DCM 2005 Preliminary Version

Token-Passing Nets: Call-by-Need for Free

François-Régis Sinot [1,2]

LIX, École Polytechnique, 91128 Palaiseau, France

Abstract

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 was 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 flow of evaluation. In particular, no artifact to represent boxes is needed. However, these encodings purposefully follow as closely as possible the implemented strategies, call-by-name and call-by-value, hence do not benefit from the ability of interaction nets to easily represent sharing. The aim of this note is to show that sharing can indeed be achieved without adding any structure. We thus present the call-by-need strategy following the same philosophy, which is indeed not any more complicated than call-by-name. 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 (INs) [4] are a graphical paradigm of computation that makes all the steps in a computation explicit and expressed uniformly. In particular, sharing is possible (as opposed to terms) and is dealt with explicitly (as opposed to termgraphs). Locality and strong confluence of reduction also make interaction nets well-suited as an intermediate formalism in the implementation of programming languages. However, despite their qualities and their popularity among theoreticians, it is sad to notice that they are less widely used by implementors of real-world programming languages. While it is difficult to say why, it is very probable that works such as those on optimal reduction have led some to think of interaction nets as a theoreticians-only

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

[2] Email: frs@lix.polytechnique.fr

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

tool. It might thus be worth it to bridge the gap between interaction nets and traditional tools such as abstract machines.

Interaction nets have been used for the implementation of optimal reduction [5,3,2] and for other efficient (non-optimal) implementations of the λ-calculus [7,8]. 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 [9]. The fact that different β-reductions may be interleaved also has the nasty consequence that the encodings need to simulate *boxes* in a more or less complex and costly way.

On the contrary, in [10] as well as in this paper, the encodings stick as closely as possible to traditional strategies. In particular, they ensure that essentially only one reduction is possible at a time. This is simpler to implement and avoids the need for boxes. 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.

In [10] the encodings are in standard interaction nets, featuring a standard agent called the evaluation token. Reductions are essentially triggered by the evaluation token, whose unicity is guaranteed, hence reduction is essentially deterministic. On the other hand, some restrictions of interaction nets are tailored to ensure strong confluence, and thus are no longer necessary if all reductions are triggered by a unique token.

In this paper, we want to apply the same technology to the call-by-need strategy. This will force us to abandon the restriction to Lafont's interaction nets, as explained below, and adopt Alexiev's formalism of interactions nets with multiple principal ports [1], or simply nets. Still, since reduction is directed by a unique token, evaluation is fully deterministic.

Call-by-need was introduced by Wadsworth [11]. The idea is relatively intuitive: a subterm should be evaluated only if it is needed, and if so, it should be evaluated only once. And so is the original formulation, in terms of graph rewriting. There have been several attempts to formalise this idea in different ways, sometimes with some loss of intuitions. In contrast with these approaches, we present another graph-based formalisation, making explicit at the object-level the rewrite strategy used in [11]. Our presentation is thus more primitive than other works; we therefore prefer to refer to the intuitive definition of Wadsworth (in terms of graphs) rather than prove properties with respect to any of the latter formalisms, contrasting with the approach of [10]. The encoding will thus be direct. It can be seen as a graph-based abstract machine, which is still strikingly close to a term representation. The encoding of terms is almost the same as for call-by-name and reduction rules are not much more complicated, thus sharing is indeed obtained "for free".

The rest of this paper is structured as follows. In Section 2, we recall some background on net rewriting. We give the encoding of terms in Section 3, the evaluation rules in Section 4 and a few properties in Section 5. We conclude in Section 6.

2 Nets

Nets have been introduced in [1] under the name interaction nets with multiple principal ports. They are roughly interaction nets [4], but where the agents are allowed to have any number of principal ports, instead of just one.

A net is a graph (not necessarily connected), whose edges are binary (i.e. they are not hyperedges) and unlabelled and whose vertices are labelled, have a fixed arity and are called *agents*. The attachments points of agents are called *ports*. Each agent has a fixed number of *principal ports*, depicted by an arrow; the other ports are called *auxiliary*. 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. Nets without agents (i.e. only with edges) are allowed and are called wirings; the extremes of wirings are also called free ports.

Two agents connected by principal ports on both sides form an *active pair* or *redex*. A net rewrite rule may replace an *active pair*, by an arbitrary net, provided that all free ports are preserved during reduction. This ensures that reduction is local, i.e. only the part of the net involved in the rewrite is modified.

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. Note that in general, no property of confluence can be expected from such a system.

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 [4]. In this case, reduction is strongly confluent.

3 Encoding of Terms

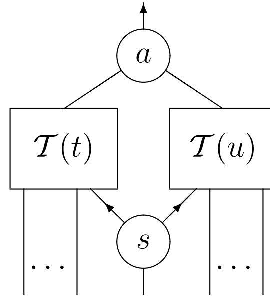
The translation $\mathcal{T}(\cdot)$ of λ-terms into interaction nets is very natural. We basically represent terms by their syntax tree, where we group together several occurrences of the same variable by agents s (corresponding to sharing) and bind them to their corresponding λ node (this is sometimes referred to as a *backpointer*). The nodes for abstraction and application are agents λ and a 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* [6] and in particular there will be no spontaneous reduction, something will have to trigger them: the *evaluation*

token.

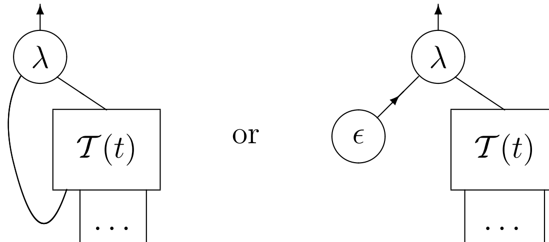
This is essentially the same encoding as in [10]. The only difference is that agents s representing sharing have two principal ports oriented upwards, so that they will not perform any copy before the evaluation token reaches them. In [10], these agents have one principal port oriented downwards, so that they can readily perform copying right after a β-reduction. These are in fact the agents c introduced below.

Variables. We consider only closed terms (open terms can be dealt with as in [10]), 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 interaction agent a whose principal port points at the root, and with $\mathcal{T}(t)$ and $\mathcal{T}(u)$ linked to its two auxiliary ports. 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 on the figure). Note that s agents have two principal ports oriented upwards, so that copy will not begin before an agent (the evaluation token) arrives from the top. These will be the only agents of the system with more than one principal port.



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.

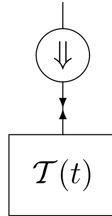
So far, we have only introduced agents λ and a strictly corresponding to the λ-calculus, as well as agents ϵ and s for the explicit resource managements necessary (and desirable: we do not want to hide such important things) in net rewriting. Also remark that the translation of a term has no active pair, hence is in normal form, whatever interaction rules are allowed. Moreover, it has exactly one principal port, at the root.

4 Evaluation by Interaction

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 it is exactly the same as in [10] (Section 4.1). There is no reason either to change the way copying and erasing are done (Section 4.3); the difference is only on *when* copying should occur, i.e. when we should activate a sharing agent into a copying agent (Section 4.2).

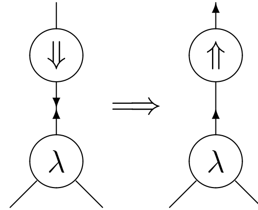
4.1 Linear part

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

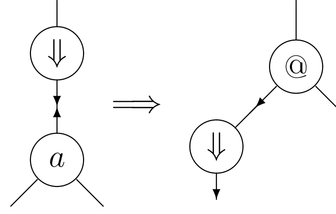


$\Uparrow\mathcal{T}(t)$ will be a net built in the same way, but with a $\Uparrow$ agent instead, with its principal port directed towards the root. In particular, $\Uparrow\mathcal{T}(t)$ is always a net in normal form. Relatively good intuition is carried by calling $\Downarrow$ “eval” and $\Uparrow$ “return”.

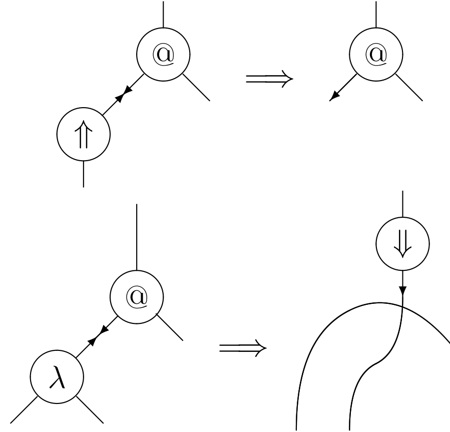
As for call-by-name, when we evaluate a term beginning by a λ, we should return that term:



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 left of the application. We also rename the agent a to @, which is still representing an application, but with its principal port no longer pointing to the root but to the left, so that interaction will be possible when the evaluation token returns.



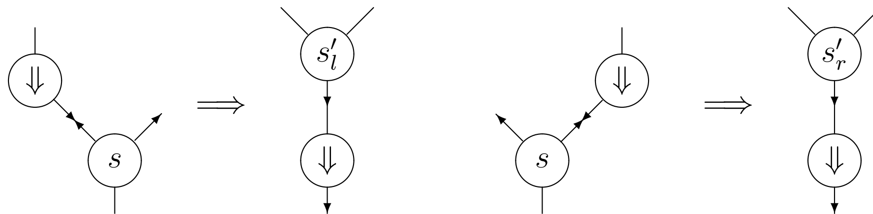
Finally, when the agent $\Uparrow$ returns from a successful evaluation to a @, then we know for sure that there is a λ just below the $\Uparrow$, and a β-reduction should be performed. Due to the restriction to binary interaction, this takes two steps:



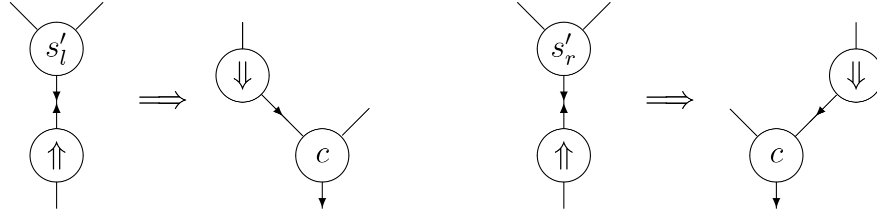
We link the variable port of the λ to the argument port of the @, which initiates the substitution; and we pursue evaluation on the body of the abstraction. This is the core of the interaction net machinery for linear λ-terms; it is the same as for call-by-name.

4.2 Sharing

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



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

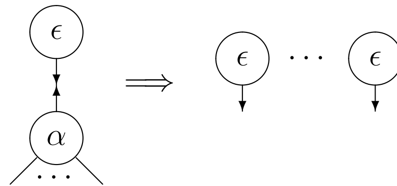


These two rules produce $\Downarrow$ agents, but they could equivalently produce $\Uparrow$ agents: we know that the agent under the initial $\Uparrow$ is a λ (possibly after some copying), so the c agent in the right-hand side of the rule will indeed copy that λ, and the $\Downarrow$ agent will be changed into a $\Uparrow$ agent. The interest of choosing to produce a $\Downarrow$ instead of a $\Uparrow$ is to force copying at that point instead of delaying it further.

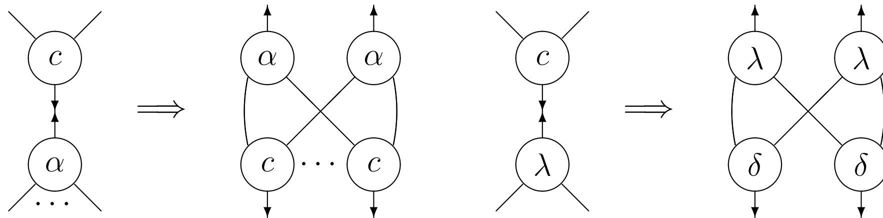
Remark 4.1 *Because of our encoding and because we follow a normal order strategy (we always go left in an application), it will often be the case that the token reaches a s on its left port. However, this is not always true, for instance in the term $(\lambda x.(\lambda y.\lambda z.z\, y)\, x\, x)\,(\lambda u.u)$. This is why we have to abandon the restriction to interaction nets.*

4.3 Copying, erasing

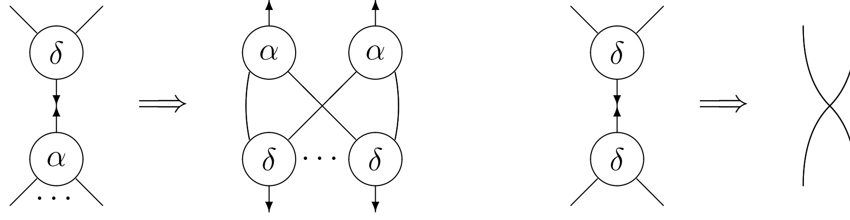
Copying and erasing are done in a classical way, by agents ϵ, c and δ. The auxiliary agent δ is introduced to duplicate abstractions, as explained below. The agent ϵ erases any agent and propagates according to the following schema (where α represents any agent except s):



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 and s.



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



A c agent always tries to copy an evaluated subnet, so agents c and s never meet (because the s would have been activated first). However, we may have to copy an agent s inside an abstraction using a δ. We know that if the agent s was in fact a c, there would be no problem. Hence the simplest (although not the most efficient) option when an agent other than the evaluation token meets an agent s is to simply activate the agent s into an agent s'' that behaves similarly to agents c, except that each reduction restores it into an agent s (instead of s''). This brings us back to the simpler framework without agents s and helps to preserve strong confluence (see below). The details (and pictures) are omitted. The sharing obtained is exactly call-by-need in the usual sense (e.g. like in Haskell): no reduction is shared inside an abstraction. A finer tuning of this issue could lead to an implementation of the *fully lazy* strategy [11], but could also make the issue of matching δ's harder.

5 Properties

Definition 5.1 *A net N is said to be valid if there exists a λ-term t such that $\Downarrow\mathcal{T}(t) \Longrightarrow^* N$.*

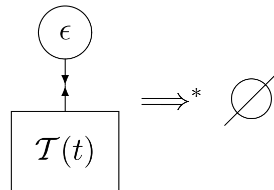
In a valid net, there may be several redexes involving agents c, δ or ϵ, however, we have the following result.

Proposition 5.2 *In a valid net there is exactly one occurrence of $\Downarrow$, $\Uparrow$ or of a λ−@ active pair.*

Proof. By induction, using the rules. □

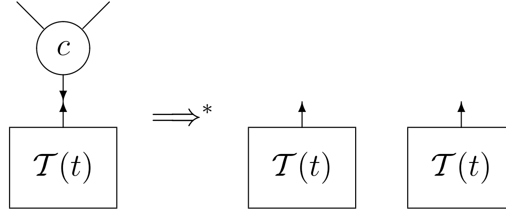
Classical results on packages [6] allow to state the two following properties:

Proposition 5.3 • *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:*



Rules can be partitioned into *evaluation rules* involving a token $\Downarrow$ or $\Uparrow$, or an active pair $\lambda - @$, and *administrative rules* involving agents c, δ, ϵ or s''. Remark that it is indeed a partition.

Proposition 5.4 *Reduction is strongly confluent on valid nets, i.e. 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 5.2), so at least one of the reduction is administrative. But notice that there is no overlap between evaluation and administrative rules, so 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. The only remaining cases involve an agent s and any agent among δ, ϵ, s'' on both its principal ports. In these cases, the agent s is simply transformed into an agent s'', and indeed $P = Q$. This completes the proof. □

Proposition 5.5 $\Downarrow\mathcal{T}(t) \Longrightarrow^* \Uparrow\mathcal{T}(v)$ *if and only if t reduces to v by the call-by-need strategy.*

Proof. There are two points:

- if the issue of sharing is ignored, the strategy followed is call-by-name (see [10]), hence by classical results, we do not evaluate a subterm unless it is needed;
- when a shared subterm is needed, the rules for sharing are clear: the subterm is evaluated first, and copied only afterwards (thanks to Proposition 5.3).□

6 Conclusion

We have presented a simple approach to express call-by-need in net rewriting. The approach is so simple that it is indeed a good alternative to working with terms. It is obtained from an encoding of call-by-name without adding any extra structure, and without much complication. 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 original formulation, and as the basis for a graph-based abstract machine.

References

[1] Alexiev, V., "Non-deterministic Interaction Nets," Ph.D. thesis, University of Alberta (1999).

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

[3] Gonthier, G., 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)* (1992), pp. 15–26.

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

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

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

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

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

[9] Pinto, J. S., *Sequential and concurrent abstract machines for interaction nets*, in: J. Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, Lecture Notes in Computer Science **1784** (2000), pp. 267–282.

[10] Sinot, F.-R., *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)*, Lecture Notes in Computer Science **3461** (2005), pp. 386–400.

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