Encoding an Object Calculus into Interaction Nets

Fabien Fleutot

Dept. of Computer Science
King’s College London
and
Université de Paris VII

Abstract

We propose an encoding of an object calculus into interaction nets in two stages. First, we make the calculus fully explicit, i.e. with explicit substitutions, duplications and erasures. Then, we use this explicit calculus to produce an interaction net encoding of objects.

Keywords: Object calculus, Interaction nets, Linearisation, Explicit substitution

1 Introduction

Concurrent programming is a challenge in computer science, partly due to the difficulty of thinking and expressing parallel algorithms for the programmer. Object oriented languages are often considered as a good paradigm to help with this task, because they help partitioning a program into relatively independent components.

Interaction nets [5] can be used, among others, as a low level graphic language into which encoding higher level languages [7]. Their evaluation scheme offers properties of a good sharing (they have been used to implement optimal β-reduction for the λ-calculus [6]), and natural concurrent evaluation capabilities.

1 Email: fabien@dcs.kcl.ac.uk
We present here an encoding of an object calculus into interaction nets. This encoding has been implemented together with a graphical interaction nets virtual machine, which can use different evaluation strategies, including a concurrent one.

Overview.

In the next section, we will present the calculus to be encoded, and extend it with explicit substitutions, duplication and erasing, so that its encoding will be as straightforward as possible. Section 3 briefly recalls interaction nets. Section 4 describes the encoding of the calculus into interaction nets.

2 Object calculus

There are quite a number of calculi which aim at capturing the essence of object oriented programming (e.g. [1,8]). Most of them rely on the idea that an object is a set of label×method pairs. A method is a term, in which a variable is bound to the whole object’s value. Then, objects support:

• Invocation of methods, i.e. returning the method term associated with a given label.
• Update of a method, i.e. changing the method associated with a given label.
• Some calculi support extension, i.e. addition of new label to method associations.

2.1 Fisher-Honsell-Mitchell calculus

This object calculus consists of the untyped λ-calculus, augmented with some basic object-oriented primitives:

• the empty object
• the addition of a labeled method to an object
• the update of a labeled method’s content
• message sending, i.e. calling an object’s method through its label

Formally, the grammar is:
\[
expr ::= x \quad \text{(Variable)} \\
| \lambda^x \expr \quad \text{(Lambda)} \\
| \expr \ expr \quad \text{(Apply)} \\
| \emptyset \quad \text{(Empty object)} \\
| \expr \triangleright (m, \expr) \quad \text{(Add method)} \\
| \expr \triangleleft (m, \expr) \quad \text{(Update method)} \\
| \expr.m \quad \text{(Method invocation)}
\]

For convenience, we may sometimes write objects as a list of methods between brackets, i.e. \([a = \lambda x.1, b = \lambda x.2]\) instead of \(\emptyset \triangleright (a, \lambda x.1) \triangleright (b, \lambda x.2)\).

The semantics of this language is as follows (the (Beta) rule uses implicit substitutions):

\[
(\lambda x.e)e' \leadsto e[x \leftarrow e'] \quad \text{(Beta)}
\]

\[
[(m_i = e_i)_{i \in [1..n]}] \triangleleft (m_j, e) \leadsto [(m_i = e_i)_{i \in [1..n] \setminus \{j\}}, m_j = e] \quad \text{(Update)}
\]

\[
[(m_i = e_i)_{i \in [1..n]}, m_j \mapsto e_j [(m_i = e_i)_{i \in [1..n]}] \quad \text{(Invocation)}
\]

2.2 Making the calculus explicit

Interaction nets only accept local rules, i.e. rules that rewrite two operators directly connected together. Moreover, being derived from linear logic [4], they need an explicit bookkeeping of all term deletion and duplication. Therefore, we will linearize the calculus and express it in terms of local rules, i.e. rules that rewrite two operators directly nested into one another. These explicit linearizers will depend on explicit substitutions, which will be introduced simultaneously. We will also ensure that the interface of terms (i.e. their sets of free variables) and their linearity are maintained during the whole reduction process. All of the rules will be given modulo \(\alpha\)-conversion.

2.2.1 Linearizers

A term is linear if each variable, either free or bound, appears exactly once in it (modulo \(\alpha\)-conversion). Obviously, not all terms are linear, and moreover, the subset of linear terms is not Turing-complete. To address this problem, we introduce two linearity operators: erasure \((E\{\}_-)_-\) and contraction \((C\cdot\{\}_-)_-\).

By inserting them at proper places in a term, and performing some variable renaming, any term can be made linear.
The former is intended to make a dummy usage of a variable, when a term has no use of this variable, but needs to have it as a free variable to be linear. For instance, \(\lambda x.\lambda y.x\) is not linear, but \(\lambda x.\lambda y.E\{y\}x\) has the same meaning and is linear.

The latter is to be used when a term needs to use a variable more than once, but cannot since it would break linearity. \(C^{x_0,x_1}\{e\}e'\) means the same as \(e'\) where \(x_0\) and \(x_1\) would have been replaced by \(e\). For instance, a linear equivalent of \((\lambda x.xx)\) is \((\lambda x.C^{x_0,x_1}\{x\}x_0\ x_1)\).

We give the syntax of the complete explicit calculus below. To the initial calculus, we added erasure, contraction, explicit substitution, and anti-erasure. The latter is to be used during explicit erasure, as we shall show below. The explicit substitution \(R^x\{e_1\}e_0\) reads “\(e_0\) in which \(x\) is replaced by \(e_1\)”.

\[
\text{expr ::= } x \quad \text{(Variable)} \\
| \lambda^x\text{expr} \quad \text{(Lambda)} \\
| \text{expr expr} \quad \text{(Apply)} \\
| \emptyset \quad \text{(Empty object)} \\
| \text{expr} \triangleright (m, \text{expr}) \quad \text{(Add method)} \\
| \text{expr} \leftarrow (m, \text{expr}) \quad \text{(Update method)} \\
| \text{expr}.m \quad \text{(Method invocation)} \\
| C^{x,}\{\text{expr}\}\text{expr} \quad \text{(Contraction)} \\
| E\{\text{expr}\}\text{expr} \quad \text{(erasure)} \\
| \tilde{E}^x\text{expr} \quad \text{(anti-erasure)} \\
| R^x\{\text{expr}\}\text{expr} \quad \text{(substitution)}
\]

Here, linearizers are defined as \(E\{\text{expr}\}\text{expr}\) and \(C^{x,}\{\text{expr}\}\text{expr}\) rather than \(E\{x\}\text{expr}\) and \(C^{x,}\{x\}\text{expr}\), which may seem surprising: linearizing a variable would be enough to turn any term into an explicitly linearized one; this is the approach usually taken, e.g. in [7]. However, in order to make the calculus reduction steps explicit, we will need the ability to linearize an arbitrary term, and not only a variable. This is also why we gave up the usual \(C^{x_0,x_1}e, E_x\) syntax: accumulation of terms in nested subscripts proved unreadable.

The set of free variables of a term is defined as follows:
Please notice that whenever a variable appears as a superscript of an operator, it is bound by this operator, hence the unusual notation $\lambda^x e$ instead of $\lambda x. e$.

Now, we can define $L(\_)$, the property of being linear, using $fv(\_)$:

\[
\begin{align*}
fv(\emptyset) &= \{\} \\
fv(x) &= \{x\} \\
fv(e.m) &= fv(e) \\
fv(e\ e') &= \\
fv(e \triangleright (m, e')) &= \\
fv(e \triangleleft (m, e')) &= \\
fv(\mathcal{E}\{e\}e') &= fv(e) \cup fv(e')
\end{align*}
\]
\[ L(x) \quad \text{is true} \]
\[ L(\lambda x e) \iff L(e) \land x \in fv(e) \]
\[ L(e e') \iff L(e) \land L(e') \land fv(e) \cap fv(e') = \{\} \]
\[ L(e \triangleright (m, e')) \iff L(e) \land L(e') \land fv(e) \capfv(e') = \{\} \]
\[ L(e \Leftarrow (m, e')) \iff L(e) \land L(e') \land fv(e) \cap fv(e') = \{\} \]
\[ L(e.m) \iff L(e) \]
\[ L(\mathcal{E}\{e\}e') \iff L(e) \land L(e') \land fv(e) \cap fv(e') = \{\} \]
\[ L(\mathcal{E}^x e) \iff L(e) \land x \in fv(e) \]
\[ L(C^{x_0..x_1}\{e\}e') \iff L(e) \land L(e') \land fv(e) \cap fv(e') = \{\} \land \{x_0, x_1\} \subseteq fv(e') \]
\[ L(\mathcal{R}^x \{e'\} e) \iff L(e) \land L(e') \land x \in fv(e) \]

2.2.2 Linearization function

We define \( T(\_ \_ \) a linearization function that transforms a term from the Fisher-Honsell-Mitchell calculus into a linear term. This function will use three auxiliary functions \( T_C, T_L \) and \( T_R \).

Let \( fv(a) \cap fv(b) = \{x_0 \ldots x_n\} \), and \( \{x'_0, x''_0 \ldots x'_n, x''_n\} \) be a set of fresh variables with respect to \( a \) and \( b \).

\[ T_C(a, b, t) = C^{x'_0..x''_0}\{x_0\} \cdots C^{x'_n..x''_n}\{x_n\}t \]
\[ T_L(a, b) = \mathcal{R}^{x_0}\{x'_0\} \cdots \mathcal{R}^{x_n}\{x'_n\}a \]
\[ T_R(a, b) = \mathcal{R}^{x_0}\{x''_0\} \cdots \mathcal{R}^{x_n}\{x''_n\}b \]
\[
T(\lambda^x e) = \lambda^x T(e) \quad x \in fv(e)
\]
\[
T(\lambda^x e) = \lambda^x E\{x\} T(e) \quad x \notin fv(e)
\]
\[
T(a \ b) = T_C(a, b, T_L(a, b) \ T_R(a, b))
\]
\[
T(\emptyset) = \emptyset
\]
\[
T(a \triangleright (m, b)) = T_C(a, b, T_L(a, b) \triangleright (m, T_R(a, b)))
\]
\[
T(a \leftarrow (m, b)) = T_C(a, b, T_L(a, b) \leftarrow (m, T_R(a, b)))
\]
\[
T(e.m) = T(e).m
\]

From now, all operations described in this paper will act exclusively on linear terms obtain by the \(T(-)\) linearization function, which implies that:

- When an expression has two sub-expression, their free variable sets are disjoint.
- Whenever a binder binds a variable, this variable is used exactly once in the binder’s sub-expression(s).

### 2.3 Reduction rules

#### 2.3.1 \(\beta\)-reduction

This is the obvious rule:

\[
(\lambda x.e) \ e' \leadsto \ R^x\{e'\} e \quad (AL)
\]

#### 2.3.2 Explicit substitutions

We now present the rules for propagating the substitutions created by \(T\) and the \(\beta\)-reduction rule. This is an extension of the linear \(\lambda\)-calculus with explicit substitutions defined in [3].
\[\mathcal{R}^x\{e\}x \leadsto e\] (RV)
\[\mathcal{R}^x\{e\}(e.m) \leadsto (\mathcal{R}^x\{e\})e\cdot m\] (RI)
\[\mathcal{R}^y\{e\}(\lambda^xe) \leadsto \lambda^x(\mathcal{R}^y\{e\})e\quad x \neq y\] (RL)
\[\mathcal{R}^x\{e''\}(e\ e') \leadsto (\mathcal{R}^x\{e''\})e\quad e' \in f\!v(e)\] (RAL)
\[\mathcal{R}^x\{e''\}(e\ e') \leadsto e\ (\mathcal{R}^x\{e''\})e'\quad e \in f\!v(e')\] (RAR)
\[\mathcal{R}^x\{e''\}(e \triangleright (m, e')) \leadsto (\mathcal{R}^x\{e''\})e \triangleright (m, e')\quad e \in f\!v(e)\] (ROL)
\[\mathcal{R}^x\{e''\}(e \triangleright (m, e')) \leadsto e \triangleright (m, \mathcal{R}^x\{e''\})e'\quad e \in f\!v(e')\] (RO_R)
\[\mathcal{R}^x\{e''\}(e \triangleleft (m, e')) \leadsto (\mathcal{R}^x\{e''\})e \triangleleft (m, e')\quad x \in f\!v(e)\] (RU_L)
\[\mathcal{R}^x\{e''\}(e \triangleleft (m, e')) \leadsto e \triangleleft (m, \mathcal{R}^x\{e''\})e'\quad x \in f\!v(e')\] (RU_R)
\[\mathcal{R}^x\{e''\}(\mathcal{E}\{e\})e' \leadsto \mathcal{E}\{\mathcal{R}^x\{e''\}\}e'\quad x \in f\!v(e)\] (RE_L)
\[\mathcal{R}^x\{e''\}(\mathcal{E}\{e\})e' \leadsto \mathcal{E}\{e\}(\mathcal{R}^x\{e''\})e'\quad x \in f\!v(e')\] (RE_R)
\[\mathcal{R}^x\{e''\}(\mathcal{C}^{x_0, x_1}\{e\})e' \leadsto \mathcal{C}^{x_0, x_1}\{\mathcal{R}^x\{e''\}\}e'\quad x \in f\!v(e)\] (RL)
\[\mathcal{R}^x\{e''\}(\mathcal{C}^{x_0, x_1}\{e\})e' \leadsto \mathcal{C}^{x_0, x_1}\{e\}(\mathcal{R}^x\{e''\})e'\quad x \in f\!v(e')\] (RC_R)

The fact that the terms are linear has some implications on the conditions applying to these rules. For instance, no term of the forms \(\mathcal{R}^x\{e\}\emptyset\), nor \(\mathcal{R}^y\{e\}x\), \(\mathcal{R}^x\{e\}(\lambda^xe)\) may appear. Moreover, among the rule pairs of the form \((\mathcal{R}_L)/(\mathcal{R}_R)\), one and exactly one of the conditions on \(f\!v(e)\) and \(f\!v(e')\) is fulfilled by a given term of the appropriate form. The rule (RL) might require some \(\alpha\)-conversion to make sure that \(x \notin f\!v(e')\).

**Property 1** Explicit substitution in our calculus terminates.

**Proof.** By a size criterion on the term on which substitution applies. \(\square\)

**Property 2** In a term which doesn’t contain any \(\mathcal{E}\), substitutions disappear by reduction.

**Proof.** By inspection of the rules. \(\square\)

Since we will show below that \(\mathcal{E}\) also disappears by reduction in terms generated by \(T\) and partially evaluated, we will then have that substitutions will not appear in full normal forms.
2.3.3 Explicit erasure

\[ \mathcal{E}\{(e\ e')\}e'' \rightsquigarrow \mathcal{E}\{e\}\mathcal{E}\{e'\}e'' \quad (EA) \]
\[ \mathcal{E}\{\emptyset\}e \rightsquigarrow e \quad (EZ) \]
\[ \mathcal{E}\{e \triangleright (m, e')\}e'' \rightsquigarrow \mathcal{E}\{e\}\mathcal{E}\{e'\}e'' \quad (EO) \]
\[ \mathcal{E}\{e \Leftarrow (m, e')\}e'' \rightsquigarrow \mathcal{E}\{e\}\mathcal{E}\{e'\}e'' \quad (EU) \]
\[ \mathcal{E}\{e.m\}e' \rightsquigarrow \mathcal{E}\{e\}e' \quad (EI) \]
\[ \mathcal{E}\{\lambda x. e\}e' \rightsquigarrow \tilde{E}^x\mathcal{E}\{e\}e' \quad (EL) \]
\[ \mathcal{E}\{C^{x_0,x_1}\{e\}e'\}e'' \rightsquigarrow \mathcal{E}\{e\}\tilde{E}^{x_0}\tilde{E}^{x_1}\mathcal{E}\{e'\}e'' \quad (EC) \]
\[ \tilde{E}^{x}\mathcal{E}\{x\}e \rightsquigarrow e \quad (EF_0) \]
\[ \tilde{E}^{x}\mathcal{E}\{y\}e \rightsquigarrow \mathcal{E}\{y\}\tilde{E}^{x}e \quad x \neq y \quad (EF_1) \]

Here, the five first rules are quite obvious: they simply make the erasure operator go down the structure of the term to erase.

However, it is slightly more complicated when it comes to erasing a binding operator: to preserve the linearity and the free variables of the term, the variable has to remain bound until it is actually erased. This is the purpose of the \( \tilde{E} \) operator: it binds the variable temporarily, to replace the binder deleted by \((EL)\) or \((EC)\), until there is nothing but this variable left (i.e. \( \mathcal{E}\{x\}e \)). Then, the binder and the dummy usage of the variable can cancel each other thanks to \((EF_0)\). Since there may be several variables bound, \((EF_1)\) allows \( \mathcal{E}\{\} \) to go through non matching \( \tilde{E} \), so that eventually all pairs that can be mutually cancelled will meet each other.

About the termination of erasure: the erasures go through all operators, except variables, substitutions, erasures and anti-erasures. Moreover substitutions eventually disappear in terms issued from \( T \), and there is no rule that allow any operator to get between \( \tilde{E} \) and the following \( \mathcal{E}\{\} \)s. Therefore, all erasures become a (possibly empty) series of \( \tilde{E} \) followed by a series of \( \mathcal{E}\{x\} \); thanks to linearity and \((EF_1)\), all \( \tilde{E} \) will eventually be deleted, therefore the only kind of erasures that will remain in a full normal form term are variable erasures \( \mathcal{E}\{x\} \).

2.3.4 Explicit contraction

As for erasure, most of contraction rules are structural rules, which just decompose a term to progressively rebuild two copies of it that respect linearity of the whole expression. We assume \( y_0, y_1, z + 0, z_1 \) are fresh variables for the
redexes in rules $CA, CO, CU, CI$.

$$C^{x_0,x_1}\{(e\ e')\}e'' \rightarrow C^{y_0,y_1}\{e\}C^{z_0,z_1}\{e'\} \rightarrow R^{x_1}\{(y_1\ z_1)\}R^{x_0}\{(y_0\ z_0)\}e'' \quad (CA)$$

$$C^{x_0,x_1}\{\emptyset\}e \rightarrow R^{x_1}\{\emptyset\}R^{x_0}\{\emptyset\}e \quad (CZ)$$

$$C^{x_0,x_1}\{e\downarrow (m, e')\}e'' \rightarrow C^{y_0,y_1}\{e\}C^{z_0,z_1}\{e'\} \rightarrow R^{x_1}\{y_1\downarrow (m, z_1)\}R^{x_0}\{y_0\downarrow (m, z_0)\}e'' \quad (CO)$$

$$C^{x_0,x_1}\{e\leftarrow (m, e')\}e'' \rightarrow C^{y_0,y_1}\{e\}C^{z_0,z_1}\{e'\} \rightarrow R^{x_1}\{y_0\leftarrow (m, z_1)\}R^{x_0}\{y_0\leftarrow (m, z_0)\}e'' \quad (CU)$$

$$C^{x_0,x_1}\{e.m\}e' \rightarrow C^{y_0,y_1}\{e\}R^{x_1}\{y_1.m\}R^{x_0}\{y_0.m\}e' \quad (CI)$$

$$C^{x_0,x_1}\{E\{e\}e'\}e'' \rightarrow E\{e\}C^{x_0,x_1}\{e'\}e'' \quad (CE)$$

$$C^{x_0,x_1}\{C^{y_0,y_1}\{e\}e'\}e'' \rightarrow C^{y_0,y_1}\{e\}C^{x_0,x_1}\{e'\}e'' \quad (CC)$$

The $(CC)$ rule might require some $\alpha$-conversion: in case $y_0$ or $y_1$ appear in $e''$, they would be bound by the transformation, and moreover, linearity would be broken.

One case is much more problematic: contraction of a $\lambda$-term. This is difficult to do in a purely local way, since during the duplication process, it is difficult to keep linearity. The naive way to implement it would have been:

$$C^{x_0,x_1}\{\lambda^x e\}e' \rightarrow C^{y_0,y_1}\{e\}R^{x_1}\{\lambda^x y_1\}R^{x_0}\{\lambda^x y_0\}e'$$

However, this does not work. Indeed, the $x$ which has to appear once in $e$ (since $\lambda^x e$ is linear) is now free in the whole expression, whereas it wasn’t. Moreover, we don’t have $x \in fv(y_0)$ nor $x \in fv(y_0)$, therefore $\lambda^x y_0$ and $\lambda^x y_1$ are not linear, and neither is the whole expression. Even much more contrived attempts of solutions, that introduce new operators to preserve linearity, fail when the variable initially bound by $\lambda$ is contracted in the body term.

Therefore, we give up with expressing contraction of $\lambda$-terms in a purely local way, and simply state:

$$C^{y_0,y_1}\{\lambda^x e\}e' \rightarrow C^{z_0,z_1}\{\lambda^x e\} \rightarrow \big( R^{y_0}\{\lambda^x z_0\} \cdots R^{z_0}\{z_0\} e \big) \ \big( R^{y_1}\{\lambda^x z_0\} \cdots R^{z_0}\{z_0\} e \big) \ e'$$

$$fv(e) = \{x, z_0 \cdots z_n\} \quad (CL)$$
Anyway, it turns out that this operation can be implemented into interaction nets rather easily, though in a way that resists to translation attempts into term rewriting.

As for termination of contraction, note that it goes through every term construct, except variables, anti-erasures and substitutions. Since substitutions and anti-erasures eventually disappear, the only kind of contractions that remain in a term in full normal form are contractions of variables, i.e. $C^{y_0,y_1}\{x\}e$.

2.3.5 Explicit update

Making update purely local is straightforward: one simply has to make the method searching explicit, and to remove the older method body with $E\{}$ once it has been found and updated:

\[
e \triangleright (m,e') \Leftrightarrow (m,e'') \leadsto E\{e'\}e \triangleright (m,e'') \quad (UO_0)
\]
\[
e \triangleright (m,e') \Leftrightarrow (m',e'') \leadsto e \Leftrightarrow (m',e'') \triangleright (m,e') \; m \neq m' \quad (UO_1)
\]

2.3.6 Invocation

As for update, invocation has to explicitly search through the list of variables. However, once the method with the matching label is found, the work is not over: this method has to be duplicated—this is done by a standard contraction operator—and one of the copies must be applied, with the whole object as a parameter. One solution would have been to somehow mark the head of the object, having a “probe” operator going down the list of methods, finding and duplicating the method body, bringing this copy back up to the marker, and doing the application. However, the “going back up” stage will easily be avoided in interaction nets, since the marker and the probe can stay connected together.

Therefore, we propose a version of the invocation that is not purely local in textual calculus, but whose translation into interaction nets will be:

\[
e.m \leadsto \text{Invo}(Cation(m,e)) \quad (I)
\]
\[
Cation(m,e \triangleright (m',e')) \leadsto Cation(m,e) \triangleright (m',e') \; m \neq m' \quad (SI_1)
\]
The invocation operator $m$ is split in two halves $Invo$ and $Cation$, the former being the marker, the later being the probe. $Cation$ goes down through the method list thanks to $(SI_1)$ until the matching method is found. Then, $(SI_0)$ creates a contraction to duplicate the method body, places one copy as the left part of the method application to the object, and the other back into the object.

### 2.4 Some results

**Lemma 2.1** If $e'$ has no anti-erasure, then $E\{e'\} \xRightarrow{*} E\{x_0\} \ldots E\{x_n\} e$, with $fv(e') = \{x_0, \ldots, x_n\}$.

**Proof.** by induction on $e'$, knowing that substitutions disappear. \hfill $\square$

**Corollary 2.2** If $fv(e') = \emptyset$ and $e'$ contains no anti-erasure, we have $E\{e'\} e \xRightarrow{*} e$.

**Lemma 2.3** If $fv(e') = \emptyset$, we have $C^{x_0, x_1} \{e'[x_0, x_n]\} e \xRightarrow{*} e[e', e']$.

**Proof.** By induction on $e'$. \hfill $\square$

**Theorem 2.4** (Simulation) Let $e$ be a closed term in the Fisher-Honsell-Mitchell calculus. If $e$ reduces to $e'$ by an outermost reduction step, then there is a reduction path from $T(e)$ to $T(e')$ in the linear calculus.

**Proof.** by inspection of the rules, knowing that $fv(e) = fv(T(e)) = \emptyset$:

- Update is straightforward.
- Invocation: the method extracted by $(I)$, $(SI_0)$ and $(SI_1)$ is closed, and a contraction of a closed term duplicates it (lemma 2.3).
- Beta-reduction: $(AL)$ generates an explicit substitution, which terminates (property 1). \hfill $\square$

**Theorem 2.5** A term obtained by full reduction of a term created by $T(\_)$ contains:

- no anti-erasure $\bar{E}e'e$,
- only erasures and contractions of variables $C^{x_0, x_1}\{x\} e$, 


• no explicit substitution $R^x\{e\}'e$.

**Proof.** We know that substitution disappears by reduction (property 1). The rest of the proof holds by induction, knowing that $\bar{E}$ are only generated by contractions and lambdas, and never by ($T\bot$) directly. The non trivial inductive cases are:

- $E\{\lambda x^n e\}'e$ where $x_n \in f\nu (e) = \{x_0..x_m\}$, which reduces as follows:
  
  $E\{\lambda x^n e\}'e \rightsquigarrow \bar{E}x^n E\{e\}'e' \rightsquigarrow^* \bar{E}x^n E\{x_0\}...E\{x_n\}e'$
  
  $\rightsquigarrow E\{x_0\}...E\{x_{n-1}\}\bar{E}x^n E\{x_n\}...E\{x_m\}e' \rightsquigarrow E\{x_0\}...E\{x_m\}e'$.

- $E\{C^{x_n,x_{n+1}}\{e\}'e\}$ where $x_n \in f\nu (e) = \{x_0..x_m\}$ will work similarly: $e$ will reduce to $E\{x_0\}...E\{x_m\}$, then the $E\{x_i\}; i < n$ will pass through $\bar{E}x^n E^{x_{n+1}}$, and finally the $\bar{E}$ will cancel themselves with their matching $E\{x_{n+1}\}$.

**Corollary 2.6** A read-back function can be defined, which suppresses variable contractions and erasures: $RB(C^{x_0,x_1}\{x\}e) = RB(e[x_0 \leftarrow x][x_1 \leftarrow x])$ and $RB(E\{x\}e) = RB(e)$. This read-back function transforms a fully reduced linear term back into an original Fisher-Honsell-Mitchell term.

**Conjecture 2.7** If a term terminates in Fisher-Honsell-Mitchell calculus, its translation through $T(\bot)$ into the linear calculus terminates, and the read-back of the fully reduced linear term is equal modulo $\alpha$-conversion to the fully reduced Fisher-Honsell-Mitchell term.

### 3 Interaction nets

#### 3.1 Background

Interaction nets [5] are a declarative, graph-based programming paradigm derived from linear logic [4], which supports concurrent evaluation.

An interaction net system consists of a graph representing a program, and a set of interaction rules that evaluate it. The graph’s vertexes are called agents, and are connected together by their ports. Each agent has exactly one principal port, and any number of auxiliary ports. Interaction rules are graph rewriting rules, whose left parts consist of two agents connected by (and only by) their respective principal ports. There is one interaction rule per pair of agent types. A agent’s principal port is marked by an arrow:

![Interaction net arrow](image)

Here is the general form of an interaction rule:
Since an interaction rule is determined by what principal port is connected to a given agent’s principal ports, an agent is involved in at most one possible interaction at a given time; therefore, the order in which interactions are reduced is not significant, and these interactions may be reduced in parallel.

3.2 Examples

Here are a couple of interaction nets examples. Integers are encoded with the agent $O$ to represent integer 0, and the successor operator $S$. The addition and the multiplication are encoded by agents $+$ and $\times$; they use $\delta$ as a duplicating agent, and $\epsilon$ as an erasing agent.

Addition:

\[
\begin{array}{c}
\alpha \\
\beta \\
\end{array}
\]

\[
\Rightarrow
\]

\[
\begin{array}{c}
S \\
+ \\
0 \\
\end{array}
\]

Multiplication:

\[
\begin{array}{c}
\times \\
\end{array}
\]

\[
\Rightarrow
\]

\[
\begin{array}{c}
\times \\
\epsilon \\
\end{array}
\]

\[
\Rightarrow
\]

\[
\begin{array}{c}
S \\
+ \\
0 \\
\end{array}
\]

\[
\begin{array}{c}
\times \\
\delta \\
\end{array}
\]

Duplication, Erasing:

\[
\begin{array}{c}
\epsilon \\
\alpha \\
\end{array}
\]

\[
\Rightarrow
\]

\[
\begin{array}{c}
\epsilon \\
\epsilon \\
\end{array}
\]

\[
\Rightarrow
\]

\[
\begin{array}{c}
\alpha \\
\delta \\
\end{array}
\]

\[
\Rightarrow
\]
### 3.3 Parameters

As a simple extension to interaction nets, we added parameters to agents. Namely, there is a denumerable set of parameters, typically the set of finite words, and each agent may be tagged by one parameter. The rewriting rules may be conditioned by parameters: a given rule may be allowed to apply only if the two agents to be transformed have the same parameter, or only if their parameters are different.

This generalization is actually not more expressive than the traditional interaction nets with an infinite denumerable set of agents together with their interaction rules. Let’s consider two parameterized agents $A[\_]$ and $B[\_]$, and $p_0, p_1 \ldots$ the denumerable, infinite set of parameters. The two conditioned rules that can apply on $A$ and $B$ are $(\forall i) A[p_i] \times B[p_i]$ and $(\forall p_i \neq p_j) A[p_i] \times B[p_j]$; they denote two denumerable set of rules on the denumerable agents set $\bigcup_i \{A[i], B[i]\}$.

### 4 Encoding into interaction nets

The explicit version of the object calculus can be encoded in a pretty straightforward way into interaction nets. The encoding proposed here is an extension of Ian Mackie’s encoding of $\lambda$-calculus [7].

#### 4.1 Static encoding of terms

##### 4.1.1 Variables

As explained above, in a linear closed term, variables appear exactly twice: the first one with a binder which “creates” them (here $\lambda x$ or $C_{x_0,x_1}\{\ldots\}$), and the second one when the variable is actually used in the term (possibly by $E\{\ldots\}$).

Variables will simply be encoded as a wire linking a variable declaration with this variable’s usage. With open variables in non-closed terms, one of the wire’s ends will be left disconnected.

A nice consequence of this encoding is that, by simply connecting a term to a variable, one performs the substitution of the variable by the connected term, which means that most of substitution rules will translate into identity in the interaction nets.

##### 4.1.2 Objects

The empty object is encoded with a dedicated agent $Z$ (Fig. 1). Other objects are built up using the $\_ \triangleright (\_ , \_ )$ operator. This operator has a main port, and
two auxiliary ports to be connected to its arguments, as shown on Fig. 2. To encode the method’s label, we use a family of $M[\ ]$ agents parameterized by the label name (which appears between brackets), instead of a single Addm agent.

It would have been possible to encode the label with a separate agent, connected to an Addm agent. However, this choice did not seem adequate, since:

- it would have complicated interaction rules
- there would still have been an infinite family of agents required to encode labels
- it appears that the retained encoding is visually easier to read.

4.1.3 Update

Update is encoded almost the same way as method addition. The corresponding node is just called $U[\ ]$ instead of $M[\ ]$, and its main port is turned down rather than up, since it is intended to modify the object below it. It can be seen on Fig. 3.
4.1.4 Invocation

Invocation is encoded with an \( I[ \ ] \) agent, parameterized with the method’s label. Its main port is turned down, where it has to find and copy the appropriate method’s body.

Fig. 4. Encoding of \( e.m \)

4.1.5 Contraction

Contraction is encoded with a \( C \) node. The main port is to be connected to the term to be duplicated, and the two auxiliary ports represent the two newly bound variables, which are to be connected to the place where they will be used.

Fig. 5. Encoding of \( C^{x0,x1}(e) e' \)

4.1.6 Erasure

Erasure is simply encoded as a \( E \) agent, with a single main port connected to the unused variable declaration. The \( \bar{E} \) is simply encoded as a single \( E \) agent as well.
4.1.7 Lambda terms

Lambda terms are more complex to encode. For reasons which shall be discussed later, the frontier of the net part encoding the lambda term has to be physically marked. Therefore, the Lam agent, which encodes lambda abstractions, has the following ports:

- the main port, which is the whole term’s interface
- an auxiliary port connected to the term inside the lambda
- an auxiliary port representing the newly bound variable, and connected to its usage into the inner term
- an auxiliary port connected to the term’s frontier.

The frontier aims to separate the term from outside, i.e. free variables declarations. This is done through B agents. These agents have four ports: two of them are used to chain them together, one of them in “inside” the term, connected to the variable usage, and the last one (which is the main port), is connected “outside” of the term, to the variable declaration, or to the inside border of another lambda term in which this one is nested. The two ends of this string are respectively connected to the Lam term, and to a terminating V term. This is shown on Fig. 7.

Fig. 6. Encoding of E{e}e’

Fig. 7. Encoding of λx e, with fv(e) = {x, y_1..y_n}
4.1.8 Application

Finally, the application is encoded with a three port App dedicated agent. Its main port is oriented to the left, where a lambda-term is expected to perform β-reduction.

Fig. 8. Encoding of (e e')

4.2 Dynamics

4.2.1 Implicit rules: (R_\_), (R_{\_L}), (R_{\_R}).

Let’s consider now how to encode textual rules into interaction net rules. Most of rules are about making substitutions travel through terms. However, this operation is really easy, and in fact most of the time implicit, into interaction nets. Indeed, finding the variable to be replaced is trivial, since the substitution is directly connected to it. Therefore, all of the (R_\_) substitution rules simply map to the identity into interaction nets, except (RL), because of the B border agents.

4.2.2 Substitution through lambda: (RL)

Substitution through lambda is not so simple, since terms have to go through the B agents marking the frontier. When a substitution is made, the corresponding free variable of the term disappears, and the corresponding B port should be removed as well. For simple values like \( \emptyset \), this is easily done as shown on Fig. 9. Objects will go through B method by method: Fig. 10 shows the rule that lets M nodes pass B; then then the terms under M[ ] will pass with their respective matching B passing rules.

However, when the substituting term contains free variables, they become free variables of the term in which the substitution occurs. The global behavior we need is the one described in Fig. 11. Basically, this operation consists of suppressing the B above the substituting term, duplicating all of the B string of this term, and including this newly created string in the containing term, instead of the removed B.
A way to do this is to create a B duplicating agent, let’s call it Db, and to make it travel through the substituting terms list. However, Db would reach the left ports of B agents, which are auxiliary ports, and so the two ports would not be able to interact. So what actually happens is that the substitution can only occur once the subterm which should enter into the superterm is closed.
There is one important exception to this inability to substitute with non-closed terms, when we apply the equivalent of the \((CL)\) duplication rule. In such a case, we shall explain that \(B\) agents are “turned” (by being momentarily changed into \(B_t\)) so as to face \(D_b\) and interact with it, as shown on Fig. 14. Finally, a rule between \(D_b\) and \(V\) is required to eliminate \(D_b\), as shown on Fig. 13.

What will happen when the term is not closed? Substitution will not terminate, until eventually the substituting term becomes closed, thanks to other substitutions. This is a key strategic feature of this encoding: it only allows to substitute variables with \textit{closed} terms. \textit{Almost always}, since there are some cases when the substitution has to occur right now, even if the term is not closed yet. In such cases, \(B\) agents will be transformed into \(B_t\) agents, whose main port is on the left. On these agents, \(D_b\) can act as shown on Fig. 14.
4.2.3 Contraction: (R...)

C agents duplicate the term they are interacting with, in quite a straightforward way, as show on Fig. 15.

But to the difficulties we had in the textual calculus with $\lambda$-terms correspond dual difficulties in interaction nets: the bound variable under the $\lambda$ would allow C agents to escape back up in the term, and there, possibly meet other C agents created from somewhere up in the term, causing an almost undermined mess. This is a classic problem in interaction nets encoding of the $\lambda$-calculus.

Here we avoid this problem by duplicating the inside of the $\lambda$-term with a dedicated D agent, which acts as the $\delta$ combinator of interaction combinators. The behavior of D the duplicator is shown on Fig. 20 and Fig. 21. The only point we still have to take care of, is to make sure that two D agents originating from two different, nested, duplication may not interact together and thus mess up both duplication processes. Due to B agents’ active ports being turned downwards, inner D cannot go out. However, outside D agents may break in. To forbid that, we give a special rule for D and B (Fig. 16): instead of duplicating B, D turns it into a $B_t$, so that D cannot enter until any contraction or substitution in which the inner $\lambda$ may be involved is terminated.

Therefore the rule of Fig. 17, which emits D agents inside the term, and a
C agent on the border. In case an outside duplication may interfere with this one, B agents are turned by the rule of Fig. 22, therefore allowing Fig. 14 and 18 to terminate substitution/copy before the outer copy occurs. In case no such copy from outside happens, the contraction only terminates once the \lambda-term is closed, with the rule of Fig. 19.

Here is an example of a term in which duplication has to terminate before the term is closed:

\[(\mathcal{C}^{k_0,k_1}\{\lambda^x \lambda^y \mathcal{E}\{y\}x\})_{k_0 1 2 3}\]

(which is the linearized form of \((k \ k \ 1 \ 2 \ 3)[k \leftarrow \lambda xy.x]\)). In the inner term \((\lambda^y \mathcal{E}\{y\}x)\), \(x\) is free; in the \(k_0\) copy, it will be replaced by \(k_1\), whereas in \(k_1\) it will eventually be replaced by 2; therefore, duplication of \((\lambda^y \mathcal{E}\{y\}x)\) has to be performed before substitutions.
4.2.4 Erasure: $(E_-)$

Erasure does not cause as many problems as contraction: it simply has to make every term connected to an $E$ disappear. This is done thanks to the rule scheme presented on Fig. 23, $\alpha$ being just any kind of agent. The erasure will terminate when $E$ agents meet unary agents (these unary agents can be $E$ agents themselves).

Please notice that $E$ encodes both $E\{\}$ and $\Tilde{E}$, and whenever two $E$ agent destroy each other, the one oriented upwards encodes a $\Tilde{E}^x$ whereas the one
Fig. 23. Interaction rule between $E$ and any other agent $\alpha$

oriented downwards encodes the dual $E\{x\}$.

4.2.5 $\beta$-reduction: (AL)

$\beta$-reduction is a pretty simple operation in explicit substitution systems: it just consists of replacing the application with the lambda term’s body, in which the bound variable is substituted with the application’s argument. In this encoding however, there is an additional operation to perform: the Bt string connected to Lam, if any, must be destroyed. This operation, called dereliction since it corresponds to the homonym box-opening operation of linear logic, is performed by the dedicated Der agent, as shown on Fig. 25 and 26. Therefore, the $\beta$-reduction will consist of two connections and the emission of a Der agent, as shown on Fig. 24.

Fig. 24. Interaction rule between App and Lam

Fig. 25. Interaction rule between Der and Bt
4.2.6 Update: \((UP_0), (UP_1)\)

Update operation directly follows the textual rule: crossing non-matching methods thanks to the rule of Fig. 27, and performing the method body replacement thanks to the one on Fig. 28. The \(E\) connected to the method body will destroy it, then connect some new \(E\) agents to every free variables, as \((UP0)\) textual rule does.

4.2.7 Invocation: \((I), (SI_0), (SI_1)\)

Invocation will follow the principle described above for the explicit textual calculus: it will create an agent at the top of the object, and send another agent to travel through and find the appropriate method; these two agents will be connected together, so that when the method body is found and copied, one of these copies can immediately be sent in front of the object. The top-marking agent is simply an application: indeed, invoking a method consist of applying the corresponding \(\lambda\)-term to the whole object. The searching agent, which is called \(Si[\ ]\) and is parameterized with the invoked label, will simply
pass through non-matching methods, as shown on Fig. 30. When it finds the right one, it will put a contraction to cause a copy, connect one of these copies to the original method carrier $M[\cdot]$, and the other one to the App agent waiting at the top of the object, as shown on Fig. 29.

![Fig. 29. Interaction rule between M[a] and Si[a]](image)

![Fig. 30. Interaction rule between M[a] and Si[b]](image)

The equivalent of (I) rule causes a problem: its direct translation would be the one shown on Fig. 31. Unfortunately, this rule contains an active pair on the right-hand side, which is not allowed for interaction rules. Moreover, this active pair will behave differently, depending on whether the first method does match or not. Therefore, the two cases have to be taken into account. By reducing the active pairs successively appearing by reducing this forbidden active pair, one finally obtain the rules on Fig. 32 and 33.

4.3 Results

Prooving that the encoding in the interaction nets simulates the original calculus is an ongoing work, which relies among others on the definition of a read-back function from the interaction nets to the calculus. However, we expect that the encoding of a term which terminates in Fisher-Honsell-Mitchell calculus terminates as well in the interaction nets encoding, and that some read-back function can be define to extract the corresponding Fisher-Honsell-Mitchell term from this interaction net final state.
5 Conclusion and future works

This encoding provided us two very low level object calculi, one textual and one as a graph reduction system. The implementation in interaction nets is interesting to work with, due to the strategies it implements: full reduction, WHNF, concurrent reduction... The implementation in interaction nets is quite deterministic in the strategies it adopts, e.g. substituting only closed
terms, and it makes sense to study the adaptation of these “interaction nets natural strategies” in the textual calculus, as well as trying to implement some “typically textual” strategies into the interaction nets. Moreover, adding imperative features to this purely functional language, as done for $\zeta$-calculus in [1], will make it much closer to usual object oriented languages.

References

[1] M. Abadi and L. Cardelli. A theory of objects. Monographs in Computer Science. Springer, 1996.

[2] M. Fernández and I. Mackie. A calculus for interaction nets. In Proceedings of PPDP’99, Paris, number 1702 in Lecture Notes in Computer Science. Springer, 1999.

[3] M. Fernández and I. Mackie. Closed reductions for the $\lambda$-calculus. In Computer Science Logic, Proceedings of CSL’99, number 1683 in Lecture Notes in Computer Science. Springer, 1999.

[4] J.-Y. Girard. Linear Logic. Theoretical Computer Science, 50(1):1–102, 1987.

[5] Y. Lafont. Interaction nets. In Proceedings, 17th ACM Symposium on Principles of Programming Languages, pages 95–108, 1990.

[6] J. Lamping. An algorithm for optimal lambda-calculus reductions. In Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages, pages 16–30. ACM, ACM Press, January 1990.

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

[8] J. Mitchell, F. Honsell, and K. Fisher. A lambda calculus of objects and method specialization. In Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science, pages 26–38. IEEE Computer Society Press, 1993.

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