Compiling the λ-calculus into Interaction Combinators

Ian Mackie and Jorge Sousa Pinto*
CNRS (UMR 7650), LIX, École Polytechnique
91128 Palaiseau Cedex, France
{mackie,pinto}@lix.polytechnique.fr

12 February 1998

Abstract

In this paper we present a case study of the use of Lafont's *interaction combinators* for the coding of the λ-calculus. We obtain a very simple, perhaps the simplest, system of rewriting which performs very well in comparison with extant interaction net implementations for the λ-calculus. The coding is shown to be correct, and we give a series of experimental results using a prototype implementation.

1 Introduction

Interaction nets [4] have been proposed by Yves Lafont as a new paradigm of computation, based on net rewriting. They can be understood as either a (graphical) programming language (analogous to term rewriting systems, for example) or as an "assembly" language for the encoding of other rewriting systems [3, 2, 1]. A key feature which makes interaction nets particularly interesting for study is that they capture explicitly *erasing* and *duplication*, which are the main components of any computational system. Once the process of duplication is built into a system, then one has access to the concept of *sharing*, for which interaction nets are very well adapted. Interaction nets can therefore be seen as capturing the fine details of computation.

In [5] Lafont proposed a finite set of *interaction combinators* which are complete — all other interaction net systems can be encoded into these basic elements. This result is analogous to the **SKI** combinators for the λ-calculus. The purpose of this paper is to present a case study of the use of these combinators of interaction for the encoding of the λ-calculus.

There are already several systems of interaction nets for coding the λ-calculus.

- Gonthier, Abadi and Lévy [3] gave an optimal one, using an infinite set of (indexed) agents. This system contains 5 agent templates, and 12 interaction rule schemes.
- In [6] an interaction system is given which uses a finite number of agents, but turns out to be quite inefficient since it fails to capture an essential part of the substitution process. Although very little sharing is captured by this system it does better than call-by-need. This system contains 8 agents, and 16 rewrite rules.
- More recently, another λ-evaluator based on interaction nets has been proposed [8], which is a variant of the above which resolves the problem of weak substitution. This system contains 9 agents and 22 rules, and thus the system is quite complex.

Each of these interaction systems provides an interesting implementation of the λ-calculus. However, our motivation here is to look for simpler systems, using a small set of agents and interaction rules. In our study of the use of interaction combinators, we have two choices:

*On leave from the University of Minho, Portugal. Research supported by Portuguese government PRAXIS XXI grant.

- either we code the systems above using the translation given by Lafont;
- or we try to use the combinators in a more direct way.

The disadvantage of the first approach is that the resulting nets become very large, and any intuition about the resulting system is quite hard to obtain. However, the proof of completeness of the combinators given by Lafont almost captures the coding of linear logic directly. By generalizing some of the results, and using an encoding of the λ-calculus into linear logic, we achieve a more direct, and more interesting, way of translating the λ-calculus into the combinators, using the second approach. The purpose of this paper is to show how this can be done, and show some properties of the resulting interaction system. This gives a system of interaction using only 3 agents and 6 rules.

The rest of this paper is structured as follows: In the next section we recall interaction nets, specifically the interaction combinators of Lafont. Section 3 is concerned with the coding of the λ-calculus into these combinators. In Section 4 we will study some simplifications of the encoding, and mention several optimizations. In Section 5 we will show that the coding captures reduction correctly. Section 6 gives some experimental results. Finally, in Section 7 we conclude the paper and suggest some directions for further study.

2 Background

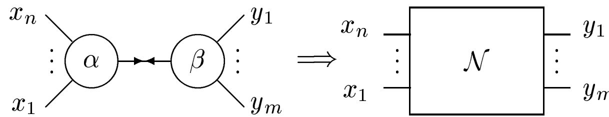
In this section we briefly recall the basics of interaction nets that we need for the rest of the paper. We refer the reader to [5] for additional details.

2.1 Interaction Nets

An interaction net system is specified by giving a set Σ of symbols, and a set *IR* of interaction rewrite rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*. An occurrence of a symbol $\alpha \in \Sigma$ will be called a *cell* (or an *agent*). If the arity of α is n, then the cell has $n+1$ *ports*: a distinguished one called the *principal port* depicted by an arrow, and n *auxiliary ports* labeled $x_1, \ldots, x_n$ corresponding to the arity of the symbol. We will say that the cell has $n+1$ *free* ports. We index ports clockwise from the principal port, hence the orientation of a cell is not important.

A net $\mathcal{N}$ built on Σ is a graph (not necessarily connected) with cells at the vertices. The edges of the graph, called *wires*, connect cells together at the ports such that there is only one edge at every port (edges may connect two ports of the same cell). The ports of a cell that are not connected to another cell are called the free ports of the net. There are two special instances of a net: a wiring (no cells), and the empty net.

A pair of agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected together on their principal ports is called an *active pair*; the interaction net analog of a redex. An interaction rewrite rule $((\alpha, \beta) \Longrightarrow \mathcal{N}) \in IR$ replaces an occurrence of the active pair (α, β) by a net $\mathcal{N}$. The rule has to satisfy a very strong condition that all the free ports are preserved during reduction. The following diagram illustrates the idea, where $\mathcal{N}$ is any net built from Σ.



It is almost trivial to see that this system of rewriting is confluent.

2.2 Interaction Combinators

In this section we will recall Lafont's system of combinators for interaction. There are 3 agents: $\Sigma = \{\gamma, \delta, \epsilon\}$. γ is a *constructor*, δ is a *duplicator*, and ϵ is an *erasing* agent. The set of interaction rules *IR* for this system is given in Figure 1. We also drop the arrow on the principal ports, which the reader can easily reconstruct from the diagrams.

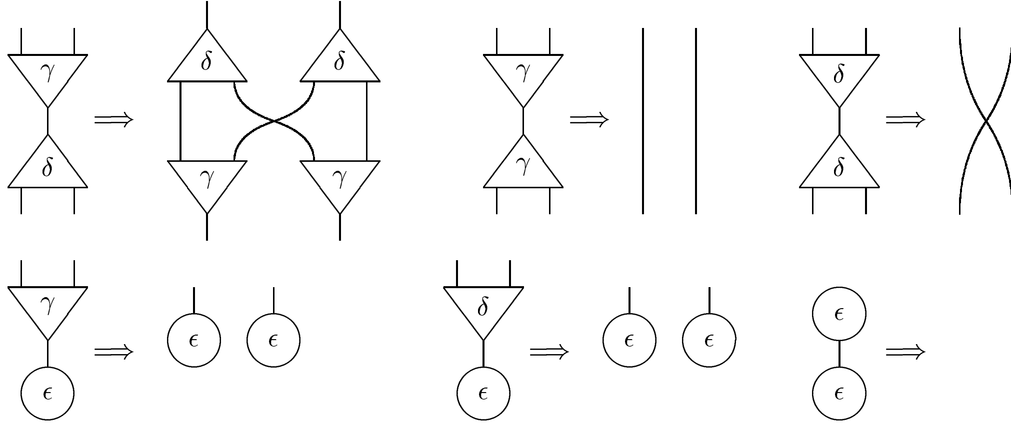
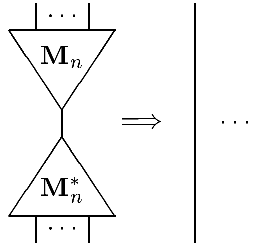


Figure 1: Interaction Rules.

2.3 Multiplexing and Copying

Following Lafont [5], we show how the basic combinators can be used to code several useful building blocks that we will use in our translation. For the moment, we can think of these as macros which should be expanded during the translation.

Multiplexing. We define nets $\mathbf{M}_n$ (multiplexors) and $\mathbf{M}_n^*$ (de-multiplexors) for any $n \in \mathbb{N}$, using only γ and ϵ, such that the following property holds:



Any coding of these nets will do — an example is given in Figure 2. The important point is that $\mathbf{M}_n$ and $\mathbf{M}_n^*$ are dual to each other such that the above property holds. In general, there are an infinite number of ways that the pair $(\mathbf{M}_n, \mathbf{M}_n^*)$ can be constructed.

Since the construction of $\mathbf{M}_n$ is arbitrary, and the differences are completely inessential, we shall take the construction modulo an equivalence. For any $\mathbf{M}'_n$ satisfying the above, we have: $(\mathbf{M}_n, \mathbf{M}_n^*) \equiv (\mathbf{M}'_n, \mathbf{M}_n^{*\prime})$, where $\equiv$ is an equivalence relation.

Packing, Duplicating and Unpacking. Next we look at ways of duplicating nets. For an arbitrary net $\mathcal{N}$ it is not possible to directly duplicate it using the δ agent since the net $\mathcal{N}$ may contain δ agents itself. However, a rather ingenious method was proposed by Lafont [5] to overcome this problem. The basic idea is to package up a net in such a way that all the δ agents are pulled out. This package can then be duplicated, and an additional construction will be given which will unpack the net to replace the δ agents. The following is just a slight variant of the presentation given in [5], to which we refer the reader for additional details.

If a net $\mathcal{N}$ contains n occurrences of δ, then it can be decomposed in the following way:

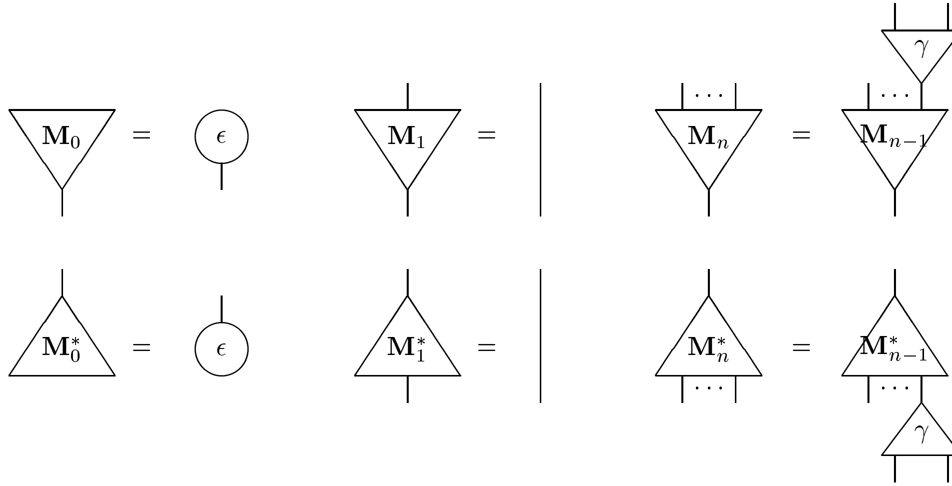
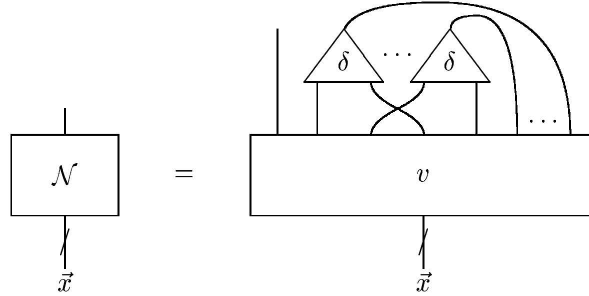
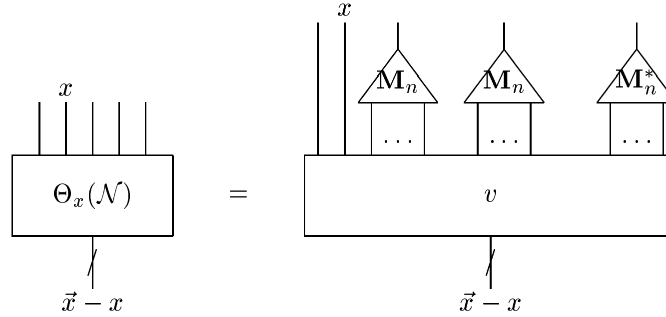


Figure 2: Example coding of the (de-)multiplexors.

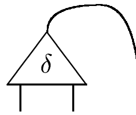


where v is "δ-free". Using the multiplexors $\mathbf{M}_n$ and the de-multiplexors $\mathbf{M}_n^*$ we can collect together all the left-hand sides, right-hand sides, and the principal ports of the δ agents as shown below, which give the packaging function $\Theta_x(\mathcal{N})$.

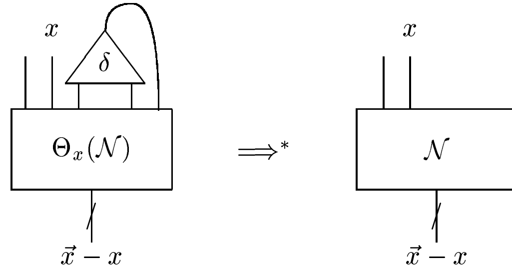


In addition, Θ_x also pulls out the edge x which will be useful for the translation which follows, but has nothing to do with pulling out the δ agents from the net $\mathcal{N}$. Using this construction, we can now duplicate this net (if it doesn't contain cycles). Remark that we allow additional free edges on the net v which is a generalization of the coding given in [5].

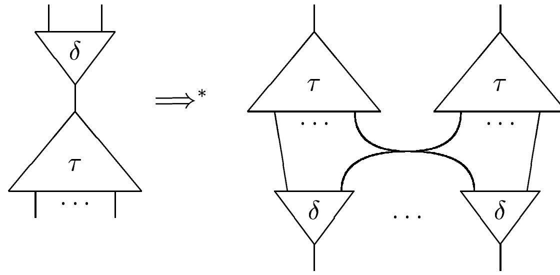
Finally, we can reconstruct the net $\mathcal{N}$ from $\Theta_x(\mathcal{N})$ using the following net which opens the package and replaces the δ agents:



This net is used in the following way, and we leave the reader to verify the reduction shown, which can also be found in [5].

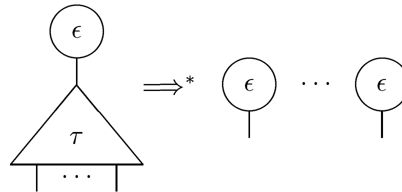


Nets that do not contain δ agents can be duplicated. However, the process of duplication can only complete if the net is of a particular form. Let τ be any net in normal form which is δ-free, and also a tree (called a principal net in [5]). Then the following sequence of reductions are possible to duplicate τ.



In general, our nets will not be of this form — they will contain cycles. We will come back to the issue of duplicating such nets in Section 5.

Similarly, the net τ can be erased using the ϵ agent as shown below.



We now have all the machinery that we need to code the λ-calculus, which is the subject of the next section.

3 Coding the λ-calculus

To simplify the compilation of the λ-calculus into interaction combinators, it is useful to extend the syntax of the λ-calculus with explicit discarding and copying constructs which are written as $[x = _]t$ and $[x = y, z]t$ respectively. The first says that x does not occur in M, and the second says that if x occurs twice in t then we call one y and the other z and they are combined by the copying construct. If x occurs more than twice then we can use the second rule repeatedly so all occurrences of the variable x get a unique name. Note that there are trivial encodings of the λ-calculus into this extension which do nothing more than variable counting. Two examples of this notation that we use later in this paper are the combinators $\mathbf{K} = \lambda xy.[y = _]x$ and $\mathbf{S} = \lambda xyz.[z = u, v]xu(yv)$. Using this notation, all variables occur exactly once in the body of a λ-term.

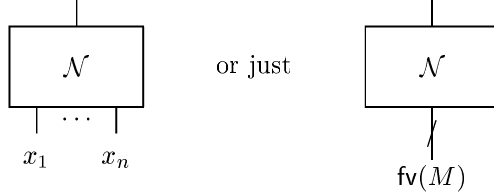
There is an obvious notion of *free variables* for this enriched λ-calculus, which we will take as an ordered sequence of variables, written as $[x; y; z]$, etc.

Definition 3.1 (Free Variables)

$$\begin{array}{lcl}
\mathsf{fv}(x) & = & [x] \\
\mathsf{fv}(\lambda x.t) & = & \mathsf{fv}(t) - [x] \\
\mathsf{fv}(tu) & = & \mathsf{fv}(t) + \mathsf{fv}(u) \\
\mathsf{fv}([x = _]t) & = & \mathsf{fv}(t) + [x] \\
\mathsf{fv}([x = y, z]t) & = & \mathsf{fv}(t) + [x] - [y; z]
\end{array}$$

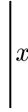
where $+$ is the obvious concatenation, and $-$ is the removal of the first occurrence of the element.

A term M in the λ-calculus, with $\mathsf{fv}(M) = [x_1; \dots; x_n]$, will be translated as a net $[\![M]\!] = \mathcal{N}$ with n free output edges which we draw as either

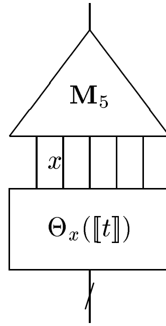


We will drop the labeling since this is derived directly from the term, and the order is preserved.

Variable. If M is a variable, say x, then $[\![M]\!]$ is simply an edge.

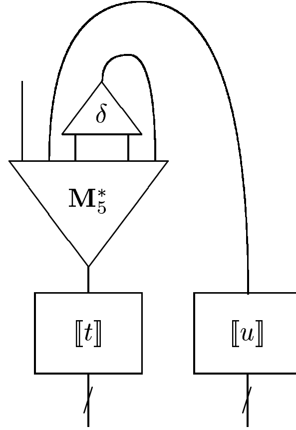


Abstraction. If M is an abstraction $\lambda x.t$, then $[\![M]\!]$ is given by the following net, using the construction Θ_x and the net $\mathbf{M}_5$.



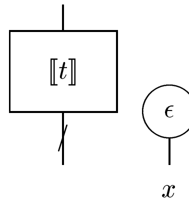
Hence the body of the abstraction t is δ-free, and the net $\mathbf{M}_5$ groups together (from left to right) the body, bound variable x, and the remaining three edges are for the occurrences of δ in the net.

Application. If M is an application tu, then $[\![M]\!]$ is given by the following net, using the net $\mathbf{M}_5^*$ and the code for opening a package.

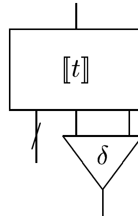


The net $\mathbf{M}_5^*$ groups together (from left to right) the result, the argument, and the code for opening a package.

Erase. If M is $[x = _]t$, then $[\![M]\!]$ is given by the following net, using the erasing agent ϵ.



Copy. If M is $[x = u, v]t$, then $[\![M]\!]$ is given by the following net, using the duplicating agent δ to group together u and v into a single edge x.



We have assumed that the occurrences of u and v were the rightmost edges of $[\![t]\!]$.

This completes the compilation of the λ-calculus into interaction combinators. There are several useful remarks that are worth mentioning:

- Θ_x will pull out all the δ's generated not only by the copy construct, but also the δ used in the coding of application.
- For readers familiar with the translations of λ-calculus into linear logic, we remark that we are using the "$D = !(D \multimap D)$" translation. More precisely, the agent γ is used for both the $\otimes$ and $\invamp$ multiplicative connectives. The construction $\Theta_x(\cdot)$ codes the promotion rule, and the δ agent for opening a package codes the dereliction. Contraction is coded with δ, and weakening with ϵ.
- There is a choice that can be made in the translation of the λ-calculus into this enriched λ-calculus. For example, $\lambda xy.y$ can be represented as either $\lambda x.[x = _]\lambda y.y$ or $\lambda xy.[x = _]y$, and there are similar choices for the use of the copying construct. Both alternatives are correct, but we have not yet established which is the best coding to use. We just remark that the performance of our evaluator can be severely influenced by the choice for a specific λ-term.

In Figure 3 we give some examples of the translation function, showing the nets representing the λ-terms: $\lambda x.x$, $\lambda xy.x$, and $\lambda xy.xy$.

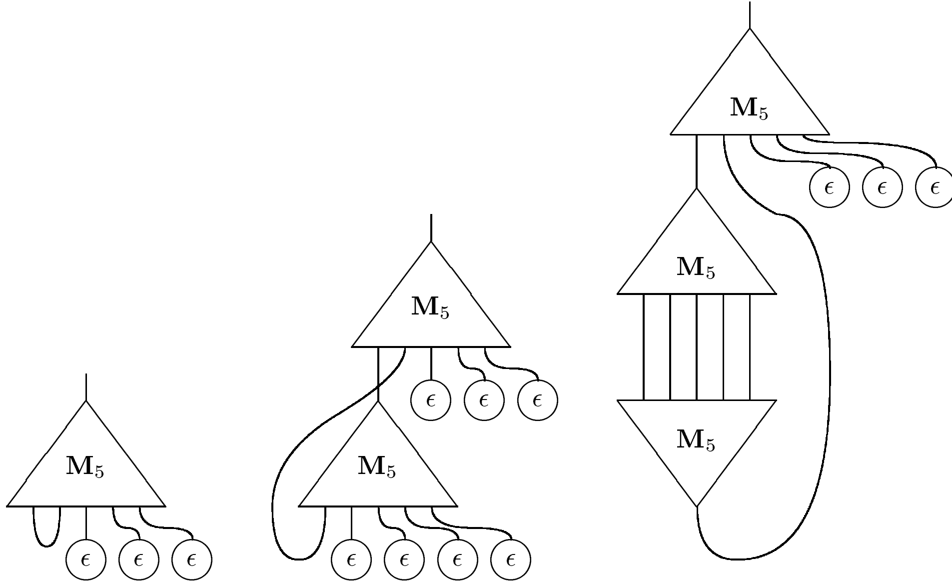


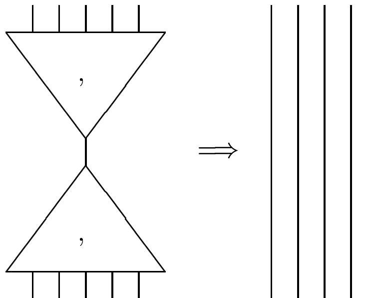
Figure 3: Example Terms.

In Figure 4 we show part of the reduction process for $(\lambda x.t)u \to_\beta t[u/x]$, which shows how application works. We come back to the issue of how the substitution process is completed in Section 5.

4 Optimizations

In this section we will look at several ways in which the resulting system can be optimized.

A study of the dynamics of this system reveals that most of the work done during execution is concerned with interactions for the multiplexing and de-multiplexing operations. A first suggestion to improve the situation is to observe that the nets $\mathbf{M}_5$ and $\mathbf{M}^*_5$ occur sufficiently often to merit the inclusion of these as agents. Each sequence of interactions between these two nets takes 4 interactions, and duplication and erasing of these nets is thus also expensive. Hence we suggest to add a new agent , $= \mathbf{M}_5 = \mathbf{M}^*_5$, with the following rewrite rule:



Note that , $, \delta, \epsilon$ is also a complete system of interaction since $\mathbf{M}_n$ and $\mathbf{M}^*_n$ can be built, for all n, using , and ϵ. The rules for , with δ and ϵ are the expected ones.

Whether it is worth adding additional agents $\mathbf{M}_n$, for each n, is not yet clear, however having several different multiplexing agents obviously makes the translations simpler and more compact.

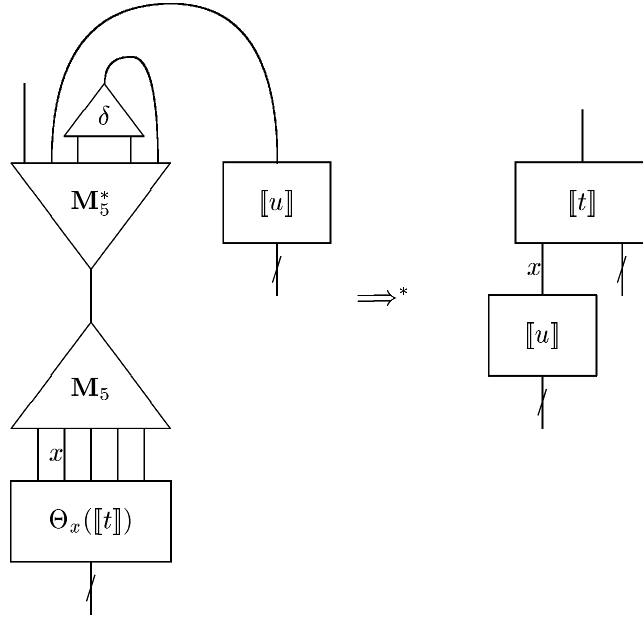
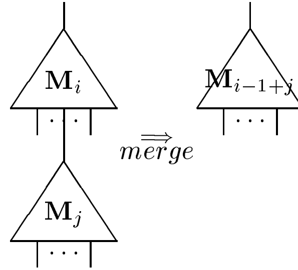


Figure 4: Example Reduction.

If we are prepared to leave the interaction net world of only having interactions on the principal ports, then we can considerably improve the dynamics of the system in the following way. During reduction, different (de-)multiplexors are generated. We could *merge* them as they are created, i.e.,

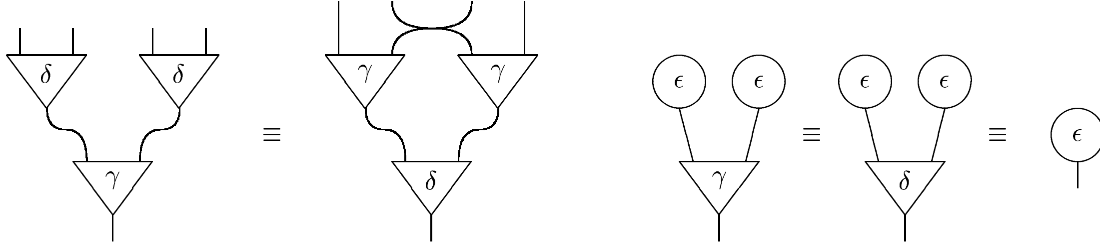


Now $\mathbf{M}_n$ can only interact with $\mathbf{M}_m$ when $n = m$, otherwise we have to wait for additional merging to be done so that the multiplexor and de-multiplexor are matched. However, we have not implemented this, and leave the details for further study.

Erasing can also be improved if we can identify disconnected nets.

5 Soundness

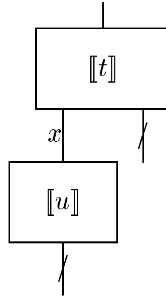
Before stating any properties of our encoding, we give a notion of equivalence of interaction nets, which was also given by Lafont in [5] (and proved correct using a path semantics).



These equivalences, together with the equivalence of multiplexors introduced in Section 2, will be used in our correctness result.

We cannot expect that our encoding will capture β-reduction in full generality, since a specific strategy is imposed. Moreover, as with other encodings using interaction nets, sharing is a key issue which imposes a weak notion of reduction, i.e. certain reductions cannot complete. However, this is precisely the advantage of interaction nets. If we try to duplicate an open term, say the application xy, then the evaluator will stop since it tries to share the potential redex — i.e. once the variable x has been substituted for an abstraction $\lambda z.t$, then the application $(\lambda z.t)y$ will only be done once. The process of duplication can then continue. Never-the-less, using the notion of equivalence introduced, we can show that the interaction system is sound.

In Figure 4 we showed the first few reductions of the (outermost) redex $[\![(\lambda x.t)u]\!]$. We were left with the following net:



Here we show how the reduction process continues from this point to complete the process of substitution. It is fruitful to factor out the *linear* case which seems particularly interesting.

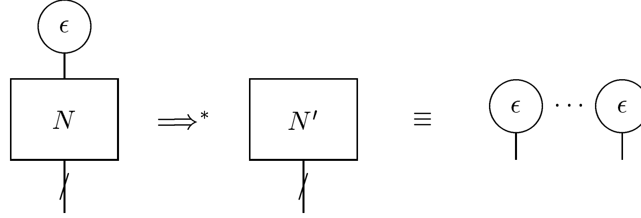
Lemma 5.1 (Substitution, linear case) *If x is linear (not copied or discarded) in t, then $[\![(\lambda x.t)u]\!] \Longrightarrow^* [\![t[u/x]]\!]$. Moreover, the sequence of interactions is nothing more than that shown in Figure 4.*

Proof: Straightforward induction, using the fact that the construction $\Theta_x(\cdot)$ does not affect linear occurrences of variables. $\square$

The remarkable point here is that linear substitutions come for free — there is no overhead at all. We invite the reader to compare this with other systems of interaction for the λ-calculus where the cost of substitution (even linear) can be very expensive indeed (worst case proportional to the size of the term being substituted).

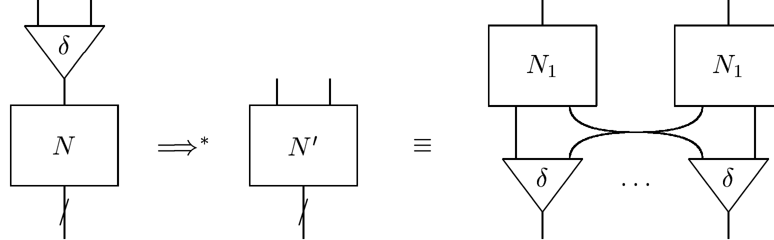
However, the general result relies on the non-linear cases. It is sufficient to show that erasing and duplication of encoded terms behave well. The following two lemmas are all that we need. Both can be proved by induction, and using $\equiv$.

Lemma 5.2 (Erasing) *If $[\![t]\!]$ normalizes to a net N, then:*



Note that if $[\![t]\!]$ is not terminating, then the process of erasing will not complete either. However, we could decide to ignore disconnected nets...

Lemma 5.3 (Duplication) *If t has a normal form u, and $[\![t]\!]$ normalizes to a net N, then:*



where $N_1 \equiv [\![u]\!]$.

Again if $[\![t]\!]$ does not terminate, then neither will the duplication process.

Using these results, we can now show the main result of this section.

Theorem 5.4 (Soundness) *If $t \to_\beta u$ is a reduction in the λ-calculus, $[\![t]\!] \Longrightarrow^* N_1$, then $[\![u]\!] \Longrightarrow^* N_2$ and $N_1 \equiv N_2$, where N_1 and N_2 are nets in normal form.*

6 Experimental Results

Here we give a set of experimental results, using an implementation of the interaction net system given. We take as our unit of measure the number of interactions performed during reduction — since each interaction is a constant time operation, this gives a convenient measure of the cost of a computation. In addition, we indicate the number of "β-reductions". We compare the results with all other interaction net implementations of the λ-calculus.

We use λ-terms coding Church numerals, which provide an excellent set of test data and generate vast computations (application is exponentiation):

$$\begin{array}{lll} \mathbf{2} & = & \lambda fx.[f = f_1, f_2](f_1(f_2x)) \\ \mathbf{3} & = & \lambda fx.[f = f_1, f'][f' = f_2, f_3](f_1(f_2(f_3x))) \\ \text{etc.} & & \end{array}$$

Since all the evaluators perform some notion of weak reduction, we force evaluation to normal form using the identity function.

In the following table the columns represent 4 different evaluators under test. Eval 1 is the optimal one, as reported in [3], Eval 2 is reported in [6], Eval 3 is reported in [8], and finally, Eval 4 is the one reported in this paper.

Term	Eval 1	Eval 2	Eval 3	Eval 4
22II	204(9)	56(11)	38(9)	66(9)
222II	789(16)	304(42)	127(20)	278(18)
3II	75(5)	17(5)	17(5)	32(5)
33II	649(15)	332(45)	87(15)	322(15)
322II	7055(21)	4457(531)	383(51)	3268(29)
223II	1750(19)	1046(132)	213(31)	869(22)
44II	3456(23)	2816(347)	148(23)	2447(23)

We remark that we have used an extended set of combinators in these results for EVAL 4, to include $\mathbf{M}_3$, $\mathbf{M}_4$ and $\mathbf{M}_5$ explicitly into the system.

We achieve better sharing than all the other evaluators, except the optimal one. However, we seem quite far from having the least number of interactions. Many other optimizations remain to be implemented, for which there seems to be a lot of scope.

7 Conclusions

In this paper we have shown that a system of interaction combinators can be used to give an encoding of the λ-calculus. We believe that this is the simplest of all the known encodings of the λ-calculus into interaction nets. Moreover, experimental results indicate a very high level of sharing of β-reduction, and the overhead is considerably less that the optimal one.

Further work needs to be done to test these ideas out using a more realistic language, extending the system with data-types, etc., so that the resulting system can be compared with other implementations of the functional languages.

Finally, we remark that the path semantics for the interaction combinators [5] may provide alternative methods of implementation in the spirit of the Geometry of Interaction Machine [7].

References

[1] Maribel Fernández and Ian Mackie. Integrating paradigms via interaction nets. In Manuel M. T. Chakravarty, Yike Guo, and Tetsuo Ida, editors, *Multi-Paradigm Logic Programming, MPLP'96. Proceedings of the JICSLP'96 Post Conference Workshop*, pages 137–146. Technical University of Berlin Report Number 96–28, August 1996.

[2] Maribel Fernández and Ian Mackie. Interaction nets and term-rewriting systems. *Theoretical Computer Science*, 190(1):3–39, 1998.

[3] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, pages 15–26, January 1992.

[4] Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 95–108, January 1990.

[5] Yves Lafont. Interaction combinators. *Information and Computation*, 137(1):69–101, 1997.

[6] Ian Mackie. *The Geometry of Implementation.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, September 1994.

[7] Ian Mackie. The Geometry of Interaction Machine. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 198–208, January 1995.

[8] Ian Mackie. YALE: Yet another lambda evaluator (based on interaction nets), 1998. In preparation.