Call-by-Need in Token-Passing Nets

F.-R. SINOT†
LIX, École Polytechnique
91128 Palaiseau cedex, France.
Email: frs@lix.polytechnique.fr

Received 18 October 2005
Revised 25 March 2006

Recently, encodings in interaction nets of the call-by-name and call-by-value strategies of the λ-calculus have been proposed. The purpose of these encodings is to bridge the gap between interaction nets and traditional abstract machines, which are both used to provide lower-level specifications of strategies of the λ-calculus, but in radically different ways. The strength of these encodings is their simplicity, which comes from the simple idea of introducing an explicit syntactic object to represent the evaluation flow. Another benefit of this approach is that no artifact is needed to represent boxes. However, these encodings purposefully follow as closely as possible the implemented strategies, call-by-name and call-by-value, hence do not benefit from the ability of interaction nets to easily represent sharing. The aim of this paper is to show that better sharing (hence efficiency) can indeed be achieved without adding much structure. We thus present the call-by-need strategy following the same philosophy, which is indeed not any more complicated than call-by-name. We also extend our approach to fully lazy reduction. This continues the task of bridging the gap between interaction nets and abstract machines, thus pushing forward a more uniform framework for implementations of the λ-calculus.

Contents

1 Introduction 2
2 Call-by-need 4
   2.1 Preliminaries 4
   2.2 Big-step semantics 4
   2.3 Small-step semantics 5
3 Nets 8
   3.1 Interaction nets 8
   3.2 Nets 8
   3.3 Token-passing nets 9
4 Encoding of terms 9

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

Interaction nets (Lafont, 1990) are a graphical, distributed model of computation with the advantage of expressing in an explicit and uniform way all the usually hidden non-linear steps of a computation, such as copying and erasing. This is inspired by proof nets and linear logic (Girard, 1987). In particular, sharing is built in (as opposed to terms) and is dealt with explicitly (as opposed to termgraphs). Locality and strong confluence of reduction contribute to making interaction nets well-suited as an intermediate formalism in the implementation of programming languages. However, despite their qualities and their popularity among theoreticians, it is sad to notice that they are less widely used by implementors of real-world programming languages. While it is difficult to say why, works such as those on optimal reduction have probably led some to think of interaction nets as a theoreticians-only tool. It might thus be worth it to bridge the gap between interaction nets and traditional tools such as abstract machines.

Interaction nets have been used for the implementation of optimal reduction (Lamping, 1990; Gonthier et al., 1992; Asperti et al., 1996) and for other efficient (non-optimal) implementations of the $\lambda$-calculus (Mackie, 1998; Mackie, 2004). All of the above encodings of the $\lambda$-calculus have in common that a $\beta$-redex is always translated to an active pair (i.e. a redex in interaction nets). Hence, paradoxically, while all reductions are equivalent, there is still the need for an external interpreter to find the redexes and manage them, which is typically implemented by maintaining a stack of redexes (Pinto, 2000), or by ad hoc methods, not documented, not part of the theory, thus error-prone. The fact that interaction steps implementing different $\beta$-reductions may be interleaved (because there is no control on the order of reductions) also has the nasty consequence that the encodings need to simulate boxes (Girard, 1987; Gonthier et al., 1992) in a more or less complex and costly way.

There is thus a gap between the world of interaction nets and that of abstract machines. In (Sinot, 2005a), we began to bridge this gap by giving encodings of the call-by-name and call-by-value strategies of the $\lambda$-calculus in interaction nets. These encodings are based on the idea of an evaluation token, which is a standard interaction agent, walking through the term as an evaluation function would do. They are very natural and stick as closely as possible to a given strategy, in a way similar to an abstract machine. In particular, they ensure that essentially only one reduction is possible at a time. This is simpler to implement than general interaction nets and contributes to avoid the need for boxes, hence to simplify the encoding.
In this paper, our goal is to bridge the gap further and to push forward a more uniform framework to describe implementations of the $\lambda$-calculus. More precisely, we will give a presentation of the call-by-need strategy in the style of (Sinot, 2005a). Seen from the point of view of this previous work, we improve the system with the ability to share reductions, which is a key aspect of interaction nets which was absent from this work. From the point of view of functional languages, we give a uniform and fully formal description of a call-by-need interpreter in interaction nets. We even extend the approach further and encode the fully lazy strategy in interaction nets as well, thus making it more formal and understandable than previous presentations (Wadsworth, 1971; Shivers and Wand, 2005). In particular, we make a connection with the work of Lang (Lang, 1998), that could not be made without our framework. To put this piece of work into context, it is important to say that, while call-by-need has more or less become a standard in implementations of functional languages, the fully lazy strategy, which allows more sharing of reductions, has not. This shows that it is important to improve our understanding of this strategy. Our work could also constitute an important step forward in understanding optimal reduction better, although the remaining path may still be difficult.

In (Sinot, 2005a) the encodings are in standard interaction nets, featuring a standard agent called the evaluation token. Roughly speaking, reductions are triggered by the evaluation token and the interaction rules guarantee that there is a unique occurrence of the token in any net, hence reduction is essentially deterministic. On the other hand, some restrictions which are part of the formalism of interaction nets are tailored to ensure strong confluence, and are thus no longer necessary if all reductions are triggered by a unique token. In this paper, we will have to abandon the restriction to Lafont’s interaction nets, as explained in Section 5.2, and adopt Alexiev’s formalism of interactions nets with multiple principal ports (Alexiev, 1999), or simply nets. Still, since reduction is directed by a unique token, evaluation is fully deterministic. An important aspect of this work is thus also to illustrate how the definition of interaction nets can be relaxed without losing important properties, such as strong confluence.

Call-by-need was introduced by Wadsworth (Wadsworth, 1971). The idea is relatively intuitive: a subterm should be evaluated only if it is needed, and if so, it should be evaluated only once. The original formulation is in terms of graph rewriting, however there have been several attempts to formalise this idea in different ways: big-step operational semantics of call-by-need have been given independently in (Launchbury, 1993) and (Seaman and Purushothaman Iyer, 1996); small-step presentations based on contexts have been presented in (Ariola et al., 1995; Ariola and Felleisen, 1997; Maraist et al., 1998). In contrast with these approaches, we present a purely graph-based formalisation, making explicit at the object-level the rewrite strategy used in (Wadsworth, 1971). To prove the correctness of our encoding, we refer to a variant of Launchbury’s big-step semantics, from which we derive another small-step semantics of call-by-need, radically different from previous works, and better suited to our needs. In this respect, we follow the approach of (Sinot, 2005a).

Another way to see our encoding using net rewriting is as a graph-based abstract machine, which is still strikingly close to a term representation. The encoding of terms is almost the same as for call-by-name and reduction rules are not much more complicated, thus sharing is almost obtained for free. Moreover, while some reductions are sequentialised thanks to the token, we do not lose all the potential for parallelism, and a distributed implementation may still reduce several
redexes at the same time. We have only made explicit which computations must be sequentialised and which ones may be performed in parallel.

This present work is a substantially extended and improved version of (Sinot, 2005b). The new features include a new small-step presentation of call-by-need (Section 2), a full proof of correspondence with the net rewrite system (Section 6) and a net rewrite system for fully lazy reduction (Section 7).

The rest of this paper is structured as follows. In Section 2, we recall Launchbury’s semantics and derive a new small-step presentation of call-by-need. In Section 3, we recall some background on net rewriting. We give the encoding of terms in Section 4, the evaluation rules in Section 5 and the correctness properties in Section 6. The issue of fully lazy reduction is tackled in Section 7. We conclude in Section 8.

2. Call-by-need

2.1. Preliminaries

We assume basic knowledge of the \( \lambda \)-calculus; we refer the reader to (Barendregt, 1984) for more details. To fix notations, the set of \( \lambda \)-terms is defined by:

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

where \( x \) ranges over a set of variables. Terms are considered modulo \( \alpha \)-conversion i.e. renaming of bound variables. We denote by \( \text{fv}(t) \) the set of free variables of a term \( t \). If \( t \) is a \( \lambda \)-term, \( \hat{t} \) is a term \( \alpha \)-equivalent to \( t \) in which all the bound variables have been renamed to fresh variables.

This set is equipped with the following rule of \( \beta \)-reduction:

\[
(\beta) \quad (\lambda x.t) \ u \rightarrow_{\beta} t\{x := u\}
\]

where \( t\{x := u\} \) denotes the substitution of \( x \) by \( u \) in \( t \), that is \( t \) where all occurrences of \( x \) have been replaced by \( u \), without name capture. We write \( \rightarrow_{\beta} \) for one-step reduction and \( \rightarrow^{*}_{\beta} \) for its reflexive transitive closure.

We call weak head normal forms terms of the form \( \lambda x.t \) or \( x \ t_1 \ldots t_n \). Closed weak head normal forms are of the form \( \lambda x.t \) and are called values. We say that \( v \) is a weak head normal form of \( t \) if \( v \) is a weak head normal form and \( t \rightarrow^{*}_{\beta} v \).

Contexts (or more precisely one-hole contexts) on \( \lambda \)-terms are terms with a hole (written \( [\ ] \)), and defined by:

\[
C[\ ] ::= [\ ] \mid \lambda x.C[\ ] \mid t C[\ ] \mid C[\ ] t
\]

The operation of filling a context \( C[\ ] \) with a term \( t \) is written \( C[t] \) and is defined as \( C[\ ] \), where the hole \( [\ ] \) has been replaced by \( t \), without renaming (name capture may occur).

2.2. Big-step semantics

The call-by-need (or lazy) strategy was introduced by Wadsworth (Wadsworth, 1971) as a graphical interpreter. To make it easier to reason about lazy evaluation, Launchbury expressed it in a natural (or big-step) semantics style (Launchbury, 1993). This semantics is briefly reviewed in
Appendix A. In this section, we introduce a variant of this semantics that is slightly simpler and whose equivalence with Launchbury’s semantics is proved in Appendix A.

Environments (e.g. $\Gamma$, $\Delta$, $\Theta$) are mappings from variables to terms. Evaluation is defined on distinctly named pairs $\Gamma : t$ of an environment and a term. Contrary to the cases of call-by-name and call-by-value, evaluation may modify the environment as a side effect. More precisely, when a value is computed for a variable, that value is stored in the environment to avoid a possible recomputation if it is needed several times. Evaluation is only defined for closed pairs $\Gamma : t$, which means that if $t$ has some free variables, they must be in the domain of $\Gamma$. Evaluation judgements are of the form $\Gamma : t \Downarrow \Delta : v$, to be read “the term $t$ in the environment $\Gamma$ reduces to the value $v$ together with the new environment $\Delta$”, as defined by the following set of deduction rules.

\[
\begin{align*}
\frac{}{\Gamma : \lambda x.t \Downarrow \Gamma : \lambda x.t} \text{ Lam} \\
\frac{\Gamma : t \Downarrow \Delta : \lambda x.t' \quad (\Delta, x \mapsto u) : t' \Downarrow \Theta : v}{\Gamma : t u \Downarrow \Theta : v} \text{ App} \\
\frac{\Gamma : t \Downarrow \Delta : v}{(\Gamma, x \mapsto t) : x \Downarrow (\Delta, x \mapsto v) : \hat{v}} \text{ Var}
\end{align*}
\]

This corresponds rather closely to the semantics given in (Launchbury, 1993) and Appendix A. The main difference is that we do not assume that terms are precompiled in a $\lambda$-calculus with $\textit{lets}$ so that the closures are already explicit. We instead generate a new binding in the environment in rule $\text{App}$ (corresponding to $\beta$-reduction). Also, as we use exactly the bound variable for this binding, we do not need to perform any substitution: everything is done using the environment. Our presentation is thus simpler: there are three rules instead of four, there is no need for a calculus with $\textit{lets}$ (recursive bindings can be dealt with using standard $\lambda$-calculus fixpoints). It is also easier to use since the evaluation relation is well-defined on all terms, not only on precompiled terms. Moreover, our semantics is equivalent to Launchbury’s one (see Appendix A), which justifies that it is indeed a semantics of call-by-need and allows us to freely borrow results from (Launchbury, 1993) when necessary. In particular, it is sufficient to perform $\alpha$-conversion only in rule $\text{Var}$ to ensure that all bound variables remain distinct.

2.3. Small-step semantics

A big-step semantics provides a high-level description of the strategy. However, the strategy itself is not observable at the level of a proposition of the form $\Gamma : t \Downarrow \Delta : v$, but in the proof tree of this proposition, using the deduction rules of the semantics. The proposition itself only expresses a relation between an input term and a result. In Sections 4 and 5, we will describe an implementation of call-by-need in interaction nets. A correctness result with respect to the big-step semantics would not be enough to make us feel able to claim that the implementation indeed follows the strategy, because there could be some differences in the way the results are obtained that cannot be observed in the result itself. That is why we introduce a small-step style...
presentation of call-by-need, that will be derived from the big-step presentation in a rather systematic way, so that it will be obvious that reductions in the small-step system correspond to deduction branches of the big-step one. This will also make the proof of correctness easier.

We want to replace the previous inductive rules by a first-order rewrite system but we also want to be as explicit about the evaluation order as in the previous system. Intuitively, the idea is to mimic with rewrite rules what happens when drawing the deduction tree of a judgement \( \Gamma : t \Downarrow \Delta : v \). Three kinds of things may happen: we may start a new branch and go up in the deduction tree (evaluate a subterm), we may finish a branch (return an intermediate result) and we may go down in the tree (put the result back in the right context).

We thus enrich the syntax of terms with the symbols \( \Downarrow \) (called "eval", corresponding to evaluation) and \( \Rightarrow \) (called "return", corresponding to the evaluation function returning). More precisely, we define the set of enriched terms as follows:

\[
e := \Downarrow \Gamma t \mid \Rightarrow \Gamma t \mid x \mid \lambda x.t \mid C[e]
\]

where \( t \) stands for a \( \lambda \)-term, \( \Gamma \) for an environment, \( x \) for a variable and \( C[\cdot] \) for a one-hole context on \( \lambda \)-terms.

We may now define the following rewrite system on enriched terms:

- \((\Downarrow \text{Lam})\)
- \((\Downarrow \text{App})\)
- \((\Rightarrow \text{App})\)
- \((\Downarrow \text{Var})\)
- \((\Rightarrow \text{Var})\)

This is a plain rewrite system: substitution is handled by the environment and reduction is allowed in any context. This is a small-step presentation of call-by-need, as will be shown by Proposition 2.2. Other small-step presentations of call-by-need include (Ariola et al., 1995; Ariola and Felleisen, 1997; Maraist et al., 1998). Ours is radically different, and arguably simpler. The simplicity comes from the fact that we make the evaluation flow explicit at the syntactic level, whereas the game played in those previous works consists in designing clever evaluation contexts to restrict reduction and in encoding environments as terms in a contrived way.

Another point of comparison is against abstract machines such as e.g. the lazy Krivine machine (Créguet, 1990). Our presentation is probably closer to these works, although it is more abstract. But again, it has exactly the right degree of abstraction to fit our needs.

Also note that, as far as we know, such a simple small-step presentation of call-by-need has not been given before; usual small-step presentations rely on inductive rules allowing reductions in a certain class of contexts, hence do not make explicit the flow of evaluation contrary to our presentation, which is crucial for the encoding into interaction nets. In some sense, our presentation is intermediate between traditional small-step semantics (which separate as much as possible reduction and strategy) and abstract machines (which may involve complex data structures). We call this presentation the token-passing semantics of call-by-need.

A \( \lambda \)-term \( t \) is always in normal form with respect to this system, and so is \( \Rightarrow \Gamma t \). To evaluate \( t \) in a context \( \Gamma \), we have to start reduction from \( \Downarrow \Gamma t \). In general, if \( \Gamma \) is undefined for some free variables of \( t \), the call-by-need evaluation of \( \Gamma : t \) may fail. This corresponds here to the
Call-by-Need in Token-Passing Nets

reduction starting from \( \downarrow_\Gamma t \) terminating on an enriched term not of the form \( \uparrow_\Delta u \). If \( t \) is closed, which is generally assumed, we may safely choose \( \Gamma \) empty.

This system enjoys the following strong invariant:

**Proposition 2.1.** If \( \downarrow_\Gamma t \rightarrow^* u \), then there is exactly one occurrence of \( \downarrow_\Delta \) or \( \uparrow_\Delta \) in \( u \) (for some \( \Delta \)).

**Proof.** By induction on the length of the reduction and by cases on the last rule applied. In rule \( \uparrow_\text{Var} \), \( v \) is copied but it has no occurrence of \( \downarrow \) or \( \uparrow \) by the induction hypothesis. 

Since a reduction always involves a \( \downarrow_\Gamma \) or \( \uparrow_\Gamma \), there is thus always at most one redex in a term obtained from reduction of \( \downarrow_\Gamma t \) and the control flow is indeed made explicit at the syntactic level.

**Proposition 2.2.** \( \Gamma : t \downarrow \Delta : v \iff \downarrow_\Gamma t \rightarrow^* \uparrow_\Delta v \)

**Proof.** Both implications are systematic.

\( \Rightarrow \) By induction on the derivation:

- **Lam:** \( \Gamma : \lambda x.t \downarrow \Gamma : \lambda x.t \) and indeed \( \downarrow_\Gamma \lambda x.t \xrightarrow{\text{Lam}} \uparrow_\Gamma \lambda x.t \).

- **App:** if \( \Gamma : t u \downarrow \Theta : v \), then there exists \( t' \) and \( \Delta \) such that \( \Gamma : t \downarrow \Delta : \lambda x.t' \) and \( (\Delta, x \mapsto u) : t' \downarrow \Theta : v \). By induction, \( \downarrow_\Gamma t \rightarrow^* \uparrow_\Delta \lambda x.t' \) and \( \downarrow_{(\Delta, x \mapsto u)} t' \rightarrow^* \uparrow_\Theta v \), hence: \( \downarrow_\Gamma (t u) \xrightarrow{\text{App}} (\downarrow_\Gamma t) u \rightarrow^* (\uparrow_\Delta \lambda x.t') u \xrightarrow{\text{App}} (\downarrow_{(\Delta, x \mapsto u)} t') \rightarrow^* (\uparrow_\Theta v) \).

- **Var:** if \( (\Gamma, x \mapsto t) : x \downarrow (\Delta, x \mapsto v) : \hat{v} \), then \( \Gamma : t \downarrow \Delta : v \). Hence by induction, \( \downarrow_\Gamma t \rightarrow^* \uparrow_\Delta v \), and finally: \( \downarrow_{(\Gamma, x \mapsto t)} x \xrightarrow{\text{Var}} (x \mapsto \downarrow_\Gamma t) \rightarrow^* (x \mapsto \uparrow_\Delta v) \xrightarrow{\text{Var}} \uparrow_{(\Delta, x \mapsto v)} \hat{v} \).

\( \Leftarrow \) By structural induction on the term \( t \):

- **Abstraction:** \( \downarrow_\Gamma \lambda x.t \xrightarrow{\text{Lam}} \uparrow_\Gamma \lambda x.t \) and indeed \( \Gamma : \lambda x.t \downarrow \Gamma : \lambda x.t \) by (Lam).

- **Application:** assume \( \downarrow_\Gamma (t u) \rightarrow^* \uparrow_\Theta v \). Thus we have \( \downarrow_\Gamma (t u) \xrightarrow{\text{App}} (\downarrow_\Gamma t) u \rightarrow^* (\uparrow_\Theta v) \).

Moreover, \( \Gamma : t \downarrow \Delta : \lambda x.t' \) by induction. Then \( \uparrow_\Delta \lambda x.t' \) \xrightarrow{\text{App}} \uparrow_{(\Delta, x \mapsto u)} t' \rightarrow^* \uparrow_\Theta v \), thus \( \Delta, x \mapsto u \) : \( t' \downarrow \Theta : v \) by induction and \( \Gamma : t u \downarrow \Theta : v \) by rule (App).

- **Variable:** assume \( \downarrow_{(\Gamma, x \mapsto t)} x \xrightarrow{\text{Var}} (x \mapsto \downarrow_\Gamma t) \rightarrow^* \uparrow_{(\Delta, x \mapsto v)} \hat{v} \). The last rule applied must be \( (x \mapsto \uparrow_\Delta v) \) \xrightarrow{\text{Var}} \uparrow_{(\Delta, x \mapsto v)} \hat{v} \), thus necessarily \( \downarrow_\Gamma t \rightarrow^* \uparrow_\Delta v \) and by induction \( \Gamma : t \downarrow \Delta : v \). Using (Var), we get \( (\Gamma, x \mapsto t) : x \downarrow (\Delta, x \mapsto v) : \hat{v} \).

\( \square \)

Hence the given rewrite system faithfully corresponds to the call-by-need strategy. In particular, as for the big-step semantics, if all bound variables are initially distinct, this is preserved by reduction. This step is crucial, as the interaction net encoding will closely follow the above small-step semantics.
3. Nets

3.1. Interaction nets

A system of interaction nets (Lafont, 1990) is specified from a set $\Sigma$ of symbols, and a set $\mathcal{R}$ of interaction rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) arity. An occurrence of a symbol $\alpha \in \Sigma$ will be called an agent. If the arity of $\alpha$ is $n$, then the agent has $n + 1$ ports: a distinguished one called the principal port depicted by an arrow, and $n$ auxiliary ports labelled $x_1, \ldots, x_n$ corresponding to the arity of the symbol. Such an agent will be drawn in the following way:

\[
\begin{array}{c}
\alpha \\
\downarrow \\
\vdots \\
x_1 \ldots x_n
\end{array}
\]

Intuitively, a net $N$ built on $\Sigma$ is a graph (not necessarily connected) with agents at the vertices. The edges of the graph connect agents together at the ports such that there is only one edge at every port. The ports of an agent that are not connected to another agent are called free. There are two special instances of a net: a wiring (no agents) and the empty net; the extremes of wirings are also called free ports.

An interaction rule $((\alpha, \beta) \rightarrow N) \in \mathcal{R}$ replaces a pair of agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected together on their principal ports (this is called an active pair or redex, and written $\alpha \bowtie \beta$) by a net $N$. Rules must satisfy two conditions: all free ports are preserved during reduction (reduction is local, i.e. only the part of the net involved in the rewrite is modified), and there is at most one rule for each pair of agents. Because of this last restriction, a rule is fully defined by its left hand-side; such a rule will thus be sometimes denoted by $\alpha \bowtie \beta$ as well. The following diagram shows the format of interaction rules ($N$ can be any net built from $\Sigma$).

\[
\begin{array}{c}
x_n \\
\vdots \\
x_1
\end{array}
\begin{array}{c}
\alpha
\end{array}
\begin{array}{c}
y_1 \\
\vdots \\
y_m
\end{array}
\begin{array}{c}
\beta
\end{array}
\begin{array}{c}
x_n \\
\vdots \\
x_1
\end{array}
\begin{array}{c}
N
\end{array}
\begin{array}{c}
y_1 \\
\vdots \\
y_m
\end{array}
\]

We use the notation $\rightarrow$ for the one-step reduction relation, or $\rightarrow^*$ if we want to be explicit about the rule used, and $\rightarrow^*$ for its transitive and reflexive closure. If a net does not contain any active pairs then we say that it is in normal form. The key property of interaction nets, besides locality of reduction, is that reduction is strongly confluent. Indeed, all reduction sequences are permutation equivalent and standard results from rewriting theory tell us that weak and strong normalisation coincide (if one reduction sequence terminates, then all reduction sequences terminate).

3.2. Nets

We define nets as interaction nets (Lafont, 1990), but where the agents are allowed to have any number of principal ports, instead of just one. They have been introduced in (Alexiev, 1999) under the name interaction nets with multiple principal ports and have also been used for instance under the name multiports interaction nets (Mazza, 2005), but they can even be traced back to (Bawden, 1986).
Each agent thus has a fixed number of principal ports, all depicted by an arrow; the other ports are still called auxiliary. An active pair still consists of two agents connected by principal ports on both sides. Reduction is still local, but in general, no property of confluence can be expected.

If all agents in a net system have exactly one principal port and at most one rule can be applied to any active pair, then it is a system of interaction nets (Lafont, 1990). In this case, reduction is strongly confluent.

### 3.3. Token-passing nets

The aim of this paper is to describe two net reduction systems implementing respectively call-by-need (Section 5) and fully lazy reduction (Section 7). However, we feel that these two systems, as well as the systems of (Sinot, 2005a), have some common points and that it would make sense to englobe them under a common denomination. We have chosen the denomination *token-passing nets*, because a particular agent is passed around and enables certain reductions. It is out of the scope of this paper to give a formal definition of token-passing nets, as well as to prove general properties about them. It is also too early for that: we should first find some other interesting examples of token-passing nets before presenting general concepts about such systems. This is thus left as future work.

### 4. Encoding of terms

In this section, we define a translation function $T(\cdot)$ from $\lambda$-terms into nets, which is indeed very natural. We basically represent terms by their syntax tree, where we group together several occurrences of the same variable by agents $s$ (for *sharing*) and bind them to their corresponding $\lambda$ node (this is sometimes referred to as a *backpointer*). The nodes for abstraction and application are agents $\lambda$ and $a$ respectively, with their principal port oriented upwards. In traditional encodings, the principal port of the application agent is oriented to the left (i.e., towards the translation of the left subterm of the application), so that interaction with an abstraction ($\beta$-reduction) is always possible. Here, on the contrary, terms are translated to *principal nets* (Lafont, 1997; Lippi, 2002) with one free port connected to a principal port, called the *root* of the net, and possibly some free ports connected to auxiliary ports, corresponding to the free variables of the term. In particular, closed terms will be translated to *packages* (Lafont, 1997; Lippi, 2002). Translations of terms are thus *reduced nets*, and something will have to trigger reduction: the *evaluation token*.

This is essentially the same encoding as in (Sinot, 2005a). The only difference is that agents $s$ representing sharing have two principal ports oriented upwards, and are thus inhibited until an agent arrives from above (the evaluation token will be in charge of activating a sharing agent into a copy agent). This contrast with usual encodings (including (Sinot, 2005a)) where a standard agent is used, with its only principal port oriented downwards, so that it may readily perform copying immediately after a $\beta$-reduction. This version in fact corresponds to the agent $c$ which will be introduced in Section 5.2.

**Variables.** We consider only closed terms (open terms can be dealt with as in (Sinot, 2005a)), hence variables are not translated as such. They will simply be represented by edges between their binding $\lambda$ agent and their grouped occurrence in the body of the abstraction, as explained below.
**Application.** The translation $T(t \ u)$ of an application $t \ u$ is simply an agent $a$ whose principal port points towards the root, and with its left auxiliary port (called *function port*) linked to the root of $T(t)$ and its right auxiliary port (argument port) linked to the root of $T(u)$. If $t$ and $u$ share common free variables, then $s$ agents (representing sharing) collect these together pairwise so that a single occurrence of each free variable occurs amongst the free edges (only one such copy is represented in the figure). Note that $s$ agents have two principal ports oriented upwards, so that copy will not begin before an agent (the evaluation token) arrives from above. These will be the only agents of the system with more than one principal port.

![Diagram of application translation](image)

**Abstraction.** For an abstraction, $T(\lambda x.t)$ is obtained by introducing an agent $\lambda$, and simply linking its right auxiliary port (called *body port*) to the root of $T(t)$ and its left one (variable port) to the unique wire corresponding to $x$ in $T(t)$. If $x$ does not appear in $t$, then the left port of the agent $\lambda$ is linked to an agent $\epsilon$ (for *erasure*).

![Diagram of abstraction translation](image)

To sum up, we represent $\lambda$-terms in a very natural way. In particular, there is no encoding of boxes. Another point worth noticing is that, because of the explicit link between a variable and its binding $\lambda$ agent, $\alpha$-conversion comes for free, as it is often the case in graphical representations of the $\lambda$-calculus. So far, we have only introduced agents $\lambda$ and $a$ strictly corresponding to the $\lambda$-calculus, as well as agents $\epsilon$ and $s$ for the explicit resource management necessary (and desirable: we do not want to hide such important things) in net rewriting. Also remark that the translation of a term has no active pair, hence is in normal form, whatever interaction rules are allowed. Moreover, it has exactly one principal port, connected to the root.

5. Evaluation by interaction

In this section, we give the dynamics of the encoding, that is, the interaction rules. The difference between call-by-name and call-by-need is only visible when sharing is involved. Consequently, the part of the encoding which does not deal with it is exactly the same as in (Sinot, 2005a) (Section 5.1). There is no reason either to change the way copying and erasing are done (Section 5.3);
the difference is only on when copying should occur, i.e. when we should activate a sharing agent into a copying agent (Section 5.2).

5.1. Linear part

We introduce two new unary agents ⇓ and ⇑. To start the evaluation, we simply build the following net, that we will denote \( \mathcal{T}(t) \).

\[ \downarrow \]

\[ \mathcal{T}(t) \]

\( \uparrow T(t) \) will be a net built in the same way, but with a \( \uparrow \) agent instead, with its principal port directed towards the root. In particular, \( \uparrow T(t) \) is always a net in normal form.

As for call-by-name, when we evaluate a term beginning by a \( \lambda \), we should return that term:

\[ \downarrow \]

\[ \mathcal{T}(t) \]

\[ \uparrow \]

\[ \lambda \]

\[ \Rightarrow \]

\[ \lambda \]

We take the opportunity of this first rule to recall that the interaction rules always have an implicit name: for instance, the above rule is called \( \downarrow \bowtie \lambda \), and reduction using this rule is written \( \downarrow \bowtie \Rightarrow \lambda \).

To evaluate a term whose head symbol is an application, we should first evaluate its left sub-term. In other words, we should move the evaluation token to the function port of the application. We also rename the agent \( a \) to \( \oplus \), which is still representing an application, but with its principal port no longer pointing towards the root but to the left, so that interaction will be possible when the evaluation token returns.

\[ \downarrow \]

\[ \lambda \]

\[ \Rightarrow \]

\[ \lambda \]

\[ \uparrow \]

\[ \oplus \]

Finally, when the agent \( \uparrow \) returns from a successful evaluation to an agent \( \ominus \), then we know for sure that there is a \( \lambda \) just below the \( \uparrow \) (this will be proved in Lemma 6.3), and a \( \beta \)-reduction should be performed. Due to the restriction to binary interaction, this takes two steps:
We link the variable port of the $\lambda$ to the argument port of the $\otimes$, which initiates the substitution; and we pursue evaluation on the body of the abstraction. This is the core of the interaction net machinery for linear $\lambda$-terms; it is the same as for call-by-name.

5.2. Sharing

Sharing is represented by agents $s$. When the evaluation token reaches an agent $s$ that means that evaluation of the shared subterm is required. This is done very simply by moving the evaluation token down to the shared subterm. The agent $s$ is then renamed to an agent $s'$ looking down, so that interaction will be possible when the token returns. We also have to remember if the token comes from the left or from the right of the agent $s$, in order to continue from the same position. This is why we in fact introduce agents $s'_l$ and $s'_r$.

When the token returns to an agent $s'$, then we initiate the copying process with a $c$ agent and propagate the token to the original position (left or right, as remembered in the agent). There is no need to resume evaluation (i.e. produce a $\Downarrow$ token) because the subterm is already evaluated (Lemma 6.3). (This will not always be the case for fully lazy reduction, see Section 7).
Remark 5.1. Because of our encoding and because we follow a normal order strategy (we always go left in an application), it will often be the case that the token reaches an agent on its left port. However, this is not always true, for instance in the term \((\lambda x. (\lambda y. \lambda z. z y) x x) (\lambda u. u)\). This is why we have to abandon the restriction to interaction nets.

5.3. Copying, erasing

Copying and erasing are done in a classical way, by agents \(\epsilon\), \(c\) and \(\delta\). The auxiliary agent \(\delta\) is introduced to duplicate abstractions, as explained below. As shown in the operational semantics of call-by-need, copying happens only on closed terms and is done at once. In other words, the interaction net implementation of the copying process should always complete, and never block on some partially copied net. The agent \(\epsilon\) erases any agent and propagates according to the following schema. For clarity, the schema is valid for \(\alpha \in \{\lambda, a, @, \epsilon, c, \delta, s''\}\) (\(\delta\) and \(s''\) are introduced below with their own schemas, which are compatible with this one):

Erasure of \(s\) agents is slightly unusual, although very natural. The intuition behind is simply that if a term is shared and one of the two copies is unused, then sharing is useless. Such rules cannot be defined in standard interaction nets, and this is indeed often considered as an equivalence in the literature.

The agent \(\epsilon\) is thus responsible for garbage collection. There are two interesting remarks to make on this subject. First, garbage collection is complete (see Proposition 6.5), because \(\beta\)-reduction, hence erasing, only ever occurs on closed terms thanks to the token. This contrasts with other implementations of the \(\lambda\)-calculus in interaction nets, where it is often needed to evaluate a net (which may not terminate) before being able to erase it. Moreover, interactions with \(\epsilon\) will never be needed to allow the token to make progress. More precisely, an \(\epsilon\) agent may appear either in a subnet disconnected from the subnet containing the unique evaluation token, or above an \(s\) agent, but not on the same side as the evaluation token. In particular, implementations are free to simply ignore disconnected subnets containing \(\epsilon\) agents or garbage collect them as a whole.

In general, the agent \(c\) duplicates any agent it meets. To duplicate an abstraction, we need an auxiliary agent \(\delta\) that will also duplicate any agent, but will stop the copy when it meets another \(\delta\) agent. Note that an agent \(\epsilon\) will thus never interact with another agent \(c\). Here, \(\alpha \in \{a, \epsilon\}\) (\(\lambda\) is excluded and the other cases do not arise).
The agent $\delta$ duplicates any agent, except itself. If it interacts with itself, it just annihilates. Here, $\alpha \in \{\lambda, a, @, \epsilon, c\}$.

A $c$ agent always tries to copy a closed, evaluated subnet, so agents $c$ and $s$ never meet (because the $s$ would have been activated first; this is proved in Lemma 6.3). However, we may have to copy an agent $s$ inside an abstraction using a $\delta$. We cannot use the generic rule $\delta \bowtie \alpha$ with $\alpha = s$ because the $s$ agent may not be surrounded by agents $\delta$, and this could cause $\delta$ agents to “escape”, as can be seen by reducing the term $(\lambda x. (\lambda y. y x) (\lambda z. z x)) (\lambda i. i)$.

When an agent $\delta$ interacts with an agent $s$, the agent $s$ is “activated” into a new agent $s''$ which behaves like a lazy agent $c$: it performs only one copy step, and is then restored into an agent $s$ (“deactivated”). Morally, this avoids the problem of copying agents $s$: we do not copy them, we instead tell them to copy. More precisely, agents $s''$ are introduced in the following way:

And they interact in the following way ($\alpha \in \{a, \epsilon, \delta\}$).
Observe that we do not attempt to preserve sharing inside an abstraction (we put $\delta$ agents instead of $s$ agents). The sharing obtained is thus exactly call-by-need in the usual sense (e.g. like in Haskell and as opposed to fully lazy reduction): there is sharing at the top-level, but no reduction is shared inside an abstraction. A finer tuning of this issue will lead to an implementation of the fully lazy strategy (Wadsworth, 1971) in Section 7.

6. Properties

6.1. Reduction

In this section, we give some properties of the interaction rules: mainly the preservation of some structural properties of nets and strong confluence.

**Definition 6.1.** Rules can be partitioned into *evaluation rules* involving a token $\downarrow$ or $\uparrow$, or an active pair $\lambda \bowtie \emptyset$, and *administrative rules* involving agents $c, s'', \delta$ or $\epsilon$. Remark that it is indeed a partition. We write $\Rightarrow_{ev}$ for reduction with an evaluation rule, and $\Rightarrow_{adm}$ for reduction with an administrative rule.

**Definition 6.2.** A net $N$ is *valid* if there exists a $\lambda$-term $t$ such that $\downarrow T(t) \Rightarrow^* N$.

From now on, all nets are supposed valid. Initially, a net corresponds to the syntax tree of a term, hence there is a natural notion of orientation of the net and of the agents in it, which can be made fully formal using types (Lafont, 1990). Moreover, this orientation is preserved by reduction, loosely for $\lambda \bowtie \emptyset$ (if $\alpha$ is below $\beta$ in $M$ and $M \Rightarrow_{\lambda \bowtie \emptyset} N$ not touching $\alpha$ and $\beta$, then, in $N$, either $\alpha$ is still below $\beta$, either they are in disconnected components), strictly for the other rules. We may now state some structural properties of valid nets.

**Lemma 6.3.** In a valid net:

1. $\downarrow$ and $\uparrow$ are never below a $\lambda$ or $\alpha$;
2. $\downarrow$ and $\uparrow$ are never above a $\emptyset$;
3. $\downarrow$ and $\uparrow$ are never below a $s, c, \delta$ or $s''$;
4. if there is a $\uparrow$, then there is a $\lambda$ below it, and there may be only $c$ agents in between;
5. if there is a $s$ below a $c$, then there is a $\lambda$ in between;
6. if there is a $\delta$ and a $\downarrow$ or $\uparrow$ in the same connex component, then the former is below the latter.

**Proof.** By induction: all properties are true for $\downarrow T(t)$ and are preserved by reduction (by checking all rules if the agents are involved in the reduction, by the above remark otherwise).

In a valid net, there may be several administrative redexes, however, we have the following result.

**Proposition 6.4.** In a valid net there is exactly one occurrence of $\downarrow$, $\uparrow$ or of a $\lambda \bowtie \emptyset$ active pair, hence there is at most one evaluation redex.

**Proof.** This is true for $\downarrow T(t)$ and all rules trivially preserve the property, except $\lambda \bowtie \emptyset$. But thanks to Lemma 6.3 (first two points), we know that there is no $\downarrow$ or $\uparrow$ in the net when this rule is used.
Classical results on packages (Lafont, 1997; Lippi, 2002) allow to state the two following properties:

Proposition 6.5.
— If \( t \) is a closed \( \lambda \)-term, then:

\[
T(t) \not\rightarrow^* \emptyset
\]

(where the right-hand side of the rule denotes the empty net).

— If \( t \) is a closed \( \lambda \)-term, then:

\[
\epsilon \\
\Rightarrow^* \\
T(t)
\]

Proof. By induction on the structure of \( t \). For the second point, we need the following lemma, easily proved by induction as well.

Proposition 6.6. Reduction (excluding erasing rules) is strongly confluent on valid nets, i.e. if \( M \) is a valid net such that \( M \not\rightarrow P \) and \( M \not\rightarrow Q \) (with \( P \neq Q \)), then there exists a net \( N \) such that \( P \not\rightarrow N \) and \( Q \not\rightarrow N \).
Proof. In a valid net, there is at most one evaluation rule applicable (by Proposition 6.4), so at least one of the reductions is administrative. But an $s$ agent cannot have a $\delta$ on one port and an evaluation token on the other, thanks to Lemma 6.3 and Proposition 6.4. Consequently, there is no overlap between evaluation and administrative rules: in this case, the reductions are independent and the diverging pair can be joined by applying the other rule. The remaining case thus involves two administrative rules. Again, if they are applied at different places, the pair is easy to join. Agents $c$ and $s$ cannot meet thanks to Lemma 6.3 (point 5), hence the only remaining cases involve an agent $s$ and agents $\delta$ or $s''$ on both its principal ports. In all of these configurations, both reductions lead to the same net where the agent $s$ has been activated into an agent $s''$. This completes the proof.

Now if we put back the garbage collection rules into the system, confluence does not hold in general because of the overlap between $\epsilon \bowtie s$ and $\downarrow\bowtie s$: if the term corresponding to the net below the agent $s$ is not weakly normalising, then the diverging reductions will never join again. This issue could be addressed by adding more principal ports to $s'_l$ and $s'_r$ and interaction rules with $\epsilon$, but there is little point in doing this, at least from an implementation point of view. However, reduction is confluent for terminating $\lambda$-terms. This can be easily seen from Corollary 6.11 (and the above counter-example shows that no better confluence result can be hoped for). It is also worth noticing that, in the folklore of interaction nets, an $\epsilon$ connected to an auxiliary port of a $c$ agent is often considered equivalent to a single edge (Asperti and Guerrini, 1998). This partly corresponds to the interaction rule $\epsilon \bowtie s$ in our setting, which is possible because $s$ has two principal ports, but it is not a surprise that it does not solve all problems.

A more general remark about token-passing nets is that reduction is more likely to be confluent than in net systems without token, as shown by Proposition 6.6 for our particular system. Another remark is that some cost in terms of number of interaction steps is due to the token agent. However, this is better seen as a cost made explicit rather than as an extra cost: in standard implementations of Lafont’s interaction nets, there is a need for some external machinery to find the next redex (Pinto, 2000). On the contrary, in token-passing nets, it is always sufficient to reduce in the neighbourhood of the token agent.

6.2. Simulation

We now want to exhibit some correctness property demonstrating that the interaction net system given in Section 5 is indeed an implementation of the rewrite system of Section 2.3, and thus of call-by-need. However, there is no point in looking for an exact correspondence: we do not care if the representations of the objects differ, as long as the results of the computations correspond. We will thus consider both terms and nets up to a certain equivalence preserving the meaning of the objects. From now on, $\Downarrow$ will stand for either $\Downarrow$ or $\Uparrow$.

Definition 6.7 (Equivalence on terms). We define $\sim$ as the least symmetric, reflexive and transitive relation such that:

- for any two-holes context $C[[]]$ and value $v$, $C[\bullet_{(\Gamma,x\mapsto v)} t][x] \sim C[\bullet_{(\Gamma,x\mapsto v)} t][v]$;
- for any context $C[\cdot]$, if $e = C[\bullet_{(\Gamma,x\mapsto u)} t]$ and $e' = C[\bullet_{(\Gamma)} t]$, if $x \notin \text{fv}(e')$ then $e \sim e'$. 


In the first clause of the definition above, we need a context with two holes because the binding \( x \mapsto v \) is not local to the subterm \( t \), but to the whole term (in the big-step notation \( \Gamma : t \), the environment \( \Gamma \) is a binder at the top-level). Moreover, there is no problem of capture since all bound variables are distinct (this is assumed in the initial term and preserved by reduction).

**Definition 6.8 (Equivalence on nets).** We define \( \approx \) as the least symmetric, reflexive, transitive relation that is stable by net contexts (i.e. plugging two equivalent nets into the same net gives two equivalent nets) and such that:

- \( a \approx @, s \approx s' \);
- \( \longrightarrow \subseteq \approx \);
- \( s \) is considered associative and commutative, i.e.:

\[
\begin{align*}
S & \approx S \\
S & \approx S
\end{align*}
\]

Another thing we have to take care of before exhibiting a simulation property is to define a sensible notion of correspondence between intermediate states during the reduction of a term and a net. This amounts to extending the translation function \( T(\cdot) \) to enriched terms instead of just \( \lambda \)-terms, which was enough to start the process.

In general, an enriched term \( e \) is of the form: \( e = C_0[x_1 \mapsto \Gamma t_1][\ldots x_n \mapsto \Gamma t_n] \) where \( t \) is a \( \lambda \)-term, \( x_1, \ldots, x_n \) are variables and \( C_0[], \ldots, C_n[] \) are contexts on \( \lambda \)-terms.

We now extend \( T(\cdot) \) step by step:

- \( T(t) \) is already defined if \( t \) is a \( \lambda \)-term. If \( t \) has free variables \( x_1, \ldots, x_n \), we assume that the corresponding free edges are temporarily labelled with the right variable. The label will not be part of the final translation: it is only used to build it.
- We define \( T(\emptyset t) \) (empty environment) as in Section 5.1 by connecting an agent \( \downarrow \) or \( \uparrow \) to \( T(t) \).
- \( T(C[\emptyset \Gamma t]) \) where \( \Gamma = (x_1 \mapsto t_1, \ldots, x_n \mapsto t_n) \) is defined by building \( T(C[\emptyset t]) \) and \( T(t_i) \) for all \( i \), and connecting every edge labelled \( x_i \) in \( T(C[\emptyset t]) \) to the root of \( T(t_i) \).
- Similarly, for \( T(C[x \mapsto t]) \), we build \( T(C[x]) \) and \( T(t) \) and connect the free edge labelled \( x \) of the former to the root of the latter.

Remark that \( T(\cdot) \) produces nets that are valid with respect to reduction only modulo \( \approx \). For instance, \( T((\uparrow_\Gamma t) u) \) has an agent \( \downarrow \) below an agent \( a \), which is not supposed to happen in a reduction: the agent \( a \) should have been transformed into \( @ \). However, everything is fine modulo \( \approx \), in the sense of the following two propositions.

**Proposition 6.9 (Completeness).** For two enriched terms \( t \) and \( u \), if \( t \rightarrow u \), then there exist \( M \) and \( N \) such that \( T(t) \approx M, M \Longrightarrow^* N \) and \( N \approx T(u) \).

**Proof.** By cases on the rule used.

- \( (\downarrow Lam): \) clear;
- \( (\downarrow App): \) clear;
— (▷App): clear (takes two steps);
— (◁Var): say ▷ (Γ,x→t)x → x ⇒ ▷Γ t, there are two cases:
  - if the occurrence of x is unique in the term or if t has already been evaluated, then the variable x is just represented by a wire, and the two translations are equal: \( T(C[Γ, x → t]) = T(C[x ⇒ ▷Γ t]) \);
  - otherwise, this corresponds to a rule ▷ ≅ s, which transforms s into \( s' \) or \( s'' \) (equivalent modulo ≈);
— (◁Var): there are again two cases:
  - in the linear case, the translations are the same on both sides;
  - in the non-linear case, this corresponds to a rule ▷ ≅ \( s' \) or ▷ ≅ \( s'' \).

We cannot impose \( M = T(t) \) in the above proposition because the principal ports in \( T(t) \) may be oriented incorrectly (as in \( T(⟨Γ, t⟩ u) \)), but there is indeed a correctly oriented representative in the class of \( T(t) \).

For the other direction, there is a detail to take care of: the rule ▷ ≅ @ creates an intermediate net that does not correspond to a valid term until the corresponding rule \( λ ⊿ @ \) is applied. Hence we do not reason on reduction \( \implies \) but rather on \( \implies \) defined as follows:

— \( M \implies N \) if there exists \( P \) such that \( M \implies P \implies N \);  
— \( M \implies N \) if \( M \implies N \) by another rule.

**Proposition 6.10 (Correctness).** If \( M \implies N \) and \( M \approx T(t) \) for some enriched term \( t \) then there exist \( t' \) and \( u \) such that \( t ≈ t', t' \rightarrow u \) and \( N \approx T(u) \).

**Proof.** By cases on the family of rules used.

— rules ▷ ≅ λ, ▷ ≅ a, ▷ ≅ @ + λ ≅ @: clear (some rules moving variables between the context and construct of the form \( x ⇒ t \) may be applied implicitly in the linear case);
— rules ▷ ≅ s: \( t' \) is of the form \( C[Γ, x → v]x \) (with \( x \) non-linear in \( t' \)), and indeed \( T(C[Γ, x → v]) \approx N \);
— rules ▷ ≅ \( s' \): similarly;
— administrative rules: true with \( t = t' = u \), since \( \approx \) contains this \( \implies \) step.

**Corollary 6.11 (Simulation of call-by-need).**

— if \( Γ : t \downarrow Δ : v \), then \( T(⟨Γ, t⟩) \implies T(⟨Δ v⟩) \);  
— if \( T(⟨Γ, t⟩) \implies T(⟨Δ v⟩) \), then there are \( v', Δ' \) such that \( v' Δ v \sim Δ v \sim Δ v' \) and \( Γ : t \downarrow Δ : v' \).

**Proof.** We simply put together the results.

— By Propositions 2.2 and 6.9, there are nets \( M \) and \( N \) such that \( T(⟨Γ, t⟩) \approx M, N \approx T(⟨Δ v⟩) \) and \( M \implies N \), and we may choose \( N \) in normal form without loss of generality. Since the token is at the top, there is no \( s' \) in \( M \) or \( N \) and there is no harm in choosing to mark applications \( a \) instead of @. So indeed \( T(⟨Γ, t⟩) \implies T(⟨Δ v⟩) \).
— First notice that the reduct has a token agent in it, hence the \( \Rightarrow^* \) reduction is also a \( \equiv^* \) reduction. We use Propositions 2.2 and 6.10, and an argument similar to the one used above allows to get rid of the \( \approx \). However, we cannot get rid of the \( \sim \) since a net may be read back into several different enriched terms.

Remark that this corollary is weaker than the propositions, since it is a simulation property only with respect to the normal forms, and not to the reductions themselves.

7. Fully lazy reduction

The call-by-need strategy, for which we have given above an implementation using net rewriting, captures only the sharing of values. For example, evaluation of the term \( (\lambda f.f I)(I I) w \) where \( I = \lambda x.x \) will evaluate the redex \( I I \) twice, because the subterm \( \lambda w.(I I) w \) will be shared, then copied as a whole when necessary. This is what happens in the implementation above, as well as in standard implementations of call-by-need, e.g. in the G-machine (Peyton Jones and Salkild, 1989). This is not usually considered as a problem, because this term can also be transformed into \( (\lambda f.f I(f I))(\lambda z.(\lambda w.z w))(I I) \) in which the redex \( I I \) will be evaluated only once. This transformation is called fully lazy \( \lambda \) -lifting (Peyton Jones, 1987, Chapter 15).

However, it is also possible to share directly the evaluation of this redex. Implementations achieving this are called fully lazy. Wadsworth was the first to provide a fully lazy interpreter (Wadsworth, 1971): he noticed that the redex \( I I \) should not be copied since no occurrence of the bound variable \( w \) occurs in it. However, his algorithm is neither intuitive or efficient; it is thus natural to try to adapt the work done so far to fully lazy sharing.

The evaluator proposed in Section 5 relies in a crucial way on the facts, on one hand, that whenever a copy is started on a closed term, it may always complete; on the other, that the token ensures that reduction happens only at a point where the term is closed (i.e. the token does not go under \( \lambda \)'s). This would not be true for a fully lazy evaluator. The difficulty lies in that we are using a distributed framework: the copying is performed by local steps, and two different copying processes may be interleaved, so it may not be obvious to know when to stop.

This problem disappears in a non-distributed framework; for instance, in (Shivers and Wand, 2005), an algorithm for fully lazy reduction is described in an imperative style. The trick is to make two passes for copying: in the first pass, nodes are copied and marked as such, and an already marked node is never copied; in the second pass marks are erased. This is clearly not applicable here.

In our framework the problem boils down to that of deciding when two agents \( \delta \) meet, if they should copy each other or cancel out. This problem is known to be hard in the context of optimal reduction (Lamping, 1990), where it is known as the problem of implementing (efficiently) the boxes of linear logic. Before even starting to work on a solution, we can already say that such a solution will not be completely satisfactory. So far, the evaluation token had two benefits: it restricted evaluation to the needed redexes and allowed to manage copying in a very simple way. In a fully lazy setting, this last point will not hold: we have reached a limit of the token-passing approach.

Now there are several known solutions to the problem of giving the right scope to copying...
agents. The framework RINO developed by Lang (Lang, 1998) is the easiest to adopt for our purpose, because he does not introduce extra agents (he rather adds some structure to already existing agents) and his evaluator is already (essentially) fully lazy, although he does not make such a claim. We will thus freely borrow his technology and results. The full details and proofs can be found in (Lang, 1998).

Lang’s technology relies on giving labels to certain agents, so we first introduce some terminology. We assume given an infinite set of atomic labels or letters (denoted $x, x_1, x_2, y, \ldots$). We then define a label or word (denoted $\sigma, \rho, \ldots$) as an ordered sequence of atomic labels, and we write $\epsilon$ for the empty label. Prefixing, postfixing and concatenation are denoted by the absence of symbol when there is no risk of confusion. We define the length of a label $\sigma$, written $|\sigma|$ as the number of letters of $\sigma$. Moreover we write $\rho \leq \sigma$ if $\rho$ is a prefix of $\sigma$.

Our starting point is the encoding of Section 4. The agents are modified in the following way.

- The agents $\lambda$ are annotated with an atomic label which is unique in the net considered: the starting net has unique atomic labels on $\lambda$ agents, and when a $\lambda$ agent is copied, the two copies are given fresh atomic labels (i.e. atomic labels that have never previously appeared in the net), see Figure 1.

- The former agents $c$ and $\delta$ are now coalesced into a single type of agent $c_\sigma$, annotated with a non-atomic label. All labels are initially empty (i.e. equal to $\epsilon$). The former $c$ agents correspond to an empty label, while the $\delta$ agents correspond to $c_\sigma$ with $|\sigma| \geq 1$. This is a consequence of the rule $c_\sigma \triangleright \lambda$, see Figure 1.

- The agents $s$ are annotated with a label as well. The previous agent $s$ was an inhibited version of the agent $c$, used to share closed subterms. Now $s_\sigma$ is also an inhibited version of $\delta$ (in the case $|\sigma| \geq 1$) and it may share possibly open subterms.

- The agent $\delta$ was used for two dual purposes: to share a subterm (i.e. as a fan in, in the literature) and to end the scope of the sharing of a context (fan out). Now these two usages are syntactically distinguished as $c_\sigma$ for the first one, and $\tau_\sigma$ for the second. This is necessary because of the possible asymmetry in the rule $c_\sigma \triangleright \tau_\rho$ (see Figure 1), contrasting with Section 5.

- There is no longer an agent $s''$. Indeed the default behaviour of the new agent $c_\sigma$ is to perform one copy step and then to turn back into a $s_\sigma$ and share the subterm until the token makes a further copy necessary, similarly to $s''$ as given in Section 5.

- And finally, there is a new type of token, written $\uparrow$, used when the evaluated subnet under it is of the form: a stack of applications beginning by an agent $\tau_\sigma$. Its role is to go up and find an agent $s'_\sigma$ (there is necessarily one), as opposed to the standard token $\uparrow$ whose role is to find an application.

Our fully lazy net interpreter consists of the rules of Section 5 (adding labels where necessary), with the modifications and additions shown in Figure 1, commented below.

- An agent $\uparrow$ is created when the token reaches a $\tau_\sigma$ and goes up the stack of applications above it until it finds an agent $s'_\sigma$ (there is necessarily one).

- When the token $\uparrow$ returns to an agent $s'_\sigma$, then we initiate the copying process with a $c_\sigma$ agent and resume evaluation from the original position (left or right, as remembered in the agent).

- When an application is copied, we know that the left subterm will be needed, hence copied. We thus prefer to generate directly a $c_\sigma$ instead of a $s_\sigma$, as a cheap optimisation.
— When a $c_{\sigma}$ agent copies a $\lambda$ agent labelled $x$, the two copies of the $\lambda$ are given fresh labels (i.e. labels that have never appeared before in the net), while the new $\tau$ and $s$ agents are labelled by $\sigma x$ ($x$ postfixed to $\sigma$), see Figure 1.

— When two agents $c_{\sigma}$ and $c_{\rho}$ are facing each other, their labels are compared for prefixing. If the label of the closing copy agent $c_{\rho}$ is a prefix of the other, then they both come from the same $\lambda$ and should annihilate (the copy is finished). Otherwise, they do not match, and they should copy each other. Note that in this case, the rule is asymmetric so that we need to introduce different agents $c_{\sigma}$ and $c_{\rho}$.

— To understand the role of the labels on an example, the puzzled reader is invited to reduce the term $(\lambda z. z \ z) (\lambda x. (\lambda z. z \ z) (\lambda y. x \ y))$: the body of the inner abstraction is copied twice, and the labels ensure that the copying agents match correctly.

In brief, the net interpreter we propose is a reformulation of the one found in (Lang, 1998). We thus lack legitimacy to spend more space on properties of this interpreter. Correctness of the labelling and simulation with respect to the $\lambda$-calculus are given in (Lang, 1998) and easily adapted. We may nevertheless notice an interesting common point between the systems developed by Lang, Lippi and ourself: there are two distinct agents for application, one that is used for $\beta$-reduction and one that is used for duplication, and these agents can be converted from one type to the other by certain interaction rules with other agents.

Lang’s implementation is essentially fully lazy, but this is not exactly true since normal order is not imposed (this is the main role of the token here), hence some useless computations may be performed. On the other hand, it uses standard interaction nets (no multiple principal ports), which is not possible with the token-passing approach, because the token may appear on any side of a sharing agent. We may briefly state this strength of our presentation:

**Proposition 7.1.** The net rewriting rules given in Figure 1 perform fully lazy reduction.

**Proof.** Fully lazy reduction means sharing of maximal free expressions (Peyton Jones, 1987). Whatever they are, one just has to notice that our implementation shares everything, until the token says that evaluation is indeed required. Our implementation is thus trivially fully lazy.

The observation that Lang’s interpreter is (almost) fully lazy would probably be difficult to make without our framework with a token; indeed Lang does not make such a claim (except that of being experimentally efficient). Once this observation has been made, it is probably fair to say that Lang’s implementation is probably more efficient than ours, in particular in a true distributed setting. The evaluation token indeed makes the strategy more explicit, but it may also have a cost, depending on how token-passing nets are implemented. Lang’s implementation does not have such (potential) extra cost, and moreover uses standard interaction nets (no multiple principal ports). Our presentation can be seen as a formalisation of what would probably be a real implementation of Lang’s interpreter in a sequential language. It is also fair to say that our presentation aims at simplicity and does not consider possible optimisations. We indeed think that our framework could be a good candidate to reason about such possible optimisations and prove them formally.
8. Conclusion

We have presented a simple approach to express the lazy and fully lazy strategies of the λ-calculus in net rewriting. For standard call-by-need, the approach is so simple that it is indeed a good alternative to working with terms and environments. It is obtained from an encoding of call-by-name without adding any extra structure, and without much complication. Moreover, some meaningless differences in the representation as pairs of term and environment are quotiented out in the net representation, which makes the approach even simpler. For fully lazy reduction, the framework is less simple and borrows some of the technology developed by Lang, thus reintroducing some mechanism to match copying agents reminiscent of the boxes of linear logic. However, this is not a weakness of our framework; this is simply the price to pay to have such a fine notion of sharing. The framework of interaction nets had to be abandoned, but no property is lost. This can be seen both as a simple formalisation of Wadsworth’s original ideas in a distributed framework, and as the basis for distributed graph-based abstract machines.
Appendix A. Equivalence of big-step semantics

We present Launchbury’s big-step semantics of call-by-need (Launchbury, 1993) and prove its equivalence with the simplified version presented in Section 2.2. The set of usual \( \lambda \)-terms is denoted \( \Lambda \).

**Definition A.1.** \( \Lambda_{\text{let}} \) is the set of \( \lambda \)-terms with \textit{lets} defined by:

\[
\begin{align*}
t, u &::= x \mid \lambda x.t \mid t x \\
&\mid \text{let } x = u \text{ in } t x \quad \text{(if } x \notin \text{fv}(t))
\end{align*}
\]

**Definition A.2.** The compilation function \((\cdot)^* : \Lambda \rightarrow \Lambda_{\text{let}}\) is defined by:

\[
\begin{align*}
x^* &= x \\
(\lambda x.t)^* &= \lambda x.(t^*) \\
(t u)^* &= \begin{cases} 
(t^*) u & \text{if } u \text{ is a variable}, \\
\text{let } x = u \text{ in } t x & \text{otherwise (} x \text{ is a fresh variable).}
\end{cases}
\end{align*}
\]

**Definition A.3.** Launchbury’s big-step semantics of call-by-need is denoted \( \Downarrow_L \) and is defined on \( \Lambda_{\text{let}} \) as follows. All bound variables are assumed to be initially different.

\[
\begin{align*}
\frac{\Gamma : \lambda x.t \Downarrow_L \Gamma : \lambda x.t}{Lam_L} \\
\frac{\Gamma : t \Downarrow_L \Delta : \lambda y.t' \quad \Delta : t'[y := x] \Downarrow_L \Theta : v}{\Gamma : t x \Downarrow_L \Theta : v \quad \text{App}_L} \\
\frac{\Gamma : t \Downarrow_L \Delta : v}{\Gamma, x \mapsto t : x \Downarrow_L (\Delta, x \mapsto v) : \hat{v} \quad \text{Var}_L} \\
\frac{\Gamma, x \mapsto u : t \Downarrow_L \Delta : v}{\Gamma : \text{let } x = u \text{ in } t \Downarrow_L \Delta : v \quad \text{Let}_L}
\end{align*}
\]

Terms of \( \Lambda_{\text{let}} \) enjoy some structural properties. First, if all variables are distinct in \( \Gamma : t \) and \( \Gamma : t \Downarrow_L \Delta : v \), then all variables are distinct in \( \Delta : v \) (Launchbury, 1993). In other words, it is sufficient to perform \( \alpha \)-conversion in rule \( \text{Var}_L \) only to keep all bound variables distinct. Moreover, for any \( t \in \Lambda_{\text{let}} \), if a subterm of \( t \) is an application \( u v \), then \( v \) is a variable, and if a subterm of \( t \) is of the form \( \text{let } x = u \text{ in } v \), then \( v = w x \) with \( x \notin \text{fv}(w) \). It is easy to see that the compilation \((\cdot)^*\) enforces these properties and that the evaluation \( \Downarrow_L \) preserves them, which justifies the above definitions, as well as this one:
Definition A.4. We define the readback function \((\cdot)^\circ : \Lambda_{\text{let}} \to \Lambda\) by:
\[
\begin{align*}
\lambda x.t^\circ &= \lambda x.(t^\circ) \\
(t x)^\circ &= t^\circ x \\
(let \ x = u \ in \ t)^\circ &= t^\circ u^\circ
\end{align*}
\]

Lemma A.5. Compilation and readback are inverses (modulo \(\alpha\)-conversion):
- \((\cdot)^\circ \circ (\cdot)^* = \text{id}_\Lambda\)
- \((\cdot)^* \circ (\cdot)^\circ = \text{id}_{\Lambda_{\text{let}}}\)

Proof. Clear.

Lemma A.6.
If \(x\) is a fresh variable (for \(\Gamma : t\) and \(u\) is a \(\Lambda_{\text{let}}\)-term, then \(\Gamma : t \Downarrow_L \Delta : v\) if and only if \((\Gamma, x \mapsto u) : t \Downarrow_L (\Delta, x \mapsto u) : v\).

Proof. Clear.

We extend \((\cdot)^*\) and \((\cdot)^\circ\) to environments and to pairs \(\Gamma : t\) in the obvious way. We also identify the relations \(\Downarrow\) and \(\Downarrow_L\) with their associated functions. We are now in a position to state the main theorem of this section.

Theorem A.7. \(\Downarrow\) and \(\Downarrow_L\) are bisimilar (modulo \(\alpha\)-conversion) in the following sense:
- \(\Downarrow = (\cdot)^\circ \circ \Downarrow_L \circ (\cdot)^*\)
- \(\Downarrow_L = (\cdot)^* \circ \Downarrow \circ (\cdot)^\circ\)

Proof. Thanks to Lemma A.5, both points are equivalent. We only develop the difficult case. Let us assume \(\Gamma : t \Downarrow \Theta : v\) and let us show that \((\Gamma : t u)^* \Downarrow_L \Theta' : v'\) with \((\Theta' : v')^\circ = \Theta : v\).

By rule \(\text{App}\), there exist \(\Delta, r\) such that \(\Gamma : t \Downarrow \Delta : \lambda x.r\) and \((\Delta, x \mapsto u) : r \Downarrow \Theta : v\). By induction, there exist \(\Delta', t'\) such that \((\Gamma : t)^* \Downarrow_L \Delta' : t'(1)\) and \((\Delta' : t')^\circ = \Delta : \lambda x.r\).

By Definition A.4, there exists \(r'\) such that \(t' = \lambda x.r'\) (2). Using Lemma A.5, \(\Delta' = \Delta^*\) and \(r' = r^*\) (3). By induction again, there exist \(\Theta', v'\) such that \((1, (\Delta, x \mapsto u) : r)^* \Downarrow_L \Theta' : v'\) (4) and \((\Theta' : v')^\circ = \Theta : v\) (5). The following derivation and fact (5) allow to conclude this case.

The other two cases are easy.
References


