Optimal Reductions in Interaction Systems*

Andrea Asperti

INRIA-Rocquencourt

Cosimo Laneve

Dip. di Informatica, Pisa

Abstract

Lamping's optimal graph reduction technique for the λ -calculus is generalized to a new class of higher order rewriting systems, called Interaction Systems. This provides a uniform description, in Lamping's style, of other basic computational constructs such as conditionals and recursion.

1 Introduction

At the end of 70's, Lévy fixed the theoretical performance of what should be considered as an optimal implementation of the λ -calculus. The optimal evaluator should always keep *shared* those redexes in a λ -expression that have a common origin (e.g. that are copies of the same redex). For a long time, no implementation achieved Lévy's performance (see [5] for a quick survey). Only recently Lamping and Kathail have solved independently the problem [14,9].

Unfortunately, both the theoretical studies by Lévy and Lamping-Kathail's evaluators take under consideration only the pure λ -calculus. This is a great limitation from the point of view of Computer Science, since essential operators as *conditionals* and *recursion* are left definitely out (in general, δ -rules: see [3], chapter 15).

The main aim of this paper is to provide optimal implementations for a much wider class of rewriting systems. In order to define the performance of the optimal evaluators of such a class we need to generalize the theoretical studies in [16]. This is particularly heavy. Here we prefer to survey rather briefly about this issue, focusing much more on the implementative aspects. See [2,15] for detailed presentations of the theory.

(*Interaction Systems*) The class of rewriting systems we are going to analyze in this paper became apparent to us by studying implementative issues. In particular, the remarkable settlement of Lamping's evaluator by Gonthier, Abadi and Lévy [7,8]. Roughly, these works establish a correspondence between Lamping's optimal implementation of λ -calculus and Girard's implementation of λ -calculus into *proof nets* [6], thus providing a (*linear*) logic foundation of Lamping's *control operators*. Such foundation made clear that λ -calculus could be split into two distinct parts: one, the *linear part*, defining the operators of the calculus (e.g. the arity, the "binding power", etc.) and the other taking care of the *management of resources* (or, better, of subexpressions). Therefore it seemed that Lamping-Gonthier's evaluation style could be *smoothly generalized* to a

*Supported in part by the ESPRIT Basic Research Project 6454 - CONFER

large class of systems just by replacing the “linear part” with an arbitrary linear (intuitionistic) calculus.

Let us explain the paradigm we have in mind. Take a generic set of logical operators. An intuitionistic presentation consists of a set of rules managing sequents of the shape $A_1, \dots, A_n \vdash B$. Roughly, these rules are partitioned into two sub-classes, according to the introduction of the logical operator happens on the left of the entail or on the right. This partition is reflected in the *language of the proofs*, too. Thus, there are proofs operators which correspond to introduction of a logical connective to the left of the entail (*destructors*) and to the right (*constructors*). As it is well-known from the Curry-Howard analogy, a rule can be simplified by *eliminating cuts* (interactions between destructors and constructors of a logical connective). These simplifications are usually formalized by means of *rewriting rules*, where left hand sides (lhs's, in brief) mention exactly the destructor/constructor pair yielding the cut. The calculi that result have a higher order nature since functional symbols own a “binding power” over their arguments (think about λ -abstraction).

The *linear* versions of intuitionistic calculi have already been studied in literature by Lafont [11] and named *Interaction Nets* (shortened into IN's). Therefore, we are just dropping the linearity constraint of IN's. In order to recall this strict analogy, we have named our systems Interaction Systems (IS's, in brief).

The higher order nature of IS's makes them a sub-class of Klop's Combinatory Reduction Systems [10,1] (CRS's). In particular, IS's are CRS's where the Curry-Howard analogy still “makes sense”. In [2] we have deepened the relations between Interaction Systems, Interaction Nets and CRS's.

(*Optimal evaluators*) The logical nature of Interaction Systems is at the basis of the optimal evaluators we shall provide. In particular, let L be the intuitionistic logic associated with an IS. We can define a “linear logic” version of L . That is, we may define a new system L_{LL} by just replacing the structural part of L with its “counterpart” in Linear Logic (i.e. giving to weakening and contraction a “logical status” by means the operators *why not* and *of course*). Then, L can be embedded into L_{LL} in essentially the same way that Intuitionistic Logic is embedded into (Intuitionistic) Linear Logic. So, by using the optimal implementation of *boxes* defined in [8], we get an optimal implementation of the original IS's.

A particular care is needed for implementing rewriting rules, since right hand sides (shortened into rhs's) must be “linearized” w.r.t. the metavariables. This is not a problem for the λ -calculus because the two metavariables occur exactly once in the rhs of the β -reduction. But, in general, this is false. For instance, as in the conditionals, a metavariable can be erased by the rule (it does not appear in the rhs). In order to preserve the *locality* of the (graph) rewriting, we must attach at the positive port of the metavariable a node performing the erasing. The rule defining the recursion operator duplicates a metavariable. Obviously, duplication invalidates optimality, since redexes internal to the expression represented by the metavariable are duplicated, too. Thus we must keep shared the duplicated expressions. The linear form of the rhs will be yielded by writing them in a substitution-free shape (by means of pseudo-operations of abstraction and application) and partially evaluating it.

We emphasize that the rewriting system describing the optimal implementation of an IS is

an Interaction Net. That is, we are providing (optimal) implementations of IS's into IN's, thus eliminating variables from symbolic computations. This generalizes recent works of Burroni [4] and Lafont [12] to higher order rewriting systems.

As much as possible we shall avoid any reference to Interaction Nets and Linear Logic. However [8] is a prerequisite for a deep understanding of the evaluators.

2 Interaction Systems

An Interaction System is defined by a *signature* Σ and a set of *rewriting rules* R .

(The signature) The signature Σ consists of a denumerable set of *variables* and a set of *forms*. The set of forms is partitioned into two disjoint sets Σ^+ and Σ^- , representing *constructors* (ranged over by c) and *destructors* (ranged over by d). Variables will be ranged over by $x, y, z, \dots$, possibly indexed. Vectors of variables will be denoted by $\vec{x}_i$ where i is the length of the vector (often omitted).

Each form can work as a binder. This means that in the arity of the form we must specify not only the number of arguments, but also, for each argument, the number of variables it is supposed to bind. Thus, the *arity* of a form f , is a finite (possibly empty) sequence of naturals (and not, as usual, a natural!). Moreover, we have the constraint that the arity of every destructor $d \in \Sigma^-$ has a leading 0 (i.e., it cannot bind over its first argument). The reason for this restriction is that, in Lafont's notation [11], at the first argument we find the *principal port* of the destructor, that is the (unique) port where we will have interaction.

Expressions, ranged over by $t, t_1, \dots$, are inductively generated by the two rules below:

- a. every variable is an expression;
- b. if f is a form of arity $k_1 \dots k_n$ and $t_1, \dots, t_n$ are expressions then $f(\vec{x}_{k_1}^1.t_1, \dots, \vec{x}_{k_n}^n.t_n)$ is an expression.

Free and bound occurrences of variables are defined in the obvious way. As usual, we will identify terms up to renaming of bound variables (α -conversion).

(The rewriting rules) Rewriting rules are described by using schemas or *metaexpressions*. A metaexpression is an expression built up also with *metavariables*, ranged over by $X, Y, \dots$, possibly indexed (see [1] for more details). Metaexpressions will be denoted by $H, H_1 \dots$.

A *rewriting rule* is a pair of metaexpressions, written $H_1 \rightarrow H_2$, where H_1 (the *left hand side* of the rule, lhs for short) has the following format

$$d(c(\vec{x}_{k_1}^1.X_1, \dots, \vec{x}_{k_m}^m.X_m), \dots, \vec{x}_{k_n}^n.X_n)$$

and $i \neq j$ implies $X_i \neq X_j$ (*left linearity*). The arity of d is $0k_{m+1} \dots k_n$ and that of c is $k_1 \dots k_m$.

The *right hand side* H_2 (rhs, for short) is every *closed* metaexpression, whose metavariables are already in the lhs and built up by the following syntax

$$H ::= x \mid f(\vec{x}_{a_1}^1.H_1, \dots, \vec{x}_{a_j}^j.H_j) \mid X_i[\vec{H}_1/x_1^1, \dots, \vec{H}_{k_i}/x_{k_i}^{k_i}]$$

The expression $X\{^{H_1}/x_1, \dots, ^{H_n}/x_n\}$ denotes a meta-operation of substitution, as in the λ -calculus.

Finally, the set of rewriting rules must be *non-ambiguous*, i.e. there exists at most one rewriting rule for every pair **d-c**.

Example 2.1 (The λ -calculus) We have a destructor $@$ of arity 00, and a constructor λ of arity 1. The only rewriting rule is β -reduction:

$$@(\lambda(x.X), Y) \rightarrow X\{^Y/x\}.$$

■

Example 2.2 Interesting Interaction Systems can be defined by enriching the λ -calculus with “ δ -rules”. For instance, it is possible to extend the λ -calculus with an *if-then-else* operator $\natural$. Let **T** and **F** be two (new) constructors of arity ε , and $\natural$ be a destructor of arity 000. The rules for the conditional are described by the following interactions between $\natural$ and **T** or **F**:

$$\natural(\mathbf{T}, X, Y) \rightarrow X$$

$$\natural(\mathbf{F}, X, Y) \rightarrow Y$$

Note that we do not have any interaction between $\natural$ and λ .

A less trivial example is provided by the recursion operator μ . We may describe it as a destructor of arity 0, interacting with λ as follows:

$$\mu(\lambda(x.X)) \rightarrow X[\mu(\lambda(y.X[y/x]))/x]$$

Note that the μ and the λ in the rhs have nothing to do with those in the lhs.

■

2.1 Graphical representations

Expressions of Interaction Systems have graphical representations that are reminiscent of Lafont’s Interaction Nets. In particular, a form **f** of arity $k_1 \dots k_n$ is represented as a node of name **f** with $1 + \sum_{i=1}^n p_i$ ports (edges); $p_i = k_i + 1$ is the i -th *partition* of **f**. The only port which does not belong to a partition is drawn with an outgoing arrow, and it is called the *principal port* of the form **f**; the other entries are called *auxiliary ports* (see [11]). The i -th partition represents the connections between **f** and its i -th argument M . In particular, one connection is with the root of M , and k_i with the variables bound by **f**. Observe that bound variables correspond to ports of the form **f**. Thus our graphs are cyclic. If **f** is a constructor, the principal port is “at the root” of the form; if **f** is a destructor, its principal port is at its first argument (recall that no binding is possible over this argument). So, interaction between constructors and destructors takes place only at principal ports (local sequentiality).

The correspondence between ports, bound variables and body of the arguments is fixed once and for all for each form. This means that all ports of a given form should be suitably “marked” (for the sake of readability, we shall generally omit to do that). For example the graphical representation of $\mathbf{c}(\langle x_1, x_2 \rangle, \mathbf{d}(x_1), \langle y \rangle, \mathbf{g}(y, y))$ is illustrated in Figure 1.(a). Variables which are not bound will be depicted as dangling edges whose ends are labeled by the name of the variables. This is the case for the variables u and v in the λ -term $(\lambda x.(xu)(xv))(\lambda y.y)$ depicted in Figure 1.(b). We recall that, in this way, several edges may have a common end, due to the multiple occurrence of a free variable in an expression.

Figure 1: Graphical representations of expressions

The reader is referred to [2] for more details about the graphical representation. A lot of examples will be found in the following pages.

3 Labelling and the family relation

This section is devoted to the generalization of Lévy's labeling [16] from λ -calculus to arbitrary Interaction Systems. Labelling allows us to define the family relation, that is the kind of “optimal” sharing the implementation should support.

Definition 3.1 Let $L = \{a, b, \dots\}$ be a countable set of *atomic labels*. The set $\mathbf{L}$ of *labels*, ranged over by $\alpha, \beta, \dots$ is defined by the following rules:

$$L \mid \alpha\beta \mid (\alpha)_s$$

where $s \in \mathbb{N}^+$. The operation of concatenation $\alpha\beta$ will be assumed *associative*. ■

Although its formalization is a bit entangled, the idea behind the following labeling is very simple. When a redex is fired, a label α is captured between the destructor and the constructor; this is the label associated with the redex. Then, the rhs of the rewriting rule must be suitably “marked” with α , in order to keep a trace of the history of the creation. Moreover, since in the rhs we may introduce new forms, we must guarantee a property similar to the initial labeling, where all labels are different. This means that all links in the rhs must be marked with a different function of α (and we shall use sequences of integers, for this purpose).

Let us come to the formal definition. Every IS (Σ, R) can be turned in a free way into a (*labeled*) CRS (Σ^L, R^L) .

The forms of Σ^L are those in $\Sigma \cup \mathbf{L}$ with the arity of every $\alpha \in \mathbf{L}$ equal to 0. If

$$d(c(\vec{x}^1.X_1, \dots, \vec{x}^m.X_m), \dots, \vec{x}^n.X_n) \rightarrow H$$

is a rule in R then, for every i and for every i -tuple $\alpha_1, \dots, \alpha_i$, the rule

$$d(\alpha_1(\dots(\alpha_i(c(\vec{x}^1.X_1, \dots, \vec{x}^m.X_m)\cdot\cdot\cdot), \dots, \vec{x}^n.X_n) \rightarrow \mathcal{L}_{\alpha_1 \dots \alpha_i}^0(H)$$

belongs to R^L , where $\mathcal{L}_{\alpha}^s$ is defined over metaexpressions as follows

$$\begin{aligned}
\mathcal{L}_\alpha^s(x) &= (\alpha)_s(x) \\
\mathcal{L}_\alpha^s(f(\vec{x}^0, H_0, \dots, \vec{x}^m, H_m)) &= (\alpha)_s(f(\vec{x}^0, \mathcal{L}_\alpha^{s0}(H_0), \dots, \vec{x}^m, \mathcal{L}_\alpha^{sm}(H_m))) \\
\mathcal{L}_\alpha^s(X[H_0/x_0, \dots, H_n/x_n]) &= (\alpha)_s(X[\mathcal{L}_\alpha^{s0}(H_0)/x_0, \dots, \mathcal{L}_\alpha^{sn}(H_n)/x_n])
\end{aligned}$$

Example 3.2 Consider again the λ -calculus. The β -reduction $@(\lambda(x.X), Y) \rightarrow X[Y/x]$ gives rise, in the labeled version, to the following rules:

$$@(\alpha_1(\dots(\alpha_i(\lambda(x.X)\dots), Y) \rightarrow \mathcal{L}_\ell^0(X[Y/x]))$$

where $\ell = \alpha_1 \dots \alpha_i$. Note that, by definition, $\mathcal{L}_\ell^0(X[Y/x]) = (\ell)_0(X[(\ell)_{00}(Y)/x])$, therefore, by replacing ℓ_0 with $\bar{\ell}$ and ℓ_{00} with $\underline{\ell}$, we easily recognize Lévy's labeling. ■

Labeled expressions are depicted by the same standard as unlabeled ones, with the agreement to write labels besides edges connecting forms.

Let (Σ, R) be an IS and let (Σ^L, R^L) be the labeled CRS built in the way described above. Given a form f in Σ and an occurrence of it in a term t of Σ^L , we say that f has *label* $\alpha_1\alpha_2\dots\alpha_i$ if, in the syntactic tree of t , $\alpha_1\alpha_2\dots\alpha_i$ is the path towards the root which links f to the less outside form in Σ (or to the root). The *degree* of a redex u is the label of the constructor (i.e. the sequence of the labels between the pair of symbols d-c of the redex u).

Definition 3.3 Two redexes are *in a same family* if and only if their degrees are the same. ■

This approach to the notion of redex-family based on labels does not give much insights about the intuitions that are behind. There are other equivalent approaches, suggested by the case of λ -calculus [16,17]. The relations among them have been discussed in detail in [2,15].

4 Sharing graphs

Let us switch to the optimal implementation of IS's. As remarked in the Introduction, the aim is to share along derivations redexes that are in the same family. This is yielded by enriching graphical representation of expressions with control operators. Such operators are described in Figure 2 and must be considered as forms. This means that each node has a principal port of

Figure 2: The control operators

interaction. In Figure 2, the principal ports are always the lower edges.

To be formal, croissants brackets and fans are of two types, according to the polarity of their principal port. When the polarity of (the principal port of) the fan is negative then the node is called *fan-in*; when the polarity is positive, the fan is named *fan-out*. Fans are the main

nodes for implementing sharing. Let us see an example. In Figure 3, we have depicted the λ -expression $@(@ (M, N), @ (M, N'))$ through a sharing graph. The reader should be wondering why this graph represents the expression $@(@ (M, N), @ (M, N'))$. Actually the reading back of the sharing graphs is the foremost problem. So let us try to retrieve the original expression. As usual with graphical representations, we start from the root and try to recover the expression by traveling along the graph. Well, the first node we meet is $@$. So the original expression must have the form $@(X, Y)$. In order to recover X , we travel along the left branch of the $@$. During this trip, we find the $\star$ -branch of the fan-in. Keep this information in mind! Continue the trip (fan nodes are discarded by the read-back procedure: they do not appear in the syntactic expression). We reach a new node $@$, thus X has the form $@(X_1, X_2)$. The subexpression X_1 is found immediately in the left branch of the lower $@$: it is M . The expression X_2 is less obvious.

Figure 3: The sharing graph representation of a λ -expression

Traveling along the right branch of $@$ we reach a fan-out node. What branch we must choose? Well, remember that the last time we traversed a fan-in, we went into a $\star$ -branch. So we decide that the right answer has to be “the $\star$ -branch of the fan out”. This answer gives N as second argument of the lower $@$ and, notice, this agrees with the original expression. Now we must determinate Y . By traveling along the right branch of the higher $@$, we are obliged to enter the o -branch of the fan-in. Now, it should be clear that, when we shall reach the paired fan-out, we shall choose the o -branch. Therefore we yield the original expression.

Summing up, the distinction between branches of fan-nodes is essential to recover correctly the original λ -expression. However it is not powerful enough to solve any possible situation that could rise computing λ -expressions (e.g. matching correctly fan-ins and fan-outs). Therefore the introduction of other control operators as brackets and croissants (see [14]).

Brackets and croissants have a simple linear logic explanation: the firsts implement boxes, the latters are used for disintegrating boxes. This should not scare too much the reader since we are encoding intuitionistic systems into linear logic systems, thus we need to implement every mechanism of the latters. An alternative way for understanding the meaning of brackets and croissants is as *context transformers*, i.e. as “transformers” of the information that is needed to travel correctly along the sharing graph. For example, the presence of indexes besides the operators indicates the *depth* where the transformation takes place. We strongly recommend the reading of Section 6 or the corresponding section in [13,14].

The rules governing the interactions between control operators are drawn in Figure 4.

Figure 4: The control rules ($0 \leq i < j$)

The root and void nodes are respectively attached to the “important” and “unimportant” dangling edges of the graph. The important edges are the root and the free variables; the useless edges are those parts of the graph that have been discarded along the derivation. Indeed, the main purpose of erasing is to connect part of the graph that are erased by a contraction in order to preserve locality of the rewriting rules. We could add rules providing garbage collection (mainly involving the erasing node), but these do not eliminate all garbage and are not essential for correctness. So we omit them.

5 Implementation

Now we have all the preliminaries to provide the implementation of a generic IS. Actually, the evaluators are Interaction Nets. But we will not be fussy on this point. So, more generically, the optimal implementations are graph rewriting systems where the nodes are the control operators and the forms of the IS. The translation of IS-expressions is discussed in Subsection 5.1 below. Subsection 5.2 will deal with the rewriting rules.

5.1 The translation of expressions

Let us translate an arbitrary expression in sharing graph. We shall essentially follow [8], that is particularly clear.

For simplicity, we shall only consider the paradigmatic case when constructors and destructors have respectively arity 1 and 01. The other cases are easily derived. The translation is a function T , inductively defined in Figure 5. In this figure, the dangling edges in the bottom represent

Figure 5: The translation function T

generic free variables which are not bound by the forms c and d . In particular, the edge outgoing the 1-indexed fan-in, represents a free variable which is common to M and N . If some bound variable does not occur in the body, the corresponding port of the binder is a plug, as usually. The above translation is a clear consequence of the linear logic implementation in [8] and the use of a type $(!D) \multimap D$. For instance, a variable x represents an axiom whose negative edge has been derelicted. The composition of these two operations, according to [8], yields the sharing graph $T(x)$ of Figure 5. The introduction of a form needs a previous step in which the body is “protected” by $!$. This because, unlike λ -calculus, the body of a binder can be used in a non linear way by the rewriting rule. Here a peculiarity deserves to be emphasized: the effect of $!$ -introduction is not felt by bound variables. Actually, we are assuming the presence of a pseudo-binder in between the form and the body as drawn below:

So, when we perform the $!$ -introduction and the c -introduction, we yield:

That is as the constructor had no bound variable! Since the pseudo-binder is a ghost, it disappears, the bracket traverses it and we obtain the translation of Figure 5. The reasons for this stuff will be clear when we will implement rewriting rules of the IS. In that moment the ghost-binder will become apparent and will play an essential role in the partial evaluation providing the correct rhs's.

Notice also that the first argument of the destructor is not inside an of-course operation. This because destructors always behave linearly w.r.t. their principal port (it will be the outermost form inside its first argument that will “take care” of its arguments).

5.2 The translation of rewriting rules

Rewriting rules may be classified in three groups.

Control Rules. These are the 12 rules in Figure 4

Interfacing Rules. These are the rules which describe the interaction between control operators and forms of the syntax. Therefore, for every form f and control operator there is a rule schema. They are drawn in Figure 6. Notice the constraint $i > 0$. It follows from the interpretation of forms

Figure 6: The interfacing rules between control operators and forms ($i > 0$)

as “generalized fans”, namely forms properly combine the informations over their ports and “make connections” when they interact. This view completely fit with that in [7,8] where abstraction

and application are 0-indexed fans. Lamping has a much more “syntactical” interpretation of forms, thus needing a further control operator in order to fulfil to correctness [13,14].

Proper Rules. These rules describe the interactions between destructors and constructors of the IS’s. These are the only rules which are dependent from the particular Interaction System under investigation. We shall define the implementation of the rewriting rules in several steps.

Let R be an IS-rewriting rule. Foremost, we must put the rhs of R in a suitable form, in order to emphasize the “interface” between the new structure introduced in the rhs and the metavariables in its lhs.

(β -expansion) The first step is to β -expand all substitutions in the rhs. For this purpose we shall use two classes of *pseudo-forms*: abstraction Abs_n and application App_n , for $n \geq 0$. These pseudo-forms must be considered similar to other forms of the syntax. Abs_n is a constructor of arity n whilst App_n is a destructor of arity 0^{n+1} . As the reader suspects, they generalize λ -calculus abstraction and application. This is clear looking at the rewriting rule:

$$\text{App}_n(\text{Abs}_n(\langle x_1, \dots, x_n \rangle. X), Y_1, \dots, Y_n) \longrightarrow X[\overset{Y_1}{/}_{x_1}, \dots, \overset{Y_n}{/}_{x_n}]$$

In the following we shall always omit the index 1 in Abs_1 and App_1 .

So, the step of β -expansion consists simply of writing the rhs of the IS-rule without using the metaoperation of substitution and exploiting pseudo-forms Abs_n and App_n .

Example 5.1 Consider the rewriting rule for μ :

$$\mu(\lambda(\langle x \rangle. X)) \rightarrow X[\overset{\mu(\lambda(\langle y \rangle. X[y/z]))}{/}_{x}]$$

The β -expansion of the rhs gives the following term:

$$\text{App}(\text{Abs}(\langle x \rangle. X), \mu(\lambda(\langle y \rangle. \text{App}(\text{Abs}(\langle x \rangle. X), y)))) \quad \blacksquare$$

Note that, after the β -expansion, every metavariable is closed by a pseudo binders, i.e. they are into expressions of the shape $\text{Abs}_n(\vec{x}. X)$.

(linearization) The next step relies on *linearizing* the rhs w.r.t. the occurrences of expressions $\text{Abs}_n(\vec{x}. X)$. This is obtained by taking, for every metavariable X_i , a fresh *pseudo-variable* w_i and replacing every occurrence of $\text{Abs}_n(\vec{x}^i. X_i)$ with w_i . In this way we yield a metaexpression T . Next T must be closed w.r.t. metavariables w_i ’s. So, let $w_{i_1}, \dots, w_{i_k}$ be the metavariables *occurring* in T . The linearization step ends by giving the metaexpression

$$\text{App}_k(\text{Abs}_k(\langle w_{i_1}, \dots, w_{i_k} \rangle. T), \text{Abs}_{n_1}(\vec{x}^1. X_{i_1}), \dots, \text{Abs}_{n_k}(\vec{x}^k. X_{i_k}))$$

where n_j is the arity of the metavariable X_{i_j} .

Example 5.2 After the linearization step, the rhs of the recursion rule becomes:

$$\text{App}(\text{Abs}(\langle w \rangle. \text{App}(w, \mu(\lambda(\langle y \rangle. \text{App}(w, y))))), \text{Abs}(\langle x \rangle. X)) \quad \blacksquare$$

(translation) This step provides the graphical representation of the rhs of the rule. It is essential that, during the translation, we may consider each subexpression $\text{Abs}_n(\vec{x}.X)$ as a “black-box”. According to the linearization step, the expression that results will have the shape $\text{App}_k(M, \text{Abs}_{n_1}(\vec{x}^1.X_{i_1}), \dots, \text{Abs}_{n_k}(\vec{x}^k.X_{i_k}))$. The translation of this expression is drawn in Figure 7, where, for simplicity, we have assumed $n_i = 1$, for every i . Notice that metavariables

Figure 7: The translation step

are not inside boxes: they are already internal to boxes, due to the translation of expressions in Figure 5, and we use this box instead of building a new box around the argument of the application. In this way, no operation around the (unaccessible!) free variables of the instance of the metavariable can be performed.

Now it is possible to gain much more intuition about our translation in Figure 5. Let us see that the trick of the ghost-binder allows us to have a much efficient implementation (in terms of the amount of control operators that are used). Take an expression $\text{Abs}(\vec{x}.X)$. The 0-indexed bracket along the principal edge of the Abs in Figure 7 must be considered as the “lifting” of the box that surround the metavariable. This means that the effect of the outermost box in X along the output of X and the bound ports must be undone. This is easy for the output, since it is enough to introduce a closed 0-indexed bracket along the auxiliary negative port of the Abs. Let us see what happens for the bound edges. Suppose to give up the “ghost-binders” in the translation of Figure 5. Then, the following sequence Υ of control operators should face the bound ports: an open 0-indexed bracket, an open 1-indexed bracket and a closed 0-indexed bracket. In order to cancel such sequence we should introduce a “dual” sequence of control operators and prove that the closed 0-indexed bracket in Υ disappears due to the interaction with a 0-indexed bracket in X . It is clear that the resulting translation step is less efficient (and less obvious).

The partial evaluation can begin as soon as we generalize the translation function T to pseudo abstractions and pseudo applications. The usual (call by name) interpretation of Abs and App can be found in Figure 8.

Remark the absence of brackets around the argument of Abs: in ordinary β -reduction, there is a linear use of the body of the abstraction.

Figure 10: The graphical representation of the rhs of the rule firing μ - λ

Remark 5.4 The previous translation could (and must) be improved. Apart from studying optimization techniques for reducing the number of sharing operators, the translation should be *relativized* to the particular IS's under investigation. In particular, some operators of the syntax could make only a linear use of some of their arguments. For instance, this is the case of the λ -calculus, where the body of the abstraction is treated linearly in β -reduction. These linear arguments have a simpler translation, since there is no need to put them inside a “box”. Furthermore, remark that the decision that some operator f behaves linearly over an argument needs the examination of *all* the interaction rules involving f . ■

6 Correctness and optimality

The evaluators we have defined are correct and optimal w.r.t. the Interaction Systems they implement. We have not space to deepen these issues, so, here, we will hint briefly to some problems we have found. We refer the reader to [15] or to the full paper for a complete discussion.

Let G be a sharing graph representing an IS-expression t . Correctness relies on showing that performing a graph-rewriting $G \rightarrow G'$ does simulate a set of IS-rewritings $t \longrightarrow t'$ such that t' is the expression represented by G' . It is clear that determinating the expression represented by a sharing graph is a prerequisite for any arguing about correctness. This is the so called *read-back* problem. It is solved in [13,7,8] by labeling edges of the sharing graphs through *contexts* and interpreting control operators (and forms in [7,8]) as *contexts transformers*. Expressions matching the sharing graphs are thus “unfoldings” of the graphs where only *consistent paths* are considered, that is paths that behave well w.r.t. contexts.

Due to the generality of the shape of IS-rewriting rules, the notion of consistent path must be further refined w.r.t. that in the λ -calculus. Actually, a contraction can create new forms and new edges (thus new paths). So, in general, a consistent path traversing a proper redex can be disconnected after a proper rewriting, thus invalidating any property of consistency. Therefore we have restricted consistent paths in order to avoid crossings of proper redexes. As a consequence, the read-back procedure $\mathcal{R}$ is more involved w.r.t. that in [14].

Theorem 6.1 The implementation $\mathcal{T}$ is correct. That is, if N is a graph yielded by a derivation starting at $\mathcal{T}(t)$ then:

- (A) $N \longrightarrow N'$ implies $\mathcal{R}(N) \longrightarrow \mathcal{R}(N')$;
- (B) N in normal form implies that also $\mathcal{R}(N)$ is in normal form. ■

Optimality is proved by showing that, if two redexes yielded by a labeled derivation have the same label, then the read-back function identifies them, that is they have the same representation in the sharing graph. The proof goes along the same lines of that in [13].

Theorem 6.2 In every derivation no two redexes destructor-constructor have the same label. Therefore the graph implementation is optimal. ■

7 Conclusions

We have generalized Lamping's optimal graph-reduction technique [14] to Interaction Systems, thus covering all the basic constructs of a real programming language. The main point of IS's w.r.t. optimality, is that it is particularly simple to "interface" the forms of the syntax with Lamping-Gonthier's control operators.

An interesting problem remains open: the efficiency. That is the amount of control operators that are used in the general approach suggested in this chapter (in order to linearize rewriting rules, in particular). These control operators cause an heavy overhead in terms of the number of control and interfacing reductions, that jeopardize optimality. For instance, take λ -calculus. According to our technique, the usual β -reduction, that does not exploits any control operator in [14,7,8], is turned into a rule with two control operators in between the external context and the functional part and six control operators in between the functional part and the argument part. That is, we gain in generality but we loose in efficiency. We suggest, case by case, to improve the translation (see also Remark 5.4).

An alternative approach relies on extending the *bus notation* ([7]) to IS's. Indeed, this notation is particularly appealing for an actual implementation, since it enormously reduces the number of "mutual crossing" between control operators.

Recently, Burroni [4] and Lafont [12] have refined usual term rewriting systems by explicitly managing variables with control operators. The advantage is that symbolic computations can be rid of variables. In this respect, our (optimal) graph implementations seem the natural generalization of Burroni-Lafont's works to higher order rewriting systems. The richer set of control operators is justifiable for coping with situations created by the mechanisms of binding and substitution.

Acknowledgements: We would like to thank J.J.Lévy and G.Gonthier for the interesting discussions and their relevant suggestions. Cosimo Laneve would like to thank INRIA for its warm hospitality during his permanence in Spring 1992.

References

- [1] P. Aczel. A general church-rosser theorem. Draft, Manchester, 1978.
- [2] A. Asperti and C. Laneve. Interaction systems 1: The theory of optimal reductions. Technical report n. 1748, INRIA-Rocquencourt, Septembre 1992.
- [3] H.P. Barendregt. *The Lambda Calculus*. North-Holland, 1985.
- [4] A. Burrioni. Higher dimensional word problems. In *Category Theory and Computer Science*, volume 530 of *Lecture Notes in Computer Science*, pages 94 – 105. Springer-Verlag, 1991.
- [5] J. Field. On laziness and optimality in lambda interpreters: tools for specification and analysis. In *Proceedings 17th ACM Symposium on Principles of Programming Languages*, pages 1 – 15, 1990.
- [6] J. Y. Girard. Linear logic. *Theoretical Computer Science*, 50, 1986.
- [7] G. Gonthier, M. Abadi, and J.J. Lévy. The geometry of optimal lambda reduction. In *Proceedings 19th ACM Symposium on Principles of Programming Languages*, page ??, 1992.
- [8] G. Gonthier, M. Abadi, and J.J. Lévy. Linear logic without boxes. In *Proceedings 7th Annual Symposium on Logic in Computer Science*, page ??, 1992.
- [9] V. Kathail. *Optimal Interpreters for lambda-calculus based functional languages*. PhD thesis, MIT, 1990.
- [10] J. W. Klop. *Combinatory Reduction System*. PhD thesis, Mathematisch Centrum, Amsterdam, 1980.
- [11] Y. Lafont. Interaction nets. In *Proceedings 17th ACM Symposium on Principles of Programming Languages*, pages 95 – 108, 1990.
- [12] Y. Lafont. Penrose diagrams and 2-dimensional rewritings. In *LMS Symposium on Applications of Categories in Computer Science*. Cambridge University Press, 1992.
- [13] J. Lamping. An algorithm for optimal lambda calculus reductions. Technical report, Xerox PARC, 1989.
- [14] J. Lamping. An algorithm for optimal lambda calculus reductions. In *Proceedings 17th ACM Symposium on Principles of Programming Languages*, pages 16 – 30, 1990.
- [15] C. Laneve. *Optimality and Concurrency in Interaction Systems*. PhD thesis, Dip. Informatica, Università di Pisa, December 1992.
- [16] J.J. Lévy. *Réductions correctes et optimales dans le lambda calcul*. PhD thesis, Université Paris VII, 1978.
- [17] J.J. Lévy. Optimal reductions in the lambda-calculus. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry, Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159 – 191. Academic Press, 1980.