The Bologna Optimal Higher-order Machine

ANDREA ASPERTI, CECILIA GIOVANNETTI
and ANDREA NALETTO

Dipartimento di Matematica, P.zza di Porta S.Donato 5, Bologna, Italy
(*e-mail:*{asperti,giovanne,naletto}@cs.unibo.it)

Abstract

The Bologna Optimal Higher-order Machine (BOHM) is a prototype implementation of the core of a functional language based on (a variant of) Lamping's optimal graph reduction technique (Lamping, 1990; Gonthier *et al.*, 1992a; Asperti, 1994). The source language is a sugared λ-calculus enriched with booleans, integers, lists and basic operations on these data types (following the guidelines of Interaction Systems – Asperti and Laneve (1993b, 1994), Laneve (1993)). In this paper, we shall describe BOHM's general architecture (comprising the garbage collector), and give a large set of benchmarks and experimental results.

Capsule Review

As lambda terms represent functional programs people have studied ways to evaluate them efficiently. It can be proved that just reducing terms step-by-step, without having other information, is not an optimal reduction strategy. Lévy (1978) has shown that using extra information gives an optimal reduction strategy (in a certain class of strategies using information). (But then, Albert Meyer has shown (unpublished) that there is no optimal strategy if one uses arbitrary Turing computable tools.) So what matters is efficiency.

The present paper describes an implementation based on an extension of ideas in Lamping (1990) by Asperti and Laneve. In a benchmark, using purposely non-optimised algorithms, it is shown that the described implementation works rather well. The resulting optimism of the authors may be compared to the more pessimistic view of Lawall and Mairson in their 'What isn't a cost model of the lambda calculus?' (1996 *ACM International Conference on Functional Programming*, pp. 92–101). This paper, available from

http://www.cs.brandeis.edu/ mairson/Papers/icfp.ps.gz

gives an overview of several approaches, including ones related to the underlying paper.

1 Introduction

This paper describes the main architecture of the Bologna Optimal Higher-order Machine (BOHM). BOHM is a prototype implementation of an extension of Lamping's optimal graph reduction technique (Lamping, 1990) for reducing λ-expressions. The source language is a sugared λ-calculus enriched with booleans, integers, lists and basic operations on these data types. The extension of Lamping's techniques to this language is essentially based on Asperti and Laneve's work on Interaction

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Systems (Asperti and Laneve, 1993b, 1994; Laneve, 1993). In particular, all syntactical operators are represented as nodes in a graph. These nodes are divided into *constructors* and *destructors*, and reduction is expressed as a local interaction (graph rewriting) between constructor-destructor pairs.

BOHM is just a high level interpreter written in C. Its purpose was not to be a real implementation, but to provide some empirical evidence about the feasibility of Lamping's technique (that, in our opinion, looks *very* promising). From this respect, some care has been devoted in supporting the user with a large set of data relative to the computation of each term (user and system time, number of interactions, storage allocation, garbage collection, and so on). The source code is available by anonymous ftp at `ftp.cs.unibo.it`, in the directory `/pub/asperti`, under the name `BOHM.tar.Z` (compressed tar format).

In this paper, we shall describe in some detail the general architecture of BOHM (comprising the garbage collector), and give a large set of benchmarks and experimental results. No theoretical topic is discussed; in particular, we shall not address the correctness and optimality issues, since they have been already proved in Asperti and Laneve (1993b).

The structure of the paper is as follows. In the next section we introduce the problem of optimal reduction of λ-terms, and Lamping's graph reduction technique. This section is rather sketchy – the paper requires some knowledge of the problem and of Lamping's algorithm. Section 3 is devoted to an introduction to interaction systems, which provide the starting point for the generalization of Lamping's technique to a richer source language. In section 4, we define BOHM's source language, its operational semantics in the form of an interaction system, the initial graph encoding of each term and the graph reduction rules corresponding to the rewriting rules of the Interaction System. Section 5 discusses the crucial notion of a safe operator, which allows us to introduce some essential optimizations into the machine. Section 6 is devoted to the garbage collector. Finally, in section 7 we present a large number of experimental results about BOHM's performance and its garbage collector.

2 Optimal reduction

The concept of optimal reduction of the λ-calculus was defined by Lévy (1978). The idea was that of formalizing the intuitive notion of 'optimal sharing' of reducible expressions, implicitly defining a lower bound to the 'intrinsic complexity' of λ-term reduction.

Lévy approached the problem of sharing from many different perspectives (copy-relation, labelling, extraction), all of them leading to the unique and crucial notion of *redex family* (see also Lévy (1980) and Asperti and Laneve (1993a)). Two redex are 'sharable' if and only if they belong to the same family. So, a redex family is composed by 'copies' of a same redex, and could be conceptually represented by a unique object (and reduced in a single atomical step). In this sense, the length of the family reduction (i.e. the 'parallel' reduction where at each step a whole redex family is reduced) would provide the above-mentioned lower bound to the complexity of λ-term reduction, *in any possible implementation.*

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

The first and natural problem raised by Lévy's thesis was that of getting some empirical evidence that his abstract notion of sharing could be feasible: in fact, no 'traditional' implementation of λ-calculus (supercombinators, environment machines, etc.) is able to share all redexes in Lévy's families. The problem challenged the functional community for over ten years, before Lamping (1990) proposed a solution based on a complex graph rewriting technique. However, Lamping did not establish any relation between the complexity of his reduction technique and the length of the family reduction. This looks like a quite different problem, since we must obviously take into account both the computational overhead introduced for the correct handling of optimal sharing, and the fact that in any case β-reduction is not an *atomic operation*, and thus any complexity measure based on this notion could eventually be misleading. So, the very abstract problem of establishing a concrete measure for the *intrinsic* complexity of λ-term reduction is still far from being solved.

Unfortunately, Lamping did not provide any computational result or benchmark of his implementation (if he had any), making it impossible to evaluate the practical interest of his technique with respect to more traditional implementation architectures. The first empirical results on (a slight variant of) Lamping's technique appeared in Asperti (1995). However, those empirical results (that looked very promising) suffered from the limitation of being relative to *pure λ-terms*, and could not be seriously considered as real 'benchmarks' for a full functional language.

For this reason, we decided to extend the prototype in Asperti (1995) to a reacher source language, adding some primitive data types (boolean, integers, lists) equipped by the usual set of operations, and a few more control structures (if-then-else, letrec, etc.). BOHM is the result of this extension. Although BOHM's source language is still very limited, it contains the full core of a 'real' functional language. In this sense, BOHM can be considered as the first empirical and practical test for optimal implementation techniques.

2.1 Graph reduction

The aim of an optimal reduction technique is that of avoiding, during the computation of a term, any duplication of reducible expressions (that would obviously imply duplication of work). The main problem is that, in higher order languages, redexes are created along the reduction in a quite complex way. So, the problem amounts not only to avoiding the duplication of 'actual' redexes at a given stage of the computation, but also the duplication of all those substructures that are *not still* a redex, but *may become* a redex later on, due to an arbitrary long sequence of reductions ('virtual' redexes).

Since, for optimal reduction, we must get some notion of physical sharing of substructures, it seems natural to look for some graph representation of λ-terms. The first work in such a direction was due to Wadsworth (1971), where λ-terms are represented as directed acyclic graphs (essentially representing their abstract syntax tree, plus the sharing information for variables). Unfortunately, Wadsworth's

technique is not optimal. The problem is that, 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 it will be instantiated by the β-reduction r, we must eventually work on a new copy (see Peyton Jones (1987)). Even if you normalize F before its duplication, you can eventually lose sharing, at this point. Suppose, for instance, we 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.

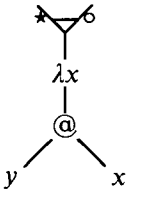
Lévy proved that this problem cannot be solved by the choice of a suitable reduction strategy. In fact, there are terms where *every* order of reduction would duplicate work. For instance, consider the following term (Lévy, 1978, p. 15):

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

where a is some constant, and the two sequences of x have both length n. **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 we just have exactly four redex families in this example!

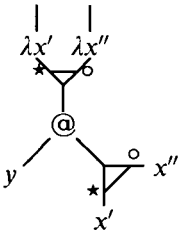
2.2 Lamping's solution

Asperti and Laneve (1993a) clarified that all redexes in the same family define a unique common *path* in the initial term of the derivation. In a sense, these paths are the 'physical' information that *must be shared* in optimal reduction systems. Lamping's idea was to define a set of local rewriting rules, on a suitable graph representation of λ-terms, that avoided duplications of these paths without preventing the reduction of the term to its normal form. Roughly, in Lamping's system, the duplication of a given term M is performed in a sort of 'lazy' way, by propagating a duplication operator (a fan) along the graph structure of M, and stopping this propagation at suitable 'critical' places (typically, just before the applications in M). Suppose, for instance, we 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 duplication of the binder implies 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:

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press



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. Note that the 'virtual' redex between the application and the unbound variable y is kept shared by this representation, while the two λ of the two instances of M have been put in evidence by the partial duplication we have performed so far, and they can now possibly interact with the surrounding context.

As noted above, when a fan interacts with a λ, a new fan (a fan-out) appears on the edge representing the bound variable. To get a simple implementation of this rule, it is convenient to have a direct connection through an explicit link between the bound variable and its binder. So, λ becomes now a ternary node, with two sons respectively pointing to the body and the bound variable. With this representation (that goes back to Bourbaki), the interaction between a λ and a fan is expressed by the rewriting rule shown in Figure 1.

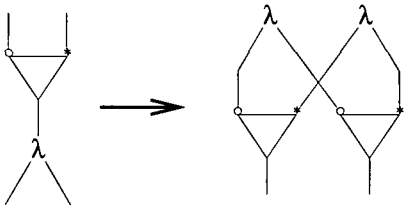


Fig. 1. Fan-lambda interaction.

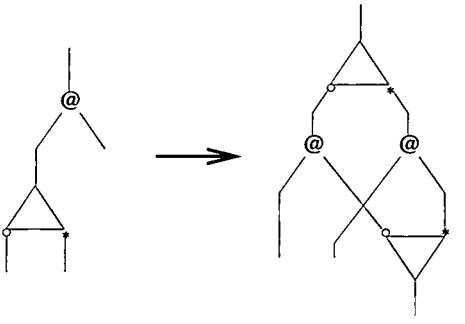


Fig. 2. Fan-application interaction.

Note that both λ and application nodes (forms) have a unique and distinguished 'active' position, where they could possibly interact with other nodes. This position (in Lafont's notation) is called the *principal port* of the form. All other ports of the

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

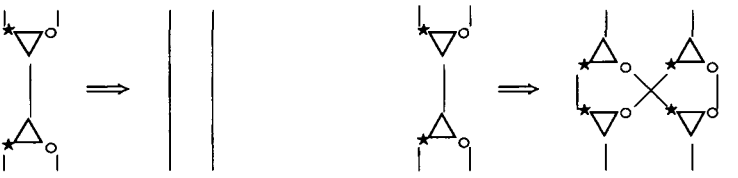
form are called *auxiliary*. The main law behind the optimal implementation strategy is the following:

(**main law**) a fan may duplicate a form if and only if it reaches the form at its principal port.

We have already provided the duplication rule for λ; the analogous rule for application is shown in Figure 2.

2.3 The problem of fans

The only problem with fans is to understand what happens when they meet 'face to face' during the reduction. Two reductions seem to be possible in this case:



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 a same 'duplication process' (i.e. the two fans are residual of a same duplication operator created by its interaction with a λ-abstraction). In all other cases, fans should 'mutually duplicate' each other, according to the right rule, above.

Unfortunately, there does not seem to be a simple way to solve the above ambiguity. At the present state-of-the-art, the solution is that of associating an integer with each fan, denoting its *sharing-level* or, if you like, 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 the same level, and according to the second rule, if their levels are different.

Matters are complicated further by the fact that levels may change dynamically during the computation. This requires the introduction of two more (indexed) control operators (brackets and croissants) for increasing and decreasing levels along the computation. The idea is that brackets and croissants define the 'interaction context' of a fan in such a way that two fans in different contexts will never be paired. From this point of view, the integer associated with each form represents the nesting level of the context; a croissant indicates that we are entering a new context and the square bracket that we are exiting from it. Operationally, we need two different operators to this purpose, since they propagate in opposite directions.

3 Interaction systems

The main problem in extending Lamping's optimal graph reduction technique from λ-calculus to a richer source language is posed by the *main rule* of section 2.2. In fact, to have a clean and local rewriting system for optimal reduction, we need all syntactical forms of our generalized source language to preserve the property of

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

having a distinguished *principal port*, where they can possibly interact with other forms (what is called 'local sequentiality' by Lafont).

To this end, Asperti and Laneve (1993b, 1994; Laneve, 1993) introduced the notion of Interaction Systems (IS).

Formally, there are two possible ways to understand Interaction Systems (IS) (Asperti and Laneve, 1993b, 1994; Laneve, 1993). From one side they are the *intuitionistic* generalization of Lafont's (1990) Interaction Nets, from which they borrow the logical setting, the bipartition of operators into *constructors* and *destructors*, and the principle of binary interaction. From the other side, ISs can be seen as a subclass of Klop's (1980) Combinatory Reduction Systems (CRS), where pattern matching in the left-hand side of a higher order rewriting rule is restricted to a very specific and simple form of binary interaction between *dual* syntactical forms. As a main corollary of this assumption, we can immediately generalize the Curry–Howard (Proofs as Propositions) analogy from lambda calculus to an arbitrary IS. This means we can associate every IS with a suitable 'intuitionistic' system: constructors and destructors respectively correspond to right and left introduction rules, interaction is cut, and computation is cut-elimination. As a consequence, ISs have a nice *locally sequential nature*: in particular, the leftmost outermost reduction strategy only reduces needed redexes.

Due to their logical (intuitionistic) nature, ISs are particularly suited to a generalization of Lamping's technique. In fact, from Gonthier *et al.* (1992a, b), it was clear that the 'core' of Lamping's algorithm (fans, croissants, brackets and their mutual interactions) provided a very abstract set of operations for implementing the *structural part* of Intuitionistic Systems (i.e. via the Curry–Howard analogy, the part charged with the management and sharing of resources in functional languages). This becomes particularly clear when we consider logical systems such as linear logic (Girard, 1986), where we have a neat distinction between the logical and structural part. The only purpose of Lamping's control operators is that of providing a completely local (and optimal) implementation of the structural operations relative to the *box* of linear logic. Then, the natural idea behind ISs was that of replacing the *logical part* of intuitionistic logic (namely, abstraction, application and their interaction through β-reduction), with a richer collection of dual operators, binarily interacting via cuts. In other words, we had to replace *linear* λ-calculus with a richer *linear* calculus, that was naturally provided by Lafont's interaction nets. In this way, we could keep Lamping's core, just adding the new 'logical' rules proper to the system. Interfacing 'structural' and 'logical' rules is then a trivial task.

3.1 The Formal definition

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

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Each form of the syntax can be a binder. In the arity of the form we must specify not only the number of arguments, but also, for each argument, the number of bound variables. Thus, the *arity* of a form **f** is a finite (possibly empty) sequence of naturals. 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 (1990) notation, at the first argument we find the *principal port* of the destructor, that is the (unique) port where we will have interaction (local sequentiality).

Expressions, ranged over by $t, t_1, \cdots$, are inductively generated as follows:

(a) every variable is an expression;
(b) if **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).

Rewriting rules are described by using schemas or *metaexpressions*. A metaexpression is an expression with *metavariables*, ranged over by $X, Y, \cdots$, possibly indexed (see Aczel (1978) 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 **d** is $0k_{m+1} \cdots k_n$ and that of **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:

$$H \quad ::= \quad x \quad | \quad \mathbf{f}(\vec{x}^1_{a_1}.H_1, \cdots, \vec{x}^j_{a_j}.H_j) \quad | \quad X_i[{}^{H_1}/_{x^1_1}, \cdots, {}^{H_{k_i}}/_{x^1_{k_i}}]$$

The expression $X[{}^{H_1}/_{x_1}, \cdots, {}^{H_n}/_{x_n}]$ denotes a meta-operation of substitution, as in λ-calculus, and defined in the obvious way.

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

Interaction Systems are a subsystem of Klop's (Orthogonal) Combinatory Reduction Systems (Klop, 1980; Aczel, 1978). We just added a bipartition of operators into constructors and destructors, and imposed a suitable constraint on the shape of the lhs of each rule. As a subclass of non ambiguous, left-linear CRSs, Interaction Systems inherit all good properties of their predecessors (Church–Rosser, finite development, etc.).

Example 3.1
(*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:

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

□

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Example 3.2
(*Booleans*) Booleans are defined by two constructors **T** and **F** of arity ε (two constants). Then you may add your favourite destructors. For instance, the logical conjunction is a destructor **and** of arity 00, with the following rewriting rules:

$$\mathbf{and}(\mathbf{T}, X) \to X \qquad \mathbf{and}(\mathbf{F}, X) \to \mathbf{F}$$

Another typical destructor is the *if-then-else* operator **ite**, of arity 000:

$$\mathbf{ite}(\mathbf{T}, X, Y) \to X \qquad \mathbf{ite}(\mathbf{F}, X, Y) \to Y$$

□

Example 3.3
(*Lists*) Lists are defined by two constructors **nil** and **cons** of arity ε and 00, respectively. The typical destructors **hd**, **tl** and **isnil** (all of arity 0) may be defined by the following rules:

$$\mathbf{hd}((\mathbf{cons}(X, Y)) \to X \qquad \mathbf{tl}((\mathbf{cons}(X, Y)) \to Y$$

$$\mathbf{isnil}(\mathbf{nil}) \to T \qquad \mathbf{isnil}((\mathbf{cons}(X, Y)) \to F$$

□

Example 3.4
(*Integers*) For arithmetics, we consider each integer n as a distinguished constructor **n**. Then, we may define arithmetical operations in constant time. The only problem is the strictly sequential nature of ISs, that imposes interaction on a distinguished port of the form. For instance, we may define

$$\mathbf{add}(\mathbf{m}, X) \to \mathbf{add}_{\mathbf{m}}(X) \qquad \mathbf{add}_{\mathbf{m}}(\mathbf{n}) \to \mathbf{k}$$

where $\mathtt{k} = \mathtt{n}+\mathtt{m}$. Note that we have an infinite number of forms, and also an infinite number of rewriting rules. □

Example 3.5
(*General recursion*) The recursion operator $\mu x.M \to M[^{\mu x.M}/_x]$ is a bit problematic, since ISs are based on a principle of *binary* interaction. The obvious idea is to introduce a suitable 'dummy' operator interacting with μ. This 'dummy' operator may be indifferently regarded as a constructor or a destructor, yielding the following encodings for μ:

$$\begin{aligned} \mathbf{d}_\mu(\mu(x.X)) &\to X[^{\mathbf{d}_\mu(\mu(x.X))}/_x] \\ \mu(\mathbf{c}_\mu, x.X) &\to X[^{\mu(\mathbf{c}_\mu, x.X)}/_x] \end{aligned}$$

In other words, we should look at μ as a constructor-destructor pair, permanently interacting with itself. □

3.2 The intuitionistic nature of ISs

In this section we shall recall the logical, intuitionistic nature of Interaction Systems, and their relation with Lafont's interaction nets. The aim of this section is to

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

introduce and clarify some essential notions of IS-forms (polarities, principal and auxiliary ports, bound ports, inputs and outputs, etc.).

3.2.1 Statics

An intuitionistic system, in a *sequent calculus* presentation (*à la* Gentzen) (see Girard *et al.*, 1989), consists of expressions, named *sequents*, whose shape is $A_1, \cdots A_n \vdash B$, where A_i and B are formulas. Inference rules are partitioned into three groups (to emphasize the relationships with ISs, we write rules by assigning terms to proofs):

Structural rules

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

$$(\textit{Contr.}) \quad \frac{\Gamma, x : A, y : A \vdash t : C}{\Gamma, z : A, \Delta \vdash t[{}^{z}/_{x}, {}^{z}/_{y}] : C} \qquad (\textit{Weak.}) \quad \frac{\Gamma \vdash t : C}{\Gamma, z : A \vdash t : C}$$

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 that allow us to introduce new formulae (i.e. new types) in the proof. The unique new formula P introduced by each logical rule is called the *principal* formula of the inference. If the principal formula is in the rhs of the final sequent, the inference rule is called *right*, and *left* otherwise. Right and left introduction rules respectively correspond with *constructors* and *destructors* in ISs. The shape of these rules is:

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

for right introduction rules (constructors), and

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

for left introduction rules (destructors). The contexts Γ_i are pairwise different.

A canonical example is implication, which gives the expressions of typed λ-calculus:

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

An immediate consequence of the above construction is that every proof of an intuitionistic system may be described by an IS-expression.

Following Lafont, we may provide a graphical representation of IS-forms. This will explain some important relations between the arity of the forms and the way

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

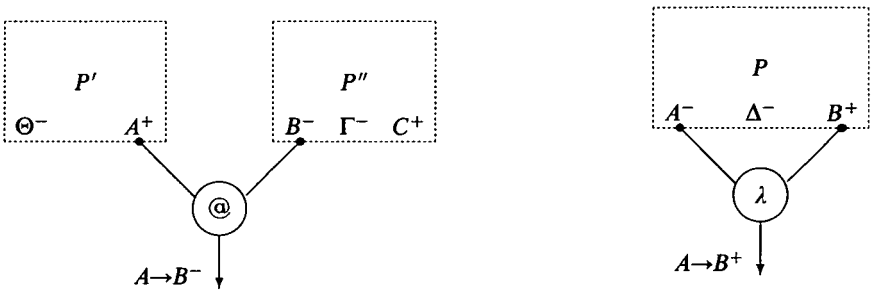


Fig. 3. Graphical representation of @ and λ.

they link upper sequents in a proof. In the notation of interaction nets, a proof of a sequent $A_1, \ldots, A_n \vdash B$ is represented as a graph with $n+1$ conclusions, n with a negative type (the inputs) and one with a positive type (the output). In particular, an axiom is just a line with one input and one output.

Every logical rule is represented by introducing a new operator in the net (a new labelled node in the graph). The operator has a *principal (or main) port*, individuated by an arrow, that is associated with the *principal formula* of the logical inference. For instance, the two logical rules for implications are illustrated in Figure 3.

The principal port of each operator may be either an input or an output. In the former case it corresponds to a new logical assumption in the left-hand side of the sequent (as for @), and the operator is a *destructor*; in the latter case, it corresponds to the right-hand side of the sequent (as for λ), and the operator is a *constructor*. The other ports of the agents are associated with the *auxiliary* formulae of the inference rule, that is, the distinguished occurrences of formulae in the upper sequents of the rule. In the two rules above, the auxiliary formulae are A and B.

The auxiliary ports of an agent may be either inputs or outputs, independent of the fact that it is a constructor or a destructor. Actually, in the general theory of interaction nets, which is inspired by *classical* (linear) logic, there is a complete symmetry between constructors and destructors, and no restriction is imposed on the type of the auxiliary ports (in other words, there is a complete symmetry between inputs and outputs). On the contrary, the fact of limiting ourselves to the intuitionistic case, imposes some obvious 'functional' constraints.

Note first that auxiliary formulae may come from different upper sequents or from a single one. For instance, the auxiliary ports of @ are in different upper sequents, while those of λ are in a same one. Lafont expresses this fact by defining *partitions* of auxiliary ports. So, in the case of @, A and B are in different partitions, while in the case of λ they are in the same partition. Note that the concept of partition is obviously related to that of binding. In particular, a partition is a singleton if and only if the arity of the form for that port is 0. Moreover, the polarity of an auxiliary port is the opposite of the polarity of the conclusion against which it has to be matched. Then, the intuitionistic framework imposes the following constraints:

- In every partition there is at most one negative port. If a negative port exists, we shall call it an *input* partition; otherwise it is an *output*. The positive ports

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

of an input partition are called *bound ports* of the form (they are the ports connected to bound variables).

- Every agent has exactly 'one output' (functionality). In particular, if the agent is a constructor, the principal port is already an output, and all the partitions must be inputs. Conversely, in the case of destructors, we have exactly one output partition among the auxiliary ports, and this partition has to be a singleton.

If $k_1 \dots k_n$ is the arity of a form, every k_i denotes the number of positive ports in the i-th *input* partition. In the case of a destructor, one input partition must be a singleton (since it corresponds to the principal port), so its arity is 0. By convention, in the concrete syntax of ISs, we have supposed that this is always the first partition of the destructor (this assumption is absolutely irrelevant). Summing up, a form with arity $k_1 \dots k_n$ is associated with an operator with $1 + \sum_{i=1}^{n}(k_i + 1)$ ports.

3.2.2 *Dynamics*

Dynamics, in logical systems, is given by the cut elimination process; the idea is that every proof ending into a cut can be simplified into another one by means of some mechanism that is characteristic of that cut (providing, in this way, a rewriting system over proofs). In particular, there exist essentially two kinds of cuts. The most interesting case is the *logical cut*, i.e. a cut between two dual (left-right) introduction rules for the same principal formula. The way such a cut should be eliminated is obviously peculiar to the system, and to the intended semantics of the formula. In all the other cases (structural cuts), intuitionistic systems 'eliminate' the cut by lifting it in the premise(s) of one of the rules preceding the cut (that becomes the last rule of the new proof). As in the case of λ-calculus, these kind of cuts are essentially 'unobservable' in ISs (they are dealt with in the metalevel definition of substitution). So, let us concentrate on logical cuts only.

A typical case of logical cut is that for implication in intuitionistic logic:

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

The elimination of the above cut consists of introducing two cuts of lesser grade (see Girard *et al.*, 1989). The rewritten proof is:

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

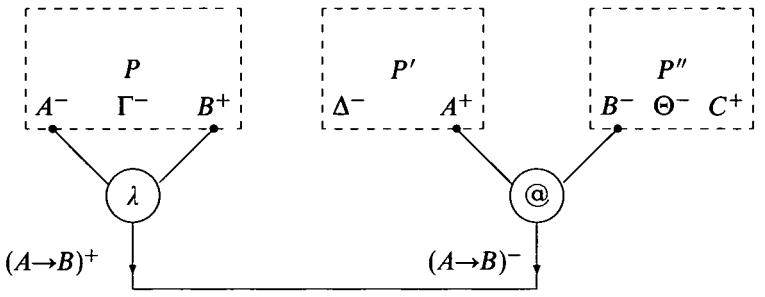
Since by assumption

$$t''[{}^{@(z,t')}/_{y}][{}^{\lambda(x.t)}/_{z}] = t''[{}^{t}/_{y}][{}^{t'}/_{x}]$$

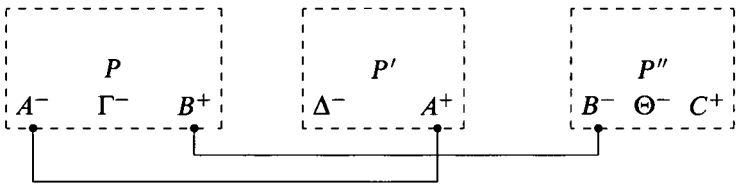
this meta-operation on proofs obviously induces the beta-reduction rule in the underlying IS.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Graphically, the above cut elimination rule can be represented as follows:



reduces to:



In general, let L and R be the left and right sequent in the cut-rule, respectively. During the process of cut-elimination, the proofs ending into the premises of L and R must be considered as 'black boxes' (only their 'interfaces', that is their final sequents, are known). During cut-elimination, one can build new proofs out of these black boxes. The unique constraint is the *prohibition of having new hypotheses* in the final sequent of rewritten proof. This has two implications:

1. the variables bound by L or R (i.e. the auxiliary formulae of the rules) must be suitably filled in (typically with cuts or introducing *new* rules binding them);
2. if a new axiom is introduced by the rewriting, then the hypothesis in the lhs must be consumed (with a cut or by binding it via a logical rule) inside the rewritten proof.

According to statics, a cut in an intuitionistic proof system corresponds to a term of the kind

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

that is just an IS's redex. The X_i represent the proofs ending in the upper sequents of L and R (the 'black boxes' above), and the above conditions on the rewritten proof are obviously reflected in ISs by left linearity and the assumption that the right-hand sides of rules must be closed expressions.

Example 3.6
(*Naturals*) If we do not like to introduce all naturals as explicit constructors, we could just work with two constructors 0 and succ, respectively associated with the following right introduction rules:

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

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

A typical destructor is add.

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

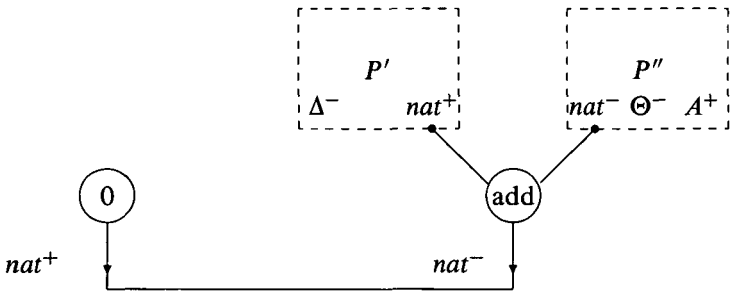
where A can be any type. The following is an example of cut:

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

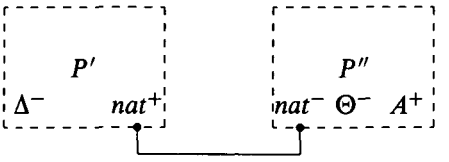
which is simplified into:

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

The above elimination induces the IS-rule $\mathsf{add}(0, X) \rightarrow X$, corresponding to the equation $t[^{\mathsf{add}(x,p)}/_y][^{0}/_x] = t[^{p}/_y]$. Graphically:



reduces to:



□

Example 3.7

(*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*:

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

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

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

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

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

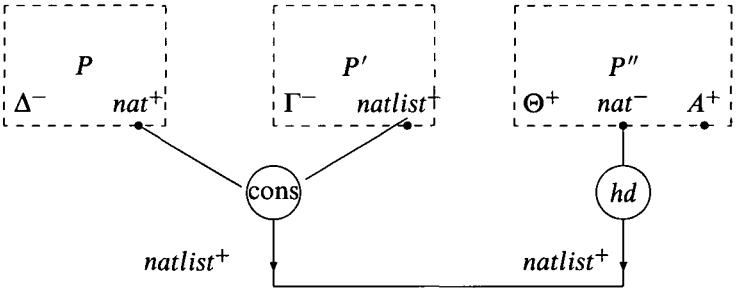
A typical cut is:

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

In this case, the cut would be eliminated in the following way

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

corresponding to the reduction rule $\mathtt{hd}(\mathtt{cons}(X, Y)) \rightarrow X$. Graphically:



reduces to:

Note here, by the way, the phenomenon of garbage creation (P'). □

4 BOHM's source language

As already mentioned, it is possible to generalize Lamping's optimal implementation technique from λ-calculus to an arbitrary interaction system. This means that

1. It is possible to translate an arbitrary IS-expression into a suitable graph representation that involves Lamping control operators for representing the

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

structural information (sharing), and new graphical form for the logical part of the system (in particular, we shall have a new graphical form for each operator of the signature).
2. Any IS-rewriting rule has an associated graph rewriting rule, corresponding to the local interaction of two forms at their principal ports.
3. The graph rewriting rules are correct and optimal w.r.t. the given Interaction System.

All three facts above have already been proved (Asperti and Laneve, 1993b, 1994), and we shall not come back to these results in this paper. We merely apply those theoretical results to a particular case, studying the implementation of a reasonable core of a functional language. In particular, we present the translation and graph reduction rules for the given source language, without ever tackling the correctness and optimality issues. On the other hand, we shall discuss in some detail their practical implementation in BOHM.

The source language we have considered is a sugared λ-calculus enriched with some primitive data types (booleans, integers, lists) equipped by the usual operations, and two more control flow constructs: an explicit fixpoint operator (letrec) and and a conditional if-then-else statement.

In this section, we start by giving the formal syntax of the language. Then we define its operational semantics in the form of an interaction system. Next we give the graph encoding of each term, according to the general paradigm described in Asperti and Laneve (1993b). Finally, we show the full set of graph reduction rules.

4.1 Syntax

The formal syntax of the language accepted by BOHM is the following:

$\langle expr \rangle ::= true$
$\quad | \; false$
$\quad | \; \langle num\_const \rangle$
$\quad | \; \langle identifier \rangle$
$\quad | \; (\langle applist \rangle)$
$\quad | \; \backslash \langle identifier \rangle . \langle expr \rangle$
$\quad | \; let \langle identifier \rangle = \langle expr \rangle in \langle expr \rangle$
$\quad | \; letrec \langle identifier \rangle = \langle expr \rangle$
$\quad | \; if \langle expr \rangle then \langle expr \rangle else \langle expr \rangle$
$\quad | \; \langle expr \rangle and \langle expr \rangle$
$\quad | \; \langle expr \rangle or \langle expr \rangle$
$\quad | \; not \langle expr \rangle$
$\quad | \; \langle expr \rangle < \langle expr \rangle$
$\quad | \; \langle expr \rangle == \langle expr \rangle$
$\quad | \; \langle expr \rangle > \langle expr \rangle$
$\quad | \; \langle expr \rangle <= \langle expr \rangle$
$\quad | \; \langle expr \rangle >= \langle expr \rangle$

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

| ⟨*expr*⟩ <> ⟨*expr*⟩
| ⟨*expr*⟩ + ⟨*expr*⟩
| ⟨*expr*⟩ − ⟨*expr*⟩
| ⟨*expr*⟩ * ⟨*expr*⟩
| ⟨*expr*⟩*div*⟨*expr*⟩
| ⟨*expr*⟩*mod*⟨*expr*⟩
| ⟨*list*⟩
| *cons*(⟨*expr*⟩, ⟨*expr*⟩)
| *head*(⟨*expr*⟩)
| *tail*(⟨*expr*⟩)
| *isnil*(⟨*expr*⟩)

⟨*list*⟩ ::= *nil*
| [⟨*exprlist*⟩]

⟨*exprlist*⟩ ::= ⟨*expr*⟩
| ⟨*expr*⟩, ⟨*exprlist*⟩

⟨*applist*⟩ ::= ⟨*expr*⟩
| ⟨*applist*⟩⟨*expr*⟩

4.2 Reduction

In this section we provide the formal operational semantics of the language, in the form of an IS. In particular, all rewriting rules will be written as destructor-constructor interactions. In some cases, this will require the introduction of a few auxiliary destructors. The full list of constructors, destructors and their respective aritites is listed below.

constructors $\lambda : 1$; *true* : ε; *false* : ε; $m : \varepsilon$, for each integer m; *nil* : ε; *cons* : 00.
destructors @ : 00; + : 00; $+_m$: 0; − : 00; $-_m$: 0; * : 00; $*_m$: 0; *div* : 00; div_m : 0; *mod* : 00; mod_m : 0; *and* : 00; *or* : 00; *not* : 0; ==: 00; == $_m$: 0; >: 00; $>_m$: 0; <: 00; $<_m$: 0; >=: 00; $>=_m$: 0; <=: 00; $<=_m$: 0; <>: 00; $<>_m$: 0; *head* : 0; *tail* : 0; *if_then_else* : 000.

As already noted, letrec should be considered as a constructor-destructor pair. Here are the proper interactions of the rewriting system:

- *Beta-reduction*
  $(\lambda x.MN) \rightarrow M[N/x]$

- *Recursive definition*
  letrec x = M → M[letrec x = M/x]

- *Arithmetical operators*
  $m+M \rightarrow +_m(M)$
  $+_m(n) \rightarrow p$, where p is the sum of m and n
  $m - M \rightarrow -_m(M)$
  $-_m(n) \rightarrow p$, where p is the difference of m and n

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

$m * M \rightarrow *_m(M)$
$*_m(n) \rightarrow p$, where p is the product of m and n
$m\ div\ M \rightarrow div_m(M)$
$div_m(n) \rightarrow p$, where p is the quotient of the division of m by n
$m\ mod\ M \rightarrow mod_m(M)$
$mod_m(n) \rightarrow p$, where p is the rest of the division of m by n

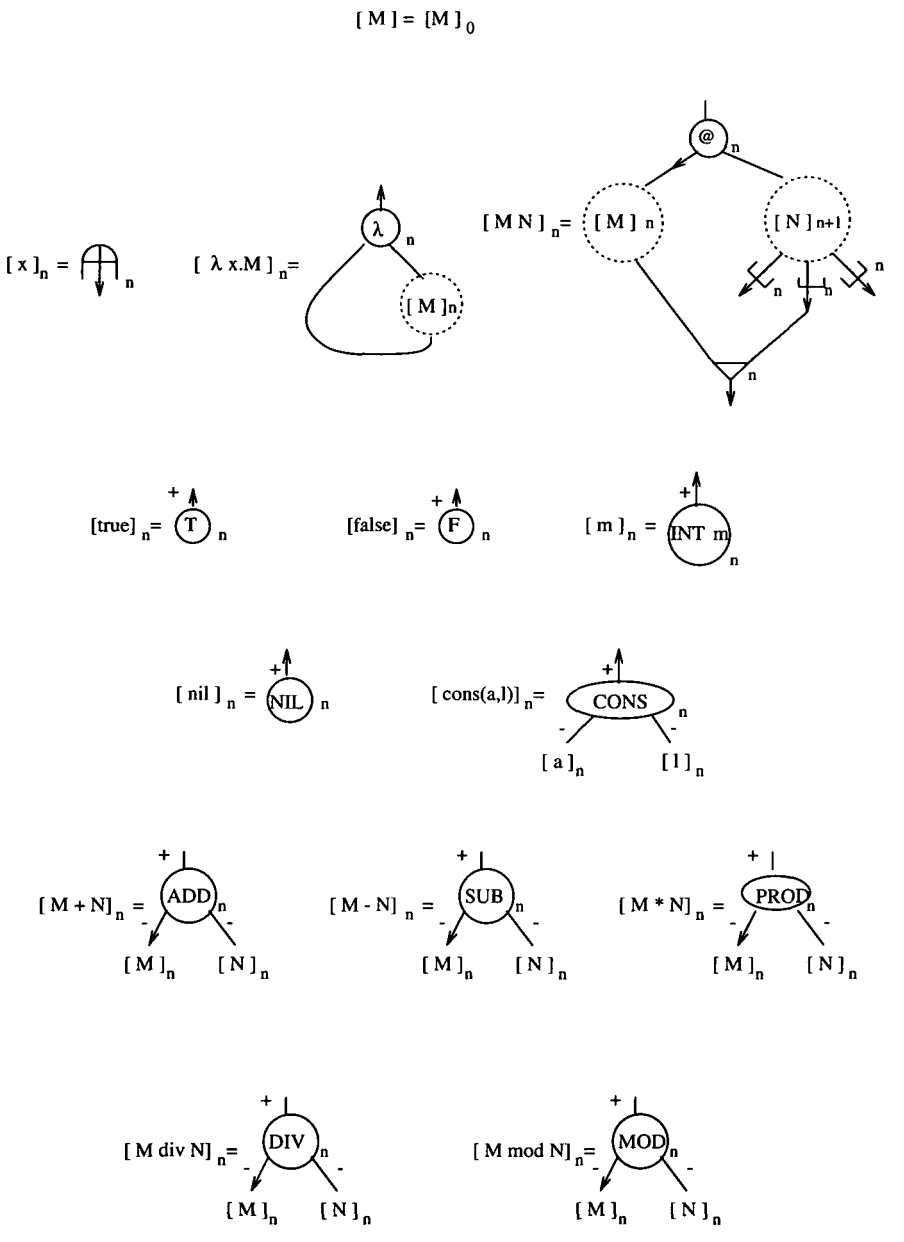
- *Boolean operators*
  true and M → M
  false and M → false
  true or M → true
  false or M → M
  not false → true
  not true → false

- *Relational operators*
  $m == M \rightarrow ==_m (M)$
  $==_m (m) \rightarrow$ true
  $==_m (n) \rightarrow$ false, if n ≠ m
  $m < M \rightarrow <_m (M)$
  $<_m (n) \rightarrow$ true, if m is less than n
  $<_m (n) \rightarrow$ false, if m is not less than n
  $m > M \rightarrow >_m (M)$
  $>_m (n) \rightarrow$ true, if m is greater than n
  $>_m (n) \rightarrow$ false, if m is not greater n
  $m <= M \rightarrow <=_m (M)$
  $<=_m (n) \rightarrow$ true, if m is not greater than n
  $<=_m (n) \rightarrow$ false, if m is greater than n
  $m >= M \rightarrow >=_m (M)$
  $>=_m (n) \rightarrow$ true, if m is not less than n
  $>=_m (n) \rightarrow$ false, if m is less than n
  $m <> M \rightarrow <>_m (M)$
  $<>_m (m) \rightarrow$ false
  $<>_m (n) \rightarrow$ true, if n ≠ m

- *Operator for list manipulation*
  head(cons(M,N)) → M
  tail(cons(M,N)) → N
  tail(nil) → nil
  isnil(cons(M,N)) → false
  isnil(nil) → true

- *The control flow operator if-then-else*
  if true then M else $N \rightarrow M$
  if false then M else $N \rightarrow M$

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

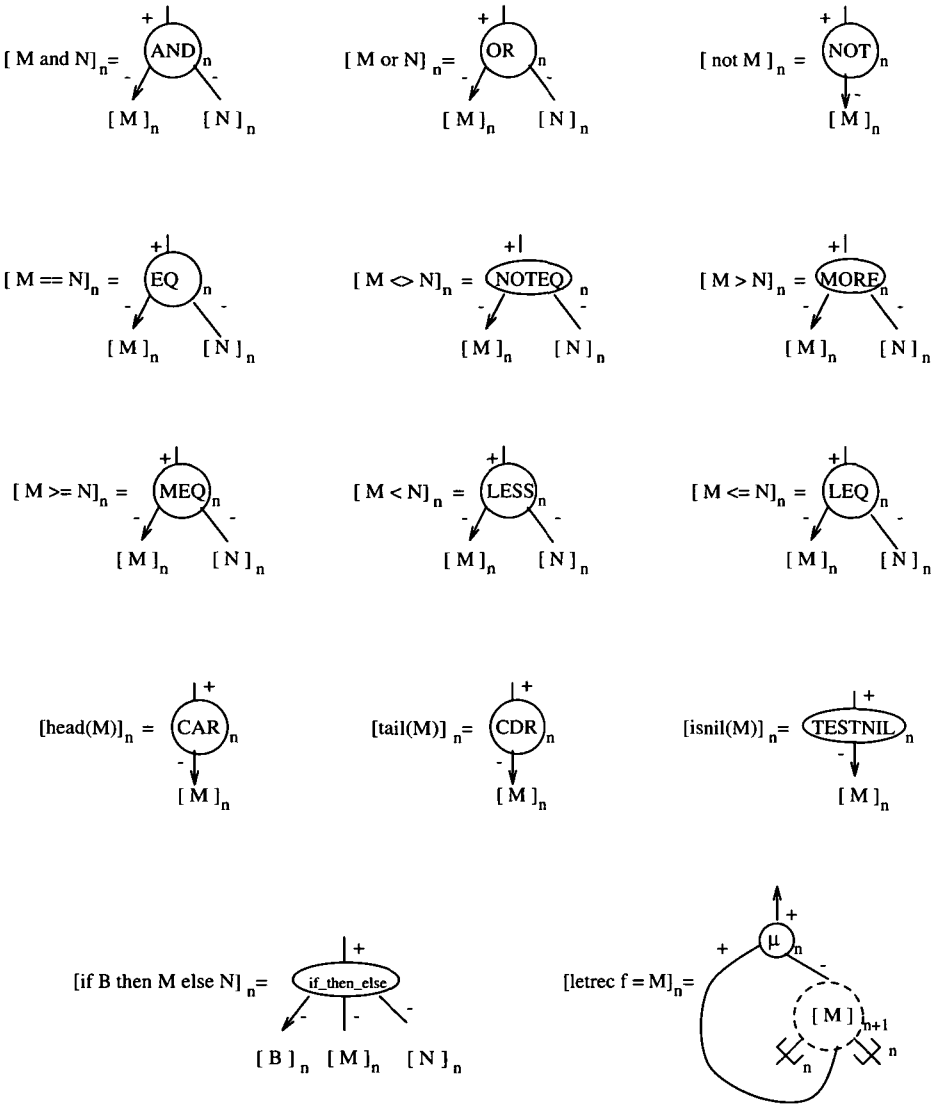


4.3 Graph encoding

In this section we shall define the translation rules for transforming the input term in its initial graphical representation.

Any term N with n free variables will be represented by a graph with $n+1$ entries (free edges): n for the free variables (the inputs), and one for the 'root' of the term (the output). The three main operators of Lamping (fan, croissant and brackets) are used for representing sharing.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press



Again, the rules below are a mere specialization to our particular source language of a general translation method for ISs described in Asperti and Laneve (1993b). The main differences with respect to Asperti and Laneve (1993b) are:

1. We shall follow Asperti's (1994) translation instead of Gonthier's; for the formal relation between these different approaches see Asperti and Laneve (1995).
2. The translation has been optimized in several places to take advantage of the *linear* behaviour of most parts of the interaction rules (see remark 7.6 in Asperti and Laneve, 1993b).

In all the previous rules, when the topmost form has more than one son, all common variables in these sons are supposed to be shared by means of fans (with

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

a level equal to the index of the translation function). For simplicitly, this has been explicitly described only in the case of the application.

4.3.1 Implementation issues

The graph created by the previous rules is unoriented (you should not confuse the arrow denoting the principal port of a form with an oriented edge). For this reason, and to facilitate graph rewriting, all edges will be represented by a double connection between nodes. In particular, for each port of a syntactical form F we must know the other form F' which is connected with it at that port, and also to which port of F' it is connected.

Obviously, each form also has a name (FAN, APP, ADD, ...), and an index. The typical representation of a ternary form may thus be described by the following Struct type in C:

```
typedef struct form
        {
         int   name,
                        /* name of the form       */
                        /* (FAN, APP, ADD,        */
                        /*  SUB, EQ, ...)         */
               index;
                        /* index of the form */
               int    nport[3];
                        /* numbers of the ports of adjacent */
                        /* forms where the three ports      */
                        /* of this form are connected to    */
               struct form *nform[3];
                        /* pointers to the forms            */
                        /* where the three ports            */
                        /* of this form are connected to    */
       }   FORM;
```

Given a pointer f to a form, the field `f->nform[i]` will denote the next form g connected with f at port i. Similarly, `f->nport[i]` says to which port of g f is connected.

In particular, `(f->nform[i])->nform[f->nport[i]] == f`.

Remark By convention, the principal port of a form has number 0.

Even if the previous implementation introduces some redundancy, it allows us to navigate in the graph in a very easy way.†

As a simple example, let us see the function that connects together two forms

† It seems possible to avoid a systematic use of bi-links, restricting them to particular edges of the graphs (typically, those representing cut or axioms). Although this solution could reasonably save some space and could also improve the performance of the system, it looks more complex to implement, and we finally rejected it in the design of our prototype.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

at two specified ports (this is the most frequently used function of BOHM; each atomic interaction calls connect 3–4 times, on average):

```
 /* the following function connects together the port portf1 of */
 /* form1 to the port portf2 of form2 */
connect(form1,portf1,form2,portf2)
        FORM        *form1;
        int         portf1;
        FORM        *form2;
        int         portf2;
{
        form1->nport[portf1] = portf2;
        form1->nform[portf1] = form2;
        form2->nport[portf2] = portf1;
        form2->nform[portf2] = form1;
}
```

4.3.2 *Global definitions*

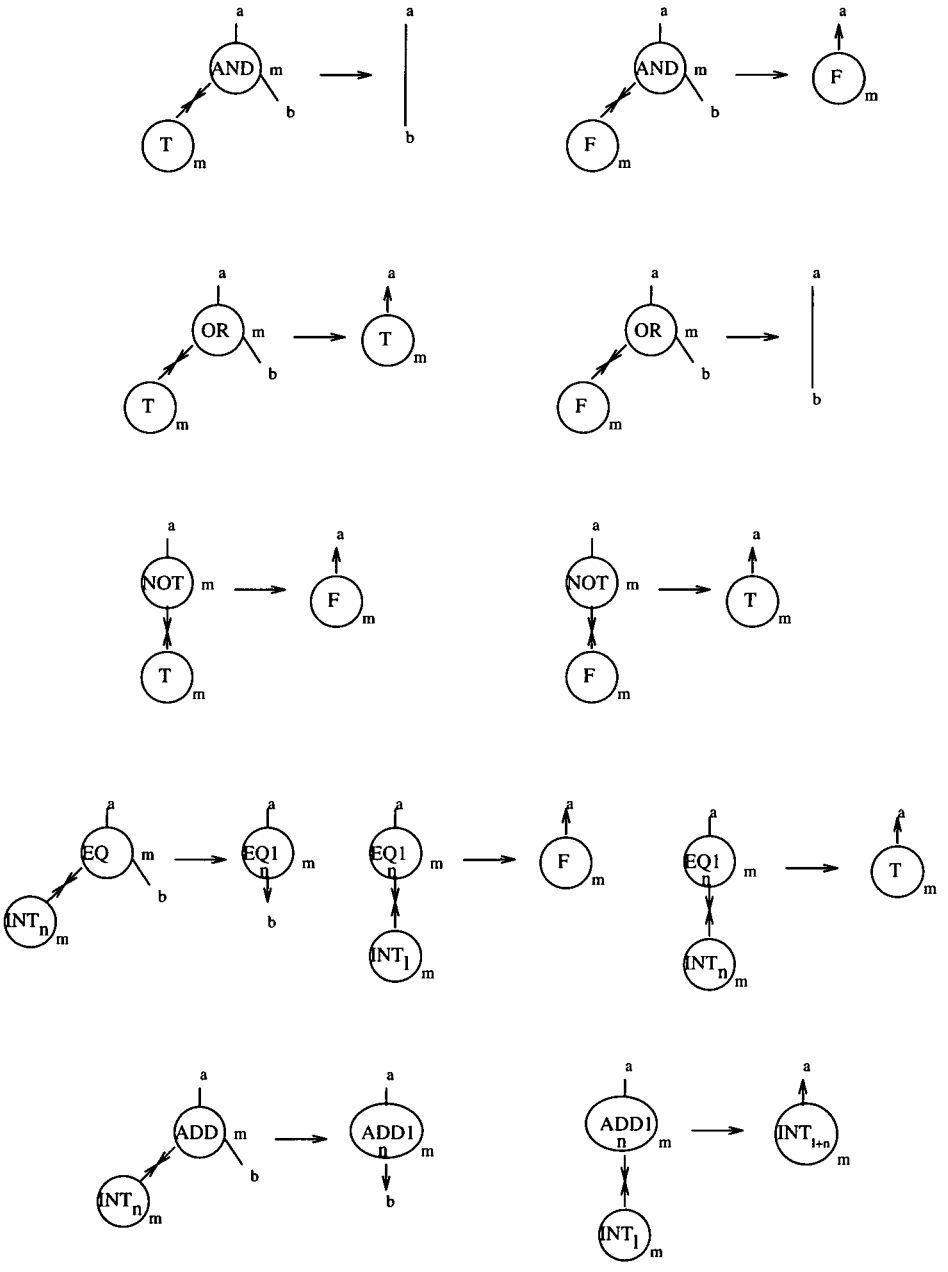
The syntactical construct *def* ⟨ *identifier* ⟩ = ⟨ *expr* ⟩ allows the user to declare an expression as a global definition, binding it with an identifier name. Every time we have a new global definition, the graph corresponding to the term is built up and closed with a ROOT form that is pointed to by the identifier in the symbol table.

When we use a previously defined global expression, we simply attach a FAN at the top of the corresponding graph, implicitly creating a new instance of the expression. In this way, the global definition is shared by all its instances. Since a global expression is a sharable data item, its graph must be created starting from level 1, and not 0 as for usual expressions. This handling of global definitions is responsible for an odd operational behaviour of the machine: actually, the second time we evaluate the same expression containing global identifiers, the reduction time can be much smaller than the first time. The reason is that we are sharing with the previous call all partial evaluations which have already been performed on global expressions. So, this choice allows us to profit from the sharing of terms and their reductions following the philosophy of optimal reduction. However, in some cases it also introduces a trade-off from the viewpoint of memory allocation. In fact, the partially evalutated global expression may need more space than the original term.

4.4 *Graph reduction*

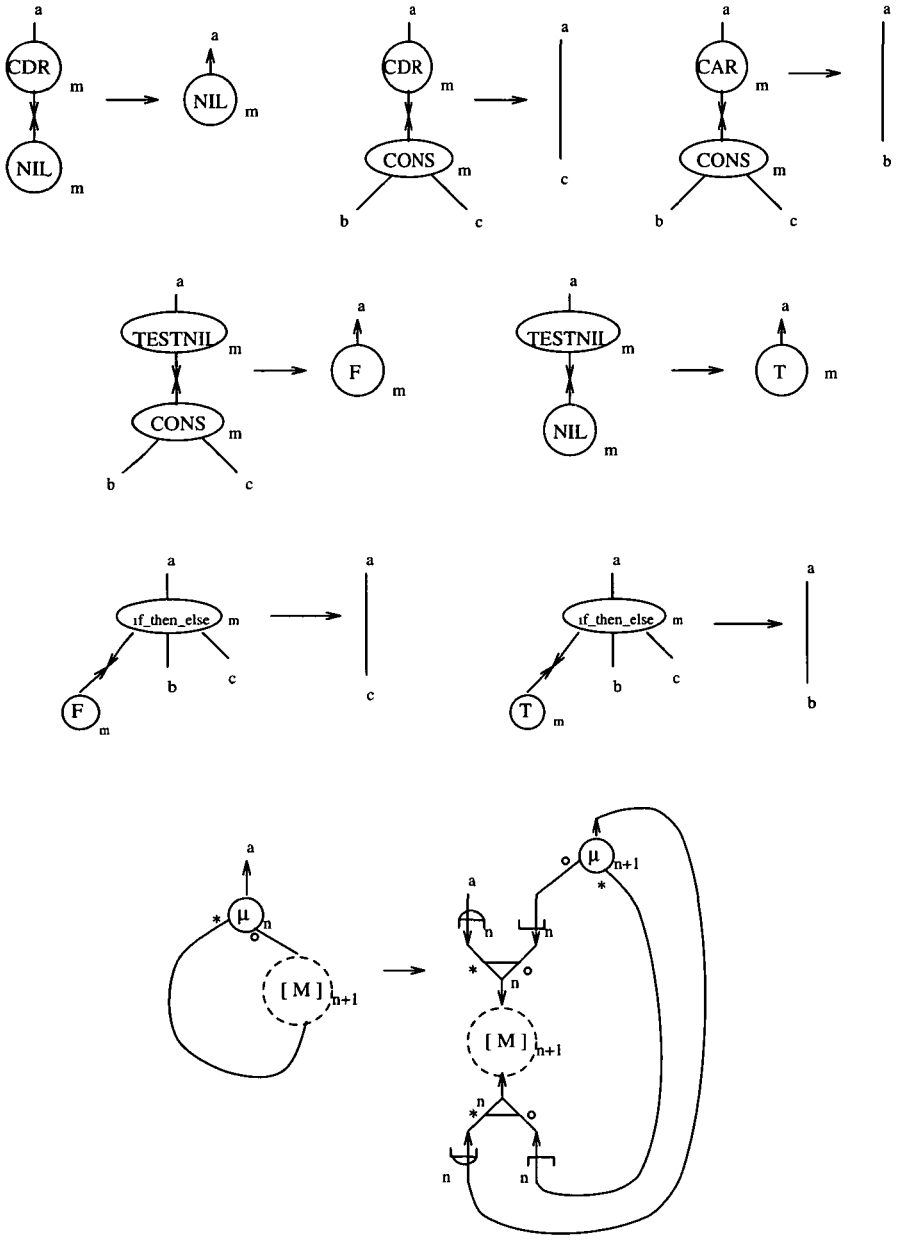
Once the graph representing an expression has been built, the reduction proceeds according to the following graph rewriting rules. Again, these rules are a special case of a general paradigm for Interaction Systems developed by Asperti and Laneve (1993b, 1994), so we shall not prove their correctness here. In fact, most of them are intuitive:

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press



The rule for the recursion operator MU (the last rule above) deserves a few words. Recall that it corresponds to the reduction rule letrec x = M → M[letrec x = M/x]. The idea is that, to preserve optimality, the two occurences of the body M in the right-hand side of the reduction rule must be shared. The best way to understand the graph rewriting rule is by 'reading back' the graph in the right-hand side. Obviously we start from the root, and immediately meet the first occurence of M; note that we are entering this occurence from the *-side of the fan-in. If travelling along M

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press



we reach the variable x, according to Lamping's context semantics, we must exit from the *-branch of the fan-out (that is, matching the initial fan-in). Now, we get a new MU operator, whose body is nothing else that another instance of M. Note that this time we are entering this instance from the o-side of the fan-in. When we exit again through the variable x, we now discover that this variable is bound by the new MU operator.

Square brackets and croissant are put in such a way as to guarantee the correct matching of the newly introduced fan nodes, and to avoid any interference between

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

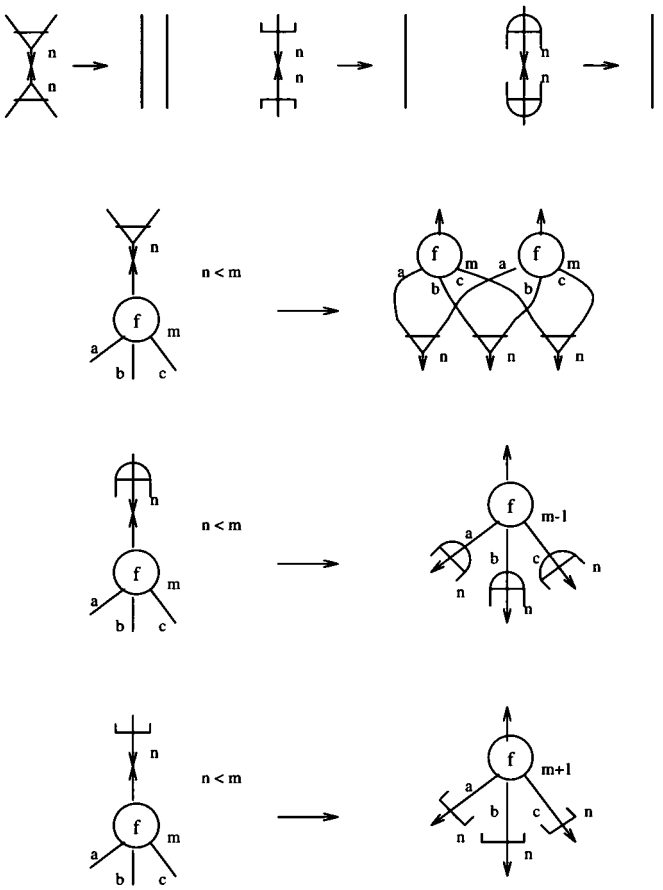


Fig. 4. Control rules.

them and other possible fans in the graph (see Asperti and Laneve (1994) for a formal correctness proof).

Having defined the proper graph rewriting rules (i.e. those rules that correpond to interactions between syntactical operators), we should finally add the rules relative to the interaction between proper forms and control nodes (or control nodes alone). Luckily, these rules have a high degree of parametricity. The idea is that, in these interactions, one operator is active, and the other is passive. The active operator is that with the lower index (in Gonthier's translation, the situation would be dual). If the active operator is a fan, it duplicates the form it is traversing; if it is a croissant it decrements the level of the form it traverses; and if it is a square bracket, it increments it. When two control operators with a same index meet together, they mutually erase each other.

According to the main law of optimal reduction, the interaction between a fan and a syntactical form must be restricted to the case where the fan reaches the form at its principal port. By analogy, this rule can be extended to all graphical forms (as proved in Gonthier *et al.* (1992a)), and we followed this 'minimal' approach in BOHM. However, note that, for brackets and croissant, this is not strictly

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

necessary from the optimality point of view. In fact, Lamping permitted a 'rapid propagation' of brackets and croissant, no matter which port they were interacting with. The practical relevance of this rapid propagation is still a matter for further investigation.

BOHM's 'control rules' are shown in Figure 4.

4.4.1 Implementation issues

BOHM is a lazy (weak) machine. It does not perform the full reduction of a term, but stops reducing when the topmost operator in the graph becomes a constructor. This halting situation is easily recognized, since it means that the root of the term becomes *directly* connected to some operator f at its *principal port* (i.e. by convention, the port with number 0). In this case, the name of f is assumed to be the result of the computation.

Reduction proceeds according to the following idea. We start looking for the leftmost outermost redex. In particular, starting from the root of the term, we traverse any operator that we reach at an auxiliary port, *always exiting from its principal port*, until we reach an operator f at its principal port. Now two cases are possible: either the previous operator was the root node (and this is the halting case); or eventually we have found a redex (since the two final operators are eventually connected at their principal ports). In this case, the redex is fired by applying the associated graph reduction rule, and we start the process again.

To avoid having to rescan the term from the root every time, it is convenient to push all operators leading to the first redex on a stack (similar to the stack of the G-machine). After firing a redex it is enough to pop the last element from this stack, and start searching for the next redex from this node.

The following code is the main loop of the reduction machine:

```
reduce_term(root)
     FORM      *root;
{
     FORM      *f1,
               *f2,
               *erase;
     int       type_error;

     type_error = FALSE;
     f1 = lo_redex(root);
     f2 = f1->nform[0];
     while ((f1 != root) && (!type_error))
        {
           if (f1->index <= f2->index)
               reduce_redex(f1,f2);
           else
               reduce_redex(f2,f1);
```

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

```
            f1 = lo_redex(pop());
            f2 = f1->nform[0];
         }
      if(!type_error) rdbk(root);
}
```

where `lo_redex` is defined as follows:

```
FORM *lo_redex(f)
     FORM    *f;
{
     FORM    *temp;

     temp = f;
     while (temp->nport[0] != 0)
        {
           push(temp);
           temp = temp->nform[0];
        }
     return temp;
}
```

`reduce_redex(f1,f2)` is just a big switch based on the name of $f1$ and $f2$. For example, let us see the code corresponding to a couple of typical interactions. This is the code in the case the first form $f1$ is AND:

```
case AND:
   switch(f2->name)
      {
          case F:
              connect(f1->nform[1],
                      f1->nport[1],
                      f2,
                      0);
              myfree(f1);
              break;

          case T:
              connect(f1->nform[1],
                      f1->nport[1],
                      f1->nform[2],
                      f1->nport[2]);
              myfree(f1);
              myfree(f2);
              break;

          default:
```

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

```
                printf("--->   type error\n");
                type_error = TRUE;
                break;
        }
```

This is the complex case of MU (the most complex reduction in BOHM):

```
case MU:
    allocate_form(&new1,CROISSANT,f1->index);
    allocate_form(&new2,FAN,f1->index);
    allocate_form(&new3,SQUARE,f1->index);
    allocate_form(&new4,CROISSANT,f1->index);
    allocate_form(&new5,FAN,f1->index);
    allocate_form(&new6,SQUARE,f1->index);

    connect(new1,1,f1->nform[0],f1->nport[0]);
    connect(new1,0,new2,1);
    connect(new3,0,new2,2);
    connect(new2,0,f1->nform[1],f1->nport[1]);
    connect(new5,0,f1->nform[2],f1->nport[2]);
    connect(new5,1,new4,0);
    connect(new5,2,new6,0);
    connect(new6,1,f1,2);
    connect(new3,1,f1,1);
    connect(new4,1,f1,0);
    f1->index++;
```

5 Safe operators

The notion of a *safe operator* was introduced for the first time in Asperti (1995). From the implementation viewpoint, the main problem of Lamping–Gonthier's optimal reduction technique is the accumulation of control operators that can easily clutter the graph, leading to an exponential explosion of the reduction time. The general idea is that there are some 'safe' cases where particular sequences of control operators can be reduced to simpler configurations. The relevance of safe operators and of the associated rules has already been proven by Asperti (1995) by means of empirical results: in many typical cases, by adding these rules we can pass from an exponential to a linear or polynomial reduction time (that is, something more than a mere 'optimization').

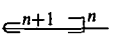
The idea of safe operators and their associated reduction rules were originally inspired by the categorical interpretation of optimal reduction passing through linear logic, described in Asperti (1994). Actually, the 'safe rules' are nothing other than the natural counterpart in sharing graphs of the three comonads equations associated with the comonad '!' (the box) of linear logic.

In this paper we shall try to avoid any reference to category theory or linear logic, providing a completely operational understanding of the problem.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

5.1 Accumulation of control operators

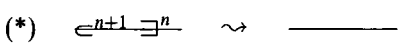
Consider the following sequence of control operators:



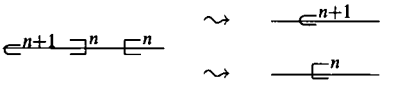
We immediately realize two important properties of such a configuration:

1. The pair of control operators may always be propagated together through other operators (i.e. they can be seen as a unique 'compound' operator).
2. The total control effect of this 'compound' operator on the nodes they are propagated through is just null.

Recall that the only problem with the optimal reduction technique is guaranteeing the correct matching of fan nodes (actually, brackets and croissants are just introduced for this purpose). Since the above 'compund configuration' has no total control effect on the mutual level of fans, the natural idea would be to reduce the above configuration to an identity, i.e. a straight line:

(*) 

Unfortunately, this is not possible in general. The problem is very simple: we forgot to take into account a possible annihilation of the innermost operator, as described by the following critical pair:

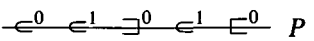


Obviously, only the the first reduction is correct, while the second one can easily lead to wrong and irreducible configurations (deadlocks). If it were possible to exclude the possibility of such an annihiliation (i.e. the existence, *at a given time*, of 'out-forms' matching a given 'in-form'), the (*)-rule above would be completely (and obviously) correct. This is exactly the idea behind a 'safe' operator.

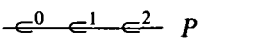
Let us prove with an example that the above case of annihilation can actually occur, in practice (this is not completely obvious). Consider the term

$$\lambda x.I(I(x\ P))\ I$$

where I is the identity and P is any term. If you reduce this term according to a leftmost outermost strategy in the system enriched by (*), after four beta reductions you are left with the following row of control operators leading to P:



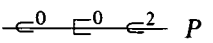
Now, the correct final configuration would be:



but pursuing a leftmost outermost reduction you would instead obtain the following

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

deadlock situation:



A completely analogous problem concerns fan and garbage nodes (see section 6).

5.2 *Safe operators*

Safe operators are control operators which have a sort of 'global status' in the graph. This means no residual of their information is still (or yet) propagating inside the graph. In other words, you may look at a safe operator as a sort of 'fresh' node; note in particular that, generalizing Lamping's distinction between fan-in and fan-out to all control operators, safe operators are eventually 'in'-forms. Precisely,

1. all control operators are safe in the initial graph;
2. an operator becomes unsafe when some rule creates 'out'-residuals of its information. This can merely happen when the safe operator interacts with a (constructor-)binder of a higher index than its own;
3. an (unsafe) operator of index n that, during its broadcasting, reaches (at their auxiliary ports) either a *safe* bracket of index n or $n-1$ or a *safe* fan of index n, becomes safe again.

From the implementation viewpoint, it is sufficient to add a tag to each node, expressing its 'safeness'. The tag is modified in the obvious way, according to the above-mentioned rules.

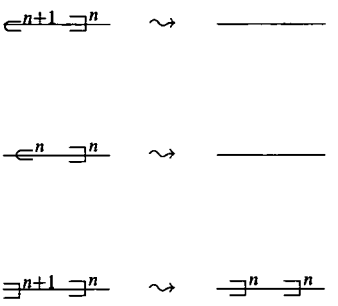
It is possible to prove that a residual of a safe operator (created by its broadcasting) may only be annihilated by another residual of the *same* operator (the residual relation is defined in the obvious way). Note, in particular, that rule 3 above is operationally justified by the fact that, in the configurations described, the only possibility to annihilate the 'outermost' form would be to annihilate the 'innermost' form, first (the 'direction' is implicitly defined by the fact that all safe operators are in-forms).

Remark Until we do not allow a 'rapid' propagation of brackets and croissants through auxiliary ports of proper forms (that does not invalidate optimality, and is completely correct), case 3 above in practice never happens. The reason for this is that the propagation of control operators is interrupted too soon inside the box, and they will never reach its extremity. In fact, this rule was not implemented in the version of BOHM described in this paper; we just report it for the sake of completeness.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Theorem 5.1

The following 'comonad' rules



are correct between *safe* operators.

Proof

An easy consequence of the following considerations:

1. all pairs of control operators in the left-hand side of a comonad equation may always be propagated together (i.e. they can be seen as a unique operator);
2. the total control effect of this 'compound' operator on the nodes they are propagated through is equal to the control effect of the right-hand side of the rule (i.e. it is null for the first two equations, and it amounts to a double level shift for the third equation);
3. if the innermost form of (a residual of) the pair is annihilated, then the outermost form can be annihilated at the next step. Indeed, the two operators are residuals of a 'compound' safe operator, and they can only be annihilated by other residuals of the same 'compound' configuration.

□

As already noted in the previous section, point 3 in the previous proof generally fails for unsafe operators.

Remark

1. According to our experimental results, only the first rule above seems to be really essential, in practice. This may be intuitively explained by noticing that, while the first comonad rule is a sort of 'β-conversion', the other rules are more akin to 'η-conversions'.
2. Remark 5.2 does not exclude the possibility of *dynamically* creating reducible configurations for the comonad rules: consider, for instance, the reduction of $\lambda x.(I\ (I\ x))$: along this reduction we can twice apply the first comonad rule, and in one case the redex is *created* along the reduction.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

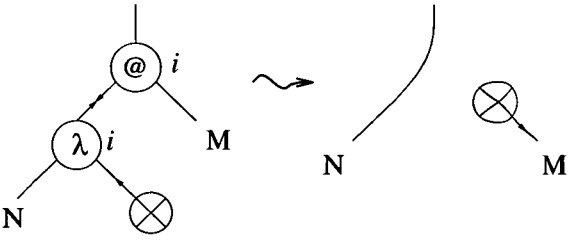


Fig. 5. Creation of garbage.

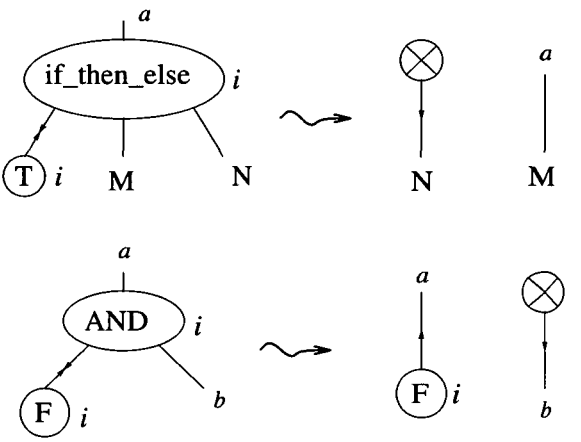


Fig. 6. Creation of garbage.

6 Garbage collection

In pure λ-calculus, the creation of garbage is related to the presence of abstraction nodes whose bound variable does not appear in the body. Such situations are represented by introducing a new node type, called the *erase operator*, which is connected to the bound port of the λ-node.

Consider the term $(\lambda x.N)M$, where x does not appear in N. As shown in Figure 5, after the β-reduction, the term M appears disconnected from the main graph, becoming garbage.

Actually, things are usually more complex, since M may share some subterms with the main graph, and these subterms cannot be regarded as garbage; moreover, the whole term M would remain connected to the main graph through the shared subterms.

As far as we consider pure lambda calculus, garbage collection is not really impelling; on the contrary, it becomes of dramatic importance in the enriched language of BOHM. In fact, a lot of BOHM's rewriting rules create garbage. A couple of typical examples are shown in Figure 6.

In the first case in Figure 6, the condition of the *if_then_else* construct is *true*, so the subterm N, corresponding to the '*else branch*' becomes garbage. In the second case, the evaluation of the boolean expression reduces immediately to *false*, and the remaining part of the expression is useless.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

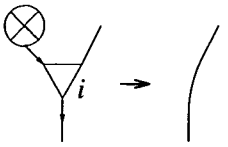


Fig. 7. Garbage-fan interaction.

Other important cases concern lists; for instance, after an application of the operator *CAR*, which selects the first element of a list, the tail of the list can be discarded.

In the context of optimal reduction, it appears difficult to implement an efficient garbage collection procedure using 'classical' algorithms (we mean 'marking active elements – collecting garbage elements'), for the following reasons:

1. marking the active elements would require a complex visit of the graph, based on the so-called context semantics, and the complexity of such a visit could be exponential in the size of the graph;
2. we could limit the collection of garbage to a physically disconnected subgraph; however, particularly with BOHM's weak reduction strategy, garbage collection would be delayed considerably in the reduction of the term, causing a possible explosion of the garbage.

The second point deserves further explanation. Consider, for instance, the case of a shared list (a fan over a cons). Suppose, moreover, that one branch of the fan has become garbage (i.e. we have a garbage operator at one auxiliary port of the fan). This situation often occurs with BOHM's rules. Obviously, we would like to eliminate the fan and garbage operator, and attach the list to the active branch of the fan. Adopting the policy in point 2 above, we could not collect any garbage until we finished duplicating the whole list. In other words, in optimal reduction, garbage should always be collected *as soon as possible.*

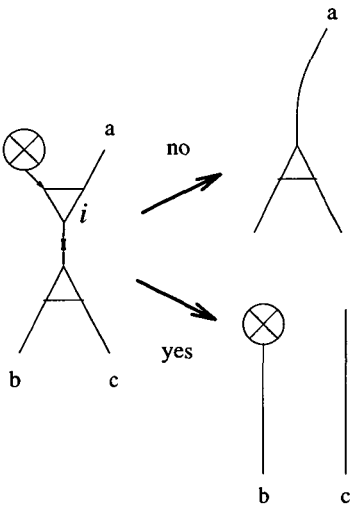
As pointed out by Lamping, an efficient way to integrate a garbage collector in the optimal graph reduction technique is based on local interactions of erase operators (⊗). The basic idea is that the erase operators are propagated along the graph, collecting every node encountered during their walk.

In BOHM, all erase operators are mantained in an appropriate data structure (a list); when garbage collection is activated, each of them proceeds interacting (i.e. erasing) the forms it encounters and adding new erase operator to the list, until possible.

Note that there are some situation in which no interaction is allowed; typically when an erase operator reaches the bound port of an abstraction, or an auxiliary port of an unsafe fan. This latter case is particularly important. Actually, when a garbage operator reaches a fan at an auxiliary port, one would be tempted to apply the rule in Figure 7.

Indeed, one of the two branches of the information shared by the fan has become garbage, and the obvious solution would seem to be to simply drop this branch.

Unfortunately, this rule is not correct in general, due to the problem described in the following diagram:



That is, if we allow a fan-in(-out) to interact with an erase operator according to the rule in Figure 7, we could preclude to some 'paired' fan-out(-in) the possibility to annihilate itself. Luckily, the rule is perfectly correct for safe fan (for the trivial reason that no 'paired' fan-out could possibly exist, in this case).

Another interesting point is the choice of an appropriate strategy for activating the garbage collection procedure. Our empirical studies seem to suggest that the best strategy is to attempt garbage collection every time some reduction rule generates new garbage, i.e. as soon as the garbage is created. Operating in this way, we obtain two important advantages:

- the size of the graph is always kept to a minimun along the reduction;
- collecting garbage as soon as possible prevents it from becoming involved in useless interactions, resulting in a sensible improvement in performance.

Figure 8 contains the complete set of garbage rules for pure lambda calculus. They are generalized in the obvious way to the other forms of the syntax. In particular, a garbage operator can efface any syntactical form, no matter at which port it reaches the form, unless it is a bound port. In this case it stops there. After collecting the form, we broadcast the garbage operator to all other ports of it, and start the process again.

Note that the erase node has no index, and its interactions are independent from the index of the adjacent node. Note also that the first two rules for fans may only be applied in case the fan is safe; otherwise, no interaction is possible, and the erase operator is stopped.

6.1 Implementation issues

To improve the efficiency of the garbage collection procedure, four new nodes have been introduced (Figure 9). They do not represent either new syntactical elements

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

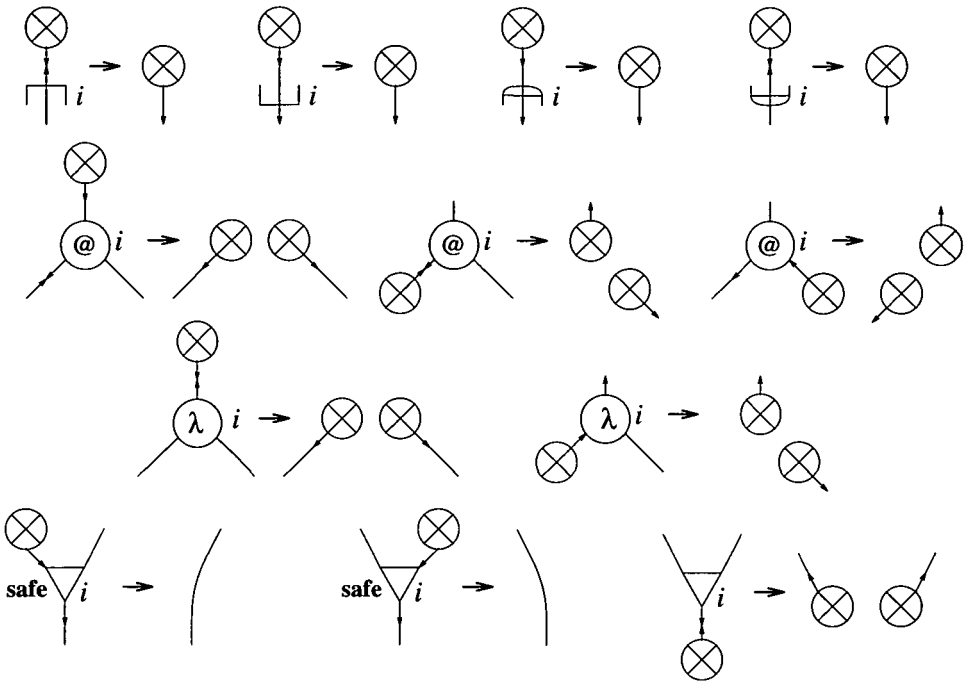


Fig. 8. Garbage collection rules.

nor new control operators; these new forms are a sort of abbreviation for particular configurations of other forms. In particular:

- The form UNS_FAN1 represents a configuration where an erase operator is 'blocked' on the 'first' auxiliary port of an unsafe fan.
- UNS_FAN2 is similar to the previous one, but relative to the 'second' auxiliary port.
- The form LAMBDAUNB represents an abstraction node connected at its bound port with an erase.
- MU_UNB is similar to the previous case, but relative to the fixed point operator.

The introduction of these new forms has two important advantages:

- All erase nodes which cannot be propagated any further are merged into their adjacent nodes, so it is easy to recognize when garbage collection is possible: an erase operator is physically present in the graph if and only if it is 'active'.
- The forms UNS_FAN1 and UNS_FAN2 prevent many useless duplication operations in the graph. Suppose X is an arbitrary operator that can interact with the fan in Figure 10; without the four new forms, things would work as shown in Figure 10. First X is duplicated by the fan, and then eventually deleted by the garbage collection procedure. Using the operator UNS_FAN2 no duplication is performed (Figure 11).

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

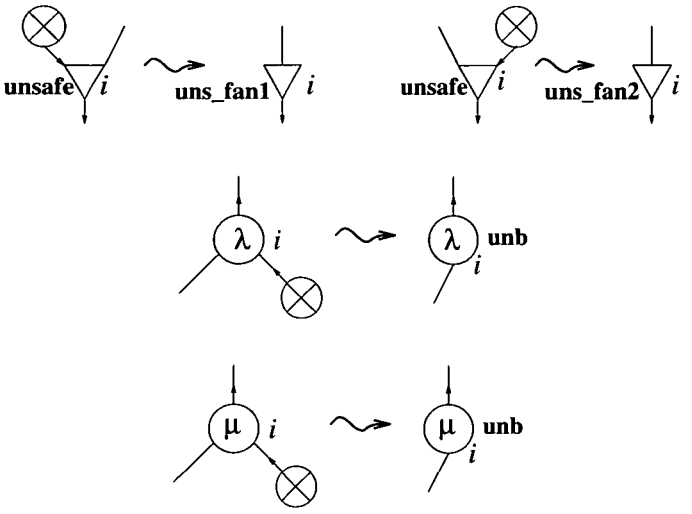


Fig. 9. New auxiliary forms.

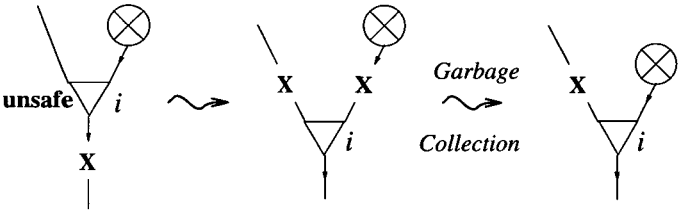


Fig. 10. Reduction without new forms.

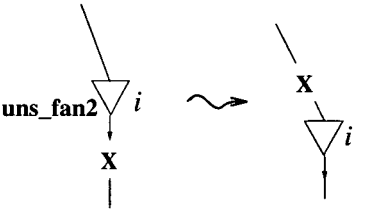


Fig. 11. Reduction with new forms.

Another annoying problem with the garbage collector is that it could inadvertedly erase some operators in the leftmost-outermost-redex stack (see section 4.3.1). Our solution has been to introduce a new tag for each form denoting if it is either in or out the stack. Then, the garbage collector, instead of erasing forms which are in the stack, will simply mark such elements as 'collected'; the form will be physically disposed of when it is eventually popped from the stack.

To evaluate the performance of the garbage collector, BOHM is enriched with some menus to select the desired collecting strategy. The possibilities are:

(1) *Maximum Garbage Collection*: the procedure is activated as soon as possible.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

(2) *Garbage Collection Depending on Memory Occupation*: the procedure is attempted when the total number of allocated operators reaches a certain limit fixed by the user.
(3) *No Garbage Collection*: garbage collection is not active.

The menus are shown by calling the program with option *-s* or by typing the directive *#option;;* in the interactive environment.

7 Examples and benchmarks

In this section we shall discuss several examples of computations, with the aim of providing a concrete measure of BOHM's performance. In particular, to give the gist of optimality and its actual power, we shall compare the performance of our system with a different, fully developed and largely diffused implementation: CamlLight.† CamlLight (Leroy and Mauny, 1992) is a small, portable implementation of the ML language (about 100K for the runtime system, and another 100K of bytecode for the compiler) developed at INRIA-Rocquencourt. In spite of its limited dimensions (versions for Macintosh and IBM PC are also available), the performance of CamlLight is quite good for a bytecoded implementation: five to ten times slower than SML-NJ.

Obviously, we chose a set of examples particularly suited to the optimal reduction technique. However, it is not our intention to keep BOHM's actual performance secret; in the *total absence* of sharing, BOHM is much slower than CamlLight‡: approximately ten times for pure lambda calculus, and even worse (up to 50 times!) for more numerical computations.

However, recall that:

1. BOHM is just an interpreter at a very high level (the graph is a real graph in C, and we are continuously allocating and deallocating nodes from the heap!);
2. numerical computations are obviously slow, due to the quite cumbersome handling of arithmetic;
3. experimentally, the slowdown of BOHM w.r.t. CamlLight looks *constant*, while in many cases CamlLight is just exponential w.r.t BOHM.

The main problem of optimal reduction is that, to get a sufficient amount of sharing, you should heavily use higher-order features in your program (like, say, iterating a higher order, non-linear functional). In other words, functions should become the real object of the computation. This is not often the case in the practice of functional programming, but nevertheless, there are some examples where optimality can be

† We had no special reason for choosing CamlLight instead of some other functional language. We just wished to offer a comparitive example.
‡ This gap has been substantially reduced with the new version of BOHM: its performance in the worst case is now *always* comparable with typical lazy implementations, such as Yale Haskell. Since the new version of BOHM contains some optimizations which are not discussed in this paper, we decided to leave the old benchmarks.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

really useful: a typical case is the incremental modifications of arrays represented as functions.†

However, the optimal handling of sharing is not the only advantage of BOHM. In fact, another essential feature of BOHM is that it joins all the advantages of lazy and strict reduction with, moreover, *full evaluation* of the argument (i.e. pursuing reduction under the lambda). The example of Tartaglia's (Pascal's) triangle should be illuminating, in this respect (see below).

All the examples discussed here can be found in a BOHM tar file, in the subdirectory `examples`. The tests have been performed on a Sparc Station 5.

7.1 Reduction

Example 7.1

Our first example is a function for computing prime numbers based on Erathostenes' sieve. In particular, (prime *nm*) is 1 if *m* is prime with respect to the *n* first prime numbers, and 0 otherwise. In other words, (prime *n*) is the *n*th approximation function in Erathostenes' sieve.

```
{The following program computes Erathostenes' sieve.}

{starting approximation function}
def constOne = \n.1;;

{(min g n m) is the n-th input value k of g (larger than m)
such that (g k)<>0 }
def min = letrec mi
          = \g.\n.\m. if ((g m) <> 0)
              then if n == 0 then m
                             else (mi g (n-1) (m+1))
              else (mi g n (m+1));;

{(minIn g n) is the n-th input value k of g such that (g k) <> 0}
def minIn = \g.\n. (min g n 1);;

{criv computes the next approximation function in Erathostenes' sieve}
def criv = \n.\g.\x. let a = (minIn g n) in
                        if ((x mod a) <> 0 or (x == a))
                        then (g x)
                        else 0;;

{(iterate n g f) = (g n (g n-1 (g n-2 (.... (g 1 f)...))))}
def iterate = letrec it
```

† Another funny example, not discussed here, where BOHM is definitely (i.e. exponentially) better than traditional implementations, is the direct evaluation of the denotational semantics for imperative languages.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Table 1. *Erathostenes' sieve*

Input	CamlLight	BOHM
(prime 2 7)	0.00 s	0.00 s 1952 interac. 125 proper interac.
(prime 2 50)	0.00 s	0.00 s 655 interac. 92 proper interac.
(prime 4 15)	0.12 s	0.08 s 5601 interac. 425 proper interac.
(prime 5 3500)	1.15 s	0.05 s 7904 interac. 755 proper interac.
(prime 6 20)	16.30 s	0.13 s 11937 interac. 1228 proper interac.
(prime 7 49)	explodes	0.23 s 16990 interac. 1934 proper interac.
(prime 10 50)		0.78 s 43463 interac. 5594 proper interac.

```
= \n.\g.\f. if n == 0 then f
                      else (g n (it (n-1) g f));;

{(prime n x) is 1 if x is prime w.r.t. the n first prime numbers,
and 0 otherwise}
def prime = \n. (iterate n criv constOne);;
```

Some examples of computation are shown in Table 1.

For BOHM, we respectively give the user time required for the reduction, the number of proper interactions (i.e. interactions between syntactical operators), and the total number of interactions (i.e. proper interactions plus control interactions). In the case of CamlLight, we just give the user time.

Example 7.2

The second example concerns computation of the transitive closure of a graph using the Roy–Warshall algorithm. Nodes are supposed to be integers, and the graph is represented by the characteristic function g of its edges, i.e. $gmn = 1$ iff there exists an arch from m to n, 0 otherwise.

Note that, with this representation, (ga) is the successor function of node a in the

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Table 2. *Transitive closure of a graph*

Input	CamlLight	BOHM
(transclos 5 g 3 2)	0.02 s	0.03 s 2439 interac. 192 proper interac.
(transclos 5 g 5 4)	0.00 s	0.02 s 1726 interac. 192 proper interac.
(transclos 10 g 2 6)	0.13 s	0.03 s 4212 interac. 274 proper interac.
(transclos 15 g 5 10)	6.58 s	0.10 s 8818 interac. 662 proper interac.
(transclos 18 g 17 18)	57.67 s	0.22 s 17901 interac. 1406 proper interac.
(transclos 20 g 5 15)	explodes	0.18 s 11709 interac. 864 proper interac.
(transclos 20 g 20 1)		0.23 s 20531 interac. 1590 proper interac.

graph. The Roy–Warshall algorithm is based on the iteration over all nodes *n* of the graph of a function (phi *ng*) that connects the nodes a and b if and only if a is connected to *n* and *n* is connected to b.

```
{(iterate n g f) = (g n (g n-1 (g n-2 (... (g 1 f))))) }
def iterate = letrec it = \n.\g.\f. if n == 0
                                    then f else (g n (it (n-1) g f));;

{Roy-Warshall's function phi}
def phi = \n.\g.\a.\b. let ga = (g a) in
                       if (((ga n) == 1) and ((g n b) == 1))
                          then 1 else (ga b);;

{transitive closure of a graph with n nodes}
def transclos = \n. (iterate n phi);;
```

As input graph, we take a graph g where each node n is connected to its predecessor n-1: `def g = \n.\m. if (n == (m+1)) then 1 else 0`. The results of some significative computations are illustrated in Table 2.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Example 7.3

Suppose we represent an array as a function, considering the index as an argument. The following code contains a mergesort algorithm for this representation of arrays. The integer 9999 is used as a marker for the end of the array. Here is the source code:

```
{merge of two "arrays"}
def merge = letrec merge1
          = \f1.\f2.\i.
               let f11 = (f1 1) in
               let f21 = (f2 1) in
                 if f11 < f21 then
                    if i == 1 then f11
                              else (merge1 \x.(f1 x+1) f2 i-1)
                 else
                    if i == 1 then f21
                              else (merge1 f1 \x.(f2 x+1) i-1)

{mergesort}
def mergesort = letrec mergesort1
              = \f.\m.\n.
              if m == n then \x.if x == 1 then (f m) else 9999
                        else let half = (m+n) div 2 in
                             let f1 = (mergesort1 f m half) in
                             let f2 = (mergesort1 f half+1 n)
                        (merge f1 f2);;
```

As inputs, we consider arrays of different dimensions, all of them initially ordered in a reversed way.

```
{examples of "arrays".}
def venti = \x.21-x;;
def quaranta = \x.41-x;;
def cinquanta = \x.51-x;;
def sessanta = \x.61-x;;
```

Some examples of computations can be found in Table 3.

Example 7.4

Our last example about BOHM's performance is a program for computing Tartaglia's triangle (also and improperly known as Pascal's triangle). The n-th element of the m-th row of this triangle is the n-th coefficient of the m-th power of a binomial. The code is straightforward. We just remark the use of the function 'eval' to 'force' the explicit evaluation of each row. This is needed to share this computation at later steps.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Table 3. *Mergesort*

Input	CamlLight	BOHM
(mergesort venti 1 20 10)	0.30 s	0.30 s 25247 interac. 1890 proper interac.
(mergesort venti 1 20 20)	1.43 s	0.53 s 43788 interac. 3719 proper interac.
(mergesort quaranta 1 40 15)	3.73 s	0.58 s 46955 interac. 3857 proper interac.
(mergesort quaranta 1 40 30)	26.22 s	1.35 s 109244 interac. 9335 proper interac.
(mergesort quaranta 1 40 40)	50.60 s	1.88 s 126713 interac. 11957 proper interac.
(mergesort cinquanta 1 50 25)	32.37 s	1.35 s 105937 interac. 8957 proper interac.
(mergesort cinquanta 1 50 40)	106.12 s	2.15 s 146007 interac. 13408 proper interac.
(mergesort cinquanta 1 50 50)	179.80 s	3.13 s 216234 interac. 20327 proper interac.
(mergesort sessanta 1 60 60)	explodes	4.65 s 293919 interac. 28406 proper interac.

```
{row 0}
def init = \x. if x == 1 then 1 else 0;;

{"eval" evaluates an input function f in the interval 0-n}
def eval = \f.\x. letrec evalaux = \n.
                  if n == 0 then 0 else
                     if x == n then (f n) else (evalaux (n-1));;

{next row in tartaglia's triangle}
def next = \f.\x. (f (x-1))+(f x);;

{tartaglia m n gives the n-th element in the m-th row of
 tartaglia's triangle}
```

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Table 4. *Tartaglia's triangle*

Input	CamlLight	BOHM
(tartaglia 9 5)	0.02 s	0.27 s 33053 interac. 884 proper interac.
(tartaglia 13 7)	0.38 s	0.62 s 81732 interac. 1841 proper interac.
(tartaglia 17 9)	5.25 s	1.12 s 163345 interac. 3280 proper interac.
(tartaglia 20 10)	37.60 s	1.88 s 264865 interac. 4850 proper interac.
(tartaglia 23 12)	explodes	2.78 s 366833 interac. 6558 proper interac.
(tartaglia 35 18)		8.75 s 1172111 interac. 18357 proper interac.
(tartaglia 40 20)		14.73 s 1758878 interac. 26197 proper interac.

```
def tartaglia = letrec tartaux = \m. if m == 0 then init else
                    \x.(eval (next (tartaux (m-1))) x (m+1));;
```

7.2 Garbage collection

In this section we provide some results about the performance of the garbage collection procedure.

Maximum allocation of nodes

In Table 5 we show the maximum dimension reached during the reduction of a same initial graph (the maximum number of nodes allocated at a given time) with and without garbage collection.

We start with some examples in pure λ-calculus. In particular, we consider two 'primitive recursive' versions of the factorial and Fibonacci functions on Church integers. Since the computation stops at weak head normal form, we shall supply some extra-parameters (identities), to get an interesting computation. Here is the source code:

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Table 5. *Maximum allocation during reduction*

	G.C. off	G.C. on
(factprim one I I)	1188	1178
(factprim two I I)	1242	1219
(factprim three I I)	1303	1276
(factprim five I I)	1515	1383
(factprim ten I I)	4544	1797
(factprim (add ten five) I I)	20029	2452
(factprim (add ten ten) I I)	73661	3226
(fiboprim one I I)	1220	1209
(fiboprim two I I)	1247	1234
(fiboprim three I I)	1296	1283
(fiboprim five I I)	1418	1381
(fiboprim ten I I)	2439	1763
(fiboprim (add ten five) I I)	26984	6228
(fiboprim (add ten ten) I I)	288003	57269

```
def I = \x.x;;
def zero = \x.\y.y;;
def one = \x.\y.(x y);;
def two = \x.\y.(x (x y));;
...
def Pair = \x.\y.\z.(z x y);;
def Fst = \x.\y.x;;
def Snd = \x.\y.y;;

def Succ = \n.\x.\y.(x (n x y));;
def Add = \n.\m.\x.\y.(n x (m x y));;
def Mult = \n.\m.\x.(n (m x));;

def Next1 = \p.let n1 = (p Fst) in
              let n2 = (Succ (p Snd)) in
                (Pair (Mult n1 n2) n2);;

def Nextfibo = \p.let n1 = (p Fst) in
                  let n2 = (p Snd) in
                  (Pair (Add n1 n2) n1);

def Factprim = \n.(n Next1 (Pair one zero) Fst);;

def Fiboprim = \n.(n Nextfibo (Pair zero one) Fst);;
```

The previous definitions can be found in the file `examples/purelambda` in BOHM's main directory. Loading this file will cause the initial allocation of 1030 nodes which obviously cannot be erased. The behaviour of the garbage collector is shown in Table 5.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Table 6. *Final allocation space*

	G.C. off	G.C. on	Garbage Op.
(factprim one I I)	1128	1112	10
(factprim two I I)	1146	1095	27
(factprim three I I)	1199	1077	55
(factprim five I I)	1454	1077	100
(factprim ten I I)	4447	1077	265
(factprim (add ten five) I I)	19812	1109	523
(factprim (add ten ten) I I)	73274	1113	838
(fiboprim one I I)	1156	1141	8
(fiboprim two I I)	1144	1101	33
(fiboprim three I I)	1172	1101	43
(fiboprim five I I)	1302	1101	88
(fiboprim ten I I)	3415	1101	623
(fiboprim (add ten five) I I)	26957	1097	6117
(fiboprim (add ten ten) I I)	287972	1101	66423

Final number of nodes

Table 6 shows, for the same terms as before, the dimension of the graph at the end of reduction, and the total number of garbage-interactions performed by the G.C.

The performance of the garbage collection procedure is, in some cases, suprisingly good. Consider, for example, the case of the λ-term *(factprim (add ten ten) I I)*; if the G.C. is active, the remaining nodes after the reduction are only 1113 against a final allocation of over 70,000 nodes. Moreover, and this is particularly interesting, this result is obtained by performing only 838 garbage collecting operations! (the reason is that garbage could be duplicated along the reduction).

Extended λ-calculus

Let us now consider an example involving some syntactical constructs of the extended source language. In particluar, we consider a quicksort algorithm for lists of integers. The function genlist takes an integer n and returns the list of the first n integers in inverse order.

```
{lenght of a list}
def lenght = letrec l = \x.if isnil(x) then 0
                                       else 1 + (l tail(x));;

{listIt f l e applies function f to all the elements of list l}
def listIt = letrec lIt =
    \f.\l.\e.if isnil(l) then e
                         else (f head(l) (lIt f tail(l) e));;

{split a list in two sublists according the specified criterion t}
def partition = \t.\l.let switch =
    \e.\l2.if (t e) then [head(l2),cons(e,head(tail(l2)))]
```

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

Table 7. *Maximum allocation space*

	G.C. off	G.C. on	Garbage Op.
(quicksort (genlist 5))	3109	2542	359
(quicksort (genlist 10))	9442	6762	1259
(quicksort (genlist 15))	20627	13557	2684
(quicksort (genlist 20))	37912	23427	4634
(quicksort (genlist 25))	62547	36872	7109
(quicksort (genlist 30))	95782	54392	10109

Table 8. *Total number of interactions*

	G.C. off	G.C. on
(quicksort (genlist 5))	5551	5393
(quicksort (genlist 10))	18446	17403
(quicksort (genlist 15))	40091	36813
(quicksort (genlist 20))	72111	64623
(quicksort (genlist 25))	116131	101833
(quicksort (genlist 30))	173776	149443

```
                else [cons(e,head(l2)),head(tail(l2))]
        in (listIt switch l [nil,nil]);;

{appends two lists}
def append = letrec a =
    \l1.\l2.if isnil(l1) then l2
                         else cons(head(l1),(a tail(l1) l2));;

{generates a list of n integers in inverted order (from n to 1)}
def genlist = letrec gen = \n.if n == 0 then nil
                                        else cons(n,(gen n-1));;
{returns the sorted list}
def quicksort = letrec qs =
   \l.if isnil(l) then nil else
      if (length l) == 1 then l else
         let l1 = head((partition \x.head(l) <= x tail(l))) in
            let l2 = head(tail((partition \y.head(l) < y tail(l))))
               in (append (qs l1) cons(head(l),(qs l2)));;
```

Table 7 gives the maximum dimension of the graph and the number of G.C. interactions.

Table 8 allows you to compare the total number of interactions with and without G.C.

https://doi.org/10.1017/S0956796800001994 Published online by Cambridge University Press

As noted above, the immediate elimination of garbage generally avoids a lot of useless operations. More precisely, this depends upon the elimination of safe fans, which prevents the duplication of garbage.

8 Conclusions

In this paper, we have presented the main architecture of the Bologna Optimal Higher-order Machine (BOHM). BOHM is the first prototype implementation of Lamping's optimal graph reduction technique extended to a reasonable core of a typical functional language. BOHM is just a high level interpreter written in C; it was not meant to be a real implementation, but just to be effective enough to provide some useful results about the actual performance of Lamping's technique. These tests and benchmarks, partially reported in this paper, look very promising.

A lot of work is obviously left, especially in optimizing the design of Lamping's algorithm and the architecture of the abstract machine. In particular, the following topics look particularly interesting:

1. studying the actual relevance of 'fast propagation rules' for brackets and croissants;
2. getting a better understanding and theoretical status for 'safe' operators;
3. investigating the possibility of collapsing long sequences of control (safe) operators into a single object.

On the other hand, the natural idea of passing from an interpreter to a compiler into native assembly code at present looks a bit premature, considering the current state-of-the-art.

References

Aczel, P. (1978) A general Church–Rosser Theorem. Draft, Manchester.

Asperti, A. (1994) Linear logic, comonads, and optimal reductions. *Fundamenta Informaticae*, Special Issue on 'Categories in Computer Science'.

Asperti, A. (1995) $\delta \circ !\epsilon = 1$: optimizing optimal λ-calculus implementations. *Proc. Sixth International Conference on Rewriting Techniques and Applications – RTA'95*, Kaiserlautern, Germany.

Asperti, A. and Laneve, C. (1993a) Paths, computations and labels in the λ-calculus. *Theoretical Computer Science*, Special issue on RTA'93, Montreal.

Asperti, A. and Laneve, C. (1993b) Interaction Systems II: the practice of optimal reductions. *Technical Report UBLCS-93-12*, Laboratory for Computer Science, Univerity of Bologna.

Asperti, A. and Laneve, C. (1994) Interaction Systems I: the theory of optimal reductions. *Mathematical Structures in Computer Science.*

Asperti, A. and Laneve, C. (1995) Relating λ-calculus translations in sharing graphs. *Proc. 2nd International Conference on Typed Lambda Calculi and Applications – TLCA'95*, Edinburgh, Scotland.

Asperti, A., Danos, V., Laneve, C. and Regnier, L. (1994) Paths in the λ-calculus: three years of communications without understandings. *Proc. LICS'94*, Paris, France.

Danos, V. and Regnier, L. (1993) Local and asynchronous beta-reduction. Proc. 8th Annual Symposium on Logic in Computer Science – LICS 93, Montreal.

Field, J. (1990) On laziness and optimality in lambda interpreters: tools for specification and analysis. *Proc. 17th ACM Symposium on Principles of Programmining Languages – POPL 90.*

Giovannetti, C. (1994) Estensione dell'implementazione ottimale del lambda calcolo con tipi di dato primitivi. *Dissertazione di Laurea*, Università degli Studi di Bologna.

Girard, J. Y. (1986) Linear logic. *Theoretical Computer Science* **50**.

Girard, J. Y. (1988) Geometry of interaction I: Interpretation of system F. In" Ferro, Bonotto, Valentini and Zanardo (eds.), *Logic Colloquium '88.* North Holland.

Girard, J. Y. Geometry of interaction II: Deadlock-free algorithms. *Proc. International Conference on Computer Logic – COLOG 88.* Springer-Verlag.

Girard, J. Y., Taylor, P. and Lafont, Y. (1989) *Proofs and Types.* Cambridge University Press.

Gonthier, G., Abadi, M. and Lévy, J. J. (1992a) The geometry of optimal lambda reduction. *Proc. 19th Symposium on Principles of Programming Languages – POPL 92.*

Gonthier, G., Abadi, M. and Lévy, J. J. (1992b) Linear logic without boxes. *Proc. 7th Annual Symposium on Logic in Computer Science – LICS'92.*

Klop, J. W. (1980) Combinatory reduction system. *PhD Thesis*, Mathematisch Centrum, Amsterdam.

Lafont, Y. (1990) Interaction nets. *Proc. 17th Symposium on Principles of Programming Languages – POPL 90*, San Francisco, CA.

Laneve, C. (1993) Optimality and concurrency in interaction systems. *PhD thesis*, Dip. Informatica, Università di Pisa.

Lamping, J. (1989) An algorithm for optimal lambda calculus reductions. Xerox PARC Internal Report.

Lamping, J. (1990) An algorithm for optimal lambda calculus reductions. *Proc. 17th Symposium on Principles of Programming Languages – POPL 90*, San Francisco, CA.

Lévy, J. J. (1978) Réductions correctes et optimales dans le lambda-calcul. *Thèse de doctorat d'état*, Université de Paris VII.

Lévy, J. J. (1980) Optimal reductions in the lambda-calculus. In: J. P. Seldin and J. R. Hindley (eds.), *To H. B. Curry, Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 159–191. Academic Press.

Leroy, X. and Mauny, M. (1992) *The Caml Light system, release 0.5. Documentation and user's manual.* INRIA Technicl Report.

Naletto, A. (1994) Progetto e realizzazione di un garbage collector per l'implementazione ottimale dei linguaggi funzionali. *Dissertazione di Laurea*, Università degli Studi di Bologna.

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages.* Prentice Hall.

Regnier, L. (1992) Lambda Calcul et Réseaux. *Thèse de doctorat*, Université Paris VII.

Wadsworth, C. P. (1971) Semantics and pragmatics of the λ-calculus. *PhD Thesis*, Oxford University.