Call-by-Name and Call-by-Value as Token-Passing Interaction Nets

François-Régis Sinot*

LIX, École Polytechnique, 91128 Palaiseau, France
frs@lix.polytechnique.fr

Abstract. Two common misbeliefs about encodings of the λ-calculus in interaction nets (INs) are that they are good only for strategies that are not very well understood (e.g. optimal reduction) and that they always have to deal in a complex way with *boxes*. In brief, the theory of interaction nets is more or less disconnected from the standard theory: we can do things in INs that we cannot do with terms, which is true [5, 10]; and we cannot do in INs things that can easily be done with terms. This paper contributes to fighting this misbelief by showing that the standard call-by-name and call-by-value strategies of the λ-calculus are encoded in interaction nets in a very simple and extensible way, and in particular that these encodings do not need any notion of box. This work can also be seen as a first step towards a generic approach to derive graph-based abstract machines.

1 Introduction

Interaction nets (INs) [9] are a graphical paradigm of distributed computation that makes all the steps in a computation explicit and expressed uniformly in the same formalism. Reduction in interaction nets is local and strongly confluent, hence reductions can take place in any order, even in parallel (see [17]). These properties make interaction nets well-suited as an intermediate formalism in the implementation of programming languages.

Indeed, interaction nets have their origins in linear logic [6], but have been most successfully used in the implementation of optimal reduction in the λ-calculus, starting from Lamping [10], Gonthier, Abadi and Lévy [8], Asperti et al. [1] to the recent work of van Oostrom et al. [19]. There have also been several other efficient (non-optimal) implementations of the λ-calculus, for instance Mackie [15, 16].

All of the above encodings of the λ-calculus have in common that a β-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 [17]. The fact that different β-reductions may be interleaved also has the nasty consequence that the encodings

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

need to simulate *boxes* in a more or less complex and costly way (see [10, 8, 13, 15, 16] for various types of such encodings). This reveals two common misbeliefs about interaction nets for λ-calculus: they are good only to express strategies that we cannot write in a term-like framework, and consequently that we do not understand very well, and they always need a complex mechanism to manage boxes. This paper intends to fight these two misbeliefs. More precisely, we will present 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 a single evaluation token, which is a standard interaction agent, walking through the term as an evaluation function would do. They are thus very natural and easy to understand.

An implementation of the λ-calculus on a sequential machine has whatsoever to perform reductions in a certain order (i.e. to follow a strategy), hence it is meaningful to give up the redex-to-redex translation in interaction nets. This has actually been done in Lippi's work [12, 11]. Lippi gives an implementation of left reduction in interaction nets, and goes even further by describing the Krivine machine in interaction nets. However, his work is based on notions of coding and decoding which makes the overall presentation difficult to understand and he does not make clear the notion of evaluation token hidden in his encoding. In particular, he does not extend his presentation to call-by-value, which indeed seems difficult with his presentation.

In contrast, our presentation is more simple and uniform: it is based on the simple idea of a single evaluation token walking through the representation of the term exactly as a functional evaluator would do, going down on recursive calls and up when exiting the recursive calls. This approach is very simple and is indeed a good alternative to working with terms, since it allows to abstract away from syntactical details such as α-conversion. Moreover it is very easily extended from call-by-name to call-by-value.

We thus provide new graph-based abstract machines for call-by-name and call-by-value, with the peculiarity that the structure of the term itself is used instead of a stack or a heap in traditional abstract machine. Moreover, our approach is so simple that there is some serious hope it can be extended to more interesting strategies. It is also nice from a theoretical point of view to (try to) bridge the gap between optimal reduction and call-by-name/value by providing a (more) uniform framework.

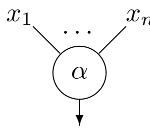
The idea of a token walking through a graph is superficially reminiscent of the geometry of interaction (GoI) [7], which has been used to implement call-by-name [14] and call-by-value [3] abstract machines. However, the details of the approach are quite different. In particular, the GoI machines avoid as much as possible to modify the graph, thus they have less freedom in the strategy. For instance, call-by-value is obtained at the price of a greater complexity. While it is relatively clear that the GoI machines can be formalised in our framework (i.e. by making the token explicit and encoding the stack with interaction agents), this does not seem to lead to a better understanding of the GoI machines (in particular to a possibly more satisfactory call-by-value machine).

Our approach is also reminiscent of continuation-passing style (CPS) transformation [18], in the sense that we simulate an evaluation strategy by forbidding certain reductions until something triggers them (the token or the continuation). However, a CPS transformation followed by a traditional encoding in interaction nets would certainly not allow to get rid of boxes, although only one β-reduction would be allowed at a time. In this respect, our framework is thus more satisfactory; it also seems easier to extend to other strategies.

The rest of this paper is structured as follows. In Section 2, we recall some background on interaction nets. In Section 3, we give the full details of our approach in the case of call-by-name for closed terms. Sections 4 and 5 adapt the presentation respectively to closed call-by-value and to open terms. Finally, we conclude in Section 6.

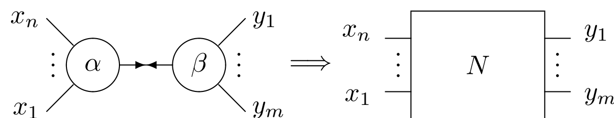
2 Interaction Nets

A system of interaction nets [9] is specified from 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 in the following way:



Intuitively, a net N built on Σ 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 $((\alpha, \beta) \Longrightarrow N) \in \mathcal{R}$ 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*), 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. 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 and $\Longrightarrow^*$ for its transitive and reflexive closure. If a net does not contain any active pairs

then we say that it is in normal form. One-step reduction ($\Longrightarrow$) satisfies the diamond property, and thus we obtain a very strong notion of confluence. 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 Call-by-Name

In this section, we give an encoding of the call-by-name strategy of the λ-calculus in interaction nets. We present the strategy with inductive rules, in a big-step style, and the first step in our encoding is to derive a more fine-grained rewrite system. In this section, we only consider closed terms (i.e. terms without free variables). Open terms will be dealt with in Section 5.

3.1 Preliminaries

We assume basic knowledge of the λ-calculus; we refer the reader to [2] for more details. To fix notations, the set Λ 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 i.e. renaming of bound variables. We denote by $\mathsf{fv}(t)$ the set of free variables of a term t.

This set is equipped with the rewrite relation:

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

where $t\{x := u\}$ denotes t where all occurrences of x are replaced by u, without name capture. We write $\to_\beta$ for one-step reduction and $\to_\beta^*$ for its reflexive transitive closure.

We call weak head normal forms (whnf) terms of the form $\lambda x.t$ or $x\ t_1 \ldots t_n$. Note that closed whnf are only terms of the form $\lambda x.t$. We say that v is a weak head normal form of t if v is a weak head normal form and $t \to_\beta^* v$.

3.2 Big-Step Style

The call-by-name strategy for closed λ-terms is specified by the following evaluation rules, as found in various textbooks:

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

This is in fact the inductive definition of an evaluation function (also known as big-step semantics), rather than a strategy: we take a λ-term t as input and we inductively find v such that $t \Downarrow v$, then v is the unique weak head normal

form of t (provided it exists) obtained by the call-by-name strategy, but the reduction path is not visible at the top-level.

Too much is hidden in these rules for a direct encoding in interaction nets. In the rule for application, we call the procedure recursively on the left term, and then we have to return to this application somehow. In a functional programming setting, this is done automatically, but this is not a free operation: when a function is entered, the current environment is saved on the stack; when it returns, this information is popped down from the stack. We will thus formalise the call-by-name strategy in a small-step style, so as to be more explicit about the control flow and to facilitate the encoding in interaction nets.

3.3 Small-Step Style

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. We thus enrich the syntax of terms with two unary symbols $\Downarrow$ (corresponding to evaluation) and $\Uparrow$ (corresponding to the evaluation function returning) and define the following rewrite system:

$$\begin{array}{lcl} \Downarrow \lambda x.t & \to & \Uparrow \lambda x.t \\ \Downarrow (t\ u) & \to & (\Downarrow t)\ u \\ (\Uparrow \lambda x.t)\ u & \to & \Downarrow t\{x := u\} \end{array}$$

Although we do not want to get into any details, it is clear how the small-step system is derived from the big-step one. In the particular case of call-by-name, omitting the symbol $\Uparrow$ gives an equivalent system (this is exactly tail-recursion optimisation), but we prefer to include it already; this point will be discussed again in Section 4. Also note that, as far as we know, such a simple small-step presentation of call-by-name has not been made before; usual small-step presentations of call-by-name and call-by-value 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-name.

A λ-term t is always in normal form with respect to this system, and so is $\Uparrow t$. To evaluate t, we have to start reduction from $\Downarrow t$.

First note that a reduction always involves a $\Downarrow$ or $\Uparrow$, hence, by the following proposition, there is always at most one redex in a term obtained from reduction of $\Downarrow t$. Thus the control flow is really made explicit at the syntactic level.

Proposition 1. *If $\Downarrow t \to^* u$, then there exists exactly one occurrence of $\Downarrow$ or $\Uparrow$ in u.*

Proof. By induction. The first two rules are easy. In the last rule, the right hand-side may have zero or more than one occurrences of u, but u has no occurrence of $\Downarrow$ or $\Uparrow$ by the induction hypothesis. □

The two systems correspond to each other in the following sense:

Proposition 2. $t \Downarrow v \iff \Downarrow t \to^* \Uparrow v$

Proof. $\Rightarrow$ By induction:

- $\lambda x.t \Downarrow \lambda x.t$ and indeed $\Downarrow \lambda x.t \to \Uparrow \lambda x.t$
- if $t\ u \Downarrow v$, then there exists t' such that $t \Downarrow \lambda x.t'$ and $t'\{x := u\} \Downarrow v$. By induction, $\Downarrow t \to^* \Uparrow \lambda x.t'$ and $\Downarrow t'\{x := u\} \to^* \Uparrow v$, hence:
  $\Downarrow (t\ u) \to (\Downarrow t)\ u \to^* (\Uparrow \lambda x.t')\ u \to \Downarrow t'\{x := u\} \to^* \Uparrow v$

$\Leftarrow$ The first part of the proposition (already proved) allows to state the following lemma: if t is a λ-term and t has a whnf, then there exists v such that $\Downarrow t \to^* \Uparrow v$ and v is a whnf (consequence of classical theorems on call-by-name). Then we can proceed by induction:

- $\Downarrow \lambda x.t \to \Uparrow \lambda x.t$ and indeed $\lambda x.t \Downarrow \lambda x.t$
- $\Downarrow (t\ u) \to (\Downarrow t)\ u$. By the lemma, if t has a whnf, there exists $\lambda x.t'$ (remember that all terms are closed), such that $\Downarrow t \to^* \Uparrow \lambda x.t'$. Moreover, $t \Downarrow \lambda x.t'$ by induction. Then $(\Uparrow \lambda x.t')\ u \to \Downarrow t'\{x := u\}$ and a similar argument (lemma and induction) allows to conclude. If t of $t'\{x := u\}$ does not have a whnf, the proposition is trivially true (we do not reach a term of the form $\Uparrow v$). □

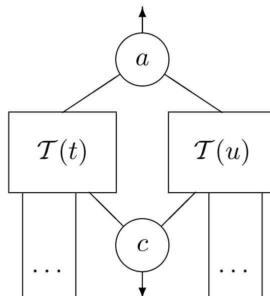
Hence the given rewrite system faithfully corresponds to the call-by-name strategy. This step is crucial, as the interaction net encoding will closely follow the small-step style system. Also remark that the method used here is very general.

3.4 Encoding of Terms

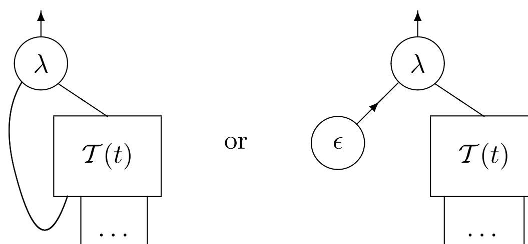
The translation $\mathcal{T}(\cdot)$ of λ-terms into interaction nets is very natural. We basically represent terms by their syntax tree, where we group together several occurrences of the same variable by agents c (corresponding to copy) and bind them to their corresponding λ node (this is sometimes referred to as a *backpointer*). The nodes for application and abstraction are agents λ and a with three ports; their principal port is directed towards the root of the term. Note that in traditional encodings, the application agent looks towards its left, so that interaction with an abstraction is always possible. Here, on the contrary, terms are translated to *packages* [12] and in particular there will be no spontaneous reduction, something will have to trigger them: the *evaluation token*.

Variables. In this section, we consider only closed terms (open terms will be dealt with in Section 5), hence variables are not translated as such. They will simply be represented by edges between their binding λ and their grouped occurrence in the body of the abstraction, as explained below.

Application. The translation $\mathcal{T}(t\ u)$ of an application $t\ u$ is simply an agent a of arity 2 pointing to the root, with $\mathcal{T}(t)$ and $\mathcal{T}(u)$ linked to its auxiliary ports. If t and u share common free variables, then c agents (representing copy) collect these together pairwise so that a single occurrence of each free variable occurs amongst the free edges (only one such copy is represented on the figure).



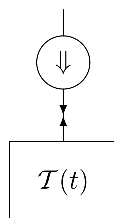
Abstraction. If $\lambda x.t$ is an abstraction, $\mathcal{T}(\lambda x.t)$ is obtained by introducing an agent λ, and simply linking its right auxiliary port to $\mathcal{T}(t)$ and its left one to the unique wire corresponding to x in $\mathcal{T}(t)$. If x does not appear in t, then the left port of the agent λ is linked to an agent ϵ.



To sum up, we represent λ-terms in a very natural way. In particular, there is no artifact to simulate boxes. Another point worth noticing is that, because of the explicit link between a variable and its binding λ, α-conversion comes from free, as it is often the case in graphical representations of the λ-calculus. So far, we only introduced agents λ and a strictly corresponding to the λ-calculus, as well as agents ϵ and c for the explicit resource managements necessary (and desirable: we do not want to hide such important things) in interaction nets. Also remark that the translation of a term has no active pair, hence is in normal form, whatever the interaction rules we allow. Moreover, it has exactly one principal port, at the root.

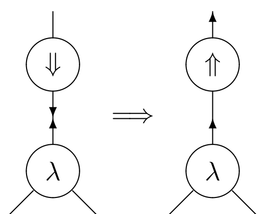
3.5 Evaluation by Interaction

We introduce two new unary agents $\Downarrow$ and $\Uparrow$. To evaluate a closed λ-term t with call-by-name, we simply build the following net, that we will denote $\Downarrow \mathcal{T}(t)$.

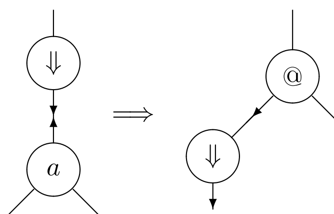


$\Uparrow \mathcal{T}(t)$ will be a net built in the same way, but with a $\Uparrow$ agent instead, with its principal port directed towards the root. In particular, $\Uparrow \mathcal{T}(t)$ is always a net in normal form.

The interaction rules will follow as closely as possible the rewrite system of Section 3.3. The first one is easy; when the evaluation token reaches a λ, it may begin to return:

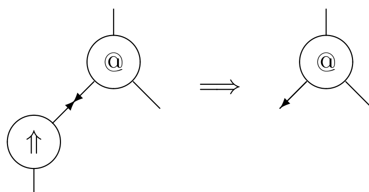


The second one is still simple, but slightly more subtle:



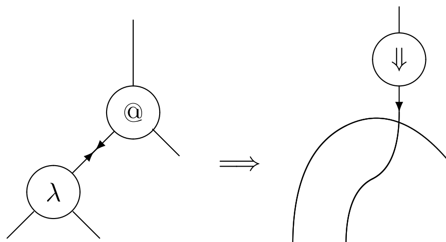
When the evaluation token reaches an application agent, we change the agent a to an agent @ still representing application, but no longer looking at its root but to the left, towards the propagated evaluation agent, waiting until it returns.

Finally, when the agent $\Uparrow$ returns from a successful evaluation to a @, then we know for sure that there is a λ just behind the $\Uparrow$, so the agent $\Uparrow$ may safely disappear, at least if λ and @ promise to create it again later. In a sense, it does not disappear, it just hides in the @ agent.



From the previous rule, it is obvious that the agent $\Uparrow$ is in fact useless. However this is the key to the generality of the translation, because we could have a different agent in the right hand-side. In particular, we will see that it is not useless for call-by-value. It is also interesting to note how striking it is in our framework that the agent $\Uparrow$ is useless, which corresponds to tail-recursion optimisation. The framework we propose is so simple that clever optimisations become obvious, hence this is indeed a good intermediate step between the inductive-style definition of a strategy and its implementation as an abstract machine.

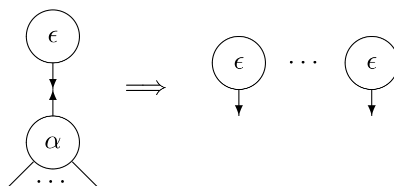
Now that the way is free between the @ and the λ, we may let them interact as usual, except that we create a new $\Downarrow$ token. We thus link together the variable port of the λ to the argument port of the @, which initiates the substitution. In brief, we follow exactly the rewrite system, except that we need two steps instead of one.



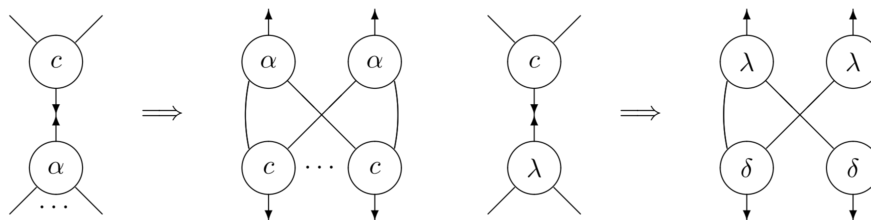
The core of the interaction net machine for call-by-name thus needs only four interaction rules, and no encoding of boxes.

3.6 Resource Management

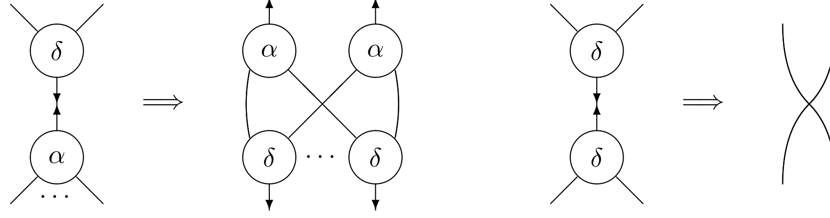
The explicit resource management typical of interaction nets is done by the agents ϵ, c and δ. The auxiliary agent δ is introduced to duplicate abstractions, as explained below. The agent ϵ erases any agent and propagates according to the following schema (where α represents any agent):



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, α represents any agent except λ.

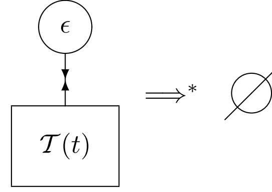


The agent δ duplicates any agent, except itself. If it interacts with itself, it just annihilates. Here, α represents any agent except δ.



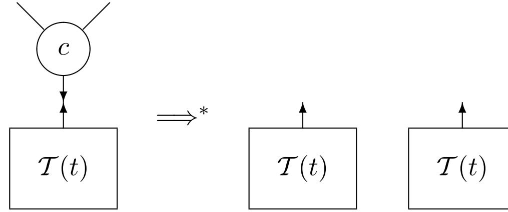
Classical results on packages [12] allow to state the two following properties:

Proposition 3. – *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:*



3.7 Properties

In a net obtained starting from $\Downarrow \mathcal{T}(t)$, there may be several redexes involving c's, δ's or ϵ's, however, we have the following result.

Proposition 4. *If $\Downarrow \mathcal{T}(t) \Longrightarrow^* N$ then in N, there is exactly one occurrence of $\Downarrow$, $\Uparrow$ or of a $\lambda - @$ active pair.*

Proof. By induction, using the rules. □

Proposition 5. $t \Downarrow v \Longleftrightarrow \Downarrow \mathcal{T}(t) \Longrightarrow^* \Uparrow \mathcal{T}(v)$

Proof. It is clear that the interaction rules closely follow the rewrite rules of Section 3.3 (using Proposition 3 for non-linear substitutions), then Proposition 2 allows to conclude. □

4 Call-by-Value

In this section, we show that we can very easily adapt the previous presentation to closed call-by-value. We follow the same organisation as Section 3, showing only the differences. In this section again, all terms are closed.

The call-by-value strategy for closed λ-terms is inductively defined by the following set of evaluation rules:

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

We may derive a small-step presentation of the strategy in a similar fashion as in Section 3. Here is the *token-passing semantics* of call-by-value:

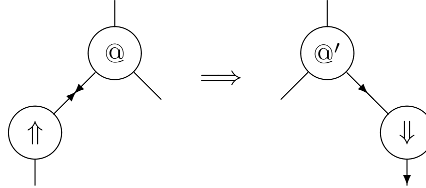
$$\begin{array}{lcl} \Downarrow \lambda x.t & \to & \Uparrow \lambda x.t \\ \Downarrow (t\ u) & \to & (\Downarrow t)\ u \\ (\Uparrow t)\ u & \to & t\ (\Downarrow u) \\ (\lambda x.t)\ (\Uparrow u) & \to & \Downarrow t\{x := u\} \end{array}$$

Here the role of $\Uparrow$ is more complex than with call-by-name: when the function part of an application is evaluated, the control is transferred to the argument. Then, when the argument is evaluated, β-reduction may be performed.

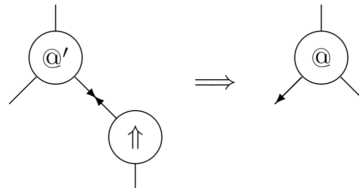
We have a similar property of simulation (the proof is also similar).

Proposition 6. $t \Downarrow v \Longleftrightarrow \Downarrow t \to^* \Uparrow v$

Some interaction rules have to change a bit, according to the small-step style system. When the left term of an application returns after evaluation, we no longer perform a β-reduction right after. Instead, we turn to evaluating the argument:



We introduce a new application agent @′ whose job is to wait until the argument of the application is evaluated. When it is, then again, we know for sure that there is a λ waiting at the left, so we may transform the agent into @ to allow the β-reduction to take place:



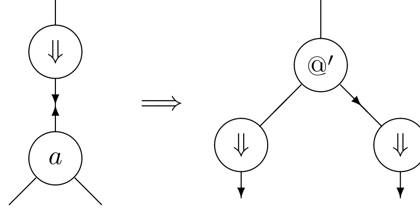
The other rules stay the same. Again, we have (the proof is easily adapted):

Proposition 7. $t \Downarrow v \iff \Downarrow \mathcal{T}(t) \Longrightarrow^* \Uparrow \mathcal{T}(v)$

To sum up, our presentation allow to adapt very easily from call-by-name to call-by-value, contrary to previous related works. There is no reason to think this approach cannot be adapted beyond to other strategies, or in a more general framework than the λ-calculus.

This interaction system is very faithful to what a sequential evaluation function would probably do. In particular, the agent $\Uparrow$ is necessary, because we have to manage explicitly the control flow in a sequential way.

Of course, interaction nets allow to evaluate the function and the argument of an application in parallel. Keeping the control flow explicit, we have to synchronise on the application node when both evaluations are completed. The system is obtained from the same rules as above, except that we replace the interaction rule $\Downarrow -a$ by:



Now in this system, it is again clear that the agent $\Uparrow$ is useless: there is no true need to synchronise both evaluations, and a β-reduction may occur even if evaluation of the argument is not yet completed.

But that is not our point. We prefer the version with sequential, explicit control (i.e. with a unique evaluation token) because it is really closer to an abstract machine: it is very easily implementable on a sequential machine, which is what we often have in practice, and does not need an external mechanism to manage a stack of active pairs.

5 Handling Open Terms

For completeness, we show how to deal with open terms. There is no difficulty, and no new idea. The presentation is done in a modular way: we only say what should be added or changed to the presentations of closed call-by-name and call-by-value to deal with open terms.

Evaluation to weak head normal form of open terms using call-by-name or call-by-value is done by adding to the corresponding system the following rules:

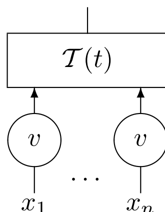
$$\frac{}{x \Downarrow x} \qquad \frac{t \Downarrow x}{t\ u_1 \dots u_n \Downarrow x\ u_1 \dots u_n}$$

Or, in a small-steps fashion (keeping the other rules):

$$\begin{array}{lll} \Downarrow x & \rightarrow & \Uparrow x \\ (\Uparrow x)\ u & \rightarrow & \Uparrow (x\ u) \\ (\Uparrow (v\ w))\ u & \rightarrow & \Uparrow (v\ w\ u) \end{array}$$

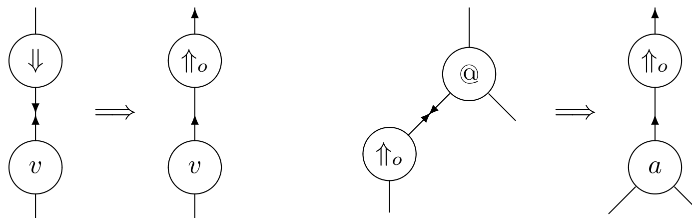
On the interaction nets side, it is clear that free variables will have to interact with the evaluation token, hence we cannot just represent them by a wire.

A term t with $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$ will be translated as a net $\mathcal{T}(t)$ with the root edge at the top, and n free edges marked by an agent v, corresponding to the free variables:

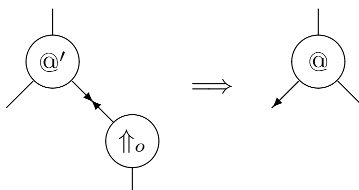


Variables. To sum up, if t is a variable x bound by a λ, then $\mathcal{T}(t)$ is just a wire (left). If it is free in the term, then $\mathcal{T}(t)$ is an agent v (right).

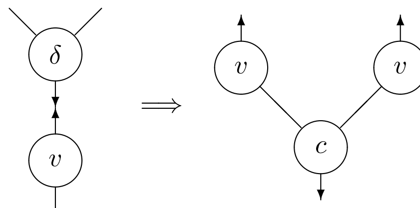
The systems for call-by-name and call-by-value are obtained by adding the two following rules, where we introduce a new agent $\Uparrow_o$ which is essentially the same as $\Uparrow$ but which remembers that the term under it is of the form $x\ t_1 \ldots t_n$ and not $\lambda x.t$. We have to introduce such an agent only because of the restriction to binary left hand-side in interaction nets (there is no deep reason behind).



In call-by-value, we also have to eliminate a possible $\Uparrow_o$ appearing to the right of an application. In this case, $\Uparrow_o$ is indeed redundant with $\Uparrow$, hence the rule is the same:



Finally, it may now happen that, in the course of the duplication of an abstraction, a δ agent meets a v agent. Then it is safer to transform it back to a c agent:



A similar result then holds (for open call-by-name and open call-by-value):

Proposition 8. $t \Downarrow v \Longleftrightarrow \Downarrow \mathcal{T}(t) \Longrightarrow^{*} \Uparrow \mathcal{T}(v)$ *or* $\Uparrow_o \mathcal{T}(v)$

6 Conclusion

We have presented a simple and extensible approach to express call-by-name and call-by-value in interaction nets. The approach is so simple that it is indeed a good alternative to working with terms, with the advantages of seeing graphically what is going on, of α-conversion for free and of explicit status and cost for the operations of substitution and copying.

Moreover, our interaction nets lie in a particular subclass of *token-passing interaction nets* that is not fully studied here, which seems very easy to implement on a sequential machine, without the usual overheads of looking for a redex and managing a stack of these. Full study of these aspects is left as future work.

We would also like to extend our approach to closed reduction [4] in order to derive an interaction nets based abstract machine for this efficient strategy.

The question whether our approach can benefit to the optimal strategy is still open. Boxes are certainly necessary, since β-reductions have to be interleaved, but controlling more tightly the evaluation flow might still be useful.

References

1. A. Asperti, C. Giovannetti, and A. Naletto. The Bologna optimal higher-order machine. *Journal of Functional Programming*, 6(6):763–810, Nov. 1996.
2. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland Publishing Company, second, revised edition, 1984.
3. M. Fernández and I. Mackie. Call-by-value lambda-graph rewriting *without rewriting.* In *Proceedings of the International Conference on Graph Transformations (ICGT'02)*, volume 2505 of *Lecture Notes in Computer Science*, Barcelona, 2002. Springer-Verlag.
4. M. Fernández, I. Mackie, and F.-R. Sinot. Closed reduction: Explicit substitutions without alpha-conversion. *Mathematical Structure in Computer Science.* to appear.

5. J. Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages (POPL '90)*, pages 1–15, San Francisco, CA, USA, Jan. 1990. ACM Press.
6. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
7. J.-Y. Girard. Geometry of interaction I: Interpretation of system F. In C. Bonotto, R. Ferro, S. Valentini, and A. Zanardo, editors, *Logic Colloquium '88*, pages 221–260. North-Holland, 1989.
8. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, Jan. 1992.
9. Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, Jan. 1990.
10. J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 16–30. ACM Press, Jan. 1990.
11. S. Lippi. Encoding left reduction in the lambda-calculus with interaction nets. *Mathematical Structures in Computer Science*, 12(6), December 2002.
12. S. Lippi. *Théorie et pratique des réseaux d'interaction.* PhD thesis, Université de la Méditerranée, June 2002.
13. I. Mackie. *The Geometry of Implementation.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, September 1994.
14. I. Mackie. The geometry of interaction machine. In *Proceedings of the 22nd Symposium on Principles of Programming Languages (POPL'95)*, pages 198–208, San Francisco, CA, USA, 1995. ACM Press.
15. I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, 1998.
16. I. Mackie. Efficient λ-evaluation with interaction nets. In V. van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 155–169. Springer-Verlag, June 2004.
17. J. S. Pinto. Sequential and concurrent abstract machines for interaction nets. In J. Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, volume 1784 of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, 2000.
18. G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
19. V. van Oostrom, K.-J. van de Looij, and M. Zwitserlood. Lambdascope: another optimal implementation of the lambda-calculus. In *Workshop on Algebra and Logic on Programming Systems (ALPS)*, Kyoto, April 2004.