_{\mathsf{out}}$) and also this one, which we shall call *inner-iter*:

$$\begin{array}{lll}
\mathsf{iter}\ \mathsf{0}\ u\ w & \rightarrow_{\mathsf{in}}\ u & \mathsf{fv}(w) = \varnothing \\
\mathsf{iter}\ (\mathsf{S}\,\mathsf{0})\ u\ w & \rightarrow_{\mathsf{in}}\ w\,u & \mathsf{fv}(w) = \varnothing \\
\mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ w & \rightarrow_{\mathsf{in}}\ \mathsf{iter}\ (\mathsf{S}\,t)\ (w\,u)\ w & \mathsf{fv}(t) = \mathsf{fv}(w) = \varnothing
\end{array}$$

We remark the relation with *fold right* and *fold left* for lists in functional programming. These operators encapsulate recursion patterns on lists, in the same way as an iterator on numbers encapsulates recursion patterns on numbers. The difference between *foldl* and *foldr* is simply the order in which the elements of the lists are accessed: left-to-right, or right-to-left. A left-to-right approach can start working on elements of lists, even infinite lists, whereas the right-to-left approach works well in the finite case (i.e. it is strict in the list). The same reasoning applies to our iterator. Of course, the origins of these operators on lists are indeed iterators on numbers (primitive recursive schemes).

With the inner-iter reduction policy, iter is strict in its first argument. For example, in an untyped calculus, if the number is not terminating, then neither is the iter (irrespectively of the evaluation order). This will not be a problem in System $\mathcal{L}$ because it is a strongly normalising calculus.

Now we have a whole collection of strategies to look at: leftmost and outermost with each of the alternatives gives different strategies. For instance, if we use inner-iter with leftmost reduction, then we get iter evaluated first. If we have outermost with outer-iter, then we compute the applications first, etc. And of course, we are interested in finding the "best" combination.

The next results show that extending System $\mathcal{L}$ with this new form of iteration does not change the calculus itself (although it gives one more way to reduce iterators), and that both ways of reducing iterators essentially use the same number of steps when the number of iterations is known.

Lemma 4.2 *For any number $n \geq 1$, any term u and any closed term w, we have:* $\mathsf{iter}\ (\mathsf{S}^n\,\mathsf{0})\ u\ w \xrightarrow{\mathsf{out}}{}^{n} w^n(u) \,{}^{n}\!\xleftarrow{\mathsf{in}}\ \mathsf{iter}\ (\mathsf{S}^n\,\mathsf{0})\ u\ w$.

Proof. Straightforward by induction on n. □

Theorem 4.3 *If we add the rules corresponding to inner-iteration ($\rightarrow_{\mathsf{in}}$) to reduction rules of System $\mathcal{L}$ ($\rightarrow_{\mathsf{out}}$), we get a new system ($\rightarrow_{\mathsf{i+o}} = \rightarrow_{\mathsf{out}} \cup \rightarrow_{\mathsf{in}}$) with the following properties:*

(i) *subject reduction;*

(ii) *strong normalisation;*

(iii) *confluence;*

(iv) *the normal form of a term is the same using* $\to_{\mathsf{out}}$, $\to_{\mathsf{in}}$ *or* $\to_{\mathsf{i+o}}$.

Proof.

(i) *Subject reduction:* Straightforward.

(ii) *Strong normalisation:* Adapt the proof for System $\mathcal{T}$ based on reducibility [14]. Let $\nu(t)$ bound the length of every normalisation sequence beginning with t, and let $l(t)$ be the maximal number of symbols in all reachable normal forms of t (there are finitely many thanks to Koenig's lemma). We prove that if t, u and w are reducible, then $\mathsf{iter}\ t\ u\ w$ is reducible, by induction on $\nu(t) + \nu(w^n(u)) + \nu(w) + l(t)$, where $\mathsf{S}^n(\mathsf{0})$ is the normal form of t.

(iii) *Confluence:* Let us first note that, because of inner-iter, we lose confluence of the untyped calculus. For example

$$w(\mathsf{iter}\ \mathsf{true}\ u\ w) \xleftarrow{\mathsf{out}} \mathsf{iter}\ \mathsf{S}(\mathsf{true})\ u\ w \xrightarrow{\mathsf{in}} \mathsf{iter}\ \mathsf{true}\ (w\,u)\ w.$$

Now for typed terms (System $\mathcal{L}$), let us consider the only critical pair:

$$w(\mathsf{iter}\ t\ u\ w) \xleftarrow{\mathsf{out}} \mathsf{iter}\ \mathsf{S}(t)\ u\ w \xrightarrow{\mathsf{in}} \mathsf{iter}\ t\ (w\,u)\ v$$

Since t is closed and typable, then $t \to^* (\mathsf{S}^n\mathsf{0})$, therefore

$$\begin{array}{ccc} w(\mathsf{iter}\ t\ u\ w) & & \mathsf{iter}\ t\ (w\,u)\ w \\ \ast\downarrow & & \ast\downarrow \\ w(\mathsf{iter}\ (\mathsf{S}^n\mathsf{0})\ u\ w) & & \mathsf{iter}\ (\mathsf{S}^n\mathsf{0})\ (w\,u)\ w \\ \ast\downarrow & & \ast\downarrow \\ w(w^n(u)) & = & w^n(w\,u) \end{array}$$

The result follows using Newman's Lemma.

(iv) *Normal forms:* $\to_{\mathsf{out}}$ and $\to_{\mathsf{in}}$ can be seen as strategies of $\to_{\mathsf{i+o}}$, i.e. the notion of normal form is the same for the three reductions. For instance, consider a term t, v a $\to_{\mathsf{i+o}}$-normal form of t and w a $\to_{\mathsf{in}}$-normal form of t (both exist because $\to_{\mathsf{i+o}}$ is strongly normalising and $\to_{\mathsf{in}} \subset \to_{\mathsf{i+o}}$). But w is also a $\to_{\mathsf{i+o}}$-normal form of t, hence $v = w$ since $\to_{\mathsf{i+o}}$ is confluent (unicity of normal forms).

□

Iteration vs. β-reduction.

In the linear λ-calculus, where each bound variable occurs exactly once, it is known that all computation is useful and is used exactly once. In System $\mathcal{L}$, this is true at the level of abstraction, but we have the power of copying and erasing at the level of the iterators. We therefore claim that the choice at the level of β-reduction is inessential; what only matters is the choice in the iterator.

Here we present several reduction strategies for iterators, following their counterpart definitions for β-reductions in the λ-calculus and functional programming languages. These reduction strategies are defined for System $\mathcal{L}$ (where every term is linear), thus the only problematic reductions are in the iterator case.

Basically, we have the choice to reduce as much as possible inside iterators before firing them, or not. But we also have the choice to give the preference to outer-iter or to inner-iter. In fact, since the inner-iter reduction policy makes iter strict, it makes a lot more sense to use either call-by-name and outer-iter together, or call-by-value and inner-iter. Below, we only show the rules for the iterator, assuming that it is properly lifted to any context.

Outer iteration by name.
Iteration by name reduces the leftmost outermost iterator first. It is closely related to Engelfriet and Schmidt's outside-in derivation for context-free grammars or first-order recursion equations [10].

$$\begin{array}{lll} \mathsf{iter}\ \mathsf{0}\ u\ w & \to u & \mathsf{fv}(w) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}\,\mathsf{0})\ u\ w & \to w\,u & \mathsf{fv}(w) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ w & \to w(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w) & \mathsf{fv}(t) = \mathsf{fv}(w) = \varnothing \end{array}$$

There is no syntactical constraint on w, so that outermost reduction is possible.

Inner iteration by value.
Iteration by value reduces leftmost innermost iterators first. It is closely related to Engelfriet and Schmidt's inside-out derivations.

$$\begin{array}{lll} \mathsf{iter}\ \mathsf{0}\ u\ v & \to u & \mathsf{fv}(v) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}\,\mathsf{0})\ u\ v & \to v\,u & \mathsf{fv}(v) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ v & \to \mathsf{iter}\ (\mathsf{S}\,t)\ (v\,u)\ v & \mathsf{fv}(t) = \mathsf{fv}(v) = \varnothing \end{array}$$

where v is some notion of normal form (in the sequel, it will be that of value).

5 The Weak System $\mathcal{L}$

5.1 Weakness of System $\mathcal{L}$ reduction

Although reduction in System $\mathcal{L}$ is allowed in any context, in particular under λ-abstractions, it is already somehow weaker than usual strong reduction for the λ-calculus, due to the use of closed reduction (free variable conditions on the rules). In particular, normal forms may still contain iterators. Note that we can however always compute the weak head normal forms of closed terms (see [1]).

We note the following:

- Normal forms of closed terms of functional type may contain iterators. For instance, $T = \lambda x.\mathsf{iter}\ (\mathsf{S}^2\mathsf{0})\ I\ x$ is a normal form.

- We also remark that T could be an argument to a function, and thus values are not the normal forms we could think of, even if we allow reduction under an abstraction. In other words, there is no strategy that will always allow us to avoid copying an iterator. For instance, in $\mathsf{iter}\ (\mathsf{S}^2 0)\ (\lambda x.x)\ (\lambda x.\mathsf{iter}\ (\mathsf{S}^2 0)\ I\ x)$, the argument $\lambda x.\mathsf{iter}\ (\mathsf{S}^2 0)\ I\ x$ is a normal form, so it will be copied by the other iterator *no matter which strategy we are using.*

5.2 Weak System $\mathcal{L}$

In the λ-calculus, two views of the notion of function coexist. One of them is that functions are ordinary syntactic objects, on which we can compute. The other sees functions as abstract objects inside which it is not sensible to compute; as pieces of programs which have to wait for their argument before executing. This opposition can be seen in the following rule:

$$\frac{t \to v}{\lambda x.t \to \lambda x.v}\ (\xi)$$

This rule is part of the λ-calculus, but a strategy of the λ-calculus is free to contain it or not: in the first case, the strategy is said to be *strong*, in the second, it is *weak*. In general, weak strategies cannot reduce beyond weak head normal form, thus they are not strategies of the λ-calculus in the sense of Definition 3.3.

Weak strategies are those used in functional programming languages [23,13]. In fact, it is more convenient to see weak strategies of the λ-calculus as strategies of a weak λ-calculus, along the lines presented in [21].

Here we present a weak version of System $\mathcal{L}$ (with outer-iter, so with the rules in page 6), which we call *weak System $\mathcal{L}$* with the same restriction as in ordinary weak reduction: do not reduce under abstractions, i.e. we remove the (ξ) rule. Similarly, reduction in the second argument of let constructs should also be prohibited. We also forbid reduction inside pairs, so as to avoid computations in the first argument of let constructs that are not needed in order to reach a pair. We do allow reduction under a S, though, as well as under cond and iter. The new calculus is defined as:

Definition 5.1 The weak System $\mathcal{L}$ is the calculus with reduction $\to_w$, defined by allowing System $\mathcal{L}$ reduction $\to$ in any weak evaluation context W, defined as follows:

$$\begin{aligned} W ::= {} & [\,] \mid W\,t \mid t\,W \mid \mathsf{S}\,W \mid \mathsf{let}\ \langle x, y\rangle = W\ \mathsf{in}\ t \\ & \mid \mathsf{cond}\ W\ u\ w \mid \mathsf{cond}\ t\ W\ w \mid \mathsf{cond}\ t\ u\ W \\ & \mid \mathsf{iter}\ W\ u\ w \mid \mathsf{iter}\ t\ W\ w \mid \mathsf{iter}\ t\ u\ W \end{aligned}$$

There is still a lot of freedom to define strategies, in particular in the iter case.

5.3 Confluence

In the λ-calculus, it is well-known that removing the (ξ) rule leads to a non-confluent calculus, as evidenced by the following diverging pair, where $I = \lambda x.x$ (see e.g. [21]):

$$\lambda y.y\,(I\,I) \leftarrow (\lambda xy.y\,x)\,(I\,I) \rightarrow (\lambda xy.y\,x)\,I \rightarrow \lambda y.y\,I$$

This has led to the introduction of frameworks such as supercombinators or explicit substitutions [21], which is not completely satisfactory either, because these systems are usually more complicated. We have the same kind of restriction here, hence non-confluence of the weak calculus is expected (as opposed to "stronger" weak calculi like [6]). However, like in other weak λ-calculi, weak System $\mathcal{L}$ is confluent for programs: closed terms of base type.

Definition 5.2 A *program* is a closed System $\mathcal{L}$ term of type Nat or Bool.

Definition 5.3 We call *values* the closed normal forms for $\rightarrow_w$.

Proposition 5.4 *Values are the closed terms of this form:*

$$v ::= \mathsf{S}^n\,\mathsf{0} \mid \mathsf{true} \mid \mathsf{false} \mid \langle t, u\rangle \mid \lambda x.t$$

Proof. By adapting the proof of adequacy (again, this result is not valid in the untyped calculus). □

In the following, a term denoted by v will always be assumed to be a value.

Proposition 5.5 $\rightarrow_w$ *is confluent on programs.*

Proof. Assume $t \rightarrow_w^* u_1$ and $t \rightarrow_w^* u_2$, where t is of base type. Consider v_1 and v_2 the $\rightarrow_w$-normal forms of u_1 and u_2 respectively. v_1 and v_2 are values of base types, hence are also normal forms for $\rightarrow$ (using Proposition 5.4). But $\rightarrow$ is confluent, thus has the property of unicity of normal forms. We conclude $v_1 = v_2$. □

6 Weak Strategies

We are now in a position to define reduction strategies for the weak System $\mathcal{L}$ similar to known strategies for the weak λ-calculus. In this section, all strategies are weak: they perform no reduction under abstraction, and, consistently, they are defined only on closed terms. We essentially just mention call-by-name and call-by-value, while we will give more details on call-by-need, which can interestingly be defined directly in the calculus, in an operational way.

6.1 Call-by-name and call-by-value

Definition 6.1 *Call-by-name* reduction is leftmost outermost weak reduction. In particular, iteration is by name. *Call-by-value* differs from call-by-name by reducing the argument of an application before contracting the redex and by using iteration by value instead of iteration by name.

Proposition 6.2 *Call-by-name and call-by-value are strategies of weak System $\mathcal{L}$. Moreover, call-by-name is normalising (in the untyped weak System $\mathcal{L}$).*

Proof. They are clearly strategies. Normalisation is as in Proposition 6.6. □

Remark 6.3 Call-by-value is not normalising in the untyped calculus: recall Ω, a (untypable) term without weak head normal form. Then $\mathsf{iter}\ 0\ (\lambda x.x)\ \Omega$ starts an infinite reduction although the term has normal form $\lambda x.x$.

6.2 Call-by-need

Under *call-by-need* (or *lazy evaluation*), an iterated term, not in normal form, is evaluated at most once, regardless of how many times the term is iterated. Thus such an iterated term may not be duplicated (by another iterator) before it has been reduced and may be reduced only if actually used.

The standard, non operational, definition of call-by-need is: reduce the argument first (i.e. use call-by-value) if it will be needed, do not reduce it otherwise (i.e. use call-by-name). In general, it is difficult to decide if an argument will be needed or not in the syntax of the λ-calculus, and extra features are added to actually implement call-by-need (sharing graphs, environments, explicit substitutions). Here, the interesting point is that we can characterise call-by-need *within the calculus*.

Definition 6.4 Call-by-need is defined by the weak strategy (still with the liberal meaning) $\to_l$. See Table 3.

Lazy evaluation contexts:

$$\begin{aligned} L ::= {} & [\,] \mid L\,t \mid \mathsf{S}\,L \mid \mathsf{let}\ \langle x, y\rangle = L\ \mathsf{in}\ t \\ & \mid \mathsf{cond}\ L\ u\ w \mid \mathsf{cond\ true}\ L\ w \mid \mathsf{cond\ false}\ u\ L \\ & \mid \mathsf{iter}\ L\ u\ w \mid \mathsf{iter}\ 0\ L\ w \mid \mathsf{iter}\ (\mathsf{S}\,t)\ u\ L \end{aligned}$$

Base cases:

$$\begin{array}{rl} (\lambda x.t)\,u \to_l t[u/x] & \mathsf{fv}(u) = \varnothing \\ \mathsf{let}\ \langle x, y\rangle = \langle t, t'\rangle\ \mathsf{in}\ u \to_l u[t/x][t'/y] & \mathsf{fv}(t) = \mathsf{fv}(t') = \varnothing \\ \mathsf{cond\ true}\ u\ w \to_l u & \\ \mathsf{cond\ false}\ u\ w \to_l w & \\ \mathsf{iter}\ 0\ u\ w \to_l u & \mathsf{fv}(w) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}\,0)\ u\ w \to_l w\ u & \mathsf{fv}(w) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ v \to_l v\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ v) & \mathsf{fv}(t) = \mathsf{fv}(v) = \varnothing, v \text{ is a value} \end{array}$$

Context rule: $\dfrac{t \to_l v}{L[t] \to_l L[v]}$

Table 3
Call-by-need

Proposition 6.5 *$\to_l$ is a strategy for $\to_w$.*

Proof. It is clear that $\to_l \subset \to_w$. Moreover, the normal forms for $\to_w$ are values in the sense of Proposition 5.4, as we can replace $\to_w$ by $\to_l$ in the proof of Adequacy (Theorem 3.8). □

Proposition 6.6 *$\to_l$ reduces only needed redexes. Hence $\to_l$ is normalising and it has the same normal forms as $\to_w$ (see Proposition 5.4).*

Proof. Say that a position is needed if it is the position of a needed redex or if it is above a needed position. By induction, it is easy to see that L only defines contexts where the hole $[\,]$ is in a needed position. In an *orthogonal* and *fully extended* CERS, reducing only needed redexes terminates [15]. System $\mathcal{L}$ is not fully extended because a non-redex can become a redex (if a term becomes closed). But it is a sub-system of a suitable CERS (where we forget the conditions), which is enough to get the result. □

Proposition 6.7 *$\to_l$ has the diamond property.*

Proof. In this proof, we simply write $\to$ for $\to_l$, and we assume that there are diverging reductions $t_1 \leftarrow_p t \to_q t_2$ at positions p and q respectively. If $p = q$, the same rule is used in both reductions, hence $t_1 = t_2$ and the reductions are not diverging. If p and q are disjoint, the pair is joined in one step on each side by applying the other rule at the corresponding position. Otherwise, one of the positions is the outermost, let's say p and write $q = p \cdot q'$. We look at all possible cases for the subterm t' at position p.

- $t' = u\,w$: by definition of L, $u \to_{q'} u'$ so $u \neq \lambda x.s$ (by Proposition 5.4), and no rule is applicable at the root of t'; this case thus does not happen.
- Similar argument for $t' = \mathsf{let}\ \langle x, y\rangle = w\ \mathsf{in}\ u$.
- $t' = \mathsf{cond}\ \mathsf{true}\ u\ w$: $u \leftarrow t' \to_{q'} \mathsf{cond}\ \mathsf{true}\ u'\ w$, then $u \to u'$, $\mathsf{cond}\ \mathsf{true}\ u'\ f \to u'$.
- Similar argument for $t' = \mathsf{cond}\ \mathsf{false}\ u\ w$ and $t' = \mathsf{iter}\ \mathsf{0}\ u\ w$.
- $t' = \mathsf{iter}\ (\mathsf{S}\,\mathsf{0})\ u\ w$: straightforward.
- $t' = \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,s))\ u\ w$: reduction at the root is allowed only when w is a value, thus the only possible innermost reduction is in s, and it is straightforward to conclude.

□

6.3 Minimality

Efficiency is a very pragmatic notion. In many cases, there is no better argument to demonstrate the efficiency of a strategy than a benchmark. On the contrary, here, System $\mathcal{L}$ gives us enough grip to actually *give a proof* of the efficiency of call-by-need. To measure efficiency, we just count the number of reduction steps; hence minimality (Definition 3.4) corresponds to the most efficient strategy. This is a more realistic notion here than in the λ-calculus, because implicit substitution

is always linear and all issues of duplication and erasure are explicit, hence taken into account when counting the number of rewrite steps.

Theorem 6.8 (Minimality) *$\to_l$ is minimal, i.e. if t is a closed term, $t \to_w^m v$ and $t \to_l^n v$ where v is a value, then $n \leq m$.*

Proof. We use Theorem 3.6 (see [26] for more details on these techniques). Throughout this proof, we write $\twoheadrightarrow$ instead of $\to_l$ and $\to$ instead of $\to_w$ to improve readability. Assume $t_1 \twoheadleftarrow_p t \to_q t_2$. We want to show that there exists t_3 such that $t_1 \to^m t_3 \twoheadleftarrow^n t_2$ with $m \leq n$. If the reductions are disjoint, this is easy because redexes are preserved by disjoint reductions.

If q is above p, then the $\to_q$ step is also a $\twoheadrightarrow_q$ step (definition of L and $\twoheadrightarrow$) and we may use the diamond property for $\twoheadrightarrow$, except in the case $\mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ w' \leftarrow \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ w \to w\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w)$ where t and w are closed and w is not a value.

This case requires some more work. First $\mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ w' \to w'\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w')$ and $w\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w) \twoheadrightarrow w'\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w)$. Using Propositions 6.6 and 5.4, and the fact that w' is closed, we have $w' \twoheadrightarrow^k \lambda x.w''$, hence $w'\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w') \to^{k+1} w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w'/x]$ and $w'\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w) \twoheadrightarrow^{k+1} w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w/x]$. Again, $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w/x] \twoheadrightarrow^n v$, where v is a value. If w is not at a needed position in $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w/x]$, then the same reduction can be mimicked on $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w'/x]$. Otherwise, for some multi-hole context C, this reduction can be decomposed as $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w/x] \twoheadrightarrow^{n_1} C[w, w, \ldots, w] \twoheadrightarrow C[w', w, \ldots, w] \twoheadrightarrow^{n_2} v$ with $n = n_1 + n_2 + 1$, and we can mimic this reduction on $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w'/x]$, omitting at least one step (because at least one residual of a needed redex is needed): $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w'/x] \twoheadrightarrow^{n_1} C[w', w', \ldots, w'] \twoheadrightarrow^{n_2'} v$ with $n_2' \leq n_2$ (technically, this $\twoheadrightarrow^{n_2'}$ reduction is the projection of the $\twoheadrightarrow^{n_2}$ reduction after the reduction $w \twoheadrightarrow w'$). In both cases, we indeed have $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w'/x] \to^m v \twoheadleftarrow^n w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w/x]$ with $m \leq n$. This concludes this case.

If q is below p and the $\to_q$ step is not also a $\twoheadrightarrow_q$ step (otherwise, use Proposition 6.7), we look at all possible cases. There are three (by looking at the definition of L and $\twoheadrightarrow$). For instance, $u \leftarrow \mathsf{cond\ true}\ u\ w \to \mathsf{cond\ true}\ u\ w' \twoheadrightarrow u$. The cases for $\mathsf{cond\ false}\ u\ w$ and $\mathsf{iter}\ \mathsf{0}\ u\ w$ are similar. The important point is that $w\,(\mathsf{iter}\ t\ u\ v) \leftarrow \mathsf{iter}\ (\mathsf{S}\,t)\ u\ v \to \mathsf{iter}\ (\mathsf{S}\,t)\ u\ v'$ is not a case to consider (v is a normal form for $\to_w$). □

Hence, thanks to Proposition 6.7, any sub-strategy (in particular any deterministic one) of $\to_l$ will also be minimal. It is already known that call-by-need is optimal in a large class of rewrite systems, including weak λ-calculi [21]. However, our present statement is much stronger because the notion of optimality in [21] takes into account parallel reduction of family of redexes. In other words, it is assumed that there is some adequate sharing mechanism that will allow all redexes of the same family to be reduced at the same time. We should also mention that this proof is a nice illustration of using the techniques of [26]. The call-by-need strategy presented here is an effective approximation of the internal needed strategy (whose minimality for orthogonal TRSs is reproved in [26]), which retains minimality (in our system; there is no hope of a similar result in general for orthogonal TRSs).

Each iterated term is evaluated at most once and it is reduced only if actually used. It is remarkable that call-by-need is easily implementable without any syntactic extension to the calculus. Note that this does not happen with standard call-by-need, which is not expressible within the syntax of the λ-calculus: one has to extend it with some explicit binding syntax (Wadsworth graph reductions, explicit let bindings or explicit heap) to express sharing of subterm evaluation.

7 Conclusion

System $\mathcal{L}$ is a calculus that isolates the linear and non-linear components of a computation. We have used this calculus to make a study of evaluation strategies in this context, where it is precisely the non-linear aspects of the computation that we need to control. This leads to a simple description of strategies and to a definition of minimal strategies within the calculus. Moreover, We anticipate that we can make heavy use of these results in current implementation work based around System $\mathcal{L}$.

Acknowledgement

We are sincerely grateful to the anonymous referees for their numerous and judicious comments (some of which could unfortunately not be followed because of space and time constraints).

References

[1] S. Alves, M. Fernández, M. Florido, and I. Mackie. The power of linear functions. In Z. Ésik, editor, *Proceedings of the 15th EACSL Conference on Computer Science Logic (CSL'06)*, volume 4207 of *Lecture Notes in Computer Science*, pages 119–134. Springer-Verlag, 2006.

[2] S. Alves, M. Fernández, M. Florido, and I. Mackie. Iterator types. In *Proceedings of Foundations of Software Science and Computation Structures, (FOSSACS'07)*, volume 4423 of *LNCS*, pages 17–31. Springer-Verlag, 2007.

[3] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, May 1997.

[4] H. Barendregt, J. A. Bergstra, J. W. Klop, and H. Volken. Degrees, reductions and representability in the lambda-calculus. Technical report, University of Utrecht, Department of Mathematics, 1976.

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

[6] T. Blanc, J.-J. Lévy, and L. Maranget. Sharing in the weak lambda-calculus. In *Processes, Terms and Cycles*, volume 3838 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2005.

[7] H. J. S. Bruggink. Residuals in higher-order rewriting. In R. Nieuwenhuis, editor, *Proceedings of Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 123–137. Springer, June 2003.

[8] L. Colson and D. Fredholm. System T, call-by-value and the minimum problem. *Theor. Comput. Sci.*, 206(1-2):301–315, 1998.

[9] U. Dal Lago and S. Martini. An invariant cost model for the lambda calculus. In A. Beckmann, U. Berger, B. Löwe, and J. V. Tucker, editors, *Logical Approaches to Computational Barriers, Second Conference on Computability in Europe, CiE 2006, Proceedings*, volume 3988 of *Lecture Notes in Computer Science*, pages 105–114. Springer, 2006.

[10] J. Engelfriet and E. Schmidt. IO and OI. *Journal of Computer and Systems Sciences*, 15:328–353, 1997.

[11] M. Fernández, I. Mackie, and F.-R. Sinot. Closed reduction: explicit substitutions without alpha conversion. *Mathematical Structures in Computer Science*, 15(2):343–381, 2005.

[12] J. Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages (POPL '90)*, pages 1–15, San Francisco, CA, USA, Jan. 1990. ACM Press.

[13] P. Fradet. Compilation of head and strong reduction. In *ESOP*, volume 788 of *Lecture Notes in Computer Science*, pages 211–224, 1994.

[14] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[15] J. R. W. Glauert, R. Kennaway, and Z. Khasidashvili. Stable results and relative normalization. *Journal of Logic and Computation*, 10(3):323–348, 2000.

[16] Z. Khasidashvili and V. van Oostrom. Context-sensitive conditional expression reduction systems. *Electronic Notes in Theoretical Computer Science*, 2:167–176, 1995.

[17] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 16–30. ACM Press, Jan. 1990.

[18] J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, Jan. 1993.

[19] J.-J. Lévy. Optimal reductions in the lambda-calculus. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 159–191. Academic Press, Inc., New York, NY, 1980.

[20] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, May 1998.

[21] L. Maranget. Optimal derivations in orthogonal term rerwiting systems and in weak lambda calculi. In *Proc. of the 1991 conference on Principles of Programming Languages*. ACM Press, 1991.

[22] M. Parigot. On the representation of data in lambda-calculus. In *CSL '89: Proceedings of the third workshop on Computer science logic*, pages 309–321, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[23] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, 1987.

[24] Z. Spawski and P. Urzyczyn. Type fixpoints: iteration vs. recursion. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 102–113, New York, NY, USA, 1999. ACM Press.

[25] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[26] V. van Oostrom. Random descent. In *Proceedings of 18th Rewriting Techniques and Applications, (RTA'07)*, volume 4533 of *Lecture Notes in Computer Science*, pages 314–328. Springer-Verlag, 2007.

[27] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Oxford University, 1971.

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