Interaction Systems II
The Practice of Optimal Reductions*

Andrea Asperti
Dip. di Matematica, Bologna

Cosimo Laneve
INRIA, Sophia Antipolis

Submitted on May 1993, Revised Version on July 1994

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. Interaction Systems provide a nice integration of the functional paradigm with a rich class of data structures (all inductive types), and some basic control flow constructs such as conditionals and (primitive or general) recursion. We describe a uniform and optimal implementation, in Lamping's style, for all these features. The paper is the natural continuation of [3], where we focused on the *theoretical* aspects of optimal reductions in Interaction Systems (family relation, labeling, extraction).

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 a same redex). For a long time, no implementation achieved Lévy's performance (see [9] for a quick survey). Only recently, Lamping and Kathail have independently solved the problem [19, 14].

Unfortunately, both Lévy's theoretical analysis and the reduction techniques proposed by Lamping and Kathail merely focus on the pure λ-calculus. This is a great limitation in view of an actual implementation, since we must eventually face the problem of extending the language with a wider range of data structures (integers, reals, records, lists, trees, ...) and some basic control flow constructs, such as conditionals, recursion and so on.

Studying the problem of extending Lamping's graph reduction technique to a more expressive language than pure λ-calculus, we discovered Interaction Systems (IS for short). At first glance, the introduction of this new class of rewriting systems, and the reason for restricting our analysis to them (IS are just a subclass of Klop's Combinatory Reduction Systems [15]), may look somewhat arbitrary. However, there is a very strong motivation behind our choice, that we would like to explain here.

In [11, 12], Gonthier, Abadi and Lévy have provided a remarkable simplification and a more satisfactory theoretical status for Lamping's graph reduction technique. Both results have been

*Partially Supported by the ESPRIT Basic Research Project 6454 - CONFER

obtained by exploiting a nice correspondence between Lamping's optimal implementation of λ-calculus and Girard's Geometry of Interaction for Linear Logic. The main idea behind Linear Logic, is that of making a clear distinction between the *logical* part of the calculus (dealing with the logical connectives) and the *structural* part (dealing with the management of hypotheses). These two different aspects of logical calculi have their obvious correspondent (via the Curry-Howard analogy) inside the λ-calculus. From one side we have the *linear* part (the linear λ-calculus), defining the syntactical operators (the arity, the "binding power", etc.) and their interaction; from the other side we have the *structural* part, taking care of the management (sharing) of resources (i.e. of subexpressions). This can be roughly summarized in the equation

$$\lambda\text{-calculus} = \text{linear } \lambda\text{-calculus} + \text{sharing.}$$

From [11, 12], it is clear that Lamping's sharing operators (fan, croissant and square bracket), provide a very abstract framework for an (optimal) implementation of the structural part, which is then interfaced to the linear (or logical) part of the calculus. Therefore it seemed that Lamping-Gonthier's evaluation style could be *smoothly generalized* to a larger class of systems by just replacing the linear λ-calculus with an arbitrary linear calculus. The only proviso to respect was to choose a linear calculus *with a strong logical foundation*, since otherwise we could immediately loose the logical analogy underlying all the previous discussion (in particular, the interface between the linear and the structural part seems to be critical). At this point, we had a natural (and up to our knowledge, unique) candidate for the linear calculus: Lafont's Interaction Nets [16].

Dropping the linearity constraint in Interaction Nets, we just obtain Interaction Systems. In the spirit of the equation above, we could write

$$\text{Interaction Systems} = \text{Interaction Nets} + \text{sharing.}$$

(this is not completely correct, since the nets we consider are *intuitionistic*, and not *classical* as in [16]; however the previous equation provides the main intuition).

Interaction Systems have been introduced in [3] (see also [20]). In the same paper we have also investigated the main theoretical aspects of optimal reductions. In particular, we have defined the notion of *redex family* via a suitable generalization of Lévy's labeling, and we have compared this definition with other well known approaches to the family relation (copy-relation, and extraction process [22]). We remark that, up to our knowledge, this has been the first attempt to generalize the theory of optimal reduction to a super-system of λ-calculus.

The technical preliminaries in [3] (that revealed some unexpected problems, especially with the copy-relation and the extraction process) was eventually aimed to provide the theoretical background for studying implementative issues (we could not avoid it: if we wish to prove that our implementation is optimal, we must eventually start with providing the formal notion of optimality!).

This work is the natural prosecution of [3]. In this paper we define the implementation of Interaction Systems in Lamping-Gonthier's style, and prove its correctness and optimality.

The structure of the paper is the following. In Section 2 we start with introducing the problem of optimal reductions, and optimal sharing in the λ-calculus. Then, we outline Lamping's graph reduction technique, and his approach to correctness (context semantics and read-back). Finally, we briefly discuss the relation between Lamping's implementation and Linear logic [11, 12], which provides the guideline for our extension to IS's.

In Section 3 we introduce Interaction Systems as a subclass of Klop's Combinatory Reduction Systems, providing several examples. We also discuss the *intuitionistic* nature of Interaction Systems. This is an essential feature of IS's; in particular, it immediately suggests the design of optimal evaluators, following the ideas in [12].

The formal definition of IS's is in Section 4. In the same section, we also introduce the generalization of Lévy's labeling to IS. Labeling is required to define the notion of redex family (two redex are in a same family if and only if their labels are equal), and thus the notion of optimality. The theoretical aspects of labeling, and its relation with other possible approaches to the notion of family, have been already discussed in [3], so we shall rapidly pass through this topic.

The implementation of IS's in Lamping-Gonthier's style is described in Section 7. Although the generalization of the implementation is not too difficult (for people confident with [12], at least), the correctness and optimality proofs are pretty entangled. The reason is that the corresponding proofs in [11] are based on particular properties of the λ-calculus (in particular, some aspects of the context semantics), which do not generalize to IS's. Roughly, we have been forced to extend Lamping's semantical approach (that looks more general, even if less elegant in the case of the λ-calculus), *but using the simplified set of operators defined in* [11]. In particular we have provided a true *read-back procedure*, which allow to recognize the expressions represented by the sharing graphs. We believe that our proof sheds new light on the correctness aspects of these implementation techniques, even in the restricted case of the λ-calculus.

2 Optimal Reduction

Intuitively, a reduction technique is *optimal* if it is able to profit of all the sharing expressed in the initial term, avoiding useless duplications. Looking for optimal reductions has a great practical interest, since the fact of loosing sharing can cause an exponential explosion of the time required for reducing the expression. Take, for instance, the term

$$\mathbf{M} = \mathtt{n}\,\mathtt{2}\,\mathbf{I}\,\mathbf{I}$$

where $\mathtt{n}$ and $\mathtt{2}$ are Church integers and $\mathbf{I}$ is the identity. Observe that the rightmost-innermost reduction strategy is obviously linear in n. However, in most of the "standard" implementations for functional languages (such as Combinatory Logic, Supercombinators – as G-Machine and TIM-machine – or Environments machines – as SECD, CAML, Krivine's machine and ZINC – or Continuation Passing Style – as SML –) the evaluation of M grows *exponentially* in n.

Of course, the explanation of this inefficiency should deserve a different analysis for each implementation. However, in this particular case, we may roughly identify the problem in the weak evaluation paradigm adopted by them: since we never reduce inside a lambda, we also loose the possibility of sharing those reductions.

In general, things are not so simple, and we cannot hope to optimize sharing by choosing a suitable evaluation strategy. In particular, Lévy has proved that there are terms, where *every* order of reduction would duplicate work. His favorite example is the following term ([21], p.15):

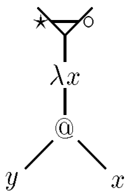
$$\mathbf{P} \;=\; (\lambda x.\, x\mathbf{I}x \ldots x)\, \lambda y.\, ((\lambda x.\, x \ldots x)\,(y\, a))$$

where a is some constant, and the two sequences of x have both length n. $\mathbf{P}$ has two redexes. If the outermost is reduced first, we eventually create n residuals of the inner one. Conversely, if we start reducing the innermost redex, n copies of $(y\,a)$ are created, and this will duplicate work later on, when **I** is passed as a parameter to y. In conclusion, any reduction strategy is at least linear in n whilst an optimal compiler should be able to get (a suitable representation of) the normal form of **P** in constant time!

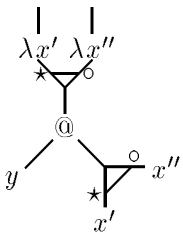
2.1 Lamping's solution

Consider Wadsworth's graph rewriting technique for the evaluation of functional expressions. Every time you have a redex $r = (\lambda x.M)N$ you should start with duplicating the functional part $F = \lambda x.M$. Indeed, F could be shared by other terms, and since we are going to instantiate it, we must eventually work on a new copy (see for instance [13]). If F contains a redex, this will be duplicated as well. The fact of reducing F first, does not help that much. The "redex" inside F could be only a "virtual" one (see [23, 5, 8] for the formal notion of "virtual" redex). Suppose for instance to have in F a subterm like (yP), where y is bound externally to r. The subterm (yP) is not a redex, but its duplication can be as useless and expensive as the duplication of an actual redex. Moreover, by the considerations in the previous section, the problem cannot be simply solved by the choice of a suitable reduction strategy.

Lamping [18, 19] proposed to duplicate F in a sort of "lazy" way, by propagating a duplication operator (a fan) along the graph structure of F, and stopping this propagation at suitable positions (typically, just before the applications in F). Suppose for instance to have the following configuration, where a duplication operator is applied to $M = \lambda x.(yx)$.

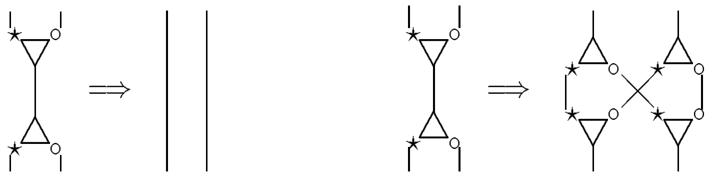


The duplication is done step by step, following the connected structure of M. The first syntactical form traversed by the fan is the binder for x. Note that the duplication of the binder implies the duplication of the bound variable (otherwise we would not know to which of the two binders we should refer). So we obtain the following term.



This term is in normal form. No one of the two fan operators can be propagated any further, until y will be instantiated to a functional term, and the corresponding redex will be fired.

Suppose to replace y by the identity. After firing the redex, we get a graph where two fans meet "face to face". Two reductions seem to be possible, now (but in this case, only the first one is correct):



By the left rule, the effacement of the two fans "completes" the duplication; this rule should be only applied when the two fans belong to the *same* "duplication process". In all the other cases, fans should "mutually cross" each other, according to the right rule, above.

The ambiguity whether applying the left or the right rule is solved by looking at the *sharing-level* of each fan (an integer, denoting the "duplication process" it belongs to). When two fans meet face to face, they will reduce according to the first rule above if they belong to a same level, and according to the second rule, if their levels are different.

Matters are furtherly complicated by the fact that levels may change dynamically during the computation. The menagement of levels requires the introduction of a suitable set of control operators (brackets and croissants) delimiting levels along the computation.

2.2 Context Semantics and Read-back

In Figure 1, we have depicted a typical example of graph (in normal form) which can be obtained as a result of a reduction in Lamping's system [19]. This term can be obtained, for instance, by reducing $\lambda f.\lambda x.(\lambda g.(g(gx))\lambda y.(fy))$. If the implementation is correct, the graph in Figure 1 should thus represent the term $\lambda f.\lambda x.(f(fx))$, but how could we retrieve this information from the graph?

This problem is known as *read-back*, and it is the foremost problem, in proving the correctness of the implementation.

Let us try to explain the general idea by "reading back" the graph in Figure 1. As usual with graphical representations, we start from the root and try to recover the expression by traveling along the graph. The two first nodes we meet are two λ. So far, so easy: the original expression must have the form $\lambda f.\lambda x.X$. The next form we meet is a fan node. In particular, we enter the fan from its $\star$-branch. This information must be recorded: it is a semantical component of a *context* associated with the path we are following in the term. Continue the trip, exiting from the principal port of the fan (fan nodes are discarded by the read-back procedure: they do not appear in the syntactic expression). We reach a @-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 the variable f. The expression X_2 is less obvious. Traveling along the right branch of @ we reach a fan-out node. What branch should we choose? We use the context semantic, to solve the problem. Remember that the last time we traversed a fan-in, we entered from a $\star$-branch (the $\star$ is the top level information in the current context). So we decide to follow the $\star$-branch of the fan out (at the same time, the $\star$ is discharged form the context). Traveling along this branch, we come back to the first fan. In this case, we enter from the $\circ$, which becomes the new top level information in the context. Next we find again the application, so X_2 must have the form $@(X_3, X_4)$. As above, we immediately recognize X_3 as the variable f. We have still to determine X_4. Since the top level control information is now $\circ$, this time we must follow the $\circ$-branch of the fan-out, finding the variable x.

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

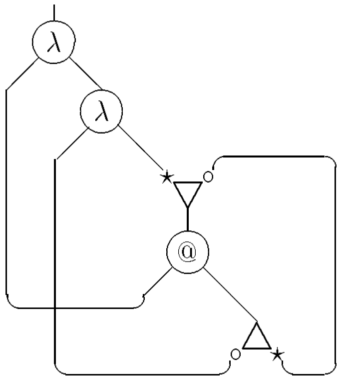


Figure 1: The sharing graph representation of a λ-expression

introduction of other control operators as brackets and croissants (see [19]). In particular, the *control information* must be structured in different levels. The semantical effect of control operators is that of creating, discharging, freezing and unfreezing levels.

2.3 Optimal Reduction and Linear Logic

There is a nice relation between Lamping's rewriting system and Girard's Linear Logic which has been pointed out in [11, 12]. The general idea is pretty simple: the "level" of each fan is related to the number of nested boxes in the Proof Net representation of the λ-term. Moreover, the control operators mark the extent of each box, implicitly defining their scope. From this point of view, we may consider Lamping's system as a local implementation of the (global) operation of duplication over boxes, in Linear Logic.

Exploiting this relation has led to a simpler rewriting system (only 12 rules), together with a much more elegant, *logical* foundation of Lamping's work.

In order to take advantage of this relation, any attempt of generalizing Lamping approach to more powerful rewriting systems, should presuppose some "logical nature" of the calculus. This is the main reason for restricting the analysis to Interaction Systems.

As a matter of fact, the intuitionistic flavour of Interaction Systems directly suggests the design of its optimal evaluator. In particular, let L be the intuitionistic logic associated with an IS (see Section 3.3). 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 of 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. Last, by using the optimal implementation of *boxes* defined in [12], we get an optimal implementation of the original IS's. Although we shall not explicitly provide the definition of L_{LL}, the reader should keep in mind the previous methodological assumption, in order to understand the translation of terms using Lamping-Gonthier's operators.

Let us just make here some remarks about this translation, which are not subsumed by the previous discussion.

First of all, we work in an untyped setting. So, as in the case of the λ-calculus, we must add to the "logical" system some implicit type-isomorphism taking care of this fact. For coding λ-calculus in proof nets of Linear Logic, there are two possible "standard" solutions, respectively based on the type-isomorphism $D \cong !(D \multimap D)$ and $D \cong (!D) \multimap D$. The first one is adopted in [19, 11], since it was closer to Lamping's original approach. The second one is suggested by the "traditional" embedding of intuitionistic implication by means of linear implication [10]. We will follow the latter, since it is closer to the logical perspective (but the former would work as well). As a consequence, our implementation of the λ-calculus will be slightly different from [11].

A second point which requires some care is the implementation of rewriting rules (cuts), since right hand sides (shortened into rhs's) must be "linearized" w.r.t. the metavariables. This problem does not appear in the λ-calculus because β-reduction is already linear in its metavariables. In IS's, in general, this is not true: metavariables can be erased and/or duplicated by each rule. The case of erasing is not particularly problematic, but duplication is more subtle, since it could affect optimality. For the sake of clarity, we shall translate each rule in several steps. We first apply a *linearization* procedure; then translate it according to the standard paradigm in [11] for representing boxes; finally we partially evaluate the rule, eliminating every redundancy which has been possibly introduced in the previous steps.

Finally, we emphasize that the rewriting system describing the optimal implementation of an IS is itself an Interaction Net. This is particularly nice: we started with IN, added sharing to get IS and implemented (optimally) this sharing inside the original systems. Since IN's get rid of variable names and implement rewriting systems in a symbolic way, our work generalizes some recent results of Burroni [7] and Lafont [17] to higher order rewriting systems.

As much as possible we shall avoid any reference to Interaction Nets and Linear Logic. However [12] is a prerequisite for a deep understanding of the evaluators.

3 Interaction Systems

As we mentioned in the Introduction, Interaction Systems should be correctly understood as the intuitionistic generalization of Lafont's Interaction Nets [16]. The relation between Interaction Systems and Interaction Nets has been deeply investigated in [3]. We shall follow here a different approach (also for avoiding repetition), explaining Interaction Systems as a subclass of Klop's [15] Combinatory Reduction Systems (CRS, for short). In particular, we shall see that Interaction Systems are just *that* subclass of CRS where the Curry-Howard (Proof as Proposition) analogy can still be applied. This will allow us to stress the *intuitionistic* nature of Interaction Systems.

3.1 IS and CRS

It is not our intention to provide the formal definition of Combinatory Reduction Systems, here: we shall just hint the main ideas (the reader is referred to [1, 15] for more details).

Combinatory Reduction Systems (CRS) are a higher order generalization of Term Rewriting Systems, where each form of the syntax may work as a binder. The main consequence is that in the right hand side of a rewriting rule can possibly appear *substitutions*, which are defined (in the obvious way) at a meta-level, as in the λ-calculus.

Since each syntactical form $\mathbf{f}$ can act as binder, its *arity* cannot be just an integer n, expressing the number of its arguments, as at the first order level. Indeed, for any argument M, we must also specify the number of variables bound by $\mathbf{f}$ in M. So the arity of a form $\mathbf{f}$ will be a sequence $k_1 \dots k_n$, where n is the number of arguments, and k_i the number of variables which are bound by $\mathbf{f}$ inside its i-th argument.

The terms of the language are then defined out of forms and variables in the obvious inductive way, according to the arity of the forms.

The next step is to introduce a notion of meta-variable, ranging over terms. This is *very much* like in the λ-calculus; the only care is in defining the proper *arity* for each metavariable X, that is, roughly, the number of distinguished free variables in X (names, not occurrences) which are "accessible" for substitution (in the λ-calculus, this is always 1). So, if the arity of X is k, we may apply to X k-ary substitution $X[^{M_1}/_{x_1}, \ldots, ^{M_k}/_{x_k}]$.

Terms containing metavariables (and substitutions) are called metaterms. A *reduction rule* is any pair (L, R) of metaterms such that:

1. the root form of L is a form;
2. L and R are closed;
3. all the metavariables in R occur already in L;
4. the metavariables in L occur only at leaf-positions in (the abstract syntax tree of) L (in particular, no substitution is applied to them).

If no metavariable occur twice in L, the reduction rule is called *left-linear*. A CRS where all reduction rules are left linear, and without critical pairs, is called *regular*.

Example 3.1 *The recursion operator μ, is a form with arity 1: it binds exactly one variable inside its unique argument. The rewriting rule for μ is expressed as follows:*

$$\mu(x.X) \rightarrow X[^{\mu(x.X)}/_x]$$

where X is a metavariable of arity 1.

Interaction Systems are obtained from CRS's by imposing the following constraints (see section 4 for the formal definition):

- in the signature, we have a bipartition of forms in *constructors* and *destructors*. Constructors have arbitrary arities, while the arity of a destructor must have a leading 0. In other words, a destructor cannot bind variables in its first argument. The reason for this restriction is that the first argument position of a destructor $\mathbf{d}$ is the (unique) place where it may *interact* with a constructor, and $\mathbf{d}$ cannot bind variables in the argument it is interacting with. We will show that this restriction has a strong logical motivation;
- in the rewriting rules, we just impose a restriction on the shape of the left hand side L, that must look as follow:
$$\mathbf{d}(\mathbf{c}(\vec{x}^1_{k_1}.X_1, \cdots, \vec{x}^m_{k_m}.X_m), \cdots, \vec{x}^n_{k_n}.X_n)$$
where $i \neq j$ implies $X_i \neq X_j$ (*left linearity*). The arity of $\mathbf{d}$ is $0k_{m+1}\cdots k_n$ and that of $\mathbf{c}$ is $k_1 \cdots k_m$.

  In other words, every rewriting rule is defined by the *interaction* of a destructor $\mathbf{d}$ with a constructor $\mathbf{c}$ (that is assumed to be unique, i.e. there exists at most one rewriting rule for every pair $\mathbf{d}$-$\mathbf{c}$).

The previous constraints may look very restrictive and somewhat arbitrary. We shall try to answer to these objections in the following subsections. We shall start with providing several examples of Interaction Systems, in order to show their expressive power. Then we shall discuss the *intuitionistic* nature of Interaction Systems, that motivated their introduction.

3.2 Examples

The most typical example of Interaction System is λ-calculus.

Example 3.2 *(***The λ-calculus***) The application @ is a destructor of arity 00, and λ is a constructor of arity 1. The only rewriting rule is β-reduction:*

$$@(\lambda(x.X), Y) \rightarrow X[^Y/_x].$$

An Interaction System where no form is a binder is called *discrete*. Discrete Interaction Systems are obviously a special case of Term Rewriting Systems. The signature of the system is a first order signature Σ partitioned in two classes: the constructors (ranged over by $\mathbf{c}$) and the destructors (ranged over by $\mathbf{d}$). The rewriting rules have the following general shape:

$$\mathbf{d}(\mathbf{c}(X_1, \cdots, X_m), \cdots, X_n) \rightarrow H$$

where H is a terms built up with forms and variables in $\{X_1, \cdots, X_n\}$.

A lot of interesting IS's are discrete.

Example 3.3 *A typical example of discrete Interaction System is* Primitive Recursion. *There are only two constructors* 0 *and* `succ`. *Composition of two functions* $\mathbf{f}(-)$ *and* $\mathbf{g}(-)$ *is obviously expressed as* $\mathbf{f}(\mathbf{g}(-))$. *The primitive recursion scheme has already the correct IS-shape:*

$$\begin{array}{l} \mathbf{d}(0, X) \rightarrow \mathbf{h}(X) \\ \mathbf{d}(\mathtt{succ}(X), Y) \rightarrow \mathbf{f}(X, Y, \mathbf{g}(X, Y)) \end{array}$$

For instance, we may define

$$\begin{array}{ll} \mathtt{add}(0, X) \rightarrow X & \mathtt{mult}(0, X) \rightarrow 0 \\ \mathtt{add}(\mathtt{succ}(X), Y) \rightarrow \mathtt{succ}(\mathtt{add}(X, Y)) & \mathtt{mult}(\mathtt{succ}(X), Y) \rightarrow \mathtt{add}(Y, \mathtt{mult}(X, Y)) \end{array}$$

In a similar way, we may define all inductive types (booleans, lists, trees, and so on).

Example 3.4 *Booleans are defined by two constructors* $\mathbf{T}$ *and* $\mathbf{F}$ *of arity* ε *(two constants). Then you may add your favorite destructors. A typical example is the* if-then-else *operator* $\natural$, *of arity* 000. *The rules for the conditional are described by the following obvious interactions between* $\natural$ *and* $\mathbf{T}$ *or* $\mathbf{F}$*:*

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

Example 3.5 *IS's may be infinite. For instance, when doing arithmetics, we would not like to use the unary notation based on 0 and* `succ`. *A simple solution is to consider each integer n as a distinguished constructor* `n`. *Then, we may reasonably define arithmetical operations in constant time. The only problem is the local sequentiality constraint, that imposes interaction on a distinguished port of the form (however, this is what occurs in practice, on a sequential machine). For instance, we may define*

$$\mathtt{add}(\mathtt{m}, X) \rightarrow \mathtt{add}_{\mathtt{m}}(X) \qquad \mathtt{add}_{\mathtt{m}}(\mathtt{n}) \rightarrow \mathtt{k}$$

where `k` = `n+m`. *Note that we have an infinite number of forms, and also an infinite number of rewriting rules.*

Let us remark a trivial but important property of Discrete Interaction Systems, that could motivate the bipartition of forms into constructors and destructors.

Proposition 3.6 *If in a Discrete Interaction System we have a rewriting rule for every pair* **d**-**c**, *then all closed terms in normal form may only contain constructors.*

In other words, constructors may be used to define the "abstract data type", whose definition is then unaffected by the introduction of new destructors (provided that the definition of each destructor is complete on the data).

Another interesting property of discrete Interaction Systems is that they have trivial optimal implementations. Indeed, since we do not have bindings, they can be represented as acyclic graphs. This means that we never introduce fan-outs during the reduction and so we do not even need control operators (brackets and croissants) to match fan-ins and fan-outs (see [7, 17], where the implementations are optimal).

There is an important point to be understood here. As just remarked, it is trivial to provide an optimal implementation of Discrete Interaction Systems. Since they are Turing-complete (there is a trivial encoding of Combinatory Logic), one may wonder what is the interest to consider higher order systems, where the correct handling of sharing becomes much more difficult. For instance, in the case of λ-calculus, we may compile a λ-term M in a term M' of Combinatory Logic, and then reduce M' in an optimal way. The problem is that the optimal reduction of M' has nothing to do with optimality in the λ-calculus! (see [3]).

Example 3.7 *Let us finally consider another example out of the discrete case: the recursion operator μ. This is a bit problematic, since IS's are based on a principle of* binary *interaction and in the case of μ we just have a sort of "unary" interaction.*

There are two "standard" ways to force a binary interaction for these kind of operators. The first consists in considering them as constructors and to introduce dummy destructors of arity 0 interacting with them. Thus, in the case of μ, we take a destructor $\mathbf{d}_\mu$ and the rewriting rule becomes:

$$\mathbf{d}_\mu(\mu(\langle x\rangle.\,X)) \quad\rightarrow\quad X[^{\mathbf{d}_\mu(\mu(\langle x\rangle.\,X))}/_{x}]$$

The dual way consists in considering operators interacting unarily as destructors, and require the existence of "dual" dummy constructors of arity ε. In the case of μ, the dummy constructor is $\mathbf{c}_\mu$ and the rewriting rule becomes:

$$\mu(\mathbf{c}_\mu,\,\langle x\rangle.\,X) \quad\rightarrow\quad X[^{\mu(\mathbf{c}_\mu,\,\langle x\rangle.\,X)}/_{x}]$$

3.3 The Intuitionistic Nature of IS

We shall now defend our claim that Interaction Systems are the subsystem of Klop's CRS, where the Curry-Howard analogy "still makes sense". We shall do that by stressing the intuitionistic nature of Interaction Systems: constructors and destructors respectively correspond to right and left introduction rules, interaction is cut, and computation is cut-elimination.

3.3.1 From Intuitionistic Systems to IS's ...

An Intuitionistic System, in a *sequent calculus* presentation (*à la* Gentzen), consists of expressions, named *sequents*, whose shape is $A_1, \cdots A_n \vdash B$ where A_i and B are formulas and the comma in the left side of the entail is interpreted as conjunction. Inference rules are partitioned into three groups (in order to emphasize the relationships with IS's, we write rules by assigning terms to proofs):

(**Structural Rules**)

$$(\textit{Exchange}) \quad \frac{\Gamma,\, x : A,\, y : B,\, \Delta\ \vdash t : C}{\Gamma,\, y : B,\, x : A,\, \Delta\ \vdash t : C}$$

$$(\textit{Contraction}) \quad \frac{\Gamma,\, x : A,\, y : A\ \vdash t : C}{\Gamma,\, z : A,\, \Delta\ \vdash t[{}^z/_x, {}^z/_y]\ : C} \qquad (\textit{Weakening}) \quad \frac{\Gamma\ \vdash t : C}{\Gamma,\, z : A\ \vdash t : C}$$

(**Identity Group**)

$$(\textit{Identity}) \quad \frac{}{x : A \vdash x : A} \qquad (\textit{Cut}) \quad \frac{\Gamma\ \vdash t : A \qquad \Delta,\, x : A\ \vdash t' : B}{\Gamma,\, \Delta\ \vdash t'[{}^t/_x] : B}$$

(**Logical Rules**) These are the "peculiar" operations of the systems. Standard operations are implication, conjunction, etc.: in the following we will discuss several examples. What is important to remark here is that logical rules are split into two groups: those introducing the logical connective "on the left" of the sequent and those introducing symbols "on the right". The former ones are called *destructors*; the latter ones are named *constructors*. The shape of these rules will be:

$$\frac{\Gamma_1,\, \vec{x}^{\,1} : \vec{A}^1\ \vdash\ t_1 : B_1 \quad \cdots \quad \Gamma_m,\, \vec{x}^{\,m} : \vec{A}^m\ \vdash\ t_m : B_m \qquad \Delta, z : C \vdash t : D}{\Gamma_1,\, \cdots,\, \Gamma_m,\, \Delta\,, y : \mathtt{T}_{\mathbf{d}}(\vec{A}^1,\, B_1,\, \cdots,\, \vec{A}^m,\, B_m, C)\ \vdash\ t[{}^{\mathbf{d}(y, \vec{x}^{\,1}.t_1, \cdots, \vec{x}^{\,m}.t_m)}/_z] : D}$$

for destructors and

$$\frac{\Gamma_1,\, \vec{x}^{\,1} : \vec{A}^1\ \vdash\ t_1 : B_1 \quad \cdots \quad \Gamma_n,\, \vec{x}^{\,n} : \vec{A}^n\ \vdash\ t_n : B_n}{\Gamma_1,\, \cdots,\, \Gamma_n\ \vdash\ \mathbf{c}(\vec{x}^{\,1}.t_1,\, \cdots,\, \vec{x}^{\,n}.t_n) : \mathtt{T}_{\mathbf{c}}(\vec{A}^1,\, B_1,\, \cdots,\, \vec{A}^n,\, B_n)}$$

for constructors. Above $\mathtt{T}_{\mathbf{d}}(\vec{A}^1, B_1, \cdots, \vec{A}^m, B_m, C)$ and $\mathtt{T}_{\mathbf{c}}(\vec{A}^1, B_1, \cdots, \vec{A}^n, B_n)$ are types built up by means of the types they take as argument and they are equal, provided they correspond to the same logical operator. The unique proviso is that no commitment is done about the contexts Γ_i, that is they are assumed pairwise different. More precisely we are generalizing the so-called *multiplicative* connectives (see [20], pg. 47 for a discussion about additives).

A standard example is implication, that gives the expressions of typed λ-calculus:

$$(\rightarrow\ \textit{left}) \quad \frac{\Delta\ \vdash t : A \qquad z : B,\, \Gamma\ \vdash t' : C}{\Delta,\, y : A\ \rightarrow B,\, \Gamma\ \vdash t'[{}^{@(y,t)}/_z] : C} \qquad (\rightarrow\ \textit{right}) \quad \frac{\Delta,\, x : A\ \vdash t : B}{\Delta\ \vdash \lambda(\langle x\rangle.t)\ :\ A \rightarrow B}$$

An easy consequence of the above construction is that every proof of an Intuitionistic System can be described by an IS-expression. Note in particular that destructors (as the application @) cannot bind at the level of the variable y (the *principal port* of destructors, in Lafont's terminology) since it is a newly added hypothesis and it is found in the lhs of the final sequent. This is the reason why, in the concrete syntax for IS's, we have assumed that destructors have arity 0 at the first argument.

An important theorem of sequent calculus is that stating the redundancy of cut-rules, i.e. every proof with instances of the cut-rule can be turned into an equivalent one without cuts. From the operational point of view this gives dynamics, because it guarantees the logical soundness of *rewriting* a proof into another one. These rewritings must be specified *a priori*: a proof ending into a cut must be remade into another one by means of some mechanism that is characteristic of that cut.

In order to ensure the possibility of eliminating *all* cuts, the cut-elimination (term rewriting) process must *terminate*. In general, this property does not hold in IS's. We just have a general correspondence between IS's and systems with an *intuitionistic nature*, but only *a posteriori* we may actually check if a particular system enjoys good "logical" properties (cut-elimination, subformula property, ...).

Obviously, we could proceed the other way round, imposing some sufficient conditions on IS's (typing, first of all) to establish a tighter relation with logic. This is surely an interesting subject, but it is out of the scope of the present paper. So, in the following, we shall merely focus on the *dynamic* aspects of cut-elimination, without worrying with termination.

Let us recall what happens in intuitionistic logic when we try to eliminate a cut between the instances of the destructor and the constructor of the implication. Here is the typical situation:

$$\dfrac{\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(\langle x\rangle.\,t) : A \rightarrow B} \qquad \dfrac{\Delta \vdash t' : A \qquad y : B, \Theta \vdash t'' : C}{\Delta, z : A \rightarrow B, \Theta \vdash t''[{}^{@(z,t')}/_{y}] : C}}{\Gamma, \Delta, \Theta \vdash t''[{}^{@(z,t')}/_{y}][{}^{\lambda(\langle x\rangle.t)}/_{z}] : C}$$

The elimination of the above cut consists in introducing two cuts of lesser grade. The rewritten proof is:

$$\dfrac{\Delta \vdash t' : A \qquad \dfrac{\Gamma, x : A \vdash t : B \qquad y : B, \Theta \vdash t'' : C}{\Gamma, x : A, \Theta \vdash t''[{}^{t}/_{y}] : C}}{\Gamma, \Delta, \Theta \vdash t''[{}^{t}/_{y}][{}^{t'}/_{x}] : C}$$

Note that this meta-operation on proofs induces a rewriting rule in the underlying IS, which is, in this case, β-reduction. Indeed, it is easy to check that the proofs $t'[{}^{@(z,t)}/_{y}][{}^{\lambda(\langle x\rangle.t)}/_{z}]$ and $t''[{}^{t}/_{y}][{}^{t'}/_{x}]$ can be proved equal via β-reduction.

Back to the discussion about a generic cut-elimination, we have to understand what kind of rewriting this process performs. Foremost, there are several kinds of cuts: the one just described is a *logical cut* (i.e. between two dual logical rules). The other forms of cut are when the rules preceding the cut are not dual. In this case, the Intuitionistic System eliminate the cut by lifting it in the premises of one of the rules (that becomes the last rule of the proof). These kind of cuts have no counterpart in IS's, since they are implicitly dealt with by the definition of substitution. So, let us concentrate on logical cuts only.

Let L and R be the left and right sequent in the cut-rule, respectively. The first observation is that, during the process of cut-elimination, the proofs ending into the premises of L and R

are considered as a whole: no assumption about them is done and every operation on any of the hypotheses (bound by L or R) must be done on the others in the same sequent, too. These sequents constitute the *interface* of the cut. Starting from the interface, one can imagine to build up a new proof, by means of arbitrary inference rules. In the case of implication we have used a sequence of two cuts. However, other choices could be possible.

The unique constraint of the cut-elimination process is the *prohibition of creating new hypotheses*. This has two implications:

1. the variables bound by L or R must be suitably filled in (typically with cuts or introducing *new* forms binding them);
2. if axioms are introduced then the variable in the premise must be consumed (with a cut or by another rule) by the proof.

What is the shape of the induced rewriting in the underlying IS? The lhs must be something of the form $\mathbf{d}(\mathbf{c}(\vec{x}^1_{k_1}.X_1,\cdots,\vec{x}^m_{k_m}.X_m),\cdots,\vec{x}^n_{k_n}.X_n)$ since a (logical) cut always involves a destructor rule and a constructor one. The X_i represent the proofs ending into the sequents in the hypotheses of L and R. In the right hand sides we may

- introduce new variables with axioms or with weakenings (denoted by x),

or, starting from proofs $H_1,\cdots,H_n$ that have been already built up,

- we can exploit proofs X_i (provided we fill the bound hypothesis: notation $X[^{H_1}/_{x_1},\cdots,^{H_n}/_{x_n}]$)
- or introduce a new logical rule (denoted as $\mathbf{f}(\vec{x}^1.H_1,\cdots,\vec{x}^n.H_n)$).

The other logical operations (contractions, cuts) are visible in the syntax under copying or interactions. The syntactical constraint reflecting the logical absence of new hypotheses is: right hand sides of rules must be closed expressions.

According to the above paradigm, every Intuitionistic System can be modeled through a suitable IS. Let us consider some example.

Example 3.8 (Naturals) *Natural numbers are defined by two constructors* $\mathtt{0}$ *and* $\mathtt{succ}$. *These constructors are respectively associated with the following right introduction rules:*

$$(nat, right_0)\quad \vdash \mathtt{0} : nat \qquad\qquad (nat, right_S)\quad \frac{\Delta, \vdash n : nat}{\Delta \vdash \mathtt{succ}(n) : nat}$$

In this case, we have two introduction rules for the type nat. *A typical destructor is* $\mathtt{add}$.

$$(nat, left_{add})\quad \frac{\Delta \vdash p : nat \qquad \Gamma, y : nat \vdash t : A}{\Delta, \Gamma\ x : nat \vdash t[^{\mathtt{add}(x,p)}/_y] : A}$$

where A can be any type. Of course, we have a different left introduction rule for each destructor. The following is an example of cut:

$$\frac{\vdash \mathtt{0} : nat \qquad \dfrac{\Delta \vdash p : nat \qquad y : nat, \Theta \vdash t : A}{\Delta, x : nat, \Theta \vdash t[^{\mathtt{add}(x,p)}/_y] : A}}{\Delta, \Theta \vdash t[^{\mathtt{add}(x,p)}/_y][^{\mathtt{0}}/_x] : A}$$

that is simplified into:

$$\frac{\Delta \vdash p : nat \qquad y : nat, \Theta \vdash t : A}{\Delta, \Theta \vdash t[{}^{p}/_{y}] : A}$$

The above elimination induces the IS-rule $\mathtt{add}(\mathtt{0}, X) \rightarrow X$*, according to which we have that* $t[{}^{\mathtt{add}(x,p)}/_{y}][{}^{\mathtt{0}}/_{x}]$ *and* $t[{}^{p}/_{y}]$ *are equal.*

Example 3.9 (Lists) *Lists are defined by means of two constructors* `cons` *and* `nil` *of arity* 00 *and* ε*, respectively. The typical destructors are* `hd` *and* `tl` *of arity* 0*. In the case of lists of integers, we may write the following introduction rules for the type* natlist*:*

$$(natlist, right_{nil}) \quad \vdash \mathtt{nil} : natlist$$

$$(natlist, right_{cons}) \quad \frac{\Delta \vdash n : nat \qquad \Gamma \vdash l : natlist}{\Delta, \Gamma \vdash \mathtt{cons}(n, l) : natlist}$$

$$(natlist, left_{hd}) \quad \frac{\Gamma, y : nat \vdash t : A}{\Gamma, x : natlist \vdash t[{}^{\mathtt{hd}(x)}/_{y}] : A}$$

$$(natlist, left_{tl}) \quad \frac{\Gamma, y : natlist \vdash t : A}{\Gamma, x : natlist \vdash t[{}^{\mathtt{tl}(x)}/_{y}] : A}$$

A typical cut is:

$$\frac{\dfrac{\Delta \vdash n : nat \qquad \Gamma \vdash l : natlist}{\Delta, \Gamma \vdash \mathtt{cons}(n, l) : natlist} \qquad \dfrac{\Theta, y : Nat \vdash t : A}{\Theta, x : natlist \vdash t[{}^{\mathtt{hd}(x)}/_{y}] : A}}{\Delta, \Gamma, \Theta \vdash t[{}^{\mathtt{hd}(x)}/_{y}][{}^{\mathtt{cons}(n,l)}/_{x}]}$$

and the obvious cut elimination rule gives:

$$\frac{\Delta \vdash n : nat \qquad \Theta, y : nat \vdash t : A}{\Delta, \Theta \vdash t[{}^{n}/_{y}] : A}$$

Again, by the reduction rule $\mathtt{hd}(\mathtt{cons}(X, Y)) \rightarrow X$*, we have* $t[{}^{\mathtt{hd}(l)}/_{y}][{}^{\mathtt{cons}(n,l)}/_{x}] = t[{}^{n}/_{y}]$*. As an exercise the reader can provide the cut between* $(natlist, right_{cons})$ *and* $(natlist, left_{tl})$ *and verify that it induces the following rewriting:*

$$\mathtt{tl}(\mathtt{cons}(X, L)) \rightarrow L$$

3.3.2 ... and back

The *vice versa*, namely interpreting an IS into an Intuitionistic System is not always possible. In particular, the main problems are due to the lack of any type discipline in IS's (we have the same problem with the pure λ-calculus).

Up to this inadequacy, it is possible to provide the generic rule corresponding to a form. The paradigm is exactly the reverse of that discussed in the previous subsection. In particular, we respectively associate with a destructor or a constructor the two introduction rules described at the beginning of section 3.3.1.

A rewriting rule, is interpreted as the elimination of a logical cut. In order to understand the way the cut is rewritten in the intuitionistic system, we reason by induction on the structure of the rhs of the IS-rule. Recall that the right hand side H of an IS-rule is a closed expression built up by the following syntax:

$$H \quad ::= \quad x \quad | \quad \mathbf{f}(\vec{x}^{\,1}.\,H_1, \cdots, \vec{x}^{\,n}.\,H_n) \quad | \quad X[{}^{H_1}/_{x_1}, \cdots, {}^{H_n}/_{x_n}]$$

The case of variables is easy: they correspond to axioms. A metaexpression $X[{}^{H_1}/_{x_1}, \cdots, {}^{H_n}/_{x_n}]$ is interpreted as a sequence of cuts between the variables x_i in X and the proofs representing H_i. A metaexpression of the shape $\mathbf{f}(\vec{x}^{\,1}.\,H_1, \cdots, \vec{x}^{\,n}.\,H_n)$, where $\mathbf{f}$ is a constructor, is interpreted by taking the proofs corresponding to $H_1, \ldots, H_n$, possibly adding weakenings if bound variables do not appear in the bodies, and adding as last rule that corresponding to $\mathbf{f}$. If $\mathbf{f}$ is a destructor, the rule corresponding to $\mathbf{f}$ take as sub-proofs those of $H_2, \ldots, H_n$. Finally a cut must be introduced between the rule of $\mathbf{f}$ and the proof of H_1. Some cuts with axioms can be eliminated in the obvious way, after this rough interpretation.

The last step consists of adding a sequence of weakenings that perform the sharing of the copies of the proofs replacing the same metavariable.

4 The formal definition of IS

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 $\mathbf{c}$) and *destructors* (ranged over by $\mathbf{d}$). Variables will be ranged over by x, y, z, $\cdots$, 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 $\mathbf{f}$, 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 $\mathbf{d} \in \Sigma^-$ has a leading 0 (i.e., it cannot bind over its first argument). The reason for this restriction is that, in Lafont's notation [16], 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 $t, t_1, \cdots$, are inductively generated by the two rules below:

a. every variable is an expression;

b. if $\mathbf{f}$ is a form of arity $k_1 \cdots k_n$ and $t_1, \cdots, t_n$ are expressions then $\mathbf{f}(\vec{x}^{1}_{k_1}.\,t_1, \cdots, \vec{x}^{n}_{k_n}.\,t_n)$ 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 $X, Y, \cdots$, possibly indexed (see [1] for more details). Metaexpressions will be denoted by $H, H_1 \cdots$.

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

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

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

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

$$H \quad ::= \quad x \quad | \quad \mathbf{f}(\vec{x}^{1}_{a_1}.\, H_1, \cdots, \vec{x}^{j}_{a_j}.\, H_j) \quad | \quad X_i[^{H_1}/_{x^i_1}, \cdots, ^{H_{k_i}}/_{x^i_{k_i}}]$$

The expression $X[^{H_1}/_{x_1}, \cdots, ^{H_n}/_{x_n}]$ 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 $\mathbf{d}$-$\mathbf{c}$.

Example 4.1 *As we already remarked, the most typical example of IS is λ-calculus. Many interesting Interaction Systems can be then defined by enriching the λ-calculus with "δ-rules". For instance, an alternative way to look at the recursion operator μ is as a destructor of arity 0 interacting with λ (i.e., a destructor alternative to application). In this case it is described by the following reduction*

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

(Note that the μ and the λ in the rhs have nothing to do with those in the lhs). We shall use this definition of μ in the rest of the paper.

4.1 Bourbaki representations

Expressions of Interaction Systems have graphical representations that are reminiscent of Lafont's Interaction Nets. In particular, a form $\mathbf{f}$ of arity $k_1 \ldots k_n$ is represented as a node of name $\mathbf{f}$ with $1 + \sum_{i=1}^{n} p_i$ ports (edges); $p_i = k_i + 1$ is the i-th *partition* of $\mathbf{f}$. The i-th partition represents the connections between $\mathbf{f}$ and its i-th argument M. In particular, one connection, that corresponding to the unique *negative* port, called the *argument port*, is with the root of M, and k_i with the variables bound by $\mathbf{f}$ (the latters will be called *bound* ports of the partition and have a *positive polarity*). Observe that bound variables correspond to (bound) ports of the form $\mathbf{f}$. Thus our graphs are cyclic. Indeed, already Bourbaki used this notation for predicate logic [6]: this is the reason for calling *Bourbaki representations* our graphical representations of expressions.

The unique port which does not belong to a partition is called the output port of $\mathbf{f}$. All the forms, have a *principal port*, which is drawn with an outgoing arrow (the arrow is omitted when it is clear from the context: see Figure 2 below). The other entries are called *auxiliary ports* (see [16]). In the case of a constructor, the principal port coincides with the output port. In the case of a destructor, the principal port is the unique edge in the first partition (recall that the arity of the first argument of a destructor is eventually 0, so this partition is a singleton and does not have bound ports).

By this definition, interactions between constructors and destructors takes place only at principal ports (local sequentiality).

Following Lafont, it is possible to add *polarities* to ports. In particular, the output port of each form is always positive. So, the principal port of a constructor is positive. On the contrary, the principal port of a destructor is negative. All bounds port have positive polarities, and all the other ports are negative. In particular, in every partition we have *exactly* one negative port. An edge may only connect forms at ports with opposite polarities.

Polarities have a strong logical motivation. They are essentially related to the connections established by the form with the formulae (the conclusions) in the upper sequents of the associated introduction rule: positive if the formula (the conclusion) is in the rhs of a sequent, and negative otherwise. Moreover, ports belonging to a same partition are eventually connected with conclusions of a *same* sequent (see [3] for more details).

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, we illustrate in Figure 2.(a) the graphical representation of $\mathbf{c}(\langle x\rangle.\, \mathbf{d}(x),\, y.\, \mathbf{g}(y,\, y))$. Variables which are not bound

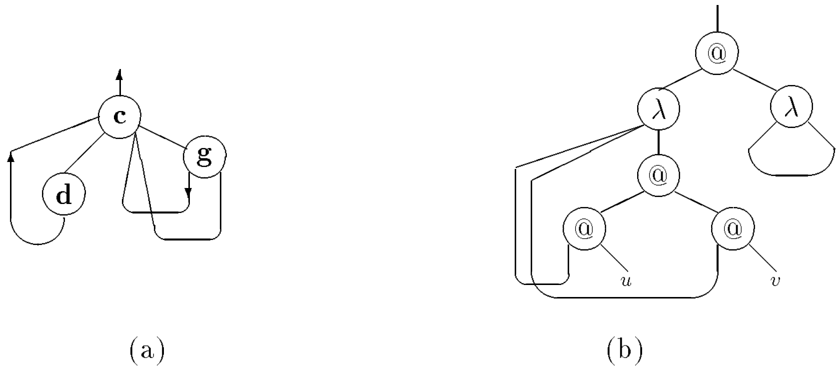


Figure 2: Graphical representations of expressions

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 2.(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.

The reader is referred to [3] for more details about the graphical representation. A lot of examples will be found in the following pages.

5 Labeling and the family relation

This section is devoted to the generalization of Lévy's labeling [21] from λ-calculus to arbitrary Interaction Systems. Labeling allows us to define the family relation, that is the kind of "optimal" sharing the implementation should support. Let $\mathbb{N}^+$ be the set of nonempty sequences of natural numbers.

Definition 5.1 *Let $L = \{a,\, b,\, \cdots\}$ be a countable set of* atomic labels. *The set* $\mathbf{L}$ *of* labels, *ranged over by $\alpha,\, \beta, \cdots$ is defined by the following rules:*

$$L \quad | \quad \alpha\beta \quad | \quad (\alpha)_s$$

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 naturals, for this purpose).

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

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

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

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

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

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

$$\begin{array}{rcl}
\mathcal{L}^s_\alpha(x) & = & (\alpha)_s(x) \\
\mathcal{L}^s_\alpha(\mathbf{f}(\vec{x}^{\,0}.\,H_0,\cdots,\vec{x}^{\,m}.\,H_m)) & = & (\alpha)_s(\mathbf{f}(\vec{x}^{\,0}.\,\mathcal{L}^{s0}_\alpha(H_0),\,\cdots,\,\vec{x}^{\,m}.\,\mathcal{L}^{sm}_\alpha(H_m)) \\
\mathcal{L}^s_\alpha(X[{}^{H_0}/_{x_0},\,\cdots,\,{}^{H_n}/_{x_n}]) & = & (\alpha)_s(X[{}^{\mathcal{L}^{s0}_\alpha(H_0)}/_{x_0},\,\cdots,\,{}^{\mathcal{L}^{sn}_\alpha(H_n)}/_{x_n}])
\end{array}$$

Example 5.2 *Consider again the λ-calculus. The β-reduction $@(\lambda(\langle x\rangle.\,X),Y)\ \rightarrow\ X[{}^Y/_x]$ gives rise, in the labeled version, to the following rules:*

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

where $\ell = \alpha_1\cdots\alpha_i$. Note that, by definition, $\mathcal{L}^0_\ell(X[{}^Y/_x]) = (\ell)_0(X[{}^{(\ell)_{00}(Y)}/_x])$, therefore, by replacing ℓ_0 with $\overline{\ell}$ 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 $(\Sigma,\ R)$ be an IS and let $(\Sigma^L,\ R^L)$ be the labeled CRS built in the way described above. Given a form $\mathbf{f}$ in Σ and an occurrence of it in a term t of Σ^L, we say that $\mathbf{f}$ has *label* $\alpha_1\alpha_2\cdots\alpha_i$ if, in the syntactic tree of t, $\alpha_1\alpha_2\cdots\alpha_i$ is the path towards the root which links $\mathbf{f}$ 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 $\mathbf{d}$-$\mathbf{c}$ of the redex u).

We will say that an expression owns the property **INIT** when the label of the forms are atomic and pairwise different.

Definition 5.3 *Two redexes yielded by a derivation starting at a labeled expression owning* **INIT** *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 [21, 22]. The relations among them have been discussed in detail in [3, 20].

6 Sharing graphs

Let us come 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 the graphical representation of expressions with control operators. Such operators are described in Figure 3 and must be considered as forms. This means that each node has a principal port of

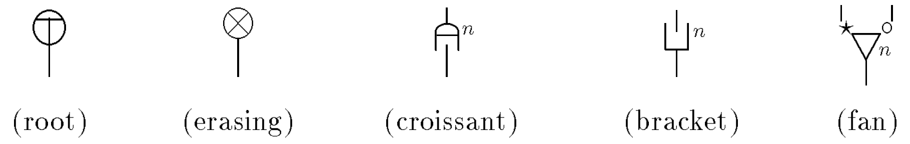


Figure 3: The control operators

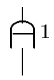
interaction. In Figure 3, the principal ports are always at the lower edges.

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.

You can get some intuition on control operators by their relation with linear logic. In this logic, every datum which has a not-linear use, must be put inside a *box*. The number of boxes enclosing a datum essentially expresses the number of different levels of sharing the datum is subject to.

The purpose of square brackets of index 0 is essentially that of marking some points of "discontinuity" in the graph which are not explicitly expressed by other control operators. Typically, when we pass from a variable to its binder, or from an application to its right argument (in both cases we are implicitly switching from a type $!(D)$ to D, or *vice versa*).

Boxes can be opened, or shifted inside other boxes. Both these operations dynamically modify the *sharing levels* in the term. So we must introduce some operators to implement these modifications. In particular, a box is opened when the datum it contains is accessed via a variable (one potential level of sharing has been dropped). This "push down" on the datum is expressed by the croissant. So, the translation of a variable will just look as follows:



A box M can be shifted inside another box N when we try to access M from some of the free variables of N. In this case, the levels of sharing M is subjected to, is augmented of 1 (the potential sharing N). Again, we need a new operator to express this modification. This is the square bracket (with index $n \geq 0$). In particular, all the time we build a box around a datum P (every time a datum can be potentially shared), we must add a square bracket of index 1 on each negative conclusion (free variable) of P.

From the semantical point of view, brackets and croissants should be understood as *context transformers*. They modify the shape of the context (adding, erasing, freezing and unfreezing levels), in order to correctly travel along the sharing graph in the read-back phase. For example, the presence of indexes besides the operators indicates the *depth* where the modification takes place in the context (see Section 8 or [18, 19]).

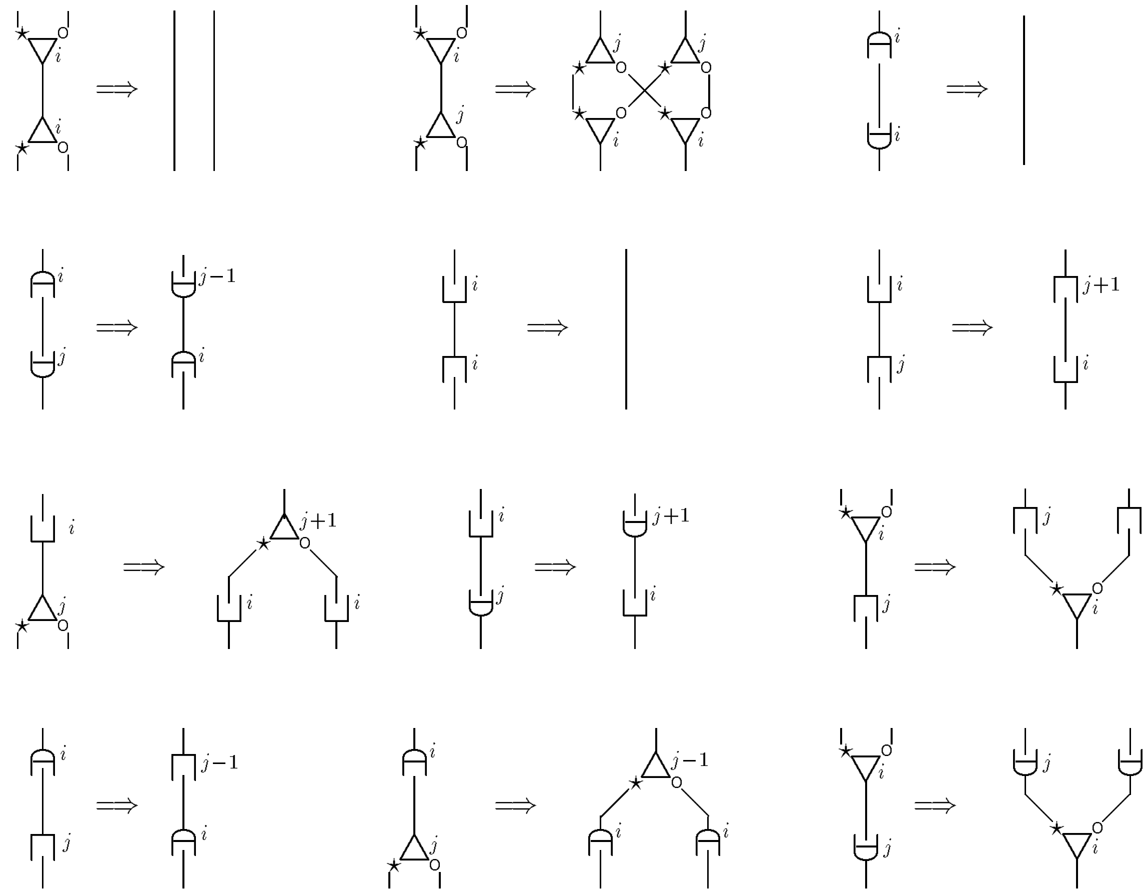


Figure 4: The control rules ($0 \leq i < j$)

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

The root and erasing 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, in order to preserve locality of the rewriting rules, the parts of the graph that are erased by a contraction are connected to erasing nodes. We could add rules providing garbage collection (mainly involving the erasing node), but these do not eliminate all the garbage and are not essential for correctness. So we omit them.

7 Implementation

Now we have all the preliminaries to provide the implementation of a generic IS. The optimal implementation is described as a graph rewriting system. The nodes of the graph are either control operators or syntactical forms of the IS. Actually, the graph rewriting system is itself an Interaction Net, inheriting all good properties of this formalism (in particular, the strong diamond property).

The translation of IS-expressions is discussed in Subsection 7.1 below. Subsection 7.2 will deal with the rewriting rules.

7.1 The translation of expressions

In the translation of an arbitrary expression in sharing graph we shall essentially follow [12]. The translation function $\mathcal{T}^+$ calls an auxiliary function $\mathcal{T}$, inductively defined in Figure 6.

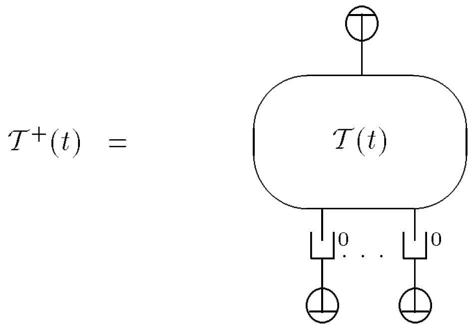


Figure 5: The encoding function $\mathcal{T}^+$

Remark 7.1 : *The definition of $\mathcal{T}$ will be slightly different from that provided in [4, 20]. Actually, there, we strictly followed the local implementation of boxes for linear logic described in the first part of [12]. The reader can verify that the translation in [4, 20] eventually introduces a redundant number of redexes between brackets of index* 0 *facing each other. Therefore expressions in normal form were encoded, in general, by sharing graphs not in normal form. A better translation can be obtained by avoiding the introduction of all these redexes. Indeed, such translation relies on the implementation of Girard's* unified logic *(a synthesis of classical, intuitionistic and linear logic) described in the second part of [12]. This is the approach we will follow here.*

For simplicity, in Figure 6, we consider the paradigmatic case when constructors and destructors have respectively arity 1 and 01. The other cases are easily derived. In this figure, the dangling

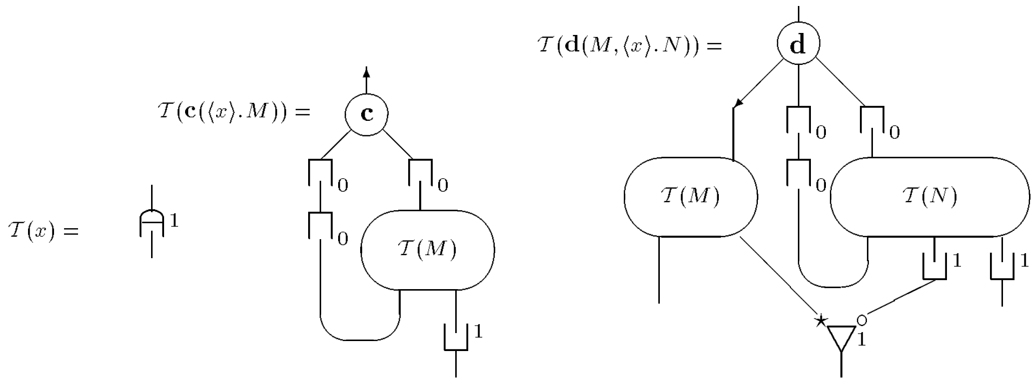


Figure 6: The translation function $\mathcal{T}$ (bound variables occur in the bodies)

edges in the bottom represent 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 connected to an erasing node, as shown in Figure 7 (logically, the variable has been introduced by

$\mathcal{T}(\mathbf{c}(\langle x\rangle.\,M)) \;=_{x\notin\mathbf{var}(M)}$

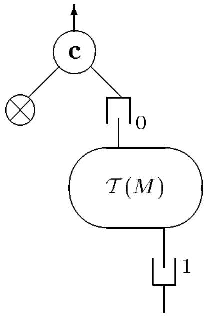
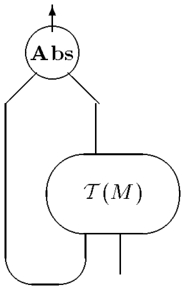


Figure 7: The translation function $\mathcal{T}$ (the bound variable does not occur in the body)

means of the weakening rule). The above translation is a more or less obvious consequence of the linear logic implementation in [12] (via a type isomorphism $D \cong (!D) \multimap D$. A variable x represents an axiom whose negative edge has been derelicted. All the arguments of forms (apart the argument at the main port of a destructor) must be put inside boxes (must be protected by !). This because, each one of these arguments may be used in a non linear way during rewriting, and/or can be used as an argument in a substitution.

Here a peculiarity deserves to be emphasized. Let us consider the case of the constructor **c** (the same considerations hold for the destructor). When we put the argument of **c** inside a box, the control operators to be added at the level of the bound variable are different from the control operators to be added to free variables. The reader should intuitively imagine to have a pseudo-binder between the form and the body of the argument, as drawn below:



In this case, when we perform the !-introduction and the **c**-introduction, we obtain the configuration illustrated in Figure 8.

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 6. The reason for proceeding in this way will become more clear when we will describe the translation of the rewriting rules of IS's. At that stage, the ghost-binder will become apparent and will play an essential role during partial evaluation.

7.2 The translation of rewriting rules

Rewriting rules may be classified in three groups. We shall discuss each group in a separate subsection.

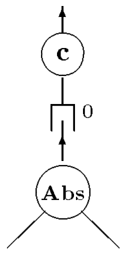


Figure 8: !-introduction with ghost-binder

7.2.1 Control Rules

These are the 12 rules in Figure 4. These rules provide the general framework for the optimal implementation of the structural part of IS's.

7.2.2 Interfacing Rules

These are the rules which describe the interaction between control operators and forms of the syntax (that is, they describe the interface between the structural and the logical part of IS's). These rules have a polimorphic nature. We define them by means of schemas, where **f** can be an arbitrary form of the syntax. The rules are drawn in Figure 9 (where $i > 0$).

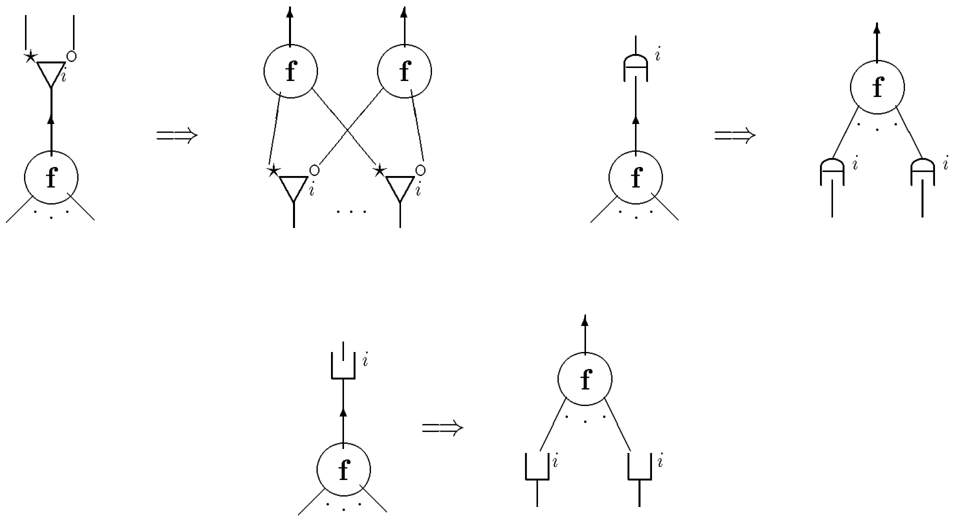


Figure 9: The interfacing rules between control operators and forms ($i > 0$)

As you see, interfacing the structural and the logical part of IS at the implementation level is *very* simple. This is a main consequence of the logical nature of IS's.

7.2.3 Proper Rules

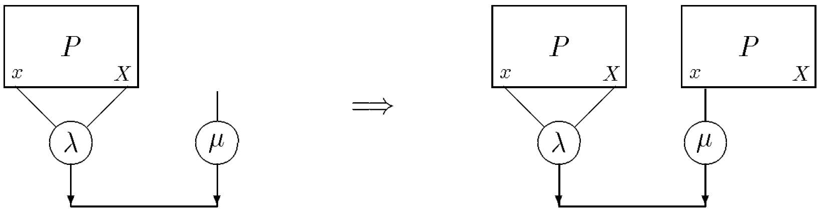
These rules describe the interactions between destructors and constructors of the IS's. These are the only rules which are dependent from the particular Interaction System under investigation, and the only ones which deserve some care, in the translation. We shall define the implementation of the rewriting rules in four steps: β-expansion, linearization, translation and partial evaluation.

The idea behind β-expansion and linearization is that of expliciting the "interface" between the new forms which have been possibly introduced in the rhs of the rule, and the metavariables in its lhs. Then, we may essentially translate the rhs as a normal term, just regarding the metavariables as "black box". Finally, we must partially evaluate the graph obtained in this way, since during β-expansion and linearization we have introduced some "pseudo-operators" which should disappear.

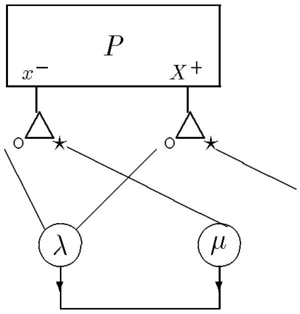
The linearization step is particularly important (β-expansion is just aimed to linearization). Consider the rewriting rule for μ:

$$\mu(\lambda(x.X)) \to X[{}^{\mu(\lambda(y.X[{}^{y}/_{x}]))}/_{x}]$$

Note that, in the rhs, we have two occurrences of the metavariable X. Intuitively, one could expect to express the previous reduction by means of a graph rewriting rule of the following kind:



However, in this way, the portion of graph in the box (corresponding to the metavariable X) should be physically duplicated. Consequently, if we had a (actual or virtual) redex inside X, it is duplicated, too. Moreover, the rewriting rule would not be "local" anymore (it would not be in the Interaction Net form), since it requires a global operation on a box. So, we must try to share the double occurrence of X in the rhs. To this aim, observe that also the variable x occurs twice (one for each instance). This means that we must be able to *unshare* the graph at this level since one occurrence has to be bound by the λ and the other has to be connected to the μ-operator. This double operation of sharing and unsharing is just the purpose of Lamping's *fan-in* and *fan-out* operators. Summing up, we expect to get an implementation of the rhs of the rule for μ that looks like:



Notice moreover that the rewriting rule has now a completely *local* behaviour: it merely modifies the connections of ports of the two forms yielding the redex.

The difficult problem, solved by the following translation procedure, is to introduce in the correct way the control operators (brackets and croissants) which ensure the right matching of fan-ins and fan-outs.

(β-**expansion**) The first step is to β-expand all substitutions in the rhs. The aim of this step is to provide a clean vision of all the metavariables in the rhs. For this purpose we shall use two classes of *pseudo-forms*: abstraction $\mathbf{Abs}_n$ and application $\mathbf{App}_n$, for $n \geq 0$. Pseudo-forms are similar to all other forms of the syntax. $\mathbf{Abs}_n$ is a constructor of arity n whilst $\mathbf{App}_n$ is a destructor of arity 0^{n+1}. As the reader could probably imagine, they generalize λ-calculus abstraction and application. Their interaction is expressed by the rule:

$$\mathbf{App}_n(\mathbf{Abs}_n(\langle x_1, \cdots, x_n\rangle. X), Y_1, \cdots, Y_n) \;\rightarrow\; X[^{Y_1}/_{x_1}, \cdots, ^{Y_n}/_{x_n}]$$

In the following we shall always omit the index 1 in $\mathbf{Abs}_1$ and $\mathbf{App}_1$.

The step of β-expansion consists in rewriting the rhs of the IS-rule by β-expanding substitutions into interactions of the pseudo-operators $\mathbf{Abs}_n$ and $\mathbf{App}_n$.

Example 7.2 *Consider the rewriting rule for μ:*

$$\mu(\lambda(x. X)) \rightarrow X[^{\mu(\lambda(y. X[^y/_x]))}/_x]$$

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

$$\mathbf{App}(\mathbf{Abs}(x. X), \mu(\lambda(y. \mathbf{App}(\mathbf{Abs}(x. X), y))))$$

Note that, after the β-expansion, all metavariables are closed by pseudo binders, i.e. they become expressions of the following kind: $\mathbf{Abs}_n(\vec{x}. X)$.

(**linearization**) The next step consists in *linearizing* the rhs w.r.t. the occurrences of expressions $\mathbf{Abs}_n(\vec{x}. X)$. This is obtained by taking, for every metavariable X_i occurring in the left hand side of the IS-rewriting rule (let them be k), a fresh *pseudo-variable* w_i and replacing every occurrence of $\mathbf{Abs}_n(\vec{x}^{\,i}. X_i)$ with w_i. In this way we yield a metaexpression T. Next T is closed w.r.t. the metavariables w_i's, and the (closed) metavariables $\mathbf{Abs}_n(\vec{x}^{\,i}. X_i)$ are passed as arguments to this term. In other words, by linearization, we get a metaexpression of the following kind, where each metavariable occur exactly once (and no substitution is applied to them):

$$\mathbf{App}_k(\mathbf{Abs}_k(\langle w_1, \cdots, w_k\rangle. T), \mathbf{Abs}_{n_1}(\vec{x}^{\,1}. X_1), \cdots, \mathbf{Abs}_{n_k}(\vec{x}^{\,k}. X_k))$$

where n_i is the arity of the metavariable X_i.

Example 7.3 *After the linearization step, the rhs of the recursion rule becomes:*

$$\mathbf{App}(\mathbf{Abs}(w. \mathbf{App}(w, \mu(\lambda(y. \mathbf{App}(w, y))))), \mathbf{Abs}(x. X))$$

We want to remark that every metavariable in the lhs of the IS-rewriting rule occurs exactly once in the linearized metaexpression yielded by the above procedure, even if it does not occur in the rhs of the IS-rewriting rule. For instance, in the case of conditionals, the linearization of the rhs of $\natural(\mathbf{T}, X, Y) \to X$ gives

$$\mathbf{App}_2(\mathbf{Abs}_2(\langle w_1,\, w_2\rangle.\, w_1),\, \mathbf{Abs}_0(X),\, \mathbf{Abs}_0(Y)).$$

The actual erasing will be performed in the following steps (see translation and partial evaluation).

(**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 $\mathbf{Abs}_n(\vec{x}.\,X)$ as a "black-box". According to the linearization step, the expression that results will have the shape

$$\mathbf{App}_k(M,\ \mathbf{Abs}_{n_1}(\vec{x}^{\,1}.\,X_{i_1}), \cdots,\ \mathbf{Abs}_{n_k}(\vec{x}^{\,k}.\,X_{i_k}))\,.$$

The translation of this expression is drawn in Figure 10, where, for simplicity, we have assumed

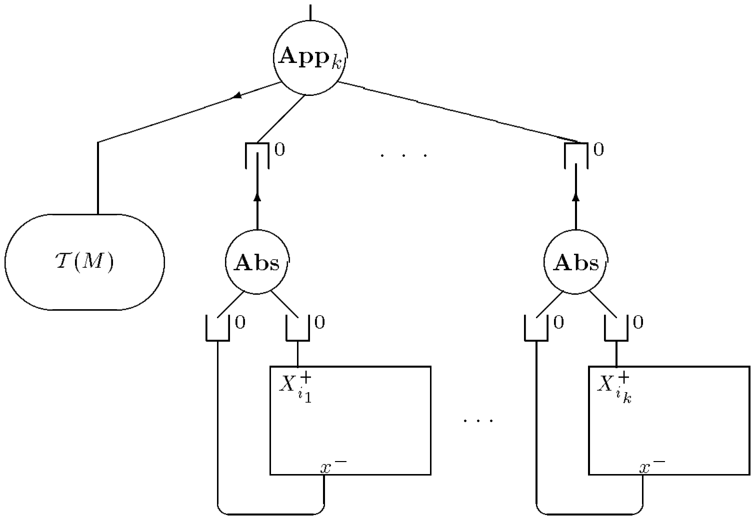


Figure 10: The translation step

$n_i = 1$, for every i. Notice that metavariables are not put inside boxes: they *are* boxes, due to the translation of expressions in Figure 6. We implicitly use this box instead of building a new box around the argument of the application. In particular, no operation around the (unaccessible!) free variables of the instance of the metavariable must be performed.

Now we can provide some more intuition about our translation in Figure 6, and the role of the "ghost-binder". In particular, ghost-binders become apparent in the translation in Figure 10: they are the **Abs** pseudo-forms. The square bracket around each **Abs** are meant to extend the box containing the metavariable up to comprising the pseudo-abstraction, according to Figure 8.

The reason for introducing ghost-binder when translating rules, instead of when translating terms, is that we may now partially evaluate the rhs, eliminating all pseudo-forms which have been just introduced for convenience. This is the purpose of the next, final phase.

However, before describing partial evaluation, we must generalize the translation function $\mathcal{T}$ to pseudo abstractions and pseudo applications. The translation follows the usual implementation of the λ-calculus (since the body of a pseudo-abstraction is used linearly in β-reduction, the translation can be slightly simplified w.r.t. the general translation of "proper" IS-forms). This is described in Figure 11.

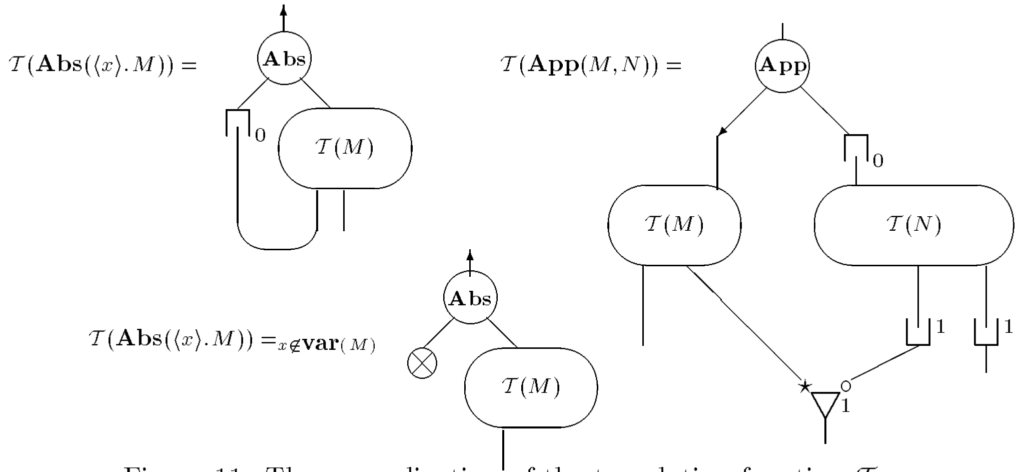
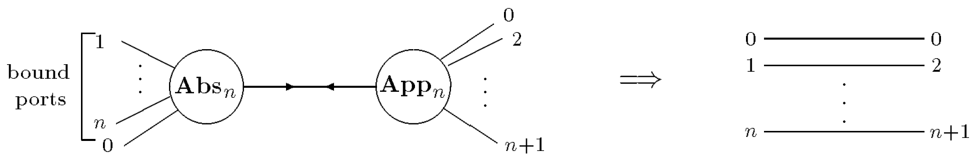


Figure 11: The generalization of the translation function $\mathcal{T}$

Now we can pursue on our running example, namely the implementation of the right hand side of the rule corresponding to redexes μ-λ. In Figure 12 we have depicted the sub-graph corresponding to the first argument of the outermost **App**, i.e. $\mathcal{T}(\mathbf{Abs}(\langle w\rangle.\,\mathbf{App}(w,\,\mu(\lambda(\langle y\rangle.\,\mathbf{App}(w,\,y))))))$.

A final observation before discussing partial evaluation. As already said, some metavariables in the lhs of the IS-rewriting rule could not occur in the rhs (e.g. the case of conditionals). According to the translation of **Abs**, the corresponding pseudo-variable introduced in the linearization step is implemented by an erasing node (since it does not occur in the body of the leftmost outermost **Abs**. This implies that, during the next phase, the corresponding expression is erased by the rule.

(**partial evaluation**) The final step is to partially evaluate the term we have obtained after the translation w.r.t. all pseudo operators. Recall that the reduction rule for pseudo application and abstraction is



Note that n can be 0. Here the rewriting rule simply consists in connecting the two edges coming into the auxiliary ports of $\mathbf{Abs}_0$ and $\mathbf{App}_0$.

Proposition 7.4 *The partial evaluation of the expressions yielded by the translation step strongly normalizes to a graph without pseudo-forms.*

PROOF: After the linearization step, we yield an expression

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

where the expression T exploits only pseudo-forms **App**. Moreover, every occurrence of these pseudo-forms, have the shape $\mathbf{App}_k(w_X,\,M)$. By firing the unique **App**-**Abs** pair in the graph yielded by the translation step, every occurrence of $\mathbf{App}_k(w_X,\,M)$ becomes "almost" a redex. That is, there is a sequence of 1-indexed fan-ins and 1-indexed croissants (0-indexed brackets can be eliminated by means of the control rules) along the path connecting the principal ports of **App**

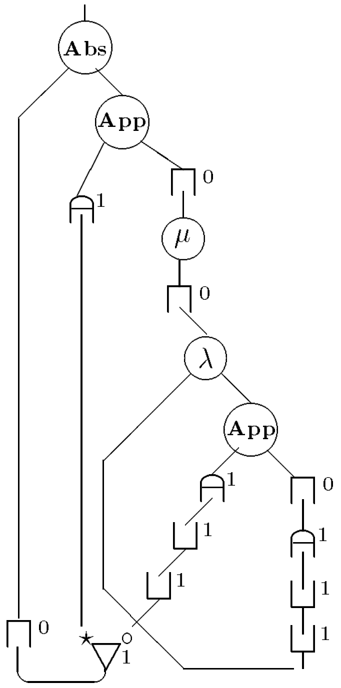


Figure 12: A part of the translation step of the rhs of μ-λ

and **Abs**. These control operators can be pushed inside the **Abs** pseudo-form by means of the interfacing rules. In this way we can fire every pair **App**-**Abs** thus yielding a normal form w.r.t. the partial evaluation. ∎

The above proposition is a more or less obvious consequence of the fact that all pseudo-operators have been created by β-expansions (and the correctness of the translation, of course).

Example 7.5 *By applying the previous technique (and some optimizations not worth discussing here) we obtain the implementation of the rhs of the rule concerning μ illustrated in Figure 13.*

Remark 7.6 *The previous translation could (and should) 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". However, in order to conclude that some operator* **f** *behaves linearly over one of its arguments we must examine* all *the interaction rules involving* **f**. *For instance, if we extend the λ-calculus with the μ operator, considering it as a destructor for λ as in the example above, the body of each λ should be put inside a box, since it can be duplicated when the λ interacts with μ. This is not the case with the other implementations of μ discussed in section 3. Thus, the choice of the rewriting system may have a big impact on the practical efficiency of the implementation.*

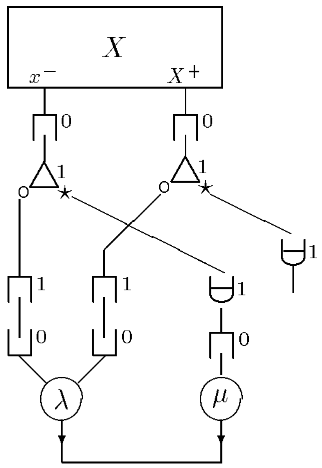


Figure 13: The graphical representation of the rhs of the rule firing μ-λ

8 Correctness

Let N be a sharing graph "representing" an IS-expression t. The implementation described in the previous section is *correct* if a graph-rewriting $N \to N'$ simulates a (possibly empty) set of IS-rewritings $t \twoheadrightarrow t'$ such that t' is the expression "represented" by N'.

It is clear that deriving the IS-expression represented by a sharing graph is an essential prerequisite for stating correctness. This is the so called *read-back* problem. It is solved in [18, 11, 12] by labeling edges of the sharing graphs through *contexts* and interpreting control operators (and forms, in [11, 12]) 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.

8.1 Context semantics and access paths

Definition 8.1 *The set of* contexts *over a set* $\mathbf{X}$ *of variables is inductively generated by the following rules:*

- $\square$ *is a context (the* empty *context);*
- *if* a *is a context then so are* $\circ \cdot a$ *and* $\star \cdot a$*;*
- *if* a *and* b *are contexts then also* $\langle a, b\rangle$ *is a context;*
- *every variable* $\mathbf{x} \in \mathbf{X}$ *is a context.*

Contexts will have the shape $A = \langle \cdots \langle a_n, a_{n-1}\rangle \cdots, a_0\rangle$ and we will say that a_n is the subcontext of A at *width* n (notation $A^n[a_n]$).

Definition 8.2 *Let* $C = \langle A, \langle B_1, B_2\rangle\rangle$*. The context* $\langle A, B_1\rangle$ *will be called the* calling context *of* C*, whilst* B_2 *will be called the* offset context *of* C*.*

Contexts are the data modified by control operators when traversed: as stated in the Definition 8.3 below, control nodes can be easily understood as context transformers. In particular, the traversal of a control node `n` can be forbidden if the external context does not allow the transformation performed by `n`. As a consequence, there are illegal paths in the sharing graph. Exploiting this idea, Lamping provides his read-back procedure (see [19]).

Since in our translation we used Gonthier's simplified set of control operators (and rewriting rules), we tried to generalize the proof in [11] from λ-calculus to IS's. The notion of consistent path in [11] is much more informative and more complex than Lamping's one since also the forms of the syntax (application and abstraction), are regarded as context transformers (actually, for the particular shape of β-reduction, they can be safely assimilated to fans). In particular, a consistent path between an application and a λ corresponds to a *virtual redex* [23, 5, 8] (a virtual redex of a term t is a redex that does not exist yet in t, but that could be created along some derivation from t; the relation between consistent paths and virtual redexes has been recently proved in [2]). The interesting invariant w.r.t. reductions is that *the consistency of a path is not changed by firing control rules in Figure 4* (or β-reductions) [11, 12]. This invariance provides a first rudimentary semantics, named *context semantics*, which gives the soundness of the graph reductions w.r.t. contexts (roughly, since we preserve virtual redexes until they are fired, the implementation respects the intended "operational behaviour" of the term).

Context semantics is still too weak w.r.t. correctness. However it is possible to use it in a judicious way. In particular, in [11], the authors take *Böhm-trees*, a standard semantics of λ-calculus that is invariant w.r.t. reductions and that gives tree-representations of λ-expressions. Then, they prove that the Böhm-tree representing a λ-term can be read-back from the sharing graph by taking *via via* consistent paths that end into the bound port of an "unmatched" abstraction. Due to the context semantics, such paths can be found directly in the initial graph, by starting at the opportune node.

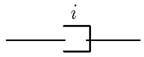
Unfortunately the generality of IS-rewriting rules does not allow any more to consider forms of the syntax as context transformers (surely they cannot be assimilated to fans). This because IS-contractions may introduce new forms and new edges (therefore new paths). As a consequence, IS-virtual redexes cannot be described as *connected paths* in the original term of the derivation, as it is the case in the λ-calculus (see [5]).

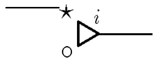
For this reason, our approach will be closer to Lamping's original one, even if we use a set of control operators that is strictly contained in those used by Lamping (so, both the implementation and the proof are quite different). On the other side, the most manifest difference w.r.t. [11] is our read-back procedure: it provides real terms rather than Böhm-trees.

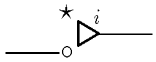
In conclusion, our proof not only generalizes the current approach to a much wider class of rewriting systems, but also, in our opinion, sheds some more light on correctness in the particular case of pure λ-calculus (putting in evidence some "magical" properties of this calculus).

Definition 8.3 (access path) *An* access path *in a sharing graph G is a* directed *path, starting and ending respectively at a negative and positive port and such that every edge of the path is labeled with a context and consecutive pairs of edges satisfy one of the following constraints:*

1. $A^i[b] \quad \text{———}\overset{i}{\sqsubset}\text{———} \quad A^i[\langle b, \Box\rangle]$

2. $A^i[\langle\langle b,\, a\rangle,\, c\rangle]$  $A^i[\langle b,\, \langle a,\, c\rangle\rangle]$

3. $A^i[\langle b,\, a\rangle]$  $A^i[\langle b,\, \star\cdot a\rangle]$

4. $A^i[\langle b,\, a\rangle]$  $A^i[\langle b,\, \circ\cdot a\rangle]$

5. *if the path enters the output port of a form with context $\langle A,\, \langle B,\, C\rangle\rangle$ then it outgoes from an auxiliary negative port of* **f** *with context $\langle A,\, \langle B,\, \mathbf{x}\rangle\rangle$, where* **x** *is a fresh variable.*

Access paths will be taken equivalent up to contexts. That is, two access paths having pairwise equal edges are considered equal, even if the contexts differ.

Note that, it is not possible to traverse a form **f** through one of its bound ports: when a path arrives in front of a bound port, it stops there. Actually the path should continue into the expression that the reduction of **f** substitutes for the bound variable, but we need the evaluation of the term in order to determine, in general, this expression. For this reason we prefer the above solution. The situation is better when we must access to the argument of **f**. In the following we will show that the *meaningful* part of the context at the principal port of **f**, when it is fired, is the same as that marking the output edge of the argument of **f**, if it will be called. So, in item 5, we are able to determine the context at the auxiliary negative port of **f**, provided we know the context at the principal edge.

We also remark that access paths are direct. This because item 5 cannot be defined for undirect ones.

The above definition puts in evidence the important role of context transformers played by the control operators. Let us see with some examples how the contexts, modulo the control nodes, guarantee the proper matching between fans. Consider the sharing graph in Figure 14.(a). In

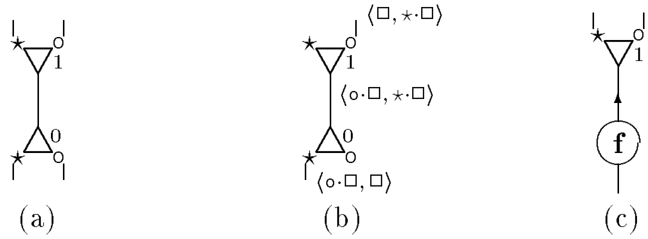


Figure 14: Sharing graphs

Figure 14.(b) we have labeled the $\circ$-branch of the 1-indexed fan, the edge connecting the two fans and the $\star$-branch of the 0-indexed fan such that the corresponding path is an access path. That is the two fans do not match.

The sharing graph in Figure 14.(c) has a node that is a form **f** with one auxiliary edge. By definition, there is an access path that traverses **f** "from the top" (i.e. from the principal port: in this case, **f** must be a constructor and the auxiliary port is negative). Provided that the auxiliary edge is positive, there are two access paths that traverse **f** "from the bottom" to the top: one outgoing the $\star$-branch of the fan and the other outgoing the $\circ$-branch (notice that **f** must be a destructor in this case). We leave to the reader the charge of finding contexts.

Definition 8.4 *A* loop *is an access path that starts at the negative port of a partition of a form and terminates at a positive (bound) port of the same partition.*

8.2 The read-back

The read-back procedure will use the following functions:

access-arg$(\mathtt{n}, C)$ gives the set of *maximal* access paths *never traversing forms* and starting at the argument ports of node n with (an instantiation of the) context C; the number of these paths is exactly the same as the number of arguments of the form n;

end-node(p), end-port(p), end-context(p) give the final node, port (better, number of the node end-node(p)) and final context of the consistent path p, respectively;

bound-endport(p) is *true* if the consistent path p ends at a bound port, *false* otherwise;

bind$(\mathtt{n}, i)$ is *true* if n makes bindings on the i-th argument, *false* otherwise;

connect$(\mathtt{n}, i, \mathtt{m}, j)$ connects with an edge the i-th argument port of the node n with the j-th port of m;

clean-up(C) replaces the offset context of C with a fresh variable;

new-node$(\mathtt{n})$ creates a new node of the same type as n.

Remark 8.5 *Observe that every path in* access-arg$(\mathtt{n}, C)$ *always starts at an argument port of* n *(which is negative) and terminates at an output port of a form or at a bound port. This because these ports are the unique ports of forms having positive polarities and, by definition of access path, their ending edge must be plugged in positive ports.*

The definition of the read-back is in Figure 15. We assume that the number 0 is the index for the output port of forms. The meaning of the arguments of read-back is the following:

- n is the node in the sharing graph we are reading-back;
- C is the context of the access path at the output port of the node n;
- m is the node in the Bourbaki graph where n is read-back into;
- $\mathcal{S}$ is a set of pairs whose first component is a node in the Bourbaki graph and the second one is a context. The set of first projections of $\mathcal{S}$ determines *exactly* the nodes which bind variables that may occur in the arguments of the form **f** represented by m. The node binding a variable occurring in the arguments of **f** will be determined by means of the contexts.

```
procedure read-back(n, C, m, S)
    let access-arg(n, C) = {p_1, ···, p_k} ;
    for i = 1 to k do
        if ¬ bound-endport(p_i) then
            { m' := new-node(end-node(p_i));
            connect(m, i, m', 0);
            if bind(n, i) then S' := S ∪ {⟨m, clean-up(end-context(p_i))⟩};
                else S' := S ;
            read-back(end-node(p_i), clean-up(end-context(p_i)), m', S');
            }
        else { let ⟨m', clean-up(end-context(p_i))⟩ ∈ S ;
            connect(m, i, m', end-port(p_i));
            }
```

Figure 15: The read-back procedure

Informally, $\mathsf{read\text{-}back}(\mathtt{n}, C, \mathtt{m}, \mathcal{S})$ behaves as follows. Initially, it is called with $\mathtt{n}$ as the root node of the sharing graph, $C = \langle \mathbf{x}, \langle \mathbf{y}, \mathbf{z} \rangle \rangle$, $\mathtt{m}$ as the root node of the Bourbaki representation of a term and $\mathcal{S} = \emptyset$. The first call to access-arg gives the path to the first form $\mathtt{n}$ of the sharing graph (this path is unique, since the root node has only one argument). Let $\mathtt{m}$ be the (newly created) node of the Bourbaki representation corresponding to the read-back of the form $\mathtt{n}$. Then we recursively descend in the graph, accessing in order each argument of $\mathtt{n}$. Let us read-back the i-th argument of $\mathtt{n}$. Let p_i be the access path starting at the i-th argument port of $\mathtt{n}$. By definition of access-path, p_i never traverses forms and ends in front of a form $\mathtt{n}' = \mathsf{end\text{-}node}(p_i)$. There are two cases:

1. The node $\mathtt{n}'$ is accessed from the output port. This case is easy: we create a new node $\mathtt{m}'$ corresponding to $\mathtt{n}'$, connect the i-th argument port of $\mathtt{m}$ and the output port of $\mathtt{m}'$. Then we reiterate the read-back with $\mathtt{n}'$, C', $\mathtt{m}'$, $\mathcal{S}'$, where C' is the "cleaning-up" of the ending-context of p_i and $\mathcal{S}'$ is the updating of $\mathcal{S}$ with the pair $\langle \mathtt{m}', C' \rangle$ when $\mathtt{m}$ performs bindings on the i-th argument (the reason for this operation is explained below).

2. The node $\mathtt{n}'$ is accessed from a bound port. Then the binder is one of the nodes got by reading-back $\mathtt{n}'$. The problem is that there could be a lot of them! By the last operation performed in the previous case, this node will eventually appear in the first component of some pair in $\mathcal{S}$. The right binder is determined by the associated context C, namely there exists *exactly one* $\mathtt{m}'$ such that $\langle \mathtt{m}', C' \rangle$ appears in $\mathcal{S}$. This fact is far from evident, and it is based on an essential invariant of the context semantics that we shall prove in the following sections. This invariant states that the calling context at the output port of any binder is the same as the calling context at the bound port. Moreover if two binders are one inside the other (the outer binder performs bindings on the argument where the inner binder appears), they will be accessed with different calling contexts.

Remark 8.6 *Observe that the initial context is built with variables. This allows us to specialize it as we fall inside arguments of forms (which are surrounded by brackets, that means requiring a further level of context). For instance, take the λ-expression $@(x, @(x, x))$ and its sharing graph representation, according to the mapping $\mathcal{T}^+$. Then, starting at the root, we can outgo from the* *second argument of the outermost @. Here we meet a closed 0-indexed bracket. This requires that the 0-level context has the shape* $\langle A, B\rangle$. *Actually this is the case. But after traversing the square bracket, the context becomes* $\langle\langle \mathbf{x}, \mathbf{y}\rangle, \mathbf{z}\rangle$. *Since the property we want to keep is that contexts have always the shape* $\langle A, \langle B, C\rangle\rangle$, *we must specialize* $\mathbf{z}$ *into* $\langle \mathbf{z}_1, \mathbf{z}_2\rangle$ *(intuitively, from the linear logic point of view, we are entering into a new box, i.e. a new area of memory). This operation should not be possible if we started with a context like* $\langle \Box, \langle \Box, \Box\rangle\rangle$, *for example.*

8.3 The correctness of the read-back

In order to prove the correctness of the read-back procedure we shall use the following four properties. Let N be a graph obtained by reducing $\mathcal{T}^+(t)$, for some t.

1. **The transparency property**: *the initial and final calling contexts of every loop in* N *are equal.*

2. **The separation property**: *there exists no access path in* N *starting at an argument edge of a form and ending at the output edge of a form such that the initial and final calling contexts are equal.*

3. **The termination property**: *Every access path in* N *is finite: the access paths that cannot be lengthened terminate at a bound port or at a root node representing a free variable. Hence the procedure* read-back *always terminates.*

4. **The consistency property**: *every access path in* N *can be lengthened consistently in order to start at the root node or at an erasing node and terminate at a bound port or at a free variable.*

The proof of these properties is delayed till the next subsection.

NOTATION: From now on the context $\langle \mathbf{x}, \langle \mathbf{y}, \mathbf{z}\rangle\rangle$ will be denoted by ∇.

Proposition 8.7 *For every form* $\mathtt{n}$ *in* $\mathcal{T}^+(t)$ *there exists a* unique *access path* p *starting at the root of* $\mathcal{T}^+(t)$ *and terminating at the output port of* $\mathtt{n}$ *such that (an instantiation of)* ∇ *is the initial context of* p.

PROOF: Straightforward consequence of the definition of $\mathcal{T}^+$. ∎

Definition 8.8 *The path* p *of the foregoing proposition is called the* spine *of* $\mathtt{n}$. *The notion of spine will be also used in Bourbaki representations (rephrased in the naive way).*

The first statement we show guarantees the static correctness of the read-back procedure w.r.t. our translation $\mathcal{T}^+(t)$.

Proposition 8.9 *Let* t *be an IS-expression and* E *be its Bourbaki representation. Let* $\mathtt{n}$ *be the root node of* $\mathcal{T}^+(t)$ *and* $\mathtt{m}$ *be a new root-node. Then* $\mathsf{read\text{-}back}(\mathtt{n}, C, \mathtt{m}, \emptyset) = E$, *where* $C = \langle A, \langle B, \mathbf{x}\rangle\rangle$, A *and* B *are generic contexts and* $\mathbf{x}$ *is a fresh variable.*

PROOF: Foremost, observe that, by definition of $\mathcal{T}^+$, there is a bijection φ from forms in $\mathcal{T}^+(t)$ and those in E such that, if $\mathtt{n}_1 \cdots \mathtt{n}_k$ is the sequence of forms traversed by the spine of $\mathtt{n}'$, then the spine of $\varphi(\mathtt{n}')$ passes the forms $\varphi(\mathtt{n}_1) \cdots \varphi(\mathtt{n}_k)$ through the corresponding ports. Notice also that the above spines are independent from the initial calling contexts.

We prove the existence of a graph isomorphism between $\textsf{read-back}(\mathtt{n}, C, \mathtt{m}, \emptyset)$ and E that fulfills this bijection.

Let G be the subgraph of E consisting of spines only (so there is no bound edge in G). It is straightforward to check that $\textsf{read-back}(\mathtt{n}, C, \mathtt{m}, \emptyset)$ is correct w.r.t. G since the procedure calculates spines and creates nodes and edges according to them. The termination property guarantees that $\textsf{read-back}(\mathtt{n}, C, \mathtt{m}, \emptyset)$ builds a graph isomorphic to G in a finite time. Moreover, if a binder at spine $p \cdot q$ is in the scope of another one at spine p, by the separation property, the final calling contexts of p and $p \cdot q$ will be accessed with different calling contexts. This means that the sets $\mathcal{S}$, in each thread of the computation of $\textsf{read-back}$, is always a graph of a bijective function.

Hence the interesting case is when the access path e starting at the h-th argument port of a node $\mathtt{n}'$ does not terminate at an output port of a form. Observe that e must terminate at a bound port. Indeed e traverse a sequence of control nodes always from the auxiliary port to the principal port. Such a sequence can be easily proved to be an access path that can be consistently labeled starting with a context as C. Moreover, let $\mathtt{m}'$ be the node related to $\mathtt{n}'$ by the read-back procedure. By definition of $\mathcal{T}^+$, $\mathtt{m}'$ must have a bound edge e' exiting from the h-th argument port.

So, if p is the spine of $\mathtt{n}'$ and $\mathtt{n}^+$ is the node connected to $\mathtt{n}'$ through e, we must be careful that there is a match between the node $\mathtt{m}^+$ calculated for $\mathtt{n}^+$ by the read-back procedure and the node $\varphi(\mathtt{n}^\flat)$ connected to the corresponding port of $\varphi(\mathtt{n}')$ in E. Let $\mathtt{m}^\flat$ be the node in $\textsf{read-back}(\mathtt{n}, C, \mathtt{m}, \emptyset)$ that is related to $\mathtt{n}^\flat$.

Assume that $\mathtt{m}^\flat \neq \mathtt{m}^+$. Let p' be the sub-spine of $\varphi(\mathtt{n}')$ that starts at $\mathtt{n}^\flat$. Then the path $p' \cdot e$ is a loop and, by the transparency property, its ending calling context must be the same as the calling context at the beginning of p'. Moreover $\mathtt{n}^\flat$, by the separation property, is the unique node that holds this property and which is outer than $\mathtt{n}'$. Hence, by definition of the procedure $\textsf{read-back}$, the thread of this procedure that calls $\textsf{read-back}(\mathtt{n}', C, \mathtt{m}', \mathcal{S})$ is such that $\mathcal{S}$ pairs the calling context at the end of e with $\mathtt{m}^\flat$. This contradicts the fact that $\mathtt{m}^\flat \neq \mathtt{m}^+$. ■

Remark 8.10 *Proposition 8.8 does not hold any more in graphs yielded reducing $\mathcal{T}^+(t)$. It is enough that a form* $\mathtt{n}$ *in N is under the scope of a sharing node to find more than one path p verifying the constraints of the proposition. In this case there will be as many instances of* $\mathtt{n}$ *in the Bourbaki graph corresponding to N as the number of spines for* $\mathtt{n}$ *in N.*

Proposition 8.8 gives the "static" correctness of the read-back. The first "dynamic" result is a sort of *weak* context semantics: the invariance of the read-back (or of access paths) w.r.t. control and interfacing rules.

Proposition 8.11 *Access paths (and hence the read-back procedure) are invariant w.r.t. control rules and interfacing rules.*

PROOF: It is easy to verify that every access path traversing a redex in Figure 4 or Figure 9 is such that the contexts before the redex and after the redex do not change along the contraction. ■

Note that the above property is false when proper redexes are considered. Take for instance the λ-expression $@(\lambda(x.\mathbf{I}), M)$. Then there is an access path from the root to the argument M. However, after the reduction, there is no access path between the root and M, since M is disconnected. We warn the reader that, in the following, despite of this limitation, we shall still call *context semantics* (forgetting the prefix “weak”) the invariant of the above proposition.

Next we must prove that, when we are reading back a sharing graph, we are always able to find access paths connecting the ports of two proper forms (see the function access-arg). A sufficient condition for this lengthening being always possible is provided by the absence of deadlock configurations.

Definition 8.12 *A deadlock is when two different control nodes with the same index or a form and a 0-indexed control node are connected through their principal port.*

Remark 8.13 *Observe that two forms may be connected through their principal port without interacting. This deadlock is* intrinsic *to the IS, i.e. it is not due to the implementation.*

The absence of deadlocks is based on the consistency property and the following lemma.

Lemma 8.14 *Let N be a sharing graph yielded by a (possible empty) derivation starting at $\mathcal{T}^+(t)$. In N every auxiliary negative port of a destructor or a constructor faces (or could face, by performing control rules)* exactly one *open 0-indexed bracket (except the pseudo-form* **Abs** *which has no 0-indexed bracket on that branch) and every bound port faces (or could face)* exactly *two open 0-indexed brackets (except* **Abs** *that has one 0-indexed bracket). Never a 0-indexed bracket is in front of a principal port of a form and any other 0-indexed bracket can be erased by firing control rules.*

PROOF: By induction on the length of the derivation σ yielding N. It is easy to verify that $\mathcal{T}^+(t)$ satisfies these properties. For the inductive step, let $N' \xrightarrow{u} N$ be the last reduction of σ. It is immediate to prove that the invariant holds when **u** is a control rule.

When **u** is an interfacing rule, we must check that the control node $\mathtt{n}$ pushed on the auxiliary edges of the form **f** does not invalidate the lemma. By definition of interfacing rule, the index of $\mathtt{n}$ must be greater than 0. Then $\mathtt{n}$ can be removed from the position in front of the auxiliary port of **f** by interacting with the 0-indexed brackets (if any). Notice that the index of $\mathtt{n}$ can never be decreased to 0: according to control rules in Figure 9, 0-indexed control nodes are generated by interacting with 0-indexed croissants only and such croissants are never created.

The case when **u** is a proper rule requires some detailed analysis of the partial evaluation. As a first step we replace the pair destructor-constructor of u with the graph G obtained by the translation step. Observe that the first argument of the outermost **App** in G is obtained by the translation $\mathcal{T}$, so it satisfies the invariant. The inductive hypothesis guarantees that the other arguments of **App** does not invalidate the invariant, too. Let us fire the pseudo-redex. Such contraction amounts to replace pseudo-variables with expressions of the shape $\mathbf{Abs}_n(\vec{x}.X)$. This reduction causes the interaction (and their erasing) of the open 0-indexed bracket on the top of $\mathbf{Abs}_n(\vec{x}.X)$ with the closed 0-indexed bracket in the bottom of the pseudo-variable. The brackets that face the (generic) metavariable X are also eventually deleted (by inductive hypothesis). Now take an access path φ connecting two pseudo-forms **App** and **Abs** (if any). Along φ there are an open 1-indexed croissant and a sequence (possibly empty) of closed 1-indexed brackets and 1-indexed fan-ins. Therefore it is possible to push them outside **Abs**. Notice that, in this way, along the auxiliary edges of the **Abs**-node, we have 1-indexed control operators.

Contracting the new pseudo-redex means that the argument of **Abs** is connected to the external environment (and this connection satisfies trivially the provisos of the induction) and the (auxiliary) arguments of **App** are connected to the bound variables of **Abs**. Along these last connections, we eventually have two 0-indexed brackets that face each other. Thus they can be erased. In this way no 0-indexed bracket can face the principal port of a form. ■

Theorem 8.15 *There is no deadlock in sharing graphs yielded by contracting the sharing graph $\mathcal{T}^+(t)$, for any t.*

PROOF: Assume a deadlock exists in N. Let e be the edge connecting the two nodes $\mathtt{n}_1$ and $\mathtt{n}_2$ of N yielding a deadlock. e is an access path (any edge is an access path). Then, by the consistency property, it can be consistently lengthened. But this is impossible, except in one case (the reader is invited to check this statement). The exception is the configuration of a 0-indexed bracket in front of a form. But such configuration is excluded by Lemma 8.13. ■

Theorem 8.16 *The implementation $\mathcal{T}^+$ is correct. That is, if N is a graph yielded by a derivation starting at $\mathcal{T}^+(t)$ then:*

(A) $t = \mathsf{read\text{-}back}(\mathtt{t}, \nabla, \mathtt{t}', \emptyset)$, *where* $\mathtt{t}$ *is the root node of* $\mathcal{T}^+(t)$ *and* $\mathtt{t}'$ *is a new root node;*

(B) $N \to N'$ *implies* $\mathsf{read\text{-}back}(\mathtt{n}, \nabla, \mathtt{m}, \emptyset) \twoheadrightarrow \mathsf{read\text{-}back}(\mathtt{n}', \nabla, \mathtt{m}', \emptyset)$, *where* $\mathtt{n}$ *and* $\mathtt{n}'$ *are the root nodes of* N *and* N', *respectively;*

(C) N *in normal form implies that also* $\mathsf{read\text{-}back}(\mathtt{n}, \nabla, \mathtt{m}, \emptyset)$ *is in normal form.*

PROOF: Item A is immediate by Proposition 8.8. Item C is easy, assuming B. Indeed, by B and definition of $\mathsf{read\text{-}back}$, every redex in $\mathsf{read\text{-}back}(\mathtt{n}, \nabla, \mathtt{m}, \emptyset)$ should have a counterimage in N which is an access path p connecting two forms. By Theorem 8.14 there is no deadlock along this path, so no control node may appear along p otherwise some interaction could be possible, invalidating the hypothesis that N is in normal form. On the other hand, the two forms connected by p should interact, by definition of the evaluator. And this is in contradiction with the hypothesis, too.

Hence let us discuss the item B. We assume that an IS-reduction

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

is actually composed of two steps: the first

$$\mathbf{d}(\mathbf{c}(\vec{x}^{\,1}.\, X_1, \cdots, \vec{x}^{\,m}.\, X_m), \cdots, \vec{x}^{\,n}.\, X_n) \to \mathbf{App}_n(\mathbf{Abs}_n(M), \mathbf{Abs}_{k_1}(\vec{x}^{\,1}.\, X_1), \cdots, \mathbf{Abs}_{k_n}(\vec{x}^{\,n}.\, X_n))$$

gives the expression obtained by the linearization step and the second, which is a "macro"-step,

$$\mathbf{App}_n(\mathbf{Abs}_n(M), \mathbf{Abs}_{k_1}(\vec{x}^{\,1}.\, X_1), \cdots, \mathbf{Abs}_{k_n}(\vec{x}^{\,n}.\, X_n)) \twoheadrightarrow H$$

where every pseudo-redex is fired. The correctness of $\mathcal{T}^+$ will be proved w.r.t. these two steps. So let $N \to N_1$ be the first macro-step in which the destructor-constructor pair is replaced by the graph obtained by the translation step and $N_1 \to N'$ be the second macro-step in which partial evaluation is accomplished.

Let us verify the correctness of $N \to N_1$. Let p be a spine in N for the destructor involved in the reduction $N \to N'$ and let q be the spine of the corresponding destructor in $E =$ **read-back**($\mathtt{n}$, ∇, $\mathtt{m}$, $\emptyset$). Let C be the context at the end of p and C' be C with the offset context replaced by a fresh variable $\mathbf{x}$. Finally let G be the graph yielded by the translation step of the pair **d**-**c** contracted along $N \to N'$ and let E' be the expression obtained by performing the linearization step of the redex at q in **read-back**($\mathtt{n}$, ∇, $\mathtt{m}$, $\emptyset$).

Then, by Proposition 8.8, the first argument of the outermost **App** is correctly read-back into the first argument of the application at q in the expression E'. For the other arguments, it is enough to instantiate the variable $\mathbf{x}$ in C' with $\langle \mathbf{x}_1, \mathbf{x}_2 \rangle$. The correctness of such arguments follows from the correctness of **read-back**($\mathtt{n}$, ∇, $\mathtt{m}$, $\emptyset$), the definition of G, the correctness of $\mathcal{T}^+$ and Theorem 8.16 (we leave to the reader this check). Notice that this reasoning is parametric w.r.t. the access path p. Indeed we must look for any spine p' in N ending into the redex **d**-**c** and contract any redex in **read-back**($\mathtt{n}$, ∇, $\mathtt{m}$, $\emptyset$) that corresponds to p'. Put E_1 as the final expression. Then $E_1 =$ **read-back**($\mathtt{n}_1$, ∇, $\mathtt{m}_1$, $\emptyset$), where $\mathtt{n}_1$ is the root node of N_1 and $\mathtt{m}_1$ is a new root node.

Let us care for the correctness of the partial evaluation $N_1 \to N'$. Notice that in this macro-step there are two steps of pseudo-reductions: $N_1 \to N_2$ consists of firing the unique pseudo-redex in G, $N_2 \twoheadrightarrow N_3$ consists of the control rules, interfacing rules and the pseudo-redexes that push the metavariables in the right position. Let E_2 be the Bourbaki graph obtained by firing the pseudo redex at spines q' that correspond to the above access paths p' in N_1. Then the correctness of $N_1 \to N_2$ w.r.t. $E_1 \twoheadrightarrow E_2$ can be proved with the same arguing as before plus the transparency property, which guarantees that the graphs connected with the pseudo-variables are accessed in the same way as in N_1.

Now Theorem 8.14 guarantees that the access paths in between the pseudo-forms in N_2 can be rid of control operators. This does not change the read-back, by context semantics. Again, the firing of pseudo-redexes can be proved to be correct by the transparency property. This concludes the proof. ■

8.4 Properties of sharing graphs

8.4.1 The transparency property

The transparency property is the counterpart of the homonymous property of Lamping [19]. In particular, Lamping had a special operator (a global bracket), for dropping the offset near the variable-end of a loop. Rephrasing this idea, with simple syntactical modifications, we obtain a stronger theorem, stating that any loop does not modify the *whole* context. However, this operator is not relevant during the computation; on the contrary, it introduces some annoying problems, since it must be properly "erased" every time we open the loop (when a substitution is performed). So we can safely rid of it provided that the information inside offset contexts is not relevant for connecting access paths. This is what we are going to prove. But let us start by giving some intuition.

Take a graph with a redex μ-λ. Firing this redex results in replacing the subgraph determined by the redex with the instance of the rhs of the rule contracting μ-λ in Figure 13. Let G' be the ending sharing graph. The two 1-indexed fans generated by the rewriting should be paired by reading-back G' (because they are generated by duplicating the same fan along the partial evaluation of the reduction). A sufficient condition for the proper matching of those fans is "what is at level greater than 0 is not modified by traveling inside an expression represented by a metavariable and the two

0-indexed brackets that surround it". This is actually the transparency property.

There is another subtle problem. Before firing the redex μ-λ no access path traversing the bound port of the λ-node does exist. But, after the contraction, the edge e ending into the bound port of the λ is connected with a 0-indexed bracket (see Figure 13). We must prove that this connection is feasible, i.e. there is no conflicting information in the contexts of the two paths. In Theorem 8.16(2) we show (roughly) that the information at the 0-level of the initial context of an access path is not meaningful for its definition. So we can connect two access paths, provided that they have the same meaningful contexts. For instance, in the case of the redex μ-λ, an access path ending into the bound port of λ can be connected with another starting at the form μ.

Theorem 8.17 *Let N be a graph yielded along a derivation starting at $\mathcal{T}^+(t)$.*

(1) **The Transparency Property**: *the initial and final calling contexts of any loop in N are equal.*

In particular, the initial context has always the shape $\langle A, \langle B, \langle B_1, \cdots \langle B_n, \mathbf{x}_{n+1}\rangle \cdots\rangle\rangle\rangle$ and the final context has the shape $\langle A, \langle B, \langle C, \langle B_k \cdots \langle B_n, \mathbf{x}_{n+1}\rangle \cdots\rangle\rangle\rangle\rangle$, for some context C.

(2) *Every access path starting at the auxiliary negative port of a form does not depend from the offset context whose shape is always $\langle B_1, \cdots \langle B_m, \mathbf{x}\rangle \cdots\rangle$ $(m \geq 0)$. That is we always obtain the same path if B_i is replaced by other contexts or $\mathbf{x}$ is instantiated with a context having a fresh variable at level 0.*

Proof: By induction on the length of the derivation σ yielding N (we also count the steps in the partial evaluation). In the basic case both (1) and (2) follow easily by structural induction over $\mathcal{T}$.

For the inductive step, let $N' \xrightarrow{u} N$ be the last reduction of σ. If u is a control rule or an interfacing rule then, by Proposition 8.10, access paths remain unchanged, thus also (1) and (2). The remaining cases are (a) when u is a proper redex (then u consists in replacing the redex with the corresponding graph obtained by the translation step) and (b) u is a pseudo-redex. The cases $(i.j)$, $i \in \{1,2\}$ means what property of the statement we are proving and $j \in \{a,b\}$ means the type of the reduction u, are discussed in order.

$(1.a)$ So N is obtained by N' replacing the redex u with the graph yielded by the translation step of **d**-**c**, the two forms which are interacting. The new loops over pseudo-forms satisfy the property (1) since either they are inside a portion of graph which is defined by means of $\mathcal{T}$ (thus we fall in the basic case), or they are internal to metavariables, and we use the inductive hypothesis (we have just replaced a binder with a pseudo-binder).

The only problematic case is that of a loop φ which starts at a binder **f** external to the redex and passes through a metavariable. Then φ can be split in three parts: the access path φ_1 from **f** to the outermost **App** of the graph yielded by the translation step of **d**-**c**, the access path φ_2 from **App** to the metavariable X, the access path φ_3 internal to the instance of the metavariable. φ_2 traverses in order an open 0-indexed bracket, an **Abs** node and a closed 0-indexed bracket. Therefore it must be that the initial context of φ_2 has the shape $\langle C_1, \langle C_2, C_3\rangle\rangle$, that is guaranteed by hypothesis. If C_3 is a variable $\mathbf{x}$ then, in order to fulfill the property (1), $\mathbf{x}$ must be instantiated into $\langle \mathbf{x}_1, \mathbf{x}_2\rangle$, with $\mathbf{x}_i$ fresh variables. Observe that, in this case, at the end of φ_2, the context is $\langle C_1, \langle C_2, \langle \mathbf{x}_1, \mathbf{x}_2\rangle\rangle\rangle$. By the inductive hypothesis on the property (2), φ_3 may be consistently labeled starting with the context $\langle C_1, \langle C_2, \langle \mathbf{x}_1, \mathbf{x}_2\rangle\rangle\rangle$. Hence $\varphi_1 \cdot \varphi_2 \cdot \varphi_3$ is a loop that verifies the transparency property.

(1.b) Hence u is a pseudo-reduction. That is any loop φ of the node **Abs** contracted by u is "opened": it is connected at one end with the edge e at the output port of the pseudo-form **App** and, at the other end, with the corresponding argument edge e' of **App**. In a sense we have grafted φ inside any access path passing through e and e'.

The problem is again the presence of a free variable x in a metavariable X of **Abs** or **App** bound by an external form **f**. Let us consider the same example as above. Let then φ_1 be the access path from **f** to **App**, φ_2 be the access path from **App** to the metavariable X and φ_3 be the access path inside the metavariable. If X is the body of **Abs** the check of (1) is straightforward because there is no control node in between **App** and **Abs**. Let us see the case when X is an argument h, $h > 1$, of **App**. Notice that, in this case φ_2 is empty.

Let φ_2' be a loop of the node **Abs**. We prove that $\varphi_1 \cdot \varphi_2' \cdot \varphi_3$ is a loop for **f** in N. By induction hypothesis (1), φ_2' does not modify the calling context. This means that the calling context at the beginning of φ_3 is the same as that at the end of φ_1.

Let $\langle C, \langle B_0, \cdots, \langle B_k, \mathbf{x}\rangle \cdots\rangle$, for some k, be the ending context of φ_2'. Recall that the initial context of φ_3 is $\langle C, \langle B_0, \langle \mathbf{x}_1 \cdots \langle \mathbf{x}_r, \mathbf{x}_{r+1}\rangle \cdots\rangle\rangle\rangle$ by inductive hypothesis. Thus take the $\mathbf{max}\{k, r\}$. Let it be k, for instance. Then, by the property (2), we can "specialize" $\mathbf{x}_i$, $1 \leq i \leq r+1$, without altering φ_3. That is $\mathbf{x}_j = B_j$ ($1 \leq j \leq r$) and $\mathbf{x}_{r+1} = \langle B_{r+1}, \cdots, \langle B_k, \mathbf{x}\rangle \cdots\rangle$. Hence $\varphi_1 \cdot \varphi_2' \cdot \varphi_3$ is a loop which enjoys the transparency property.

The case when $r = \mathbf{max}\{k, r\}$ is similar.

(2.a) This is an immediate consequence of the shape of graphs yielded by the translation process, the basic case and the inductive hypothesis.

(2.b) We restrict to access paths created by the pseudo-reduction, since the property is true by induction for the other ones. The pseudo-reduction may create new access paths by

- connecting an access path in the body of **Abs** and terminating at a bound port of **Abs** with an access path starting at argument port of **App**;
- "grafting" some loop ψ over **Abs** inside an old access path traversing **App** and going into some of its arguments.

The reasoning is essentially the same as case (1.b), and it is omitted. ∎

8.4.2 The separation property

Using a linear logic terminology, the separation property guarantees that, if two binders are one inside the other, then the inner one is inside a "deeper" box w.r.t. the outer one. This nesting of boxes means in sharing graph that we find (at least) a 0-indexed bracket in the way connecting the two forms. The 0-indexed bracket allows to properly change the calling context, exactly.

We recall that the argument port of a form is any negative port which is different from the principal port (in the case of destructors).

Theorem 8.18 *(***The separation property***) Let N be a sharing graph obtained by reducing $\mathcal{T}^+(t)$, for some t. There exists no access path in N starting at an argument edge of a form and ending at the output edge of a form such that the initial and final calling contexts are equal.*

PROOF: By induction on the length of the derivation σ yielding N (also counting the steps of the partial evaluation). The basic case is immediate by definition of $\mathcal{T}^+$. The inductive step depends on the last rule $N' \xrightarrow{u} N$ fired. By cases on the last rule u:

1. If u is a control rule the theorem follows by the context semantics.
2. If u is a proper rule the theorem is a consequence of the shape of the graph obtained by the translation process, the basic case and the inductive hypothesis.
3. If u is a pseudo reduction then the separation property is obtained by the inductive hypothesis and the transparency property.
4. The contraction u is an interfacing rule. Let us discuss the case when the access path φ starting at an argument edge of a form $\mathbf{f}$ terminates, without traversing other forms, at the output edge of $\mathbf{g}$ which is involved in the reduction u (the other cases can be reduced to this one). By Lemma 8.13 [1] a 0-indexed bracket $\mathtt{n}$ may eventually face the argument port of $\mathbf{f}$ by contracting control rules only (which do not modify the initial and final contexts of φ). Moreover, by the same lemma, there is no other 0-indexed bracket along φ which cannot be erased by firing control rules. So we assume that φ has exactly one 0-indexed bracket.

   Suppose that φ starts with the context $\langle A, \langle B, \mathbf{x}\rangle\rangle$ and let $\varphi = e \cdot e' \cdot \varphi'$, where e and e' are the edges entering the principal port of $\mathtt{n}$ and outgoing the auxiliary port of $\mathtt{n}$, respectively. The context of e' is $\langle\langle A, B\rangle, \langle \mathbf{x}_1, \mathbf{x}_2\rangle\rangle$, where $\mathbf{x}$ has been instantiated with two new fresh variables. Notice that the variable $\mathbf{x}_1$ now appears in the calling context of e'. The unique way to eliminate the variable $\mathbf{x}_1$ from the calling contexts of edges in φ' is to meet a 0-indexed bracket, which is impossible.

   So the variable $\mathbf{x}_1$ will appear in the calling context of the auxiliary edge of the control node interacting with $\mathbf{g}$. This calling context will mark the output edge of $\mathbf{g}$ after $\mathbf{u}$. Hence it cannot be the same as $\langle A, B\rangle$ since the variable $\mathbf{x}_1$ does not occur inside this latter one. ■

Remark 8.19 *The separation property holds for proper forms only. It is easy to give a counterexample involving an* **Abs** *node and a proper form.*

8.4.3 The termination property

Perhaps the termination property is the most evident property which differentiates our access paths from Gonthier's consistent paths in [11]. Indeed it is easy to find a consistent path in Gonthier's graphical representation of $(\lambda x.\, xx)(\lambda x.\, xx)$ which can be always lengthened (take the path starting at the root node). This is due to the interpretation of forms @ and λ as fans, hence no notion of bound port is present in the above mentioned paper. On the other hand, keeping separated nodes @ and λ from fans underlies our notion of access path (which generalizes the corresponding notion on syntax trees) and, as we have seen, supports generalizations to IS's.

[1] Notice that in the proof of Lemma 8.13 we have used neither the transparency property nor the separation property nor the termination property nor the consistency property. So we can safely use it here.

Theorem 8.20 *(***The termination property***) Let N be obtained by evaluating the sharing graph $\mathcal{T}^+(t)$. Every access path in N is finite: the access paths that cannot be lengthened terminate at a bound port or at a root node representing a free variable. Hence the procedure* read-back *always terminates.*

PROOF: By induction on the length of the derivation yielding N (the steps of the partial evaluation are counted, too). The basic case is proved by a structural induction on the term t. Clearly, by definition of $\mathcal{T}$, every access path is finitely lengthened. Notice that a free variables may become bound at a higher level. This does not change the termination of any access path p ending at the free variable: at the higher level p will end at a bound port.

The inductive step is based on a case analysis of the last rule $\mathbf{u}$ of σ. If $\mathbf{u}$ is a control or interfacing rule then the termination property is an immediate consequence of the context semantics. When $\mathbf{u}$ is a proper rule the statement follows by the shape of the expression replacing the proper redex and the inductive hypothesis. If $\mathbf{u}$ is a pseudo-reduction, the lengthening of paths terminating at the bound port of the **Abs** contracted by $\mathbf{u}$ is finite, by definition of the pseudo-reduction and the inductive hypothesis. ∎

8.4.4 The consistency property

This property excludes the presence of edges that cannot be consistently lengthened. It is clear that such edges are very harmful since, when they are met by the algorithm read-back, it is not possible to pursue the process of reading back on.

Remark 8.21 *In the proof of the consistency property we shall use Proposition 8.8. This is safe because the proof of this proposition is based only on the separation and termination properties.*

Theorem 8.22 *(***The consistency property***) Let N be a sharing graphs obtained by a (possible empty) derivation starting at $\mathcal{T}^+(t)$, for any t. Every access path in N can be lengthened consistently in order to start at the root node or at an erasing node and terminate at a bound port or at a free variable.*

PROOF: We restrict our analysis to the subgraph connected with the root of N. The proof for the disconnected parts is similar (by taking the erasing node instead of the root node). Take a derivation σ yielding N (in σ we also count the reductions of the partial evaluation of rewriting rules). The lemma is proved by induction on the length of σ.

The basic case follows easily by definition of $\mathcal{T}^+$ and Proposition 8.8. The inductive step depends on the last rule $N' \xrightarrow{u} N$ which is fired. If $\mathbf{u}$ is a control rule or an interfacing rule, the lemma is an immediate consequence of the context semantics. Let us see the case of proper rules and pseudo-redexes.

Foremost we must prove that the graph G yielded by the translation rule of some redex **d**-**c** satisfies the lemma when it is properly connected with the interface of the proper redex. Remember that the outermost node of G is a form $\mathbf{App}_k$. Take an access path q internal to the h-th argument of $\mathbf{App}_k$. There are two cases

1. ($h > 1$) The non trivial case is when q is internal to some metavariable. Then, by induction, there exist p_1 and p_2 such that $p_1 \cdot q \cdot p_2$ starts at the output of the metavariable and terminates at some bound edge or a free variable (the case when p_1 starts at an erasing node is immediate).

Let p be the path starting at the root node such that $p \cdot p_1 \cdot q \cdot p_2$ satisfies the constraint of the lemma in N'. Let $\langle A, \langle B, C\rangle\rangle$ be the context which is at the end of p (output port of **d**). So, by definition, p_1 starts with a context $\langle A, \langle B, \mathbf{x}\rangle\rangle$ ($\mathbf{x}$ is a fresh variable). By Theorem 8.16 $p_1 \cdot q \cdot p_2$ does not change if we start with a context $\langle A, \langle B, \langle \mathbf{x}_1, \mathbf{x}_2\rangle\rangle\rangle$, i.e. if we instantiate the variable $\mathbf{x}$. Indeed this instantiation happens at the interface in between the h-th negative argument port of $\mathbf{App}_k$ and the metavariable. Notice also that the transparency property guarantees that an edge of the metavariable bound by **d** (or **c**) can be lengthened till the bound port of the **Abs** form. This terminates the above check in this case.

2. ($h = 1$) The path q is internal to the graph $\mathcal{T}(G)$. By definition of $\mathcal{T}$, there exist p_1 and p_2 such that $p_1 \cdot q \cdot p_2$ starts at the root of $\mathcal{T}(G)$ and terminates at a bound port (there are no free variables in G). Moreover, by Proposition 8.8, $p_1 \cdot q \cdot p_2$ does not change if we begin with the context $\langle A, \langle B, \mathbf{x}\rangle\rangle$, which is the final context of p (see the previous item) [2]. So the path $p \cdot p_1 \cdot q \cdot p_2$ satisfies the lemma.

Finally we have to verify that contracting **App**-**Abs** redexes does not invalidate the lemma. By definition such rule connects the body of the abstraction with the context and the i-th bound port with the $i+1$-th argument of **App**. So we must check that the "grafting" of the body of **Abs** inside the path connecting an argument of **App** with the edge at its output port does not change the property of the lemma. We leave to the reader verifying that this follows immediately by inductive hypothesis, the transparency property and Theorem 8.16(2) (this theorem is needed in order to lengthen an access path terminating at the bound edge of **Abs**). ■

9 Optimality

Optimality can be split into two tasks. The first is the existence of an effective evaluation strategy for IS's which always contracts redexes that any other evaluation strategy should eventually reduce (*call-by-need*). This is easy, since IS's are a subclass of Klop's left-normal Combinatory Reduction Systems and these systems own the property that the leftmost-outermost evaluation order is a call-by-need strategy [15]. The remaining task relies on showing that every redex in the sharing graph always represents a maximal family of redexes in the read-back expression. This can be yielded by switching to labeled expressions.

In particular, let us consider graphical representation of labeled expressions and labeled rewritings as described in Section 4.1. We recall from Section 5 that the initial labeled graph has edges marked by atomic labels and labels are pairwise different (property **INIT**). In the discussion that follows we will allow ourselves a bit of inaccuracy, switching from labeled to unlabeled expressions and back without explicitly stating it.

We must prove that, if two redexes yielded by a labeled derivation have the same label, then they have the same representation in the sharing graph. As in λ-calculus (see [18]), it is possible that duplication of labels goes ahead w.r.t. the reduction of proper redexes. The problem is due to the fact that fan-nodes may duplicate labeled edges (e.g. when a fan is along a redex edge **d**-**c**). In order to cope with such situations, we must determine, for every redex, a part of it that is never

[2] Actually Proposition 8.8 is about $\mathcal{T}^+$. But starting at the root node of M or at the outermost form is the same as far as the context is concerned. Moreover M has no free variable, by definition, so the graphs of $\mathcal{T}^+(M)$ and $\mathcal{T}(M)$ are the same up-to the root node which misses in the latter one.

duplicated by propagation of fans. To this aim, Lamping [19] introduces the notion of *prerequisite chain* of a form $\mathbf{f}$ in the Bourbaki representation (not in the shared graph!). Such notion is smoothly generalizable to IS's.

Definition 9.1 *A* prerequisite chain *for* $\mathbf{f}$ *is a path* in the Bourbaki representation of an expression *starting at the principal port of* $\mathbf{f}$*, ending at the principal port of a form and traversing forms from auxiliary ports to principal ports.*

So, for instance, if $\mathbf{d}$ is a destructor involved in a redex, the prerequisite chain of $\mathbf{d}$ stops at the port of the constructor (it is just an edge). In the expression $\mathbf{d}(\mathbf{d}'(\mathbf{c}))$, where $\mathbf{d}$ and $\mathbf{d}'$ are destructors and $\mathbf{c}$ is a constructor, the prerequisite chain starting at $\mathbf{d}$ consists of the edge connecting $\mathbf{d}$ and $\mathbf{d}'$ and the edge connecting $\mathbf{d}'$ and $\mathbf{c}$. Notice that a prerequisite chain can traverse bound port. This is the case for the prerequisite chain of the innermost @ in $@(\lambda(x.\,@(x,\,M)),\,N)$.

It is clear that the representation of a prerequisite chain in a sharing graph can never be totally duplicated by propagation of fans. Indeed, fans (the control nodes performing duplication) cannot enter the chain from the ends since the principal edges of the ending forms are links of the prerequisite chain. Notice that, when a fan is in between a redex, we have two different prerequisite chains in the read-back subgraph, each corresponding to the two forms on the branches of the fan.

Remark 9.2 *According to Lamping, prerequisite chains are not paths, since bound variables are not connected to their binders. Our graphical representation allow to overcome smartly such problem, thus yielding a simpler definition of prerequisite chain.*

In the next theorem we shall assume, as in the proof of Theorem 8.15, that an IS-reduction

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

is composed of two steps: the first

$$\mathbf{d}(\mathbf{c}(\vec{x}^{\,1}.\,X_1,\,\cdots,\,\vec{x}^{\,m}.\,X_m),\,\cdots,\,\vec{x}^{\,n}.\,X_n) \to \mathbf{App}_n(\mathbf{Abs}_n(\vec{w}.\,M),\,\mathbf{Abs}_{k_1}(\vec{x}^{\,1}.\,X_1),\cdots,\mathbf{Abs}_{k_n}(\vec{x}^{\,n}.\,X_n))$$

gives the expression obtained by the linearization step and the second, which is a "macro"-step,

$$\mathbf{App}_n(\mathbf{Abs}_n(\vec{w}.\,M),\,\mathbf{Abs}_{k_1}(\vec{x}^{\,1}.\,X_1),\cdots,\mathbf{Abs}_{k_n}(\vec{x}^{\,n}.\,X_n)) \twoheadrightarrow H$$

where every pseudo-redex is fired.

The finer analysis of proper reductions allows a correspondingly fine process of labeling. Indeed, the labeling of the rhs of a reduction can be obtained by marking opportunely the expression yielded by the linearization step. To this aim, generalize the labeling function $\mathcal{L}^s_\alpha$ defined in Section 5 with the following rule:

$$\mathcal{L}^s_\alpha(\mathbf{App}_n(w, H_0\,,\cdots,\,H_n)) \;=\; (\alpha)_s(\mathbf{App}_n(w,\,\mathcal{L}^{s0}_\alpha(H_0),\,\cdots,\,\mathcal{L}^{sn}_\alpha(H_n))$$

(notice that the labeling never mark pseudo-variables, which always appear in the first argument of pseudo forms $\mathbf{App}$).

Proposition 9.3 $\mathbf{App}_n(\mathbf{Abs}_n(\vec{w}.\,\mathcal{L}^0_\alpha(M)),\,\mathbf{Abs}_{k_1}(\vec{x}^{\,1}.\,X_1),\cdots,\mathbf{Abs}_{k_n}(\vec{x}^{\,n}.\,X_n)) = \mathcal{L}^0_\alpha(H)$.

That is the labeling $\mathcal{L}^0_\alpha(M)$ gives $\mathcal{L}^0_\alpha(H)$ when pseudo-reductions are performed (we omit the proof because straightforward). Notice that the foregoing marking of M is such that no pseudo redex is ever labeled.

Let us generalize the notion of prerequisite chain in order to be invariant w.r.t. pseudo-reductions. More precisely, we want that the prerequisite-chains in H are the same of those in

$$H' = \mathbf{App}_n(\mathbf{Abs}_n(\vec{w}.\, M),\, \mathbf{Abs}_{k_1}(\vec{x}^{\,1}.\, X_1), \cdots, \mathbf{Abs}_{k_n}(\vec{x}^{\,n}.\, X_n))\,.$$

To this purpose we must decide how to continue a prerequisite chain in H' that arrives in front of a pseudo form. We specify this by cases. Let e be the pseudo-redex in H' and let ψ be a prerequisite chain that arrives in front of a pseudo-form:

1. if ψ enters the output port of the pseud-application $\mathbf{App}_n$ of e then it must exit from the body of the pseudo-abstraction $\mathbf{Abs}_n$ of e and *vice versa*;

2. if ψ enters the i-th argument port of the pseud-application $\mathbf{App}_n$ of e then it must exit from the i-th bound port of the pseudo-abstraction $\mathbf{Abs}_n$ of e and *vice versa*;

3. if ψ passes through the output port of a metavariable then, when it arrives in front of a pseudo-form $\mathbf{App}$ in M, it must exit from the output port and *vice versa*;

4. if ψ passes through the i-th bound port of a metavariable then, when it arrives in front of a pseudo-form $\mathbf{App}$ in M, it must exit from the i-th argument port and *vice versa.*

Proposition 9.4 *The prerequisite chains of an IS-expression where a redex* **d**-**c** *is replaced by the rhs H are the same of those where the redex is replaced by the linearization H' of H. Moreover let t' be the expression obtained from t by replacing H' for the redex* **d**-**c** *and t'' be the expression obtained by contracting e. Constrain every prerequisite chain in t'' that traverses a pseudo-redex e' by the provisos 1 and 2 (where e is replaced by e') above. Then the prerequisite chains in t'' are the same of those in t'.*

PROOF: Easy consequence of the definitions. Indeed prerequisite chains have been defined in order to go ahead the pseudo-reductions. ∎

Definition 9.5 *The label of a prerequisite chain φ beginning at* **f** *is the sequence of labels found along φ, if* **f** *is a destructor, or the reverse of such sequence if* **f** *is a constructor.*

Two prerequisite chains are in the same family *if they traverse in the same order nodes of the same type through the same ports and they have the same labels.*

Remark 9.6 *Actually, in [3] we proved that the label of an edge e in E uniquely determines the types of the forms and the ports where e is connected, provided E is yielded by a derivation starting at an expression holding* **INIT** *(see Theorem 11.5 in the above mentioned paper). Hence, in the above definition, we could remove the constraint that prerequisite chains "traverse in the same order nodes of the same type through the same ports". However, in order to make the paper self-contained, we shall avoid this simplification.*

Theorem 9.7 *Let N be a sharing graph obtained by evaluating $\mathcal{T}^+(t)$ and let E be the (labeled) Bourbaki representation* read-back(n, ∇, m, $\emptyset$)*, where* n *is the root of N and* m *is the root of E. If two prerequisite chains of E are in the same family then they have the same representation in N and* vice versa*, if a path in N is read-back into two prerequisite chains then these latter ones are in the same family.*

Moreover the label ℓ of a prerequisite chain is different from the label of a prerequisite chain in a different family and there is no edge in E which has $(\ell)_s$ as sub-label, for any s.

Proof: As we have said, we shall consider IS-reductions as composed by two "macro" steps, like in Theorem 8.15. The theorem will be proved by induction over the length of the derivation σ yielding N and counting in σ the steps of the partial evaluations.

The basic case is trivial (it is assumed that the Bourbaki representation of t holds **INIT**). Let us prove that the theorem is preserved by any rule $N' \xrightarrow{u} N$ of the evaluator. Let E' be the expression in which N' is read-back.

1. The control rules preserve the paths through the nodes matched by the rule, that is the read-back is invariant w.r.t. them. Thus the theorem is obvious.

2. Among interfacing rules, the interesting case is when a fan interacts with a form. Here the form (together with its edges) is duplicated. Let us see what happens to the prerequisite chains. There are two cases: that of a chain coming into the principal port (and stopping there) and that of a chain coming into an auxiliary port and traversing the principal edge of the form. In both cases the prerequisite chain has to traverse the fan. By induction, the two paths φ and ψ corresponding to the traversal of the two branches of the fan have images which are not in the same family, provided that the images of φ and ψ are prerequisite chains. So the duplication of the form involved in the interfacing rule does not affect the theorem.

3. **u** is a pseudo-reduction. Then the theorem holds because the definition of prerequisite chain is invariant w.r.t. pseudo-reduction.

4. If **u** is a proper rule contracting **d**-**c**. Then u is read-back into a set U of prerequisite chains in E' that, by induction, have the same label ℓ_u and every prerequisite chain in E' labeled by ℓ_u belongs to U. Notice that this property is inherited by the labels $(\ell_u)_s$ created by **u**. This is a consequence of the inductive hypothesis. Indeed if a prerequisite chain ξ had label $(\ell_u)_s$ then ξ should be a single edge, because, by definition of label of prerequisite chain, if ξ consisted of several edges the label should be $\ell_1 \cdots \ell_n$. But no edge is labeled by $(\ell_u)_s$, by induction, exactly.

   Let E be the expression obtained from E' by replacing the redexes in U with the expression $\mathbf{App}_n(\mathbf{Abs}_n(M), \mathbf{Abs}_{k_1}(\vec{x}^{\,1}. X_1), \cdots, \mathbf{Abs}_{k_n}(\vec{x}^{\,n}. X_n))$. Take two prerequisite chains φ and ψ in E which are in the same family. Then both φ and ψ traverse (the instances of) M or do not traverse one instance of M. This because edges in M are marked by $(\ell_u)_s$ and such labels do not appear elsewhere.

   Let φ' and ψ' be the paths in N which are read-back into φ and ψ, respectively. Assume $\varphi' \neq \psi'$ and let a be the first edge of φ' which is different from the corresponding edge b of ψ'. W.l.o.g. we can reduce to the cases when a and b are inside the graphical representation of M or they are both outside M and are not connected with pseudo-forms.

If a and b are inside the graphical representation of M then it is not possible that their read-back edges in E have the same labels, since pairwise different edges of M are read-back into different labeled edges, by definition (the case when a and b are bound edges of the abstraction $\mathbf{Abs}_n(M)$ is not primitive and can be easily reduced to the previous one).

If a and b are outside M the property follows by inductive hypothesis. Indeed, in this case, exactly one of the following items must hold for the ancestors in N' of the prerequisite chains of φ' and ψ':

- they terminate in front of one of the forms involved in u;
- they do not traverse u at all.

In both cases it is possible to prove a contradiction with the fact that families of prerequisite chains in E' are uniquely represented in N'.

The other direction, i.e. prerequisite chains that are read-back from the same path are in the same family, can be proved by reverting the above reasoning.

Finally we must verify that the labels ℓ_φ and ℓ_ψ of two prerequisite chains φ and ψ in E are different, provided that φ and ψ are in different families. Moreover no edge is labeled by $(\ell_\varphi)_s$, for every s. There is a case analysis:

(a) if φ and ψ are not affected by $\mathbf{u}$ (i.e. they are in the context or in some metavariable) then the property follows by induction. Notice that no edge in E is labeled by $(\ell_\varphi)_s$: this follows by induction or because the edges created (or modified) by $\mathbf{u}$ have $(\ell_u)_{s'}$ as sub-label.

(b) φ is created by the contraction of the redexes in U (i.e. φ is entirely inside a copy of M). Then ℓ_φ is a sequence of labels $(\ell_u)_s$, with different s. If also ψ is inside a copy of M then the property follows by definition of the labeling of M. Otherwise ψ must traverse an auxiliary edge v of a pseudo application in M. These edges, by definition of $\mathcal{L}$, are marked by $(\ell_u)_s$, for some s. Let $(\ell_u)_v$ be the label of v. φ cannot traverse v otherwise it could not be inside M. Hence the label $(\ell_u)_v$ does not occur in ℓ_φ, therefore $\ell_\psi \neq \ell_\varphi$. It is straightforward to check that there is no edge in E having $(\ell_\varphi)_s$ as sub-label.

(c) both φ and ψ traverse a copy of M. The reasoning of this case is similar to the one used for proving the first part of the theorem. So we omit it.

This terminates the proof. ∎

Observe that redexes are prerequisite chains. Hence, an immediate consequence of the above theorem is that redexes having the same label will be represented by the same structure in the corresponding sharing graph.

Corollary 9.8 *In every derivation of the sharing graph $\mathcal{T}^+(t)$ no two proper redexes are read-back into edges with the same label. Therefore the graph implementation is optimal.* ∎

10 Conclusions

We have generalized Lamping's optimal graph-reduction technique [19] to a new class of higher order term rewriting systems: Interaction Systems. In Interaction Systems, we may define most of the common data structures used in practice (in particular, all inductive types) and many useful constructs of real programming languages (jumps, recursions, and so on). Actually, the only constraint of IS seems to be *local sequentiality* (in the sense of Berry). For instance, the parallel or is not expressible in IS.

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. This is a consequence of their *logical* (intuitionistic) nature, which has been deeply investigated in this paper.

Interaction Systems are a subclass of Klop's orthogonal Combinatory Reduction Systems. So the expected extension of our work is the generalization of the results described here to orthogonal Combinatory Reduction Systems. A prerequisite for fulfilling this aim is the definition of the family relation. In particular the degree of a redex becomes much more involved since the label of a tree (instead of an edge) must be taken into account.

For the same reason, we cannot hope to describe the optimal implementation of CRS in the form of an Interaction Net (since forms do not have principal ports, the decision if traversing a form with a fan cannot be local any more, but it depends from the context surrounding the form). Another problem is matching lhs of rewriting rules with the graph representing the term (if we unfold the term, we could loose sharing; if not, the sharing in the lhs must be preserved in the rhs, furtherly complicating the rewriting step). Moreover, where the translation should put boxes? Is there any linearity in CRS's? When should a box be opened? All these questions have a natural answer in IS's, due to their intuitionistic nature.

The main problem of "optimal" implementations is, however, the *efficiency*. In particular, the accumulation of control operators (that could be exponential). During reduction, we may create sequences of control operators whose global control effect could be neglected. Thus the sequence of control operators could be safely removed. Unfortunately, this simplification can be performed only in suitable positions of the graph, without affecting Church-Rosser. Individuating these positions (and the configurations to be be reduced) does not seems to be an easy task. Some work in this direction was already done by Lamping, but his rewriting rules are not complete.

Burroni [7] and Lafont [17] have recently 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 is a first attempt of generalizing Burroni-Lafont's works to higher order rewriting systems. The richer set of control operators is motivated by the presence of binding and substitution. By exploiting this analogy, one could also imagine to provide a more algebraic account of optimality.

Acknowledgements: We would like to thank J.J. Lévy and G. Gonthier for the interesting discussions.

References

[1] P. Aczel. A general Church-Rosser theorem. Draft, Manchester, 1978.

[2] A. Asperti, V. Danos, C. Laneve, and L. Regnier. Paths in the λ-calculus. In *Proceedings* 9^{th} *Annual Symposium on Logic in Computer Science,* Paris, pages 426 – 436, 1994.

[3] A. Asperti and C. Laneve. Interaction Systems I: the theory of optimal reductions. Technical Report 1748, INRIA-Rocquencourt, September 1992. A revised version will appear on the journal *Mathematical Structures in Computer Science.*

[4] A. Asperti and C. Laneve. Optimal Reductions in Interaction Systems. In *TapSoft '93*, volume 668 of *Lecture Notes in Computer Science*, pages 485 – 500. Springer-Verlag, 1993.

[5] A. Asperti and C. Laneve. Paths, Computations and Labels in the λ-calculus. In *RTA '93*, volume 690 of *Lecture Notes in Computer Science*, pages 152 – 167. Springer-Verlag, 1993.

[6] N. Bourbaki. *Théorie des ensembles.* Hermann & C. Editeurs, 1954.

[7] A. Burroni. Higher dimensional word problems. In *Category Theory and Computer Science*, volume 530 of *Lecture Notes in Computer Science*, pages 94 – 105. Springer-Verlag, 1991.

[8] V. Danos and L. Regnier. Local and asynchronous beta-reduction. In *Proceedings* 8^{th} *Annual Symposium on Logic in Com puter Science,* Montreal, pages 296 – 306, 1993.

[9] J. Field. On laziness and optimality in lambda interpreters: tools for specification and analysis. In *Proceedings* 17^{th} *ACM Symposium on Principles of Programmining Languages*, pages 1 – 15, 1990.

[10] J. Y. Girard. Linear Logic. *Theoretical Computer Science*, 50, 1986.

[11] G. Gonthier, M. Abadi, and J.J. Lévy. The geometry of optimal lambda reduction. In *Proceedings* 19^{th} *ACM Symposium on Principles of Programmining Languages*, pages 15 – 26, 1992.

[12] G. Gonthier, M. Abadi, and J.J. Lévy. Linear logic without boxes. In *Proceedings* 7^{th} *Annual Symposium on Logic in Computer Science*, 1992.

[13] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International, Englewood Cliffs, New Jersey, 1987.

[14] V. Kathail. *Optimal Interpreters for lambda-calculus based functional languages.* PhD thesis, MIT, 1990.

[15] J. W. Klop. *Combinatory Reduction System.* PhD thesis, Mathematisch Centrum, Amsterdam, 1980.

[16] Y. Lafont. Interaction Nets. In *Proceedings* 17^{th} *ACM Symposium on Principles of Programmining Languages*, pages 95 – 108, 1990.

[17] Y. Lafont. Penrose diagrams and 2-dimensional rewritings. In *LMS Symposium on Applications of Categories in Computer Science.* Cambridge University Press, 1992.

[18] J. Lamping. An algorithm for optimal lambda calculus reductions. Technical report, Xerox PARC, 1989.

[19] J. Lamping. An algorithm for optimal lambda calculus reductions. In *Proceedings* 17^{th} *ACM Symposium on Principles of Programmining Languages*, pages 16 – 30, 1990.

[20] C. Laneve. *Optimality and Concurrency in Interaction Systems*. PhD thesis, Dip. Informatica, Università di Pisa, March 1993. Technical Report TD – 8/93.

[21] J.J. Lévy. *Réductions correctes et optimales dans le lambda calcul.* PhD thesis, Université Paris VII, 1978.

[22] 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.

[23] L. Regnier. *Lambda Calcul et Réseaux.* PhD thesis, Université Paris VII, 1992.