YALE: Yet Another Lambda Evaluator Based on Interaction Nets

Ian Mackie

CNRS (UMR 7650) and École Polytechnique
Laboratoire d'Informatique (LIX)
91128 Palaiseau Cedex, France
mackie@lix.polytechnique.fr

Abstract

Interaction nets provide a graphical paradigm of computation based on net rewriting. They have proved most successful in understanding the dynamics of reduction in the λ-calculus, where the prime example is the implementation of optimal reduction for the λ-calculus (Lamping's algorithm), given by Gonthier, Abadi and Lévy. However, efficient implementations of optimal reduction have had to break away from the interaction net paradigm. In this paper we give a new *efficient* interaction net encoding of the λ-calculus which is not optimal, but overcomes the inefficiencies caused by the bookkeeping operations in the implementations of optimal reduction. We believe that this implementation of the λ-calculus could provide the basis for highly efficient implementations of functional languages.

1 Introduction

It is well-known that the λ-calculus can be seen as a prototypical functional programming language, and moreover as the foundation for an implementation. However, from the point of view of a compiler writer, the theory falls a long way short of this ideal, because of two "defects":

1. The reduction steps are too "big" in that β-reduction $(\lambda x.t)u \to t[u/x]$ already takes us out of the realms of the pure theory since substitutions, which are the hard part to implement, are not captured by the basic theory. Several possible solutions to the problem that have been put forward include:
   - Combinators $(\mathbf{S}, \mathbf{K})$ eliminate the problem of variables and substitutions, but the steps are still far from being atomic: $\mathbf{S}PQR \to PR(QR)$, $\mathbf{K}PQ \to P$. In the first we have to duplicate R and in the second we have to erase Q. Both of these operations cannot be seen as atomic computation steps.
   - Explicit substitution calculi (for instance the $\lambda\sigma$-calculus [1]) are another approach, but they seem to be too tied up with syntactical problems (which have nothing to do with the actual computation) to be of practical use.
   - A third solution is graph reduction, specifically Lamping's algorithm for optimal reduction [16], and the reformulation of the algorithm as an interaction net, as developed by Gonthier, Abadi and Lévy [11]. However, the total cost of reduction includes expensive "bookkeeping" operations which makes the algorithm impractical in general.
2. There is no implicit or explicit notion of a strategy. This is of course a strength and a weakness. A strength because we have the flexibility to do reductions how we please, but a weakness in that there is no built in notion of *sharing*, which is essential for any efficient implementation.

   It is also well known that neither call-by-value nor call-by-name is the "best" strategy to use, simply because there is always the risk that we either duplicate work, or do work that will be erased later. Call-by-need [24, Chapter 4] is one solution to this problem. However, there is an unfortunate problem in that the reduction is usually to *weak head normal form* and thus any reductions inside the λ-abstraction will often be duplicated. Again, Lamping's algorithm can be seen as a solution to this problem, since it overcomes the problem of which is the best strategy to take, but introduces a huge overhead to ensure that the minimum number of β-reductions are done.

Summarizing these points, we claim that a coding of the λ-calculus requires that we impose a *strategy* which captures a natural notion of sharing, and moreover we should give a set of *atomic* computation steps for reduction.

Linear logic [9] offers a new and exciting perspective on these issues, since it offers new insights into sharing computations and evaluation strategies. In particular, the cut-elimination process in linear logic makes explicit the duplication and erasing of terms which gives a handle on sharing and garbage collection. For functional languages there are basically two approaches to using linear logic, which we classify as follows:

Intuitionistic Linear Logic. The typed λ-calculus relates to intuitionistic logic by the Curry-Howard Isomorphism. If we shift to intuitionistic linear logic, then we can construct various linear λ-calculi which correspond to linear logic proofs. Several languages have been proposed based in these ideas, amongst others: Holmström [12], Lafont [13], Abramsky [3], Wakeling [25], and the language Lilac [18] of the present author. More recently, abstract machines based on linear explicit substitution calculi have been developed [4].

Classical Linear Logic. A second approach is classical linear logic, which offers tools such as proof nets [9] and the geometry of interaction [10]. Using translations of the λ-calculus into proofs of classical linear logic, we can reason about β-reduction in a completely different way.

Proof nets also gave birth to interaction nets [14], which have proved most successful for the implementation of optimal reduction in the λ-calculus, as given by Gonthier, Abadi, Lévy [11].

Implementations of the λ-calculus via the geometry of interaction have also been proposed: local and asynchronous β-reduction [7], and the geometry of interaction machine [19].

It seems that to date the approach based on classical linear logic has been more fruitful. Whether this is just a consequence of the fact that real implementations have not been developed to take advantage of the theory, or whether there are inherent limitations to the first approach remains an open question. But many of the original claims and hopes of these ideas do not seem to have materialized. With classical linear logic there is now a clear understanding of optimal reduction, new evaluation strategies, and new ways of implementing.

In this paper we add another way of implementing the λ-calculus using ideas based on classical linear logic. Specifically, an analysis of the cut-elimination process for linear logic [20] suggests an obvious strategy for the λ-calculus. Roughly, the strategy that we introduce never copies terms with free variables, or containing redexes. This gives a new strategy for reduction in the λ-calculus which captures a great deal of sharing (but not optimal) and overcomes many of the inefficiencies mentioned above.

Specifically, we give yet another λ-evaluator (YALE) based on interaction nets. There are already several systems of interaction nets for the λ-calculus:

- Gonthier, Abadi and Lévy [11] gave an optimal one, using an infinite set of (indexed) agents. However, this system turns out to be very expensive in practice.
- In [17] an interaction system is given which uses a finite number of agents, but it does not implement substitution through λ-abstractions, which is essential to obtain sharing. Although very little sharing is captured by this system it does better than call-by-need.
- In [21] another approach is developed based on the interaction combinators of Lafont [15]. Although the system of rewriting is perhaps the simplest one, the "atomicity" of the system leads to a great deal of work.

The system of interaction nets that we present in this paper is an attempt to improve on all of these systems. The key ingredients of this work are:

- We introduce a new calculus of explicit substitutions based on linear logic, which appears to overcome many of the syntactical problems usually associated with such calculi.
- We give a new encoding of the λ-calculus in interaction nets, which is derived directly from the calculus of explicit substitutions. Moreover, it has a solid foundation from cut-elimination in linear logic (we refer the reader to [20]).
- We extend the theory to cover a simple programming language with conditional and recursion (**PCF**).

Related Work. This work can be seen as an alternative thread of research to that of the successful implementation of optimal reduction given in [5]. Our work relates to this in that we start with a system of interaction nets for linear logic, and implement the λ-calculus. The key difference is that we stick with interaction nets throughout. Moreover, this work is not based around optimal reduction, but on a natural encoding of linear logic into interaction nets.

Remark. No knowledge of linear logic is required to understand this paper since it can be understood directly as a simple graph reduction mechanism. However, it is fruitful to know that there is a solid logical background to this implementation.

Overview. The rest of this paper is structured as follows. In the next section we recall the basic notions of interactions nets. In Section 3 we introduce a calculus of explicit substitutions for the λ-calculus. In Section 4 we introduce our interaction system for the λ-calculus. Section 5 in concerned with the dynamics of the system. Sections 6 gives some extensions to the language **PCF**. In Section 7 we give some benchmark results, comparing our interaction net evaluator with optimal reducers based on the same technology. We conclude the paper in Section 8.

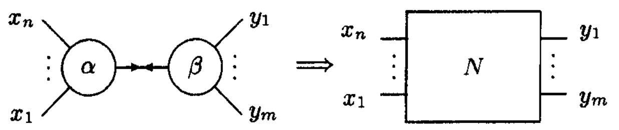
2 Background

We assume that the reader is familiar with both interaction nets [14, 15] and λ-calculus [6]. Here we set up the notation and conventions for interaction nets.

Interaction Nets. An interaction net system is specified by giving 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 $\alpha \in \Sigma$ is 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* labeled $x_1, \dots, x_n$ corresponding to the arity of the symbol. We will say that the agent has $n+1$ *free* ports. We index ports clockwise from the principal port, hence the orientation of an agent is not important.

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 (edges may connect two ports of the same agent). The ports of an agent that are not connected to another agent are called the free ports of the net. There are two special instances of a net: a wiring (no agents), 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 rule $((\alpha, \beta) \Longrightarrow N) \in \mathcal{R}$ replaces an occurrence of the active pair (α, β) by a net N. The rule has to satisfy a very strong condition: all the free ports are preserved during reduction, and there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.



We do not require that there is a rule for each pair of agents, but if we create a net with an active pair for which there is no rewrite rule, then we have a deadlock. The interaction net system that we present in this paper will be deadlock-free in this sense. An interaction net is in normal form when there are no active pairs.

3 Explicit Substitutions for the Simply Typed λ-calculus

It will be convenient, so that we can express certain strategies in the typed λ-calculus, to use a calculus of explicit substitutions. Moreover, since we are interested in controlling duplication and erasing of terms explicitly during reduction, we use a notation inspired by various calculi for linear logic, and include explicit syntactical operations for duplication and erasing. Hence, we have a calculus where we can monitor precisely the propagation of a substitution.

We will be concerned with weak reduction in the first instance, then mention extensions of the results to strong reduction later. In the λ-calculus, weak reduction (also known as lazy reduction [2]) imposes that we do not reduce under a λ-abstraction (where λ is seen as a constructor), whereas in the calculus of explicit substitutions this is often interpreted as not pushing substitutions through a λ-abstraction.

There are various motivations for weak reduction: one is that we can distinguish between $\lambda x.\Omega$ and Ω, where Ω is a non-terminating term. A second is that the process of pushing a substitution through a λ-abstraction is a delicate operation since free variables may become bound, thus renaming may be necessary (or "shifting" operations in the $\lambda\sigma$-calculus) which can be regarded as an expensive syntactical overhead on the calculus.

We have two objections to these restrictions for our work, which are motivated purely from an efficiency point of view:

1. Avoiding reduction under a λ-abstraction may cause duplication of work later. A typical example of this situation is the term:

   $$(\lambda y.y(y\mathbf{I}))(\lambda x.\underline{\mathbf{I}x})$$

   where $\mathbf{I} = \lambda x.x$. The underlined redex $\mathbf{I}x$ will be contracted twice, whereas allowing reduction under a λ-abstraction only requires one contraction of this redex.

   Of course, reducing under a λ introduces a weakness into the system since internal reductions may not terminate. However, for the first part of our work we will only consider the simply typed λ-calculus where all reduction sequences terminate. Later we will introduce recursion, and come back to the issue of non-termination.

2. The restriction of not pushing substitutions through a λ-abstraction seems too strong, since the only time we need to worry about variable capture problems is when there are free variables in the term being substituted. There seems to be no fundamental reason to block substitutions which do not contain free variables (*i.e.*, closed terms).

   Avoiding substitution through a λ-abstraction may also cause duplication of work later. A simple instance of this is the following term:

   $$(\lambda z.(\lambda y.y(y\mathbf{I}))(\underline{x\mathbf{I}}))[\mathbf{I}/x]$$

   The underlined term $x\mathbf{I}$ will become a redex after substitution, and can thus be contracted before duplication. If we postpone the substitution then two substitutions will need to be made, thus duplicating the redex.

In addition, copying terms with free variables introduces a weakness in the strategy in that redexes that might be created later during reduction will also be duplicated. An obvious solution to this would simply be to only copy terms that have no free variables, and moreover, when they are in normal form.

Putting all of these ideas together suggests a strategy for implementing the simply typed λ-calculus. The strategy is clearly a weak one since it will not always be the case that substitutions will be closed terms, but we shall show that it is adequate for the evaluation of programs (*i.e.*, terms of ground base type). We will also suggest ways of extending this strategy to cope with strong reduction.

This strategy in fact comes directly from general observations about efficient strategies for cut-elimination in linear logic, for which we refer the reader to [20].

We now give the calculus of explicit substitutions that we will use, and give the corresponding reduction rules. This provides the basis for an implementation in terms of interaction nets which is well suited to capture this strategy.

Definition 3.1 (Types) *Types are built from the following grammar, where σ, τ are used to range over type metavariables.*

$$\sigma \quad ::= \quad \mathbf{int} \mid \mathbf{bool} \mid \sigma \rightarrow \tau$$

where **int** *is the type of integers,* **bool** *for booleans, and $\rightarrow$ is the (right associative) function type constructor.*

Definition 3.2 (Typed Terms) *We write $t : \sigma$ for a typed term t as usual. The syntax of our calculus is shown in Figure 1, where we define in parallel the term construction, type constraint, variable constraint, and free variables of a term.*

Convention 3.3 (Linearity) *We adopt a strong version of the usual variable convention [6] in that all variables (free and bound) are chosen to be different. Thus in a term t all the variables occur exactly once.*

A few remarks on the notation are in order. Abstraction $\lambda x.t$ enforces that the variable actually does occur at least once free in the term t. The construct $[x = \_]t$ is a term t where we make the variable x occur free explicitly. Application tu enforces that the free variables in t and u are disjoint, thus do not occur more than once. The construction $[x = y, z]t$ ensures that all variables have a unique name: If t has two occurrences of the variable x, then we rename one to y, the other to z and then use the construct. If x occurs more than twice, then we can use this construct repeatedly. The notation $t[u/x]$ is our notation for substitution.

There are obvious translations of the usual λ-calculus into this notation which we will not elaborate. One of the key advantages of this notation is that the variable counts have already been done, which will be of significant use when we give the translation into interaction nets. Hence from one perspective this is nothing more than a convenient notation that serves as an intermediate language for our translations, where the main bureaucratic issue of counting occurrences of variables in a term has already been taken into account.

We give some examples of terms in this calculus.

$$\begin{array}{lcl}
\mathbf{I} & = & \lambda x.x \\
\mathbf{K} & = & \lambda x.\lambda y.[y = \_]x \\
\mathbf{S} & = & \lambda x.\lambda y.\lambda z.[z = z_1, z_2](xz_1)(yz_2) \\
\mathbf{2} & = & \lambda f.\lambda x.[f = f_1, f_2](f_1(f_2 x))
\end{array}$$

Definition 3.4 (Equivalences) *We write $t \approx u$ if term t and u are related by the following structural equivalence. We write X for an explicit operation $[x = \_]$, $[x = y, z]$, and S for a substitution $[v/x]$. In the following, the obvious variable constraints are implied: i.e., no variable capture occurs.*

$$\begin{array}{rcll}
(tS_1)S_2 & \approx & (tS_2)S_1 & (1) \\
X_1(X_2t) & \approx & X_2(X_1t) & (2) \\
(Xt)u & \approx & X(tu) & (3) \\
t(Xu) & \approx & X(tu) & (4) \\
(Xt)S & \approx & X(tS) & (5)
\end{array}$$

In addition, if t and u are related by the obvious notion of α-equivalence (renaming), then $t \approx u$. These equivalences apply in any context.

We give several examples to illustrate the idea:

$$\begin{array}{lcl}
[x = \_]([y = \_]t) & \approx & [y = \_]([x = \_]t) \\
(t[u/x])[v/y] & \approx & (t[v/y])[u/x] \\
([x = \_]([y = y_1, y_2]t))[w/z] & \approx & ([y = y_1, y_2]([x = \_](t[w/z])))
\end{array}$$

Definition 3.5 (Values) *A closed term t is a value* (value(t)) *if it is a weak head normal form:*

$$\lambda x.t' \quad \overline{n} \quad \mathbf{tt} \quad \mathbf{ff}$$

where t' is any term such that fv$(t') = \{x\}$.

Definition 3.6 (Weak reduction) *We define a weak notion of reduction $\rightsquigarrow_w$, as shown in Figure 2. We write* w-nf(t) *if no reduction rule can be applied to t, i.e., $t \not\rightsquigarrow_w$.*

These reductions are applied modulo the structural congruence:

$$\frac{t \approx t' \qquad t' \rightsquigarrow_w u' \qquad u' \approx u}{t \rightsquigarrow_w u}$$

and moreover, can be applied in any context $C[\cdot]$, including within substitutions and under λ.

$$\frac{t \rightsquigarrow_w u}{C[t] \rightsquigarrow_w C[u]}$$

Remark 3.7 *There are many conditions on the reduction relation $\rightsquigarrow_w$, here we try to informally motivate them.*

The application $(\lambda x.t)u$ is restricted to the case when the only free variable occurring in t is x. Thus before application can take place we must perform all the substitutions for the other free variables for t. This forces substitutions to take place before β-reduction. Only values *can be substituted through a λ which avoids all problems due to variable capture, and requires that we have at least reduced the substitution to weak head normal form. Terms can only be copied when there are no free variables, and moreover when the term is in weak normal form. This condition avoids duplication of reduction, and moreover avoids the duplication of potential redexes. The final rule allows substitutions to be moved inside other substitutions, which is essential for the previous rules explained above.*

Definition 3.8 (Programs) *A* program *is a closed term of ground type (***int** *or* **bool***).*

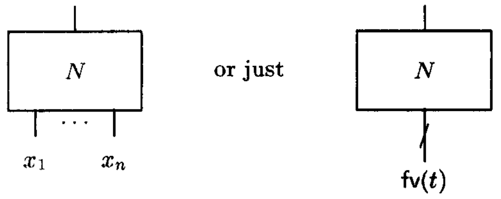
Theorem 3.9 (Adequacy) *If t is a program, then exactly one of the following holds:*

- $t \rightsquigarrow_w^* \overline{n}$, *for some unique $\overline{n}$, if t :* **int**.
- $t \rightsquigarrow_w^*$ **tt**, *if t :* **bool**.
- $t \rightsquigarrow_w^*$ **ff**, *if t :* **bool**.

Proof: This can be proved by first showing that the calculus preserves types during reduction (Subject Reduction), and then showing for a closed term of ground type all substitutions will eventually complete, *i.e.*, that there are enough closed substitutions. □

4 Nets for λ-calculus

We now show how we can implement our λ-calculus, by giving an encoding as an interaction net. We give the translation $\mathcal{T}(\cdot)$ of the typed λ-calculus into interaction nets. The agents required for the translation will be introduced on demand; in Section 5 we give the interaction rules. A term t in the λ-calculus, with fv$(t) = \{x_1, \dots, x_n\}$, will be translated as a net $\mathcal{T}(t) = N$ with the root edge at the top, and n free edges corresponding to the free variables, which we draw as either



We will drop the labeling on the edges since they are derived directly from the term, and the order is preserved.

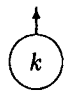
Constants at Base Type. Natural numbers and Boolean constants $k \in \{n, \mathbf{tt}, \mathbf{ff}\}$ are all coded by introducing a new unary agent for each constant. The only port is the principal port, and we slightly abuse notation and overload the agents which will simplify our presentation. These constants are drawn as follows:

Term	Type Constraint	Variable Constraint	Free Variables
$\overline{n}$: **int**	–	–	–
tt, **ff** : **bool**	–	–	–
$x^\sigma : \sigma$	–	–	$\{x\}$
$\lambda x^\sigma.t : \sigma \to \tau$	$t : \tau$	$x \in \mathsf{fv}(t)$	$\mathsf{fv}(t) - \{x\}$
$(tu) : \tau$	$t : \sigma \to \tau, u : \sigma$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) + \mathsf{fv}(u)$
$([x^\sigma = \_]t) : \tau$	$t : \tau$	$x \notin \mathsf{fv}(t)$	$\mathsf{fv}(t) + \{x\}$
$([x^\sigma = y, z]t) : \tau$	$t : \tau, y : \sigma, z : \sigma$	$x \notin \mathsf{fv}(t), y \neq z, \{y, z\} \subseteq \mathsf{fv}(t)$	$\mathsf{fv}(t) - \{y, z\} + \{x\}$
$t[u/x] : \tau$	$t : \tau, x : \sigma, u : \sigma$	$x \in \mathsf{fv}(t)$	$\mathsf{fv}(t) - \{x\} + \mathsf{fv}(u)$

Figure 1: Syntax: Typed λ-calculus

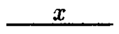
Reduction			Condition
$(\lambda x.t)u$	$\rightsquigarrow_w$	$t[u/x]$	if $\mathsf{fv}(t) = \{x\}$
$x[v/x]$	$\rightsquigarrow_w$	v	–
$(tu)[v/x]$	$\rightsquigarrow_w$	$(t[v/x])u$	if $x \in \mathsf{fv}(t)$
$(tu)[v/x]$	$\rightsquigarrow_w$	$t(u[v/x])$	if $x \in \mathsf{fv}(u)$
$(\lambda y.t)[v/x]$	$\rightsquigarrow_w$	$\lambda y.t[v/x]$	if $\mathsf{value}(v)$
$([x = y, z]t)[v/x]$	$\rightsquigarrow_w$	$t[v/y][v/z]$	if $\mathsf{fv}(v) = \varnothing$ and $\mathsf{w\text{-}nf}(v)$
$([x = \_]t)[v/x]$	$\rightsquigarrow_w$	t	if $\mathsf{fv}(v) = \varnothing$ and $\mathsf{w\text{-}nf}(v)$
$(t[w/y])[v/x]$	$\rightsquigarrow_w$	$t[w[v/x]/y]$	if $x \in \mathsf{fv}(w)$

Figure 2: Weak Reductions

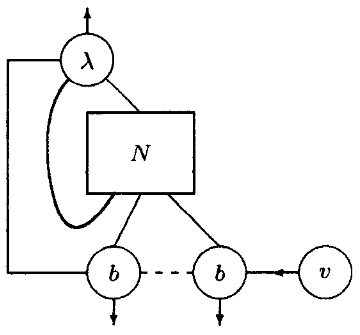


Hence we have introduced an agent for **tt**, an agent for **ff** and an infinite set of agents corresponding to the natural numbers n.

Variable. If t is a variable, then $\mathcal{T}(t)$ is translated simply into an edge.



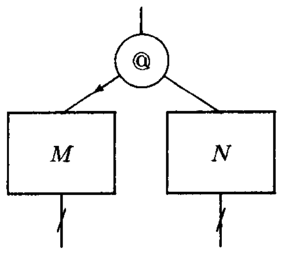
Abstraction. Let $\mathcal{T}(t) = N$, then $\mathcal{T}(\lambda x.t)$ is given by the following net, where we have introduced three different kinds of agent. First, an agent λ of arity 3, which corresponds to abstraction. The remaining two kinds of agents represent a list of the free variables of the term. We use the agent b, one for each free variable, and an agent v which represents the end of this list.



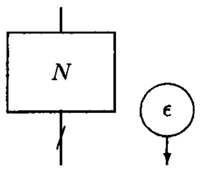
The key idea is that the coding contains a pointer to all the free variables of the abstraction; the body of the abstraction is encapsulated in a box structure.

We have assumed, without loss of generality, that the (unique) occurrence of the variable x is in the left-most position of N.

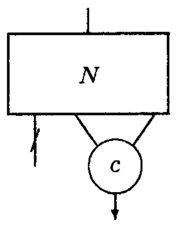
Application. Let $\mathcal{T}(t) = M$, and $\mathcal{T}(u) = N$, then $\mathcal{T}(tu)$ is given by the following net, where we have introduced an agent @ of arity 2 which corresponds to an application agent in the usual graph representations of the λ-calculus.



Erasing. Let $\mathcal{T}(t) = N$, then $\mathcal{T}([x = \_]t)$ is given by the following net using a new agent ϵ, of arity 0.

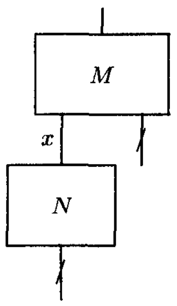


Duplication. Let $\mathcal{T}(t) = N$, then $\mathcal{T}([x = y, z]t)$ is given by the following net using a new agent c, of arity 2.



We have assumed, without loss of generality, that the (unique) occurrences of the variables y, z are in the rightmost positions of N.

Substitution. Let $\mathcal{T}(t) = M$, and $\mathcal{T}(u) = N$, then $\mathcal{T}(t[u/x])$ is given by the following net, where we simply connect the free edge x from the net M to the net N.



The following result shows that the structural equivalence introduced on terms becomes an equivalence in nets, which can be proved by a straightforward case analysis on the definition of $t \approx u$.

Proposition 4.1 *If $t \approx u$ then $\mathcal{T}(t) = \mathcal{T}(u)$.*

We thus see an advantage of using nets rather than λ-terms.

Remark 4.2 *For readers familiar with the translations of λ-calculus into linear logic, we remark that we are using the "$D = !(D \multimap D)$" translation, but we have hidden (as much as possible) the linear logic decomposition by using hybrid agents wherever we could (the agent λ corresponds to $\invamp$ and promotion, and @ corresponds to $\otimes$ and dereliction). We feel that this approach leads to a representation that is much closer to standard graph representations of the λ-calculus, but we can still gain from the linear logic foundation for the dynamics of the system. The reader is referred to [20] for the corresponding system of interaction for linear logic proofs.*

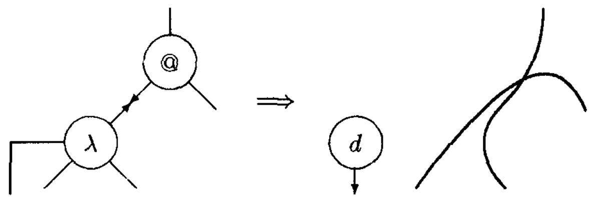
We end this section with several examples of the translation. In Figure 3 we show the nets corresponding to $\mathbf{I} = \lambda x.x$, $\mathbf{K} = \lambda x.\lambda y.x$ and the function twice $\mathbf{2} = \lambda f.\lambda x.f(fx)$.

5 Dynamics

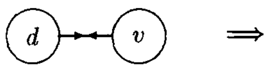
In this section we give the interaction rules for our λ-evaluator. We begin by giving the *weak* system, then go on to show how we can improve to rewriting process by giving the encoding of strong reduction. We show how each rewrite rule $\rightsquigarrow_w$ in the calculus is implemented in the interaction system, which is sufficient to show that the encoding is correct.

First, we note that 4 of the reduction rules are implicit. The translation of the rule $x[v/x] \rightsquigarrow_w v$ is implicit in this system of interaction, since $\mathcal{T}(x[v/x]) = \mathcal{T}(v)$, thus no interaction rules are required. The rules for pushing a substitution through an application: $(tu)[v/x] \rightsquigarrow_w (t[v/x])u$ and $(tu)[v/x] \rightsquigarrow_w t(u[v/x])$ are also implicit. The last rule $(t[w/y])[v/x] \rightsquigarrow_w t[w[v/x]/y]$ is equally implicit. We leave the reader to check that the translation of both sides of the rules yield the same interaction net. We therefore only need to encode the remaining 4 rewrite rules.

The first interaction rule is the linear part of β-reduction. This single rule captures the notion of connecting the body of the abstraction to the root, and the argument to the variable occurrences. A new agent d is introduced which serves to erase the box structure used in the translation, which will be explained in the following.

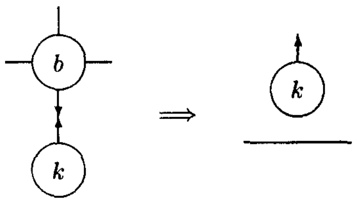


The second rule shows that if the term t was a closed λ-term in the above β-reduction, then the d agent introduced simply erases the v agent which marks the empty list.

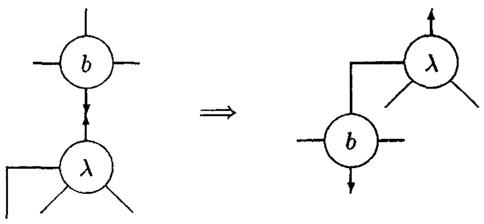


Thus if $\mathsf{fv}(t) = \{x\}$, then clearly we have $\mathcal{T}((\lambda x.t)u) \Longrightarrow^* \mathcal{T}(t[u/x])$, thus correctly implementing the first rule for $\rightsquigarrow_w$.

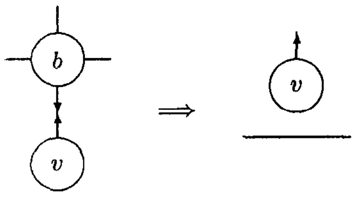
The next three rules are for the substitution of a value v through an abstraction: $(\lambda y.t)[v/x] \rightsquigarrow_w \lambda y.t[v/x]$. The first allows a base value ($k \in \{n, \mathbf{tt}, \mathbf{ff}\}$) to be moved inside an abstraction, corresponding to $(\lambda y.t)[k/x] \rightsquigarrow_w \lambda y.t[k/x]$, in a single interaction:

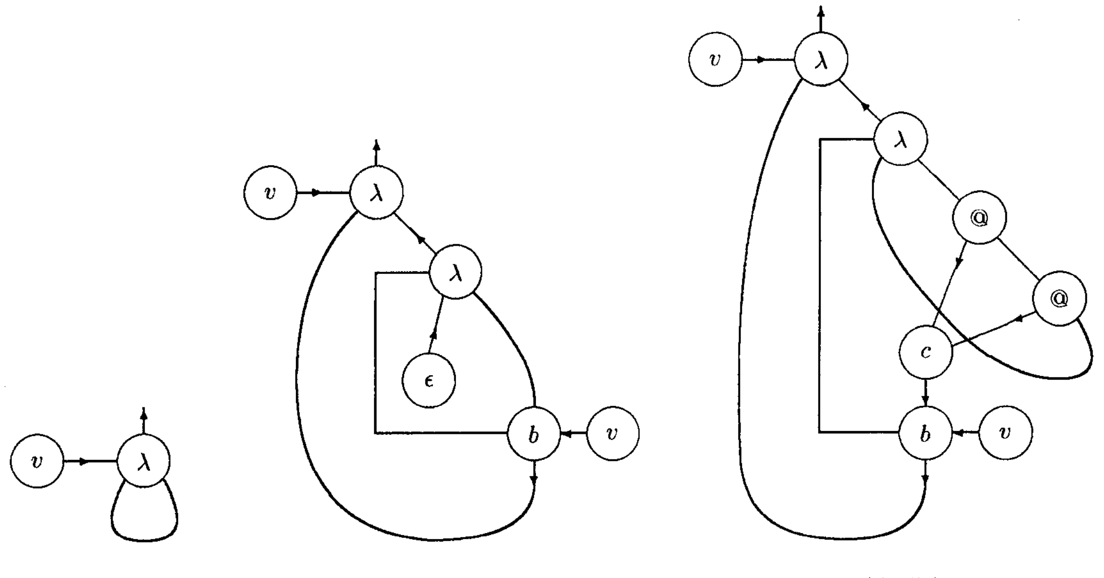


For abstraction values, we require two interactions to complete the substitution. The first interaction rule almost performs the substitution $(\lambda x.t)[(\lambda y.u)/z] \rightsquigarrow_w \lambda x.t[(\lambda y.u)/z]$.



Since the abstraction $(\lambda y.u)$ has no free variables, then the reduction successfully completes with the following interaction which erases the b agent corresponding to the free variable z.



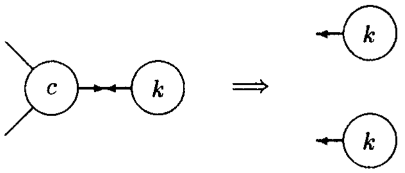


(a) $\mathcal{T}(\mathbf{I})$ (b) $\mathcal{T}(\mathbf{K})$ (c) $\mathcal{T}(\mathbf{2})$

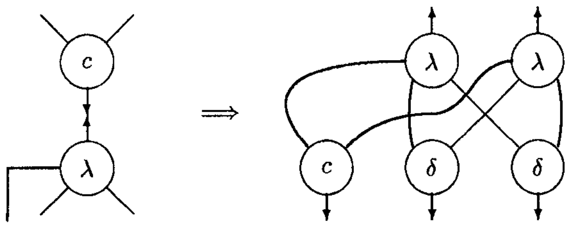
Figure 3: Example nets

Thus substitution of an abstraction value is translated into two interactions.

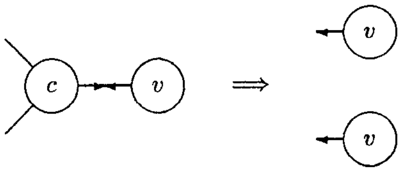
Thus far we have only been concerned with linear reductions (without copying or erasing). The following interaction rules shows how a w-nf with no free variables can be duplicated, corresponding to the term reduction $([x = y, z]t)[v/x] \rightsquigarrow_w t[v/y][v/z]$. The first rule shows that the copying agent can duplicate base values k:



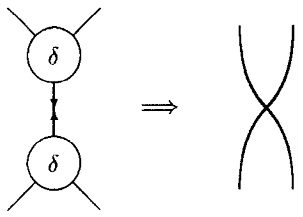
The next rule is when a copying agent meets an abstraction. We copy the λ, and propagate δ agents inside the body of the abstraction for which we show the dynamics later. We also propagate the c agent along the list of free variables.



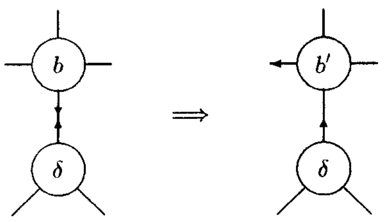
If the term is closed, then the following reduction duplicates the end of list agent.



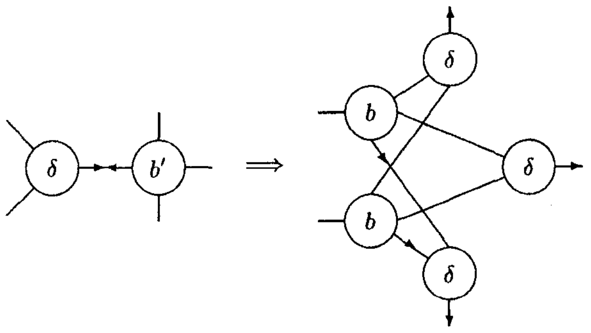
Duplication. We next give the dynamics of the rules for δ which complete the duplication of a term. There are three rules, and one rule schema which captures all the other cases. The first shows the special case when two δ agents meet and simply cancel each other out. This indicates that the duplication process is complete.



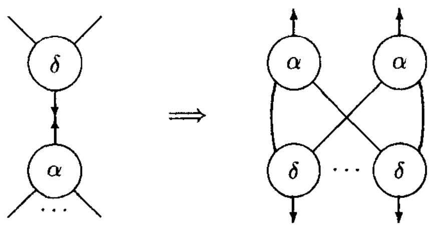
The second rule performs a blocking of the duplication process. The correctness of this system of interaction relies heavily on the principle that all δ agents are well-balanced inside the body of an abstraction during the duplication process, thus we should not allow δ agents to enter the body in this way. The blocking is achieved by introducing a new agent b' which will allow other interactions to complete, as we shall see below.



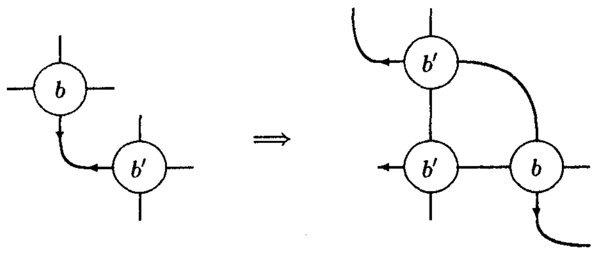
The third rule allows the list of free variable agents to be duplicated by δ agents. A δ agent is propagated along the list of free variables, inside the body of the abstraction, and also along the free variable edge. In the process of duplication we also convert the b' agent back to a b agent, which will allow for further substitutions to be made.



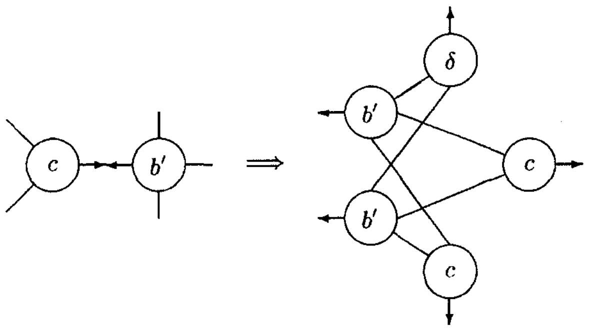
The final rule, or rather rule scheme, shows that all the remaining interactions with δ simply duplicate the agent and propagate along all the auxiliary ports.



Next we show the remaining rules for interaction with the agent b' which basically allow many of the blocked reductions to complete. The following rule allows some substitutions to complete. Specifically, an abstraction which has free variables marked with b' can be pushed through another abstraction. We introduce another b' agent which corresponds to "lifting" (or shifting) in the terminology of explicit substitutions.

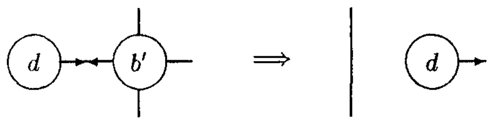


The next rule allows abstractions with free variables marked with b' to be copied:



Remark that a δ agent is introduced to duplicate the body of the abstraction, in the same way as we copied the abstraction agent.

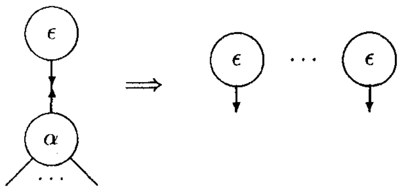
The final rule allows the b' agents to be erased by the d agent which opens the box of the abstraction.



The following result states that we can duplicate values correctly.

Proposition 5.1 *If N is a net in normal form, then there is a sequence of interactions that duplicates the net using δ agents connected to every free edge of the net N.*

Garbage collection. The final set of rules for the system concerns the process of erasing. We give a single rule scheme, which simply indicates that ϵ interacting with anything simply erases the agent, and then erases all the net connected to the auxiliary ports.



Remark that if the arity of α is 0, then the right-hand side of the rule is the empty net. One particular case of this is when α is an ϵ agent itself; in these cases the interaction marks the end of the erasing process. These rules provide the garbage collection mechanism for interaction nets.

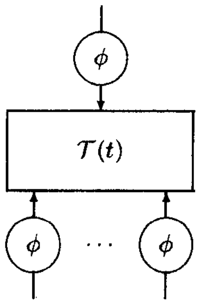
Proposition 5.2 *If N is a net in normal form, then there is a sequence of interactions that erases the net N completely using ϵ agents connected to every free edge of the net N.*

This completes the dynamics of the system of interaction, and as we have illustrated, correctly implements the term reduction $\rightsquigarrow_w$.

Proposition 5.3 *Let t be a program. If $t \rightsquigarrow_w^* u$, for some normal form u, then there is a finite sequence of interactions such that $\mathcal{T}(t) \Longrightarrow^* \mathcal{T}(u)$.*

5.1 Strong Reduction

Our evaluator of course does something on any term, not only on closed terms of ground type. However, the net will usually be in a form that makes is quite difficult to extract the corresponding term: for instance a net may be partially duplicated, then blocked waiting for a free variable to be bound. Here we will show a way of extending our weak reduction mechanism to allow for complete reductions to normal form. The intuition is that the reduction system stops short of normal form since there are free variables in the term, or an argument is required to complete reduction because the duplication process cannot complete until all terms are closed. To this end we introduce a new agent ϕ, and a set of interaction rules, which will be used to force reductions. It can be thought of as supplying dummy arguments to a term to allow reductions to complete (in a similar way that environment machines, for instance, can be supplied dummy arguments to force reduction to normal form). The translation $\mathcal{T}'(t)$ of a term t, to obtain full reduction, is now the following:



In Figure 4 we show the interaction rules for the new agent ϕ. This new agent essentially behaves like an identity agent—it simply passes over all agents without effect, as shown by the first interaction rule. However, there are three special cases. The first special case is when ϕ interacts with itself, in this case both instances just cancel each other out, indicating that the forcing is complete. The final two rules are the most important: one converts the b agent into a b' agent which will be used to allow other reductions to complete, and the last rule converts the b' agent back to a b agent. The intuition of the entire process is nothing other than allowing blocked computations to complete: when no further substitutions can be done, the free variables of an abstraction are allowed to be duplicated and erased. Using these additional agents, we can show that we get strong reduction in the λ-calculus.

Proposition 5.4 *Let t be any term. If $t \to_\beta^* u$, for some β-normal form u, then there is a finite sequence of interactions such that $\mathcal{T}'(t) \Longrightarrow^* \mathcal{T}(u)$.*

6 Extensions

In this section we show how to extend the ideas illustrated thus far so that we can deal with more realistic languages. In particular, we show how to extend to a language with conditionals and recursion. To keep the concepts as simple as possible, we will use a minimalist language which is rich enough to explain the basic ideas, but simplistic at the level of syntax. We shall use the syntax of the language **PCF** [23], which is well suited for this purpose. We have already introduced the constants at base type for **PCF** (natural numbers and booleans). In Figure 5 we show the remaining constructions of the language. The reduction rules for the constants of **PCF** are the following (also known as δ-rules):

pred $n+1$	$\rightsquigarrow$	n	**pred** 0	$\rightsquigarrow$	0
iszero $n+1$	$\rightsquigarrow$	**ff**	**iszero** 0	$\rightsquigarrow$	**tt**
cond tt $t\ u$	$\rightsquigarrow$	t	**cond ff** $t\ u$	$\rightsquigarrow$	u
$\mathbf{Y}t$	$\rightsquigarrow$	$t(\mathbf{Y}t)$	**succ** n	$\rightsquigarrow$	$n+1$

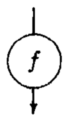
In addition to these reduction rules, we also give the rules for substitution with the new constants. We write f for all of the constant functions except conditional.

$$f(t)[v/x] \rightsquigarrow_w f(t[v/x])$$
$$\mathbf{cond}(b,t,u)[v/x] \rightsquigarrow_w \mathbf{cond}(b[v/x],t,u) \text{ if } x \in \mathsf{fv}(b)$$

Remark that there are *no* rules for pushing substitutions inside the branches of a conditional. The intuition is that since the variables are shared between the branches, then we do not want to duplicate any term which is just going to be erased later. Thus we just postpone all substitutions until the appropriate branch is known.

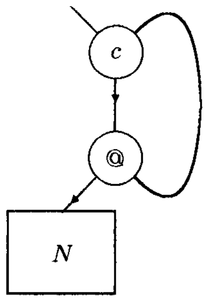
We will now extend the system of interaction to handle these additional features. Remark that we have already introduced the constants $k \in \{\mathbf{tt}, \mathbf{ff}, n\}$.

Arithmetic Functions. The constant functions $f \in \{\mathbf{succ}, \mathbf{pred}, \mathbf{iszero}\}$ are binary agents which we will draw as:

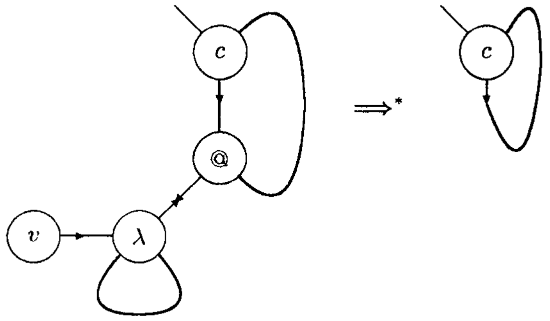


The intended meaning of the ports are: the result (at the top); and the principal port (at the bottom) where communication takes place with a constant of base type.

Recursion. We code recursion without needing to introduce any additional agents. To code $\mathbf{Y}(t)$, let $\mathcal{T}(t) = N$, then $\mathcal{T}(\mathbf{Y}t)$ is given by the following configuration:



This cyclic structure explicitly "ties the knot", and corresponds exactly to an encoding of recursion in graph reduction, see [22] for instance. We now get the coding of the **PCF** constant Ω as **YI** which is represented as the following cyclic structure, using two interactions.



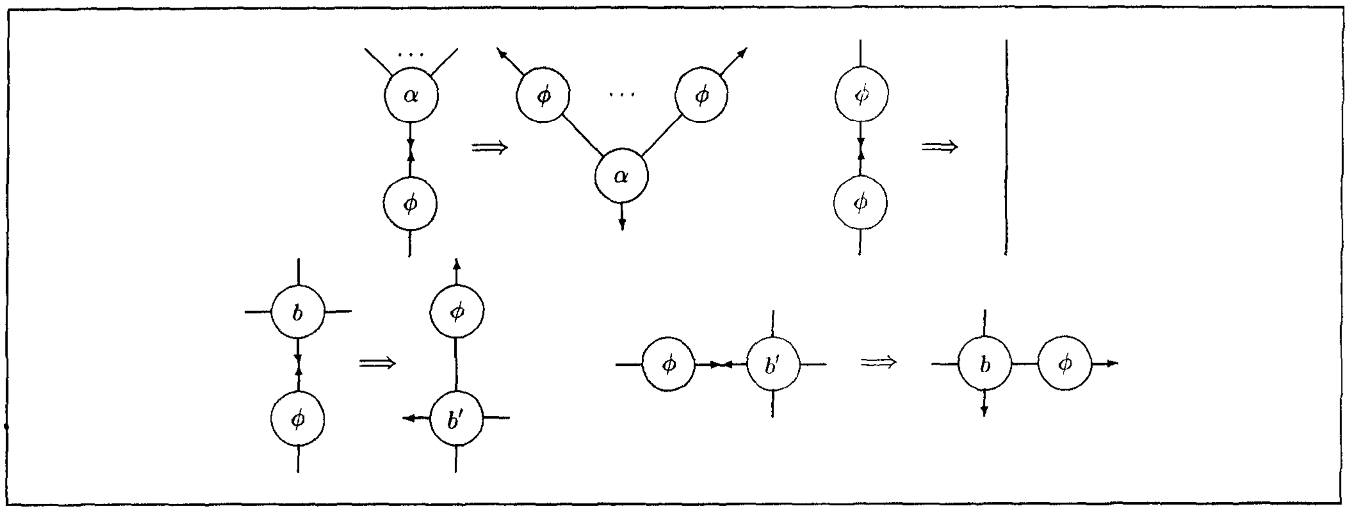


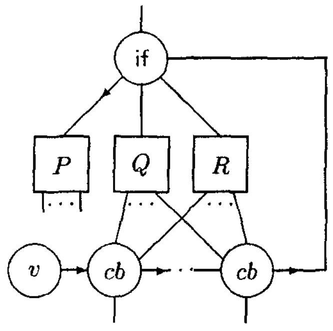
Figure 4: Forcing Reduction

Term	Type Constraint	Variable Constraint	Free Variables
succ(t) : **int**	t : **int**	–	fv(t)
pred(t) : **int**	t : **int**	–	fv(t)
iszero(t) : **bool**	t : **int**	–	fv(t)
cond$(b,t,u) : \tau$	b : **bool**, $t : \tau, u : \tau$	$\mathsf{fv}(t) = \mathsf{fv}(u), \mathsf{fv}(t) \cap \mathsf{fv}(b) = \varnothing$	$\mathsf{fv}(b) \cup \mathsf{fv}(t)$
Y$(t) : \sigma$	$t : \sigma \to \sigma$	–	fv(t)

Figure 5: Syntax: **PCF** Extensions

It is an interesting phenomenon that a non-terminating program in **PCF** terminates in this interaction system (there are no possible interactions, so the net is in normal form). However, the resulting net contains a cycle so now the problem has been pushed into the extraction of the result. This is reminiscent of "black hole" detection in functional languages.

Conditional. The coding of the conditional corresponds to the use of the *additives* in linear logic. This implies that we need some kind of "box", in the same spirit as we used for abstraction. We thus compile the conditional **cond**(b,t,u) in the following way. Let $\mathcal{T}(b) = P$, $\mathcal{T}(t) = Q$, $\mathcal{T}(u) = R$, then we build the following net, using three kinds of agent.



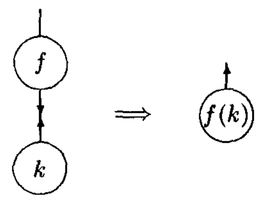
We use the agent if of arity 4, which combines together the boolean test, true and false branch. In addition, we have a list of variables occurring in the branches (cb) of the conditional, in a similar way as we used for the coding of abstractions. The idea is that since we will only need one of the branches, we will allow the free variables to be shared until we know which branch we require. This will become clear when we give the interaction rules for the conditional below.

Remark however, that there is no way that we can substitute inside the branches of the conditional, since there are no possible interactions with the box structure.

6.1 Dynamics for PCF

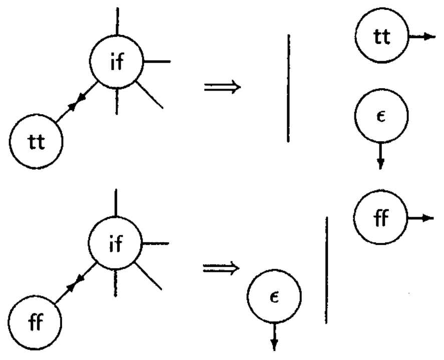
We now show how these additional agents interact with each other. We assume terms are well typed which means that there is no possibility for interactions of, for example, **succ** and tt. There are several new interaction rules that we need to add to this rewrite system, and we need to extend the rules for interactions with δ and ϵ.

For the arithmetic functions we overload the notation and write f for **succ, pred** and **iszero**, and k for a constant. The general scheme of the interaction rule is given by the following:

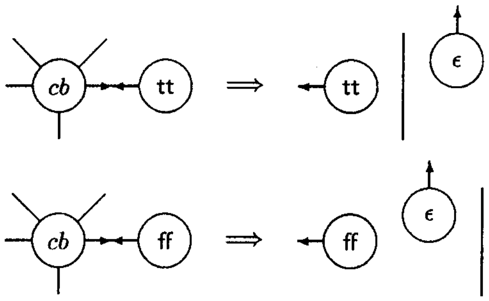


where $f(k)$ is the reduction rule for **PCF**. We can read this in two ways. The first is to allow an infinite number of rules, so that in fact we are giving interactions for **succ** n for all n. A second way is to think that there is just one agent for natural numbers with holds the value n, and the interaction with a constant actually applies the function to the agent. Either way gives the same result, but the latter is more intuitive from an implementation point of view.

The next two rules show the dynamics of the conditional. We synchronize on the boolean test, then connect the result of the conditional to the appropriate branch. An erasing agent is then introduced to garbage collect the unused branch. We propagate the **tt** and **ff** agents along the list of free variables of the conditional, which will be explained below.



The next two rules complete the dynamics of the conditional by connecting the free variables of the conditional to the appropriate branch.



These interactions work along the list of free variables until they reach its end, at which point they just cancel out the agent v.

We next need to show how these additional agents for **PCF** interact with the existing ones. However, this is very straightforward in that the interactions for k, f, and **cond** with ϵ and δ (for erasing and duplication) follow the rule scheme already given.

To complete this section, we mention issues related to the correctness of this implementation. Here, of course, the real problem is that we need a strategy for reducing interaction nets, since now we have the possibility of non-terminating computation, particularly when used in conjunction with a conditional. The unfortunate aspect is that non-terminating nets will become disconnected, but we have no automatic mechanism for detecting this. We refer the reader to [8] for a detailed study of an operational account of interaction nets, and possible strategies which overcome this problem.

7 Experimental Results

There is no point in giving a new method of reduction without actually demonstrating how useful it is. We have implemented this system of interaction, and we show a set of benchmark results. Church numerals provide an excellent way of generating large λ-terms, since application corresponds to exponentiation: $nm = m^n$. In the following table $n = \lambda f.\lambda x.f^n x$, and $I = \lambda x.x$. We apply Church numerals to II which is sufficient to force reduction to full normal form.

The following table gives a summary of a number of benchmark results that we have obtained. We compare our results with the optimal system of interaction of Gonthier, Abadi and Lévy. The first column shows the λ-term under test, and the next two columns give the total number of interactions for our algorithm (YALE) and for Gonthier, Abadi and Lévy's (GAL), respectively. Of the interactions performed, the number of redex families reduced is given in parenthesis (thus the value given for 'GAL' is the optimal one).

We also show the results for BOHM [5], which is an optimized version of Lamping's algorithm, but it is important to note that this is not an interaction net (the numbers written [·] correspond to the number of non-interaction rewrite rules applied during reduction).

Term	YALE	GAL	BOHM
2 2 I I	43(9)	204(9)	37[3](9)
2 2 2 I I	128(20)	789(16)	90[10](16)
3 I I	18(5)	75(5)	17[2](5)
3 3 I I	88(15)	649(15)	80[7](15)
3 2 2 I I	385(51)	7055(21)	157[27](21)
2 2 3 I I	214(31)	1750(19)	125[20](19)
4 4 I I	149(23)	3456(23)	136[13](23)
5 5 I I	226(33)	33971(33)	208[21](33)

These benchmark results indicate that, although the algorithm is sometimes far from optimal, the number of interactions, which we take as our measure of computation time, is always less than the one for optimal reduction. We also remark that if the term is linear (*i.e.*, built from the combinators **I**, **B**, and **C**) then the algorithm is always optimal, and moreover always does less work than any extant interaction net implementation of the λ-calculus, thus indicating that the overheads of this implementation are very small. We have not been able to compute a term which generates more work for our algorithm, since known examples where optimality is really useful explode all these evaluators (and others implementations of the λ-calculus). This supports the claim that realistic programs do not need all the power of optimality to be efficient.

As a final remark, we find it quite surprising that the magnitude of our best results corresponds very closely to that of BOHM. Maybe this suggests that these figures are representing more or less the minimum amount of work that is required to implement the λ-calculus. It remains for us to implement optimizations of our algorithm to see what can be gained.

8 Conclusions

We have presented a new algorithm for the implementation of the λ-calculus, based on a system of interaction nets. The design of this system comes directly from general observations about efficient strategies for implementing the cut-elimination procedure in linear logic.

Our experience with an implementation indicates that this system may provide a more realistic starting point for implementations of languages based on the λ-calculus using interaction nets rather than ones based on Lamping's. For most programs written in functional languages, the cases where we fail to be optimal, and where the optimal reducers do better, simply do not arise. It remains for us to extend these ideas to more realistic functional languages so that a detailed comparison of performance can be obtained.

Further work is underway to improve (optimize) the translations. One approach is to use a language where the programmer can explicitly declare linear terms which can avoid the use of the b agents, which are only there for non-linear terms, and cause most of the overheads in the reduction. For instance we could use the language Lilac [18]. There is also a lot of scope for optimizing the reduction system by adding additional rules which do not break the correctness of the system, but leave the interaction net framework.

Finally, we remark that the explicit substitution calculus that we introduced in this paper seems to enjoy a lot of interesting properties, and deserves further study.

Acknowledgements

I am most grateful to Maribel Fernández and Jorge Sousa Pinto for comments and careful proof reading of this paper.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[2] Samson Abramsky. The lazy λ-calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 4, pages 65–117. Addison Wesley, 1990.

[3] Samson Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993.

[4] Francisco Alberti. An abstract machine based on linear logic and explicit substitutions. Master's thesis, University of Birmingham, 1997.

[5] Andrea Asperti, Cecilia Giovannetti, and Andrea Naletto. The bologna optimal higher-order machine. *Journal of Functional Programming*, 6(6):763–810, November 1996.

[6] Henk 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.

[7] Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction (an analysis of Girard's execution formula). In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science (LICS'93)*, pages 296–306. IEEE Computer Society Press, 1993.

[8] Maribel Fernández and Ian Mackie. A calculus for interaction nets, 1998. Available from `http://lix.polytechnique.fr/~mackie`.

[9] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[10] Jean-Yves Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium 88*, Studies in Logic and the Foundations of Mathematics. North Holland Publishing Company, Amsterdam, August 1989.

[11] 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 (POPL'92)*, pages 15–26. ACM Press, January 1992.

[12] Sören Holmström. Linear functional programming. In T. Johnsson, Simon L. Peyton Jones, and K. Karlsson, editors, *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, pages 13–32, 1988.

[13] Yves Lafont. The Linear Abstract Machine. *Theoretical Computer Science*, 59(1,2):157–180, 1988.

[14] Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, January 1990.

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

[16] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 16–30. ACM Press, January 1990.

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

[18] Ian Mackie. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, October 1994.

[19] Ian Mackie. The geometry of interaction machine. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 198–208. ACM Press, January 1995.

[20] Ian Mackie. Linear logic *with* boxes. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS'98)*, pages 309–320. IEEE Computer Society Press, June 1998.

[21] Ian Mackie and Jorge Sousa Pinto. Compiling the λ-calculus into interaction combinators, January 1998. Available from `http://lix.polytechnique.fr/~mackie`.

[22] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International, 1987.

[23] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[24] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* PhD thesis, Oxford University, 1971.

[25] David Wakeling. *Linearity and Laziness.* PhD thesis, University of York, 1990.