Math. Struct. in Comp. Science (2006), *vol.* 16, *pp.* 639–666. © 2006 Cambridge University Press
doi:10.1017/S0960129506005408 Printed in the United Kingdom

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 deliberately follow the implemented strategies (call-by-name and call-by-value) as closely as possible, and hence do not benefit from the ability of interaction nets to represent sharing easily. 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, no 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.

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 non-linear steps of a computation, such as copying and erasing, which are normally hidden. This is inspired by proof nets and linear logic (Girard 1987). In particular, sharing is built in (as opposed to through the use of terms) and is dealt with explicitly (as opposed to through the use of termgraphs). The 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 not widely used by implementors of real-world programming languages. While it is difficult to say why this is the case, it may well be that work such as that on optimal reduction has led some to think of interaction nets as a tool for theoreticians alone. For these reasons, we believe it is worth bridging 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)

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

implementations of the λ-calculus (Mackie 1998; 2004). A common feature of all these encodings of the λ-calculus is that a β-redex is always translated to an active pair (that is, a redex in interaction nets). Hence, paradoxically, while all reductions are equivalent, an external interpreter is still required to find the redexes and to manage them. This interpreter is typically implemented by maintaining a stack of redexes (Pinto 2000), or by *ad hoc* methods, which are not documented and not part of the theory, and thus error-prone. The fact that interaction steps implementing different β-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 worlds of interaction nets and abstract machines. In Sinot (2005), we began to bridge this gap by giving encodings of the call-by-name and call-by-value strategies of the λ-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 avoiding the need for boxes, and hence to simplifying the encoding.

In this paper, our goal is to bridge the gap further and to push forward a more uniform framework for describing implementations of the λ-calculus. More precisely, we will give a presentation of the call-by-need strategy in the style of Sinot (2005). 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 that was absent from the earlier 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 also encode the fully lazy strategy in interaction nets, 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), which could not be made without our framework. To put this 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 better understanding optimal reduction, although the remaining path may still be difficult.

In Sinot (2005) 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, so reduction is essentially deterministic. On the other hand, some restrictions that 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. However, since reduction is directed by a unique token, evaluation is still fully deterministic. Thus, another important aspect of this work is 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 it is, it should be evaluated once only. The original formulation is in terms of graph rewriting, but 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) and 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, which is radically different from previous work, and better suited to our needs. In this respect, we follow the approach of Sinot (2005).

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 the reduction rules are not much more complicated, so 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.

The present work is a substantially extended and improved version of Sinot (2006). 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 give conclusions in Section 8.

2. Call-by-need

2.1. *Preliminaries*

We assume a basic knowledge of the λ-calculus; see Barendregt (1984) for more details. To fix notation, the set of λ-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 α-conversion, that is, renaming of bound variables. We use $\mathsf{fv}(t)$ to denote the set of free variables of a term

t. If t is a λ-term, $\hat{t}$ is a term α-equivalent to t in which all the bound variables have been renamed to fresh variables.

This set is equipped with the following rule of β-reduction:

$$(\beta) \qquad (\lambda x.t)\ u \to_\beta t\{x := u\}$$

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 $\to_\beta$ for one-step reduction and $\to_\beta^*$ for its reflexive transitive closure.

We call terms of the form $\lambda x.t$ or $x\ t_1 \ldots t_n$ weak head normal forms. 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 $t \to_\beta^* v$.

Contexts (or more precisely one-hole contexts) on λ-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 (for example, Γ, Δ, Θ) are mappings from variables to terms. Evaluation is defined on distinctly named pairs $\Gamma : t$ of an environment and a term. Unlike call-by-name and call-by-value evaluation, call-by-need (or lazy) 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 Γ. Evaluation judgements are of the form $\Gamma : t \Downarrow \Delta : v$, which is to be read 'the term t in the environment Γ reduces to the value v together with the new environment Δ', as defined by the following set of deduction rules:

$$\frac{}{\Gamma : \lambda x.t \Downarrow \Gamma : \lambda x.t}\ \mathit{Lam}$$

$$\frac{\Gamma : t \Downarrow \Delta : \lambda x.t' \quad (\Delta, x \mapsto u) : t' \Downarrow \Theta : v}{\Gamma : t\ u \Downarrow \Theta : v}\ \mathit{App}$$

$$\frac{\Gamma : t \Downarrow \Delta : v}{(\Gamma, x \mapsto t) : x \Downarrow (\Delta, x \mapsto v) : \hat{v}}\ \mathit{Var}$$

This corresponds fairly 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 λ-calculus with *lets* so that the closures are already explicit. We instead generate a new binding in the environment in rule *App* (corresponding to β-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, and there is no need for a calculus with *lets* (recursive bindings can be dealt with using standard λ-calculus fixpoints). It is also easier to use since the evaluation relation is well-defined on all terms, not just on precompiled terms. Moreover, our semantics is equivalent to Launchbury's (see Appendix A), which justifies our saying that it is indeed a semantics of call-by-need, and allows us to borrow results freely from Launchbury (1993) when necessary. In particular, we can ensure that all bound variables remain distinct by carrying out α-conversions in rule *Var* alone.

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 just 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, which will be derived from the big-step presentation in a fairly 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) or we may go down in the tree (put the result back in the correct context).

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

$$e ::= \Downarrow_{\Gamma} t \mid \Uparrow_{\Gamma} t \mid x \Mapsto e \mid C[e]$$

where t stands for a λ-term, Γ for an environment, x for a variable and $C[\,]$ for a one-hole context on λ-terms.

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

$$\begin{array}{clcl}
(\Downarrow Lam) & \Downarrow_\Gamma \lambda x.t & \to & \Uparrow_\Gamma \lambda x.t \\
(\Downarrow App) & \Downarrow_\Gamma (t\ u) & \to & (\Downarrow_\Gamma t)\ u \\
(\Uparrow App) & (\Uparrow_\Gamma \lambda x.t)\ u & \to & \Downarrow_{(\Gamma, x \mapsto u)} t \\
(\Downarrow Var) & \Downarrow_{(\Gamma, x \mapsto t)} x & \to & x \Mapsto \Downarrow_\Gamma t \\
(\Uparrow Var) & x \Mapsto \Uparrow_\Gamma v & \to & \Uparrow_{(\Gamma, x \mapsto v)} \hat{v}.
\end{array}$$

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) and Maraist *et al.* (1998). However, our approach 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 the previous work consists of designing clever evaluation contexts to restrict reduction and in encoding environments as terms in a contrived way.

We can also compare our work with work on abstract machines, such as, for example, the lazy Krivine machine (Crégut 1990). Our presentation is probably closer to this approach, 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. Small-step presentations usually rely on inductive rules that allow reductions in a certain class of contexts, and hence, unlike in our presentation, they do not make the flow of evaluation explicit, which is crucial for the encoding into interaction nets. In some sense, our presentation is intermediate between traditional small-step semantics (which separate reduction and strategy as much as possible) and abstract machines (which may involve complex data structures). We call this presentation the *token-passing semantics* of call-by-need.

A λ-term t is always in normal form with respect to this system, and so is $\Uparrow_\Gamma t$. To evaluate t in a context Γ, we have to start reduction from $\Downarrow_\Gamma t$. In general, if Γ is undefined for some free variables of t, the call-by-need evaluation of $\Gamma : t$ may fail. This corresponds here to the reduction starting from $\Downarrow_\Gamma t$ terminating on an enriched term that is not of the form $\Uparrow_\Delta u$. If t is closed, which is generally assumed, we may safely choose Γ to be empty.

This system enjoys the following strong invariant.

Proposition 2.1. If $\Downarrow_\Gamma t \to^* u$, then there is exactly one occurrence of $\Downarrow_\Delta$ or $\Uparrow_\Delta$ in u (for some Δ).

Proof. The proof is by induction on the length of the reduction and by cases on the last rule applied. In rule $\Uparrow Var$, v is copied but, by the induction hypothesis, it has no occurrence of $\Downarrow$ or $\Uparrow$. □

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. $\Gamma : t \Downarrow \Delta : v \iff \Downarrow_\Gamma t \to^* \Uparrow_\Delta v$.

Proof. Both implications are systematic.

$\Rightarrow$ We use induction on the derivation:

– *Lam*: $\Gamma : \lambda x.t \Downarrow \Gamma : \lambda x.t$ and indeed $\Downarrow_\Gamma \lambda x.t \xrightarrow[\Downarrow Lam]{} \Uparrow_\Gamma \lambda x.t$.

– *App*: If $\Gamma : t\ u \Downarrow \Theta : v$, then there exists t' and Δ such that $\Gamma : t \Downarrow \Delta : \lambda x.t'$ and $(\Delta, x \mapsto u) : t' \Downarrow \Theta : v$. By induction, $\Downarrow_\Gamma t \to^* \Uparrow_\Delta \lambda x.t'$ and $\Downarrow_{(\Delta,x\mapsto u)} t' \to^* \Uparrow_\Theta v$, hence:

$$\Downarrow_\Gamma (t\ u) \xrightarrow[\Downarrow App]{} (\Downarrow_\Gamma t)\ u \to^* (\Uparrow_\Delta \lambda x.t')\ u \xrightarrow[\Uparrow App]{} \Downarrow_{(\Delta,x\mapsto u)} t' \to^* \Uparrow_\Theta v.$$

– *Var*: If $(\Gamma, x \mapsto t) : x \Downarrow (\Delta, x \mapsto v) : \hat{v}$, then $\Gamma : t \Downarrow \Delta : v$. Hence, by induction, $\Downarrow_\Gamma t \to^* \Uparrow_\Delta v$, and finally

$$\Downarrow_{(\Gamma,x\mapsto t)} x \xrightarrow[\Downarrow Var]{} (x \Mapsto \Downarrow_\Gamma t) \to^* (x \Mapsto \Uparrow_\Delta v) \xrightarrow[\Uparrow Var]{} \Uparrow_{(\Delta,x\mapsto v)} \hat{v}.$$

$\Leftarrow$ We use structural induction on the term t:

– Abstraction: $\Downarrow_\Gamma \lambda x.t \xrightarrow[\Downarrow Lam]{} \Uparrow_\Gamma \lambda x.t$ and, indeed, $\Gamma : \lambda x.t \Downarrow \Gamma : \lambda x.t$ by (*Lam*).

– Application: Assume $\Downarrow_\Gamma (t\ u) \to^* \Uparrow_\Theta v$. Thus we have

$$\Downarrow_\Gamma (t\ u) \xrightarrow[\Downarrow App]{} (\Downarrow_\Gamma t)\ u \to^* \Uparrow_\Theta v.$$

The final result is no longer an application, and the rule $\Uparrow App$ is the only one that can make this happen. Thus we must have $\Downarrow_\Gamma t \to^* \Uparrow_\Delta \lambda x.t'$ for some Δ and t'. Moreover, $\Gamma : t \Downarrow \Delta : \lambda x.t'$ by induction. Then

$$(\Uparrow_\Delta \lambda x.t')\ u \xrightarrow[\Uparrow App]{} \Downarrow_{(\Delta,x\mapsto u)} t' \to^* \Uparrow_\Theta v,$$

so $(\Delta, x \mapsto u) : t' \Downarrow \Theta : v$ by induction and $\Gamma : t\ u \Downarrow \Theta : v$ by rule (*App*).

– Variable: Assume

$$\Downarrow_{(\Gamma,x\mapsto t)} x \xrightarrow[\Downarrow Var]{} (x \Mapsto \Downarrow_\Gamma t) \to^* \Uparrow_{(\Delta,x\mapsto v)} \hat{v}.$$

The last rule applied must be

$$(x \Mapsto \Uparrow_\Delta v) \xrightarrow[\Uparrow Var]{} \Uparrow_{(\Delta,x\mapsto v)} \hat{v},$$

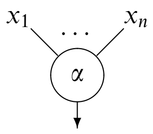
so $\Downarrow_\Gamma t \to^* \Uparrow_\Delta v$, necessarily, and, by induction, $\Gamma : t \Downarrow \Delta : v$. Using (*Var*), we get $(\Gamma, x \mapsto t) : x \Downarrow (\Delta, x \mapsto v) : \hat{v}$. □

Hence the given rewrite system faithfully corresponds to the call-by-need strategy. In particular, as with 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

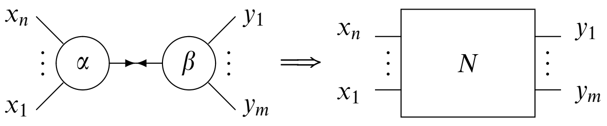
3.1. *Interaction nets*

A system of interaction nets (Lafont 1990) is specified using a set Σ 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 α 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 as follows:



Intuitively, a net N built on Σ is a (not necessarily connected) graph with agents at the vertices. The edges of the graph connect agents together at the ports such that there is only one edge at each 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 $((\alpha, \beta) \Longrightarrow N) \in \mathcal{R}$ 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, that is, 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; so such a rule will sometimes also be denoted by $\alpha \bowtie \beta$. The following diagram shows the format of interaction rules (N can be any net built from Σ):



We use the notation $\Longrightarrow$ for the one-step reduction relation, or $\underset{\alpha \bowtie \beta}{\Longrightarrow}$ if we want to be explicit about the rule used, and $\Longrightarrow^*$ for its transitive and reflexive closure. If a net does not contain any active pairs, 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 ones in which the agents are allowed to have any number of principal ports, instead of just one. They were 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).

So, each agent has a fixed number of principal ports, each of which is 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 call-by-need (Section 5) and fully lazy reduction (Section 7), respectively. However, we feel that these two systems, as well as the systems of Sinot (2005), share some common features, so it makes sense to include them all under a common heading. We have chosen to call them *token-passing nets*, because a particular agent is passed around and enables certain reductions. It is beyond the scope of this paper to give a formal definition of token-passing nets, or to prove any general properties for them. It is also too soon: we should first find some other interesting examples of token-passing nets before presenting general concepts about such systems – we leave this for future work.

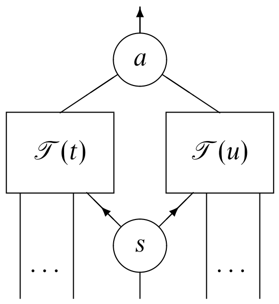
4. Encoding of terms

In this section, we define a translation function $\mathscr{T}(\cdot)$ from λ-terms into nets, which is indeed very natural. We basically represent terms by their syntax trees, where we group together several occurrences of the same variable using agents s (for *sharing*) and bind them to their corresponding λ node (this is sometimes referred to as a *backpointer*). The nodes for abstraction and application are agents λ and a, respectively, with their principal port oriented upwards. In traditional encodings, the principal port of the application agent is oriented to the left (that is, towards the translation of the left subterm of the application), so that interaction with an abstraction (β-reduction) is always possible. Here, however, 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*.

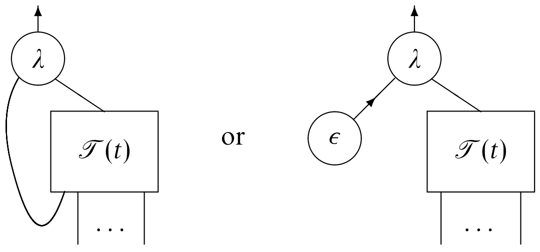
This is essentially the same encoding as in Sinot (2005). The only difference is that agents s representing sharing have two principal ports oriented upwards, and are thus inhibited until an agent arrives from above (the evaluation token will be in charge of activating a sharing agent to become a copy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

Variables. We just consider closed terms (see Sinot (2005) for how we could handle open terms), so variables are not translated as such. They will simply be represented by edges between their binding λ agent and their grouped occurrence in the body of the abstraction, as explained below.

Application. The translation $\mathscr{T}(t\ u)$ of an application $t\ u$ is simply an agent a whose principal port points towards the root, whose left auxiliary port (its *function port*) is linked to the root of $\mathscr{T}(t)$ and whose right auxiliary port (its *argument port*) is linked to the root of $\mathscr{T}(u)$. 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 copying 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.



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



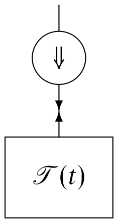
To sum up, we represent λ-terms in a very natural way. In particular, there is no encoding of boxes. Another point worth noticing is that, because of the explicit link between a variable and its binding λ agent, α-conversion comes for free, as is often the case in graphical representations of the λ-calculus. So far, we have only introduced agents λ and a strictly corresponding to the λ-calculus, as well as agents ϵ and s for the explicit resource management necessary (and desirable: we do not want to hide such important things) in net rewriting. Also note that the translation of a term has no active pair, so it is in normal form, whatever interaction rules are allowed. Moreover, it has exactly one principal port, which is 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 that does not deal with sharing (Section 5.1) is exactly the same as in Sinot (2005). There is no reason either to change the way copying and erasing are done (Section 5.3); the difference lies only in *when* copying should occur, that is, when we should activate a sharing agent to become a copying agent (Section 5.2).

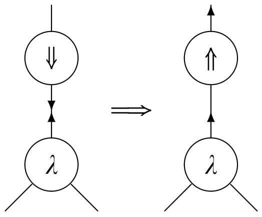
5.1. *Linear part*

We introduce two new unary agents $\Downarrow$ and $\Uparrow$. To start the evaluation, we simply build the following net, which we will denote $\Downarrow\mathscr{T}(t)$:



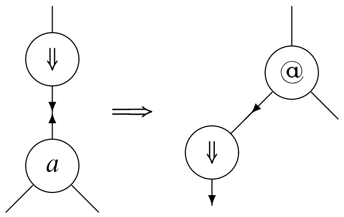
$\Uparrow\mathscr{T}(t)$ will be a net built in the same way, but with a $\Uparrow$ agent instead, with its principal port directed towards the root. In particular, $\Uparrow\mathscr{T}(t)$ is always a net in normal form.

As for call-by-name, when we evaluate a term beginning with a λ, we should return that term:

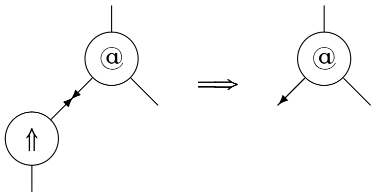


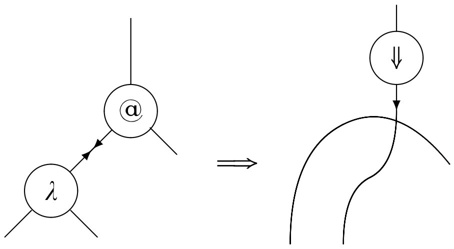
We take the opportunity of this first rule to recall that the interaction rules always have an implicit name: for instance, the above rule is called $\Downarrow \bowtie \lambda$, and reduction using this rule is written $\underset{\Downarrow\bowtie\lambda}{\Longrightarrow}$.

To evaluate a term whose head symbol is an application, we should first evaluate its left subterm. In other words, we should move the evaluation token to the function port of the application. We also rename the agent a to @, 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:



Finally, when the agent $\Uparrow$ returns from a successful evaluation to an agent @, we know for sure that there is a λ just below the $\Uparrow$ (this will be proved in Lemma 6.3), and a β-reduction should be performed. Due to the restriction to binary interaction, this takes two steps:

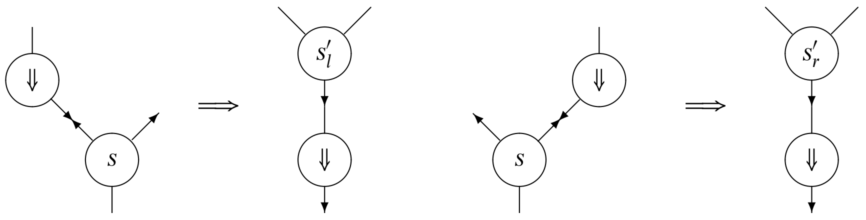




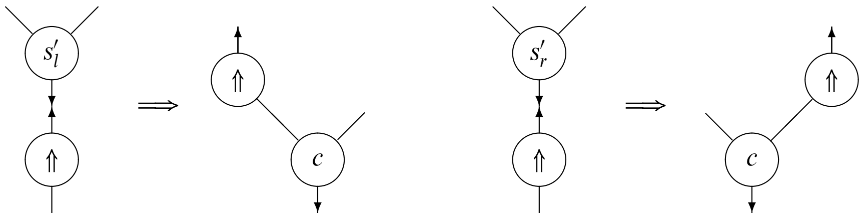
We link the variable port of the λ to the argument port of the @, which initiates the substitution; and we pursue evaluation on the body of the abstraction. This is the core of the interaction net machinery for linear λ-terms; it is the same as for call-by-name.

5.2. *Sharing*

Sharing is represented by agents s. When the evaluation token reaches an agent s, the shared subterm must be evaluated. 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 whether the token came from the left or right of the agent s so that we can continue from the same position. This is why we actually introduce two agents: s'_l and s'_r.



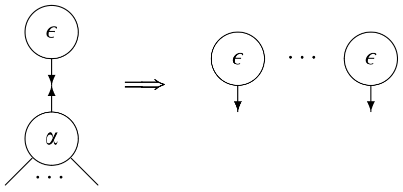
When the token returns to an agent s', we initiate the copying process with a c agent and propagate the token to the original position (left or right, as remembered in the agent). There is no need to resume evaluation (that is, produce a $\Downarrow$ token) because the subterm has already been evaluated (Lemma 6.3). (This will not always be the case for fully lazy reduction, see Section 7).



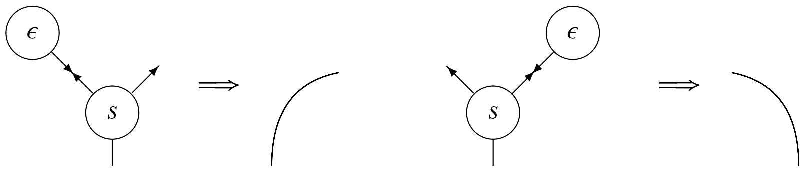
Remark 5.1. Because of our encoding and because we follow a normal order strategy (we always go left in an application), it will often be the case that the token reaches an agent s on its left port. However, this is not always the case, for instance, it is not true for the term $(\lambda x.(\lambda y.\lambda z.z\,y)\,x\,x)\,(\lambda u.u)$. This is why we have to abandon the restriction to interaction nets.

5.3. *Copying and erasing*

Copying and erasing are done in a classical way, by agents ϵ, c and δ. The auxiliary agent δ 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 immediately. In other words, the interaction net implementation of the copying process should always complete, and never block on some partially copied net. The agent ϵ erases any agent and propagates according to the following schema. For clarity, the schema is valid for $\alpha \in \{\lambda, a, @, \epsilon, c, \delta, s''\}$ (δ and s'' are introduced below with their own schemas, which are compatible with this one):

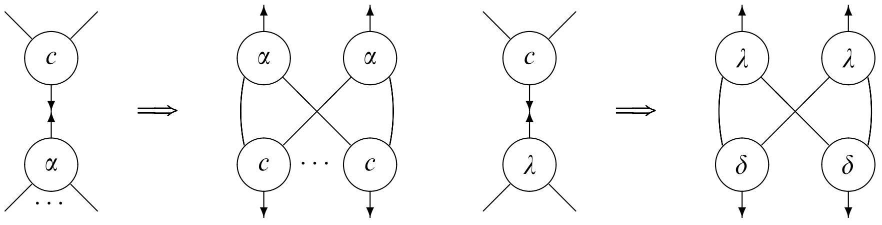


Erasure of s agents is slightly unusual, though very natural. The intuition behind it is simply that if a term is shared and one of the two copies is unused, then sharing is useless. Such rules cannot be defined in standard interaction nets, and this is indeed often considered as an equivalence in the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

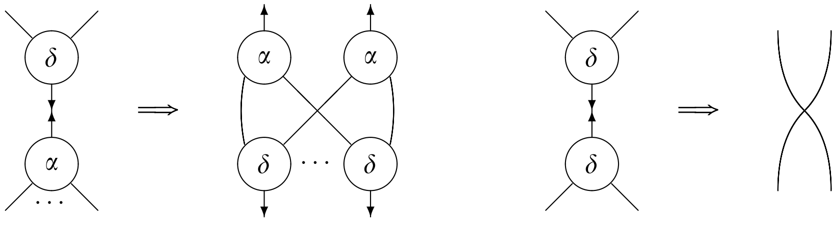


The agent ϵ is thus responsible for garbage collection. There are two interesting remarks to make on this subject. First, garbage collection is complete (see Proposition 6.5), because β-reduction, hence erasing, only ever occurs on closed terms because of the token. This contrasts with other implementations of the λ-calculus in interaction nets, where it is often the case that the net must be evaluated (with the possibility of non-termination) before it can be erased. Moreover, interactions with ϵ will never be needed to enable the token to make progress. More precisely, an ϵ 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 ϵ agents, or to 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 δ that will also duplicate any agent, but will stop the copy when it meets another δ agent. Note that an agent c will thus never interact with another agent c. Here, $\alpha \in \{a, \epsilon\}$ (λ is excluded and the other cases do not arise).

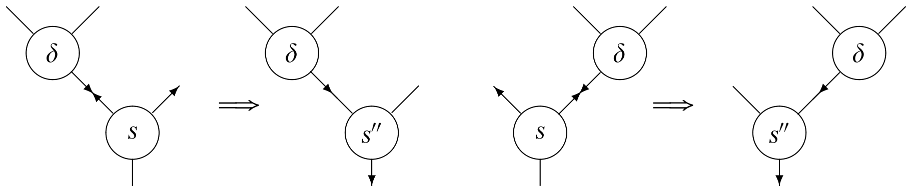


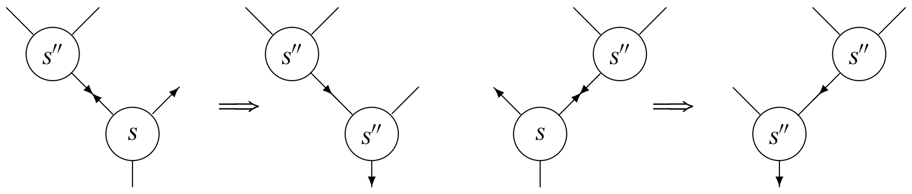
The agent δ duplicates any agent, except itself. If it interacts with itself, it just annihilates. Here, $\alpha \in \{\lambda, a, @, \epsilon, c\}$.



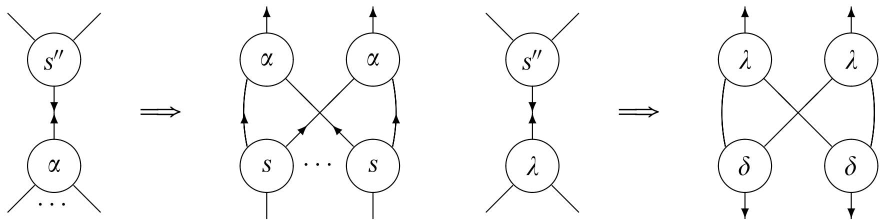
A c agent always tries to copy a closed, evaluated subnet, so agents c and s never meet (because the s would have been activated first; this is proved in Lemma 6.3). However, we may have to copy an agent s inside an abstraction using a δ. We cannot use the generic rule $\delta \bowtie \alpha$ with $\alpha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

When an agent δ 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 ('deactivated') to become an agent s again. The result is that we can avoid the problem of copying agents s: we do not copy them, instead we tell them to copy. More precisely, agents s'' are introduced as follows:





And they interact as follows ($\alpha \in \{a, \epsilon, \delta\}$):



Note that we do not attempt to preserve sharing inside an abstraction (we use δ agents instead of s agents). The sharing obtained is thus exactly call-by-need in the usual sense (for example, as 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 $\lambda \bowtie @$, and *administrative rules* involving agents c, s'', δ or ϵ. Note that it is indeed a partition. We write $\underset{ev}{\Longrightarrow}$ for reduction with an evaluation rule, and $\underset{adm}{\Longrightarrow}$ for reduction with an administrative rule.

Definition 6.2. A net N is *valid* if there exists a λ-term t such that $\Downarrow\mathscr{T}(t) \Longrightarrow^* N$.

From now on, we assume all nets are 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 $\lambda \bowtie @$ (if α is below β in M and $M \underset{\lambda\bowtie@}{\Longrightarrow} N$ not touching α and β, then, in N, either α is still below β, either they are in disconnected components); and 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 λ or a.
2 $\Downarrow$ and $\Uparrow$ are never above a @.
3 $\Downarrow$ and $\Uparrow$ are never below a s, c, δ or s''.
4 If there is a $\Uparrow$, then there is a λ below it, and there may be only c agents in between.
5 If there is a s below a c, then there is a λ in between.
6 If there is a δ and a $\Downarrow$ or $\Uparrow$ in the same connex component, then the former is below the latter.

Proof. The proof is by induction. All properties are true for $\Downarrow\mathscr{T}(t)$ and are preserved by reduction (by checking all rules if the agents are involved in the reduction, by the above remark otherwise). □

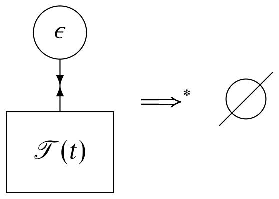
In a valid net, there may be several administrative redexes. However, we have the following result.

Proposition 6.4. In a valid net there is exactly one occurrence of $\Downarrow$, $\Uparrow$ or of a $\lambda \bowtie$ @ active pair, hence there is at most one evaluation redex.

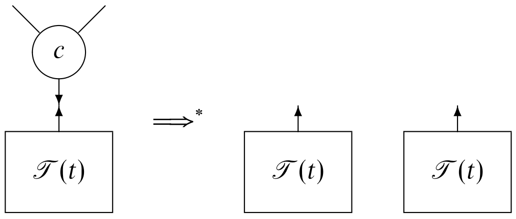
Proof. This is true for $\Downarrow\mathscr{T}(t)$ and all rules trivially preserve the property, except $\lambda \bowtie$ @. But thanks to Lemma 6.3 (first two points), we know that there is no $\Downarrow$ or $\Uparrow$ in the net when this rule is used. □

Classical results on packages (Lafont 1997; Lippi 2002) allow us to state the two properties in the following proposition.

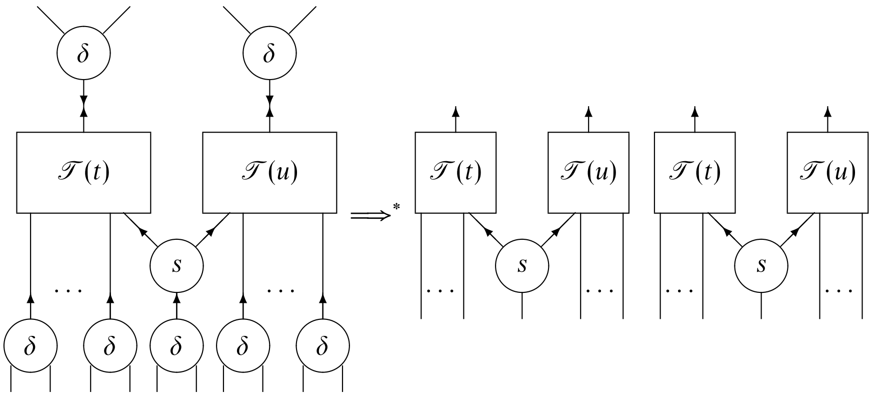
Proposition 6.5.
— If t is a closed λ-term, then



(where the right-hand side of the rule denotes the empty net).
— If t is a closed λ-term, then



Proof. The proof is by induction on the structure of t. For the second point, we need the following lemma, which is also easily proved by induction:



$\square$

It is only because of the agent s that our system fails to be an interaction net system in the sense of Lafont, but it is enough to invalidate strong confluence. However, strong confluence can be restored by imposing two limitations:

— by considering valid nets only (and thus nets with only one token agent); and
— by removing the rules for garbage collection (rules involving ϵ).

We stress again that these rules are not essential for the progress of evaluation (failing to apply them will not block the evaluation token).

Proposition 6.6. Reduction (excluding erasing rules) is strongly confluent on valid nets, that is, if M is a valid net such that $M \Longrightarrow P$ and $M \Longrightarrow Q$ (with $P \neq Q$), then there exists a net N such that $P \Longrightarrow N$ and $Q \Longrightarrow N$.

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, from Lemma 6.3 and Proposition 6.4, an s agent cannot have a δ on one port and an evaluation token on the other. 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. Lemma 6.3 (point 6) means agents c and s cannot meet, so the only remaining cases involve an agent s and agents δ or s'' on both its principal ports. In all of these configurations, both reductions lead to the same net, in which the agent s has been activated into an agent s''. This completes the proof. $\square$

Now, if we put the garbage collection rules back 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, 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 ϵ, but there is little point in doing this, at least from an implementation point of view. However, reduction is confluent for terminating λ-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 ϵ 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 no 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 tokens, as shown by Proposition 6.6 for our particular system. Another point is that some cost in terms of the number of interaction steps comes from the token agent. However, this is better seen as a cost made explicit rather than as an extra cost: standard implementations of Lafont's interaction nets require some external machinery to find the next redex (Pinto 2000). By contrast, in token-passing nets, it is always sufficient to reduce in the neighbourhood of the token agent.

6.2. *Simulation*

We will now show a 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 an implementation 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, $\Updownarrow$ will stand for either $\Downarrow$ or $\Uparrow$.

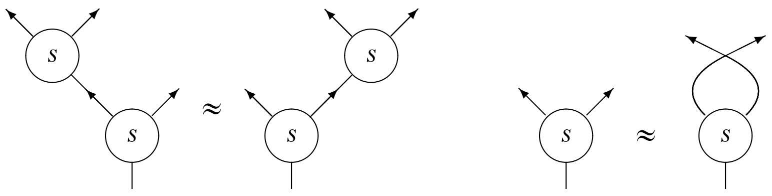
Definition 6.7 (Equivalence on terms). We define $\sim$ as the least symmetric, reflexive and transitive relation such that:

— For any two-holes context $C[\,][\,]$ and value v, $C[\Updownarrow_{(\Gamma,x\mapsto v)} t][x] \sim C[\Updownarrow_{(\Gamma,x\mapsto v)} t][v]$.
— For any context $C[\,]$, if $e = C[\Updownarrow_{(\Gamma,x\mapsto u)} t]$ and $e' = C[\Updownarrow_{\Gamma} t]$, if $x \notin \mathsf{fv}(e')$, 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 Γ is a binder at the top-level). Moreover, there is no capture problem 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 in net contexts (that is, plugging two equivalent nets into the same net gives two equivalent nets) and such that:

— $a \approx @$, $s \approx s'_l \approx s'_r$;
— $\underset{adm}{\Longrightarrow} \subset \approx$;
— s is considered associative and commutative, that is,



Another thing we have to take care of before demonstrating 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 $\mathscr{T}(\cdot)$ to enriched terms instead of just λ-terms, which was enough to start the process.

In general, an enriched term e is of the form: $e = C_0[x_1 \Mapsto C_1[\dots x_n \Mapsto C_n[\Updownarrow_\Gamma t]\dots]]$, where t is a λ-term, $x_1,\dots,x_n$ are variables and $C_0[\,],\dots,C_n[\,]$ are contexts on λ-terms.

We now extend $\mathscr{T}(\cdot)$ step by step:

— $\mathscr{T}(t)$ is already defined if t is a λ-term. If t has free variables $x_1,\dots,x_n$, we assume that the corresponding free edges are temporarily labelled with the appropriate variable. The label will not be part of the final translation: it is only used to build it.

— We define $\mathscr{T}(\Updownarrow t)$ (empty environment) as in Section 5.1 by connecting an agent $\Downarrow$ or $\Uparrow$ to $\mathscr{T}(t)$.

— $\mathscr{T}(C[\Updownarrow_\Gamma t])$ where $\Gamma = (x_1 \mapsto t_1,\dots,x_n \mapsto t_n)$ is defined by building $\mathscr{T}(C[\Updownarrow t])$ and $\mathscr{T}(t_i)$ for all i, and connecting every edge labelled x_i in $\mathscr{T}(C[\Updownarrow t])$ to the root of $\mathscr{T}(t_i)$.

— Similarly, for $\mathscr{T}(C[x \Mapsto t])$, we build $\mathscr{T}(C[x])$ and $\mathscr{T}(t)$ and connect the free edge labelled x of the former to the root of the latter.

Note that $\mathscr{T}(\cdot)$ produces nets that are valid with respect to reduction only modulo $\approx$. For instance, $\mathscr{T}((\Downarrow_\Gamma t)\ u)$ 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 $t \to u$, then there exist M and N such that $\mathscr{T}(t) \approx M$, $M \Longrightarrow^* N$ and $N \approx \mathscr{T}(u)$.

Proof. The proof is by cases on the rule used:

— $(\Downarrow Lam)$: This case is clear.

— $(\Downarrow App)$: This case is clear.

— $(\Uparrow App)$: This case is clear (it takes two steps).

— $(\Downarrow Var)$: Say $\Downarrow_{(\Gamma,x\mapsto t)} x \to x \Mapsto \Downarrow_\Gamma t$. There are two cases:

  – If the occurrence of x is unique in the term or if t has already been evaluated, the variable x is just represented by a wire, and the two translations are equal:
$$\mathscr{T}(C[\Downarrow_{(\Gamma,x\mapsto t)}\ x]) = \mathscr{T}(C[x \Mapsto \Downarrow_\Gamma\ t]).$$

  – Otherwise, this corresponds to a rule $\Downarrow \bowtie s$, which transforms s into s'_l or s'_r (which are equivalent modulo $\approx$).

— $(\Uparrow Var)$: 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 $\Uparrow \bowtie s'_l$ or $\Uparrow \bowtie s'_r$. $\square$

We cannot impose $M = \mathscr{T}(t)$ in the above proposition because the principal ports in $\mathscr{T}(t)$ may be oriented incorrectly (as in $\mathscr{T}((\Downarrow_\Gamma t)\ u)$), but there is indeed a correctly oriented representative in the class of $\mathscr{T}(t)$.

For the other direction, there is a detail to take care of: the rule $\Uparrow \bowtie$ @ creates an intermediate net that does not correspond to a valid term until the corresponding rule

$\lambda \bowtie @$ is applied. Hence, we do not reason using reduction $\Longrightarrow$ but rather $\Rrightarrow$ defined as follows:

— $M \Rrightarrow N$ if there exists P such that $M \underset{\Uparrow\bowtie @}{\Longrightarrow} P \underset{\lambda\bowtie @}{\Longrightarrow} N$.
— $M \Rrightarrow N$ if $M \Longrightarrow N$ by another rule.

Proposition 6.10 (Correctness). If $M \Rrightarrow N$ and $M \approx \mathscr{T}(t)$ for some enriched term t, then there exist t' and u such that $t \sim t'$, $t' \to^* u$ and $N \approx \mathscr{T}(u)$.

Proof. The proof is by cases on the family of rules used:

— Rules $\Downarrow \bowtie \lambda$, $\Downarrow \bowtie a$, $\Uparrow \bowtie @ + \lambda \bowtie @$: These cases are clear (some rules moving variables between the context and construct of the form $x \Mapsto t$ may be applied implicitly in the linear case).
— Rules $\Downarrow \bowtie s$: In this case t' is of the form $C[\Downarrow_{(\Gamma, x \mapsto v)} x]$ (with x non-linear in t'), and, indeed, $\mathscr{T}(C[x \Mapsto \Downarrow_\Gamma v]) \approx N$.
— Rules $\Uparrow \bowtie s'_{l/r}$: This case is similar to the previous one.
— Administrative rules: These cases are true with $t = t' = u$, since $\approx$ contains this $\Rrightarrow$ step. □

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

— If $\Gamma : t \Downarrow \Delta : v$, then $\mathscr{T}(\Downarrow_\Gamma t) \Longrightarrow^* \mathscr{T}(\Uparrow_\Delta v)$.
— If $\mathscr{T}(\Downarrow_\Gamma t) \Longrightarrow^* \mathscr{T}(\Uparrow_\Delta v)$, then there are v', Δ' such that $\Uparrow_\Delta v \sim \Uparrow_{\Delta'} v'$ and $\Gamma : t \Downarrow \Delta' : v'$.

Proof. We simply put together previous results.

— By Propositions 2.2 and 6.9, there are nets M and N such that $\mathscr{T}(\Downarrow_\Gamma t) \approx M$, $N \approx \mathscr{T}(\Uparrow_\Delta v)$ and $M \Longrightarrow^* N$, 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 $\mathscr{T}(\Downarrow_\Gamma t) \Longrightarrow^* \mathscr{T}(\Uparrow_\Delta v)$.
— First note that the reduct has a token agent in it, hence the $\Longrightarrow^*$ reduction is also a $\Rrightarrow^*$ reduction. We use Propositions 2.2 and 6.10, and an argument similar to the one used above allows us 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. □

Note 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 just given an implementation using net rewriting, just captures the sharing of *values*. For example, evaluation of the term

$$(\lambda f.fI(fI))(\lambda w.(II)w)$$

where $I = \lambda x.x$ will evaluate the redex II twice, because the subterm $\lambda w.(II)w$ 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, for example, in the

G-machine (Peyton Jones and Salkild 1989). This is not usually considered to be a problem, because this term can also be transformed into

$$(\lambda f.fI(fI))((\lambda z.(\lambda w.z\,w))(II))$$

in which the redex II will be evaluated only once. This transformation is called *fully lazy λ-lifting* (Peyton Jones 1987, Chapter 15).

However, it is also possible to share the evaluation of this redex directly. Implementations achieving this are called *fully lazy*. Wadsworth was the first to provide a fully lazy interpreter (Wadsworth 1971): he noticed that the redex II 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 crucially, on the one hand, on the fact that whenever a copy is started on a closed term, it may always complete, and, on the other hand, on the fact that the token ensures that reduction happens only at a point where the term is closed (that is, the token does not go under λ's). This would not be true for a fully lazy evaluator. The difficulty lies in the fact 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 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 deciding whether, when two agents δ meet, they should copy each other or cancel each other 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. So before even starting work on a solution, we can already say that the solution will not be completely satisfactory. So far, the evaluation token has had two benefits: it restricted evaluation to the needed redexes and allowed us to manage copying in a very simple way. In a fully lazy setting, this last point will not hold: we have reached a limit for the token-passing approach.

Now there are several known solutions to the problem of restricting copying agents to the correct scope. The framework RINO developed by Lang (Lang 1998) is the easiest to adopt for our purpose, because he does not introduce extra agents (instead, he 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 begin by introducing some terminology. We assume we are given an infinite set of *atomic labels* or *letters* (denoted x, x_1, x_2, $y,\dots$). We then define a *label* or *word* (denoted σ, $\rho,\dots$) as an ordered sequence of atomic labels, and we write ϵ for the *empty label*. Prefixing, postfixing and concatenation are denoted without a symbol when there is no risk of confusion. We define

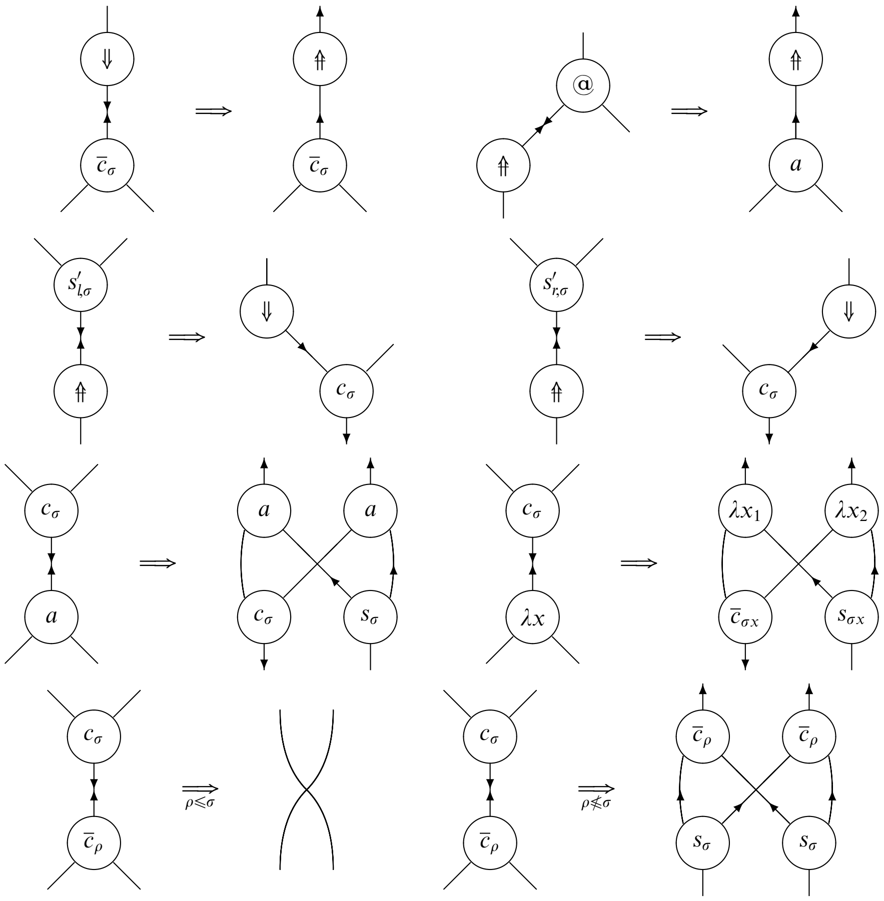


Fig. 1. Additional net rewriting rules for fully lazy reduction

the *length* of a label σ, written $|\sigma|$, as the number of letters of σ. Moreover, we write $\rho \leqslant \sigma$ if ρ is a prefix of σ.

Our starting point is the encoding of Section 4. The agents are modified as follows:

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

— The former agents c and δ are now coalesced into a single type of agent c_σ, annotated with a non-atomic label. All labels are initially empty (that is, equal to ϵ). The former c agents correspond to an empty label, while the δ agents correspond to c_σ with $|\sigma| \geqslant 1$. This is a consequence of the rule $c_\sigma \bowtie \lambda$: see Figure 1.

— The agents s are also annotated with a label. The previous agent s was an inhibited version of the agent c, and was used to share closed subterms. Now s_σ is also an inhibited version of δ (in the case $|\sigma| \geqslant 1$), but it may share open subterms.

— The agent δ was used for two dual purposes: to share a subterm (that is, 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_σ for the first, and $\overline{c}_\sigma$ for the second. This is necessary because of the possible asymmetry in the rule $c_\sigma \bowtie \overline{c}_\rho$ (see Figure 1), in contrast with Section 5.

— There is no longer an agent s''. Indeed, the default behaviour of the new agent c_σ is to perform one copy step and then turn back into an s_σ and share the subterm until the token makes a further copy necessary, in a similar way to s'' as given in Section 5.

— Finally, there is a new type of token, written ⇞, which is used when the evaluated subnet under it is in the form of a stack of applications beginning with an agent $\overline{c}_\sigma$. Its role is to go up to find an agent s'_σ (there must be one), as opposed to the role of the standard token $\Uparrow$, which 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 – the following comments provide some further explanations:

— An agent ⇞ is created when the token reaches a $\overline{c}_\sigma$ and goes up the stack of applications above it until it finds an agent s'_σ (there must be one).

— When the token ⇞ returns to an agent s'_σ, we initiate the copying process with a c_σ 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, and hence should be copied. So, as a cheap optimisation, we prefer to generate a c_σ directly instead of a s_σ.

— When a c_σ agent copies a λ agent labelled x, the two copies of the λ are given fresh labels (that is, labels that have never appeared before in the net), while the new $\overline{c}$ and s agents are labelled by σx (x postfixed to σ), see Figure 1.

— When two agents c_σ and $\overline{c}_\rho$ are facing each other, their labels are compared for prefixing. If the label of the closing copy agent $\overline{c}_\rho$ is a prefix of the other, they must both come from the same λ 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 we need to introduce different agents c_σ and $\overline{c}_\sigma$.

— To understand the role of the labels using 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 interpreter found in Lang (1998), so it is not appropriate to take more space here describing the properties of this interpreter. Correctness of the labelling and simulation with respect to the λ-calculus are given in Lang (1998) and easily adapted. We may, nevertheless, note an interesting common point between our system and the systems developed by Lang and Lippi: there are two distinct agents for application, one used for β-reduction and the other for duplication, and these agents can be converted from one type to the other by certain interaction rules with other agents.

Though Lang's implementation is 'essentially' fully lazy, it is not completely so since normal order is not imposed (this is the main role of the token here), and thus 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 a strength of our presentation in the following proposition.

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 for the claim 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 does indeed make 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 believe that our framework could be a good candidate for reasoning about such possible optimisations and for proving them formally.

8. Conclusions

We have presented a simple approach to expressing 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 term/environment pairs 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 a 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 for having 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 to the simplified version presented in Section 2.2. The usual set of λ-terms is denoted Λ.

Definition A.1. Λ_{let} is the set of λ-terms with *lets* defined by

$$\begin{aligned} t, u &::= x \mid \lambda x.t \mid t\ x \\ &\quad \mid let\ \ x = u\ in\ t\ x \quad (\text{if } x \notin \mathsf{fv}(t)). \end{aligned}$$

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

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

Definition A.3. Launchbury's big-step semantics of call-by-need is denoted $\Downarrow_L$ and is defined on Λ_{let} as follows. All bound variables are assumed to be initially different:

$$\frac{}{\Gamma : \lambda x.t \Downarrow_L \Gamma : \lambda x.t}\ Lam_L$$

$$\frac{\Gamma : t \Downarrow_L \Delta : \lambda y.t' \quad \Delta : t'\{y := x\} \Downarrow_L \Theta : v}{\Gamma : t\ x \Downarrow_L \Theta : v}\ App_L$$

$$\frac{\Gamma : t \Downarrow_L \Delta : v}{(\Gamma, x \mapsto t) : x \Downarrow_L (\Delta, x \mapsto v) : \hat{v}}\ Var_L$$

$$\frac{(\Gamma, x \mapsto u) : t \Downarrow_L \Delta : v}{\Gamma : let\ \ x = u\ in\ t \Downarrow_L \Delta : v}\ Let_L.$$

Terms of Λ_{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, to keep all bound variables distinct, it is sufficient to perform α-conversion in rule Var_L alone. Moreover, for any $t \in \Lambda_{let}$, 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 $let\ \ x = u\ in\ v$, then $v = w\ x$ with $x \notin \mathsf{fv}(w)$. 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 the following one.

Definition A.4. We define the readback function $(\cdot)^\circ : \Lambda_{let} \to \Lambda$ by

$$\begin{aligned} x^\circ &= x \\ (\lambda x.t)^\circ &= \lambda x.(t^\circ) \\ (t\ x)^\circ &= t^\circ\ x \\ (let\ \ x = u\ in\ t\ x)^\circ &= t^\circ\ u^\circ. \end{aligned}$$

Lemma A.5. Compilation and readback are inverses (modulo α-conversion):

— $(\cdot)^\circ \circ (\cdot)^* = \mathrm{id}_\Lambda$
— $(\cdot)^* \circ (\cdot)^\circ = \mathrm{id}_{\Lambda_{let}}$.

Proof. The proof is obvious. □

We also need the following auxiliary lemma.

Lemma A.6. If x is a fresh variable (for $\Gamma : t$) and u is a Λ_{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. The proof is obvious. □

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 α-conversion) in the following sense:

— $\Downarrow\ = (\cdot)^\circ \circ \Downarrow_L \circ\ (\cdot)^*$
— $\Downarrow_L = (\cdot)^* \circ \Downarrow\ \circ\ (\cdot)^\circ$.

Proof. Lemma A.5 means both points are equivalent. We only develop the difficult case. Let us assume $\Gamma : t\ u \Downarrow \Theta : v$ and show that $(\Gamma : t\ u)^* \Downarrow_L \Theta' : v'$ with $(\Theta' : v')^\circ = \Theta : v$. By rule *App*, there exist Δ and r such that $\Gamma : t \Downarrow \Delta : \lambda x.r$ and $(\Delta, x \mapsto u) : r \Downarrow \Theta : v$. By induction, there exist Δ', t' such that

$$(\Gamma : t)^* \Downarrow_L \Delta' : t' \tag{1}$$

and $(\Delta' : t')^\circ = \Delta : \lambda x.r$. By Definition A.4, there exists r' such that

$$t' = \lambda x.r'. \tag{2}$$

Using Lemma A.5,

$$\Delta' = \Delta^* \text{ and } r' = r^*. \tag{3}$$

By induction again, there exist Θ', v' such that

$$((\Delta, x \mapsto u) : r)^* \Downarrow_L \Theta' : v' \tag{4}$$

and

$$(\Theta' : v')^\circ = \Theta : v. \tag{5}$$

The following derivation and fact (5) allow us to conclude this case.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(1)+(2)}{\Gamma^* : t^* \Downarrow_L \Delta' : \lambda x.r'}}{(\Gamma^*, y \mapsto u^*) : t^* \Downarrow_L (\Delta', y \mapsto u^*) : \lambda x.r'}\,(A.6) \qquad \dfrac{\dfrac{\dfrac{(4)}{(\Delta^*, x \mapsto u^*) : r^* \Downarrow_L \Theta' : v'}}{(\Delta', x \mapsto u^*) : r' \Downarrow_L \Theta' : v'}\,(3)}{(\Delta', y \mapsto u^*) : r'\{x := y\} \Downarrow_L \Theta' : v'}\,(=_\alpha)}{(\Gamma^*, y \mapsto u^*) : t^*\ y \Downarrow_L \Theta' : v'}\,(App_L)}{\Gamma^* : let\ \ y = u^*\ \ in\ \ t^*\ y \Downarrow_L \Theta' : v'}\,(Let_L)}{(\Gamma : t\ u)^* \Downarrow_L \Theta' : v'}\,(A.2)$$

The other two cases are easy. □

References

Alexiev, V. (1999) *Non-deterministic Interaction Nets*, Ph.D. thesis, University of Alberta.

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

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

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

Asperti, A. and Guerrini, S. (1998) *The Optimal Implementation of Functional Programming Languages*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press.

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

Bawden, A. (1986) Connection graphs. In: Gabriel, R. P. (ed.) *Proceedings of the ACM Conference on LISP and Functional Programming*, ACM Press 258–265.

Crégut, P. (1990) An abstract machine for lambda-terms normalization. In: *Lisp and Functional Programming 1990*, ACM Press 333–340.

Girard, J.-Y. (1987) Linear Logic. *Theoretical Computer Science* **50** (1) 1–102.

Gonthier, G., Abadi, M. and Lévy, J.-J. (1992) The geometry of optimal lambda reduction. In: *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, ACM Press 15–26.

Lafont, Y. (1990) Interaction nets. In: *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, ACM Press 95–108.

Lafont, Y. (1997) Interaction combinators. *Information and Computation* **137** (1) 69–101.

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

Lang, F. (1998) *Modèles de la β-réduction pour les implantations*, Ph.D. thesis, École Normale Supérieure de Lyon.

Launchbury, J. (1993) A natural semantics for lazy evaluation. In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* 144–154.

Lippi, S. (2002) *Théorie et pratique des réseaux d'interaction*, Ph.D. thesis, Université de la Méditerranée.

Mackie, I. (1998) YALE: Yet another lambda evaluator based on interaction nets. In: *Proceedings of the 3rd International Conference on Functional Programming (ICFP'98)*, ACM Press 117–128.

Mackie, I. (2004) Efficient λ-evaluation with interaction nets. In: van Oostrom, V. (ed.) Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA'04). *Springer-Verlag Lecture Notes in Computer Science* **3091** 155–169.

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

Mazza, D. (2005) Multiport interaction nets and concurrency. In: Abadi, M. and de Alfaro, L. (eds.) Proceedings of CONCUR'05. *Springer-Verlag Lecture Notes in Computer Science* **3653** 21–35.

Peyton Jones, S. and Salkild, J. (1989) The spineless tagless G-machine. In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, ACM Press 184–201.

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*, Prentice Hall International.

Pinto, J. S. (2000) Sequential and concurrent abstract machines for interaction nets. In: Tiuryn, J. (ed.) Proceedings of Foundations of Software Science and Computation Structures (FOSSACS). *Springer-Verlag Lecture Notes in Computer Science* **1784** 267–282.

Seaman, J. and Iyer, S. P. (1996) An operational semantics of sharing in lazy evaluation. *Science of Computer Programming* **27** (3) 289–322.

Shivers, O. and Wand, M. (2005) Bottom-up beta-reduction: uplinks and lambda-DAGs. In: Proceedings of the 14th European Symposium on Programming (ESOP'05). *Springer-Verlag Lecture Notes in Computer Science* **3444**.

Sinot, F.-R. (2005) Call-by-name and call-by-value as token-passing interaction nets. In: Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA'05). *Springer-Verlag Lecture Notes in Computer Science* **3461** 386–400.

Sinot, F.-R. (2006) Token-passing nets: Call-by-need for free. In: Proceedings of Developments in Computational Models. *Electronic Notes in Theoretical Computer Science* **135** (3) 129–139.

Wadsworth, C. P. (1971) *Semantics and Pragmatics of the Lambda-Calculus*, Ph.D. thesis, Oxford University.