Safe Operators: Brackets Closed Forever

Optimizing optimal λ-calculus implementations

Andrea Asperti[1⋆] — Juliusz Chroboczek[2]

[1] Dipartimento di Matematica
Piazza di Porta S. Donato 5, Bologna, Italy
asperti@cs.unibo.it

[2] École Normale Supérieure
45 r. d'Ulm, 75005 Paris, France
jch@clipper.ens.fr

Abstract. Considerations from category theory described in [As94] have permitted to add new rewriting rules for optimal reductions of the λ-calculus [Lam90, GAL92a]. These rules produce an impressive improvement in the performance of the reduction system, and provide a first step towards the solution of the well known and crucial problem of accumulation of control operators. In this paper, after an introduction to optimal reductions, we exhibit the aforementioned problem and prove the correctness of the new rules.

1 Introduction

Almost twenty years ago, Lévy [Le78] proposed a complex notion of *redex family* to formalize the intuitive idea of optimal sharing in the λ-calculus (see also [Le80, AL93b]). The redexes of a term obtained during the reduction would be linked into families, and all members of a single family reduced in parallel, or, equivalently, represented by a single piece of structure. Arguably, the length of his *family reduction* would provide a lower bound to the *intrinsic complexity* of λ-term reduction, i.e. its complexity *in any possible implementation*.

Lévy's work raised two different kinds of problems:

- to get some evidence that his abstract notion of sharing could be feasible; actually, no "traditional" implementation of λ-calculus (environment machines, supercombinators etc.) is able to share all redexes in Lévy's families (see [Fie90] for a discussion).
- to understand whether the lower bound to the complexity of the reduction provided by the length of the family reduction could be actually reached (or at least approached, say, up to some polynomial order).

Note that a positive answer to the first problem is an *essential prerequisite* for a positive answer to the second one. However, it does not imply it, for at least two reasons:

⋆ Partially supported by the ESPRIT Basic Research Project 6454 — CONFER.

– the computational overhead introduced for handling sharing;
– the complexity of beta-reduction, which, in the λ-calculus (and all the more so in family reduction), is not an atomic operation (due to the meta-operation of substitution).

The first problem was solved by Lamping [Lam90] using a complex graph reduction technique. However, he did not provide any measure of the complexity of his normalization algorithm (which actually fails to solve the second problem above).

Lamping's graph rewriting rules can be classified in two main groups:

1. the rules involving application, abstraction and sharing nodes (fan) that are responsible for β-reduction and duplication (we call it the *abstract algorithm*);
2. some rules involving control nodes (brackets and croissants), which are required for the correct application of the first class of rules.

In a sense, the first set of rules requires an "oracle" to discriminate the correct rule in a critical pair between fans. The second set of rules can be seen as an effective implementation of this oracle. Similarly, the complexity problem can be divided into two clearly distinguished problems: the complexity of the abstract algorithm, and the complexity of implementing the oracle.

This distinction seems particularly appealing for the following reason. Since Lamping's seminal work, many different reduction techniques have been proposed in the literature; all these techniques share the "abstract algorithm" and differ from each other only in the way the oracle is implemented.

As a matter of fact, Lamping's technique fails in general to be polynomial in the number of family reductions *already at its abstract level*. Here, we will merely provide an example of this behaviour, devoting this important subject to a forthcoming paper; on the contrary, the real issue of this paper will be about techniques for reducing the complexity of the "oracle". In particular, the "simplified" implementation proposed in [GAL92a], which is based on a restricted set of rewriting rules, is much less efficient than Lamping's original one: there are examples where this implementation of the oracle may introduce a further *exponential* overhead with respect to Lamping's solution.

Remark. The previous comment is not meant at all to dimimish [GAL92a], whose theoretical relevance is out of question. By pointing out the relations between optimal reductions, Linear Logic [Gi86] and the Geometry of Interaction [Gi88a, Gi88b], it provided an essential breakthrough in the topic, suggesting new and absolutely innovative perspectives. The interested reader might want to consult [GAL92b, As94, AL93b, ADLR94]).

In [As94], we explained that the well known and crucial problem of accumulation of control operators [GAL92a] (the exponential explosion of the bookkeeping work required for implementing the oracle) is due to the absence of a few simple rules. However, adding these rules to the rewriting system seemed problematic (in some cases, it generates deadlocks, that is "wrong" configurations where interacting nodes can never be eliminated).

In this paper, we partially solve this problem by proposing a set "safe" rules, so called because their use is constrained by a safeness attribute of certain operators. Their introduction in the rewriting system produces a really amazing improvement of its performance.

This paper is organized as follows. In Section 2, we provide a few but significant examples of reductions, to show the practical relevance of the new "safe" rules. Moreover, in order to give the gist of optimality and its actual power, we also compare the performance of our system (the Bologna Optimal Higher-order Machine), with two standard (eager and lazy) implementations. Section 3 is devoted to a (hopefully) friendly introduction to the optimal graph reduction technique. In Section 4, we introduce control operators and give the whole set of graph rewriting rules; in Section 5, we present the initial encoding of λ-terms (following [As94]).

The problem of accumulation of control operators is discussed in Section 6. Finally, in Section 7, we explain our solution, and the use of the safe rules, providing their formal semantics and proving their correctness.

2 Examples and Benchmarks

The Bologna Optimal Higher-order Machine (BOHM) is the first prototype implementation of (a variant of) Lamping's graph reduction technique. Although BOHM's source language constitutes the full core of a typical dynamically-typed lazy functional language, including a few data types, conditional expressions, recursive values and lazy pattern matching, we will mainly present examples from pure λ-calculus.

BOHM is a simple interpreter written in the C programming language. The source code is available by anonymous FTP, at `ftp.cs.unibo.it`, in the directory `/pub/asperti`. It should compile with no modifications on most systems having a C compiler, including many personal computers.

The aim of this section is to show the practical relevance of the safe rules by comparing, on a few simple examples the relative performances of the rewriting system with and without these rules. As you will see, the introduction of the safe rules is actually something more than a mere "optimization": in some cases it makes the reduction time linear instead of exponential. With these additional rules (and only with them), Lamping's technique becomes really competitive with respect to more traditional implementation techniques. We shall give some evidence of this fact, comparing BOHM's performance with two fully developed and largely diffused implementations: Caml Light and Yale Haskell.

Caml Light [LM92] is a small, portable implementation of a dialect of the ML language (about 100K for the runtime system, and another 100K of bytecode for the compiler, versions for the Macintosh and the IBM PC are also available) developed at the *Institut National de Recherche en Informatique et Automatique*

(INRIA) at Rocquencourt (France). In spite of its limited dimensions, the performance of Caml Light is quite good for a bytecoded implementation: about five to ten times slower than SML-NJ. We used Caml Light v. 0.5.

Yale Haskell [Ya94] is a complete implementation of the Haskell language developed at Yale University. The Haskell compiler is written in a small Lisp dialect similar to Scheme which runs on top of Common Lisp. Haskell programs are translated into Lisp, and then compiled by the underlying Lisp compiler. We used Yale Haskell Y2.3b-v2 built on CMU Common Lisp 17f.

The choice of Caml Light and Haskell has also been motivated by the fact that they are respectively eager and lazy implementations, as it seems to come as a surprise to many people that optimal reductions can be better than both types of implementations on the same examples.

Our first example is a "primitive recursive" version of the factorial function (on Church integers).

```
let Succ = \n.\x.\y.(x (n x y));;
let Mult = \n.\m.\x.(n (m x));;
let Pair = \x.\y.\z.(z x y);;
let Fst = \x.\y.x;;
let Snd = \x.\y.y;;
let Nextfact = \p.let n1 = (p Fst) in
                  let n2 = (Succ (p Snd)) in
                      (Pair (Mult n1 n2) n2);;
let Fact = \n.(n Nextfact (Pair one zero) Fst);;
```

Since the evaluation stops at weak head normal forms, we have to supply enough arguments (identities, for instance) to get an interesting computation. The result of the test is in Fig. 1. The first column of the table is the input term. The two following columns respectively refer to the optimal implementation without the safe rules (old version), and with them (new version). Note that the performance of our old version (the one described in [As94]) is similar to the performance of the reduction system considered in [GAL92a] (the formal, syntactical correspondence between these translations has been recently settled in [AL95]). The last columns refer to Caml Light and Haskell respectively. Each entry of the table contains the user time required for reducing the input by each system (on a Sun Sparcstation 5). In the case of optimal systems, we also give the number of fan interactions (a measure of the complexity of the abstract algorithm), as well as the total number of interactions (i.e. of elementary operations) executed during the computation, and the length of the family reduction. The latter datum is the number of redex-families (i.e. *optimally shared* β-reductions) required for normalizing the term. This is the same as the total number of interactions between application and lambda nodes in the graph. Note that this is obviously the same for the two optimal implementations.

The improvement provided by the safe rules is amazing. The old version explodes for input seven, Caml Light for input ten, while with the new version we compute the factorial of twenty in less than six seconds.

Input	Optimal Implementation		Caml Light	Haskell
	As94 (GAL92)	new version		
(fact one I I)	0.00 s. 561 interact.	0.00 s 273 interact.	0.00 s	0.00 s
	71 fan interactions 20 family reductions			
(fact three I I)	0.04 s. 3375 interact.	0.01 s 983 interact.	0.00 s	0.01 s
	251 fan interactions 45 family reductions			
(fact five I I)	0.23 s. 17268 interact.	0.04 s 2380 interact.	0.02 s	0.05 s
	412 fan interactions 74 family reductions			
(fact seven I I)	explodes	0.13 s 5625 interact.	0.17 s	0.84 s
	685 fan interactions 107 family reductions			
(fact nine I I)		0.26 s 12402 interact.	10.60 s	explodes
	1062 fan interactions 144 family reductions			
(fact ten I I)		0.38 s 17916 interact.	explodes	
	1297 fan interactions 164 family reductions			
(fact twenty I I)		5.58 s 307839 interact.		
	6139 fan interactions 425 family reductions			

Fig. 1. Factorial

Unfortunately, things are not always that good for BOHM. Our next example is a computation of the Fibonacci sequence, defined as follows:

```
let Add = \m.\n.\x.\y.(m x (n x y));;
let Nextfibo = \p.let n1 = (p Fst) in
                  let n2 = (p Snd) in
                  (Pair (Add n1 n2) n1);;
let Fibo = \n.(n Nextfibo (Pair zero one) Fst);;
```

The result of the test is in Fig. 2. Again, there is a substantial improvement with respect to the old version, but in this case Caml Light is still more efficient.

Note that the total number of family reductions (fr) grows quadratically in the case of the factorial function ($\mathrm{fr}(n) = (n^2 + 21n + 18)/2$), while it grows as

Input	Optimal Implementation		Caml Light	Haskell
	As94 (GAL92)	new version		
(fibo one I I)	0.01 s. 513 interact.	0.00 s. 248 interact.	0.00 s.	0.00 s.
	72 fan interactions 20 family reductions			
(fibo four I I)	0.06 s. 3152 interact.	0.02 s. 868 interact.	0.00 s.	0.01 s.
	244 fan interactions 55 family reductions			
(fibo seven I I)	0.19 s. 12749 interact.	0.05 s. 2209 interact.	0.01 s.	0.01 s.
	513 fan interactions 98 family reductions			
(fibo ten I I)	0.74 s. 52654 interact.	0.19 s. 6456 interact.	0.02 s.	0.04 s.
	1260 fan interactions 173 family reductions			
(fibo thirteen I I)	3.08 s. 230548 interact.	0.71 s. 24487 interact.	0.04 s.	0.15 s.
	4023 fan interactions 390 family reductions			
(fibo sixteen I I)	explodes	2.94 s. 95199 interact.	0.12 s.	0.85 s.
	15242 fan interactions 1177 family reductions			
(fibo nineteen I I)		13.11 s 385911 interact.	0.44 s.	3.69 s.
	62349 fan interactions 4404 family reductions			

Fig. 2. Fibonacci

Fibonacci for the Fibonacci function ($\mathrm{fr}(n) = \mathrm{fib}(n)+11n+8$). In the first case, the optimal compiler is able to profit of the big amount of sharing, obtaining very good performance on big input values, while in the second case the computation is already intrinsically exponential, and Caml Light takes advantage of its simpler abstract reduction model, avoiding book-keeping operations that in this case are useless. Note in particular that the execution time in the Caml Light system is *linear* in the number of family reductions (i.e. optimal in the strongest sense) in the case of Fibonacci, so this term is surely *one of the best examples we could choose for Caml Light.*

Having considered an example particularly favorable to Caml Light and other standard implementation models, let us now see a couple of examples particularly suited to Lamping's technique.

Let us first consider the term $g = \lambda n.(n\ two\ I\ I)$, where two is a Church integer, and I is the identity. In this case (see Fig. 3) the (new version of the) optimal interpreter works linearly w.r.t. its input, while Caml Light and Haskell are exponential (just like the old implementation).

Input	Optimal Implementation		Caml Light	Haskell
	As94-GAL92	new version		
(g one)	0.00 s. 124 interact.	0.00 s 78 interact.	0.00 s	0.00 s
	23 fan interactions 7 family reductions			
(g four)	0.00 s. 608 interact.	0.00 s 221 interact.	0.00 s	0.01 s
	62 fan interactions 16 family reductions			
(g seven)	0.04 s. 3052 interact.	0.00 s 386 interact.	0.00 s	0.09 s
	101 fan interactions 25 family reductions			
(g ten)	0.43 21176 interact.	0.01 s 551 interact.	0.03 s	0.70 s
	140 fan interactions 34 family reductions			
(g thirteen)	explodes	0.02 s 831 interact.	0.20 s	6.12 s.
	179 fan interactions 49 family reductions			
(g sixteen)		0.02 s 973 interact.	1.60 s	explodes
	218 fan interactions 58 family reductions			
(g nineteen)		0.03 s 1106 interact.	12.65 s	
	257 fan interactions 67 family reductions			

Fig. 3. g = λ n.(n two I I)

Things are even more impressive if we consider the λ-term $f = \lambda n.(n\ two\ two\ II)$. Notice that, in this case, the number of fan interactions is exponential in the number of family reductions: this is therefore a case in which the abstract algorithm is exponential in Lévy's complexity measure.

Before we end this section, we would like to add two real-world examples in which BOHM really shines. For benchmarking those, we used a new experimental version of BOHM which uses the optimizations hinted at at the end of Section

<table>
<tr><th rowspan="2">Input</th><th colspan="2">Optimal Implementation</th><th rowspan="2">Caml Light</th><th rowspan="2">Haskell</th></tr>
<tr><th>As94-GAL92</th><th>new version</th></tr>
<tr><td rowspan="2">(f one)</td><td>0.00 s.
282 interact.</td><td>0.00 s
135 interact.</td><td rowspan="2">0.00 s</td><td rowspan="2">0.00 s</td></tr>
<tr><td colspan="2">42 fan interactions
12 family reductions</td></tr>
<tr><td rowspan="2">(f two)</td><td>0.00 s.
765 interact.</td><td>0.00 s
230 interact.</td><td rowspan="2">0.00 s</td><td rowspan="2">0.02 s</td></tr>
<tr><td colspan="2">67 fan interactions
17 family reductions</td></tr>
<tr><td rowspan="2">(f three)</td><td>0.04 s.
7001 interact.</td><td>0.00 s
442 interact.</td><td rowspan="2">0.00 s</td><td rowspan="2">0.18 s</td></tr>
<tr><td colspan="2">100 fan interactions
22 family reductions</td></tr>
<tr><td rowspan="2">(f four)</td><td>explodes</td><td>0.00 s
947 interact.</td><td rowspan="2">1.02 s</td><td rowspan="2">53.01 s</td></tr>
<tr><td colspan="2">149 fan interactions
27 family reductions</td></tr>
<tr><td rowspan="2">(f five)</td><td>explodes</td><td>0.02 s
2495 interact.</td><td rowspan="2">explodes</td><td rowspan="2">explodes</td></tr>
<tr><td colspan="2">230 fan interactions
32 family reductions</td></tr>
<tr><td rowspan="2">(f six)</td><td></td><td>0.04 s
7944 interact.</td><td rowspan="2"></td><td rowspan="2"></td></tr>
<tr><td colspan="2">375 fan interactions
37 family reductions</td></tr>
<tr><td rowspan="2">(f ten)</td><td></td><td>8.87 s
1676204 interact.</td><td rowspan="2"></td><td rowspan="2"></td></tr>
<tr><td colspan="2">4283 fan interactions
57 family reductions</td></tr>
</table>

Fig. 4. f = λ n.(n two two I I)

7 and described in [Chr95, Gru95]. As they no longer are pure λ-calculus examples, their execution involves reductions which, while not involving control operators, are not family reductions (e.g. boolean and arithmetic operations). In the following tables, we therefore include the total number of *proper* interactions.

The first example is the computation of the transitive closure of a graph represented by its adjacency matrix using Roy-Warshall's algorithm. The BOHM code is as follows (see the execution times in Fig. 5):

```
{(iterate n g f) = (g n (g n-1 (g n-2 (... (g 1 f))))) }
def iterate = rec 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 tranclos = \n. (iterate n phi);;

{the following function represents a graph where each node n
 is connected to its predecessor}
def g = \n.\m. if (n == (m+1)) then 1 else 0;;
```

Input	Optimal Implementation Prototype Version	Caml Light	Haskell
(tranclos 10 g 10 1)	0.04 s 2992 interact. — 2287 fan interact. 649 proper interact. — 78 family reduct.	0.05 s	0.8 s
(tranclos 12 g 12 1)	0.08 s 4138 interact. — 3201 fan interact. 871 proper interact. — 92 family reduct.	0.15 s	3.6 s
(tranclos 15 g 15 1)	0.12 s 6242 interact. — 4897 fan interact. 1264 proper interact. — 113 family reduct.	1.20 s	28.33 s
(tranclos 20 g 20 1)	0.17 s 10842 interact. — 8657 fan interact. 2079 proper interact. — 148 family reduct.	38.40 s	explodes
(tranclos 30 g 30 1)	0.40 s 24592 interact. — 20127 fan interact. 4309 proper interact. — 218 family reduct.	explodes	
(tranclos 40 g 40 1)	0.92 s 45242 interact. — 37697 fan interact. 7339 proper interact. — 288 family reduct.		
(tranclos 50 g 50 1)	1.62 s 73792 interact. — 62367 fan interact. 11169 proper interact. — 358 family reduct.		

Fig. 5. Transitive closure of a graph

Denotational Semantics can be seen as giving a systematical means to derive a functional program from any one written in a language whose semantics is formally defined. However, the programs so generated are never executed for obvious efficiency reasons. The next example is the direct execution of the Denotational Semantics of a toy imperative language. The language has variables (which actually vary), arrays, assignment and `while` loops, and its denotational semantics is produced by a small program written in Lisp (the machine state being simply represented as a function from integer addresses to values). Here is an Erastothenes' sieve implemented in this way (we give the code in Pascal syntax; the syntax we use is a little more unusual). The execution times are shown in Fig. 6.

```
function sieve30: integer;
var i,j: integer;
    a: array[0..30] of integer;
begin
  i:=0;
  while(i<=30) do begin
    a[i]:=1; i:=i+1
  end;
  i:=2;
  while(i<=6) do begin
    j:=2*i;
    while(j<=30) do begin
      a[j]:=0; j:=j+i
    end;
    i:=i+1
  end;
  i:=2; j:=0;
  while(i<=30) do begin
    j:=j+a[i]; i:=i+1
  end;
  sieve30:=j
end;
```

3 Sharing graphs

Initially, in the optimal graph reduction technique, a λ-term is essentially represented by its abstract syntax tree (like in ordinary graph reduction). There are however two main differences:

1. there is an explicit node for sharing;
2. variables are explicitly connected to their respective binders.

Such a data structure will be called *sharing graph*, or simply *graph* for short. For instance, the graph in Fig. 7(1) is the initial representation of the λ-term $M = (two\ \delta)$, where $two = \lambda x.\lambda y.(x(xy))$ and $\delta = \lambda x.(x\,x)$.

The triangle (we will call it *fan*) is used to express the sharing between different occurrences of a same variable. All variables are connected to their respective binders (we always represent this connection on the left of the connection to the

Input	Optimal Implementation Prototype Version	Caml Light	Haskell
sieve10	0.38 s 26649 interact. — 21842 fan interact. 1802 proper interact. — 72 family reduct.	0.58 s	332.77 s
sieve15	0.62 s 56020 interact. — 46747 fan interact. 3453 proper interact. — 72 family reduct.	159.53 s	explodes
sieve20	1.45 s 114251 interact. — 97931 fan interact. 6192 proper interact. — 74 family reduct.	explodes	
sieve25	2.08 s 171133 interact. — 147341 fan interact. 9026 proper interact. — 74 family reduct.		
sieve30	3.48 s 277214 interact. — 242544 fan interact. 13462 proper interact. — 76 family reduct.		

Fig. 6. Execution of the denotational semantics of an imperative program

body). Since multiple occurrences of a same variable are shared by fans, we have a single edge leaving a λ towards its variables. Each node in the graph (@, λ and fan) has therefore exactly three distinguished sites (*ports*) through which it will be connected to other nodes.

We will now illustrate the main ideas of Lamping's optimal graph reduction technique by showing how the abstract algorithm would work on our sample term ($two\,\delta$). As we will see, a crucial issue will remain unresolved, and we shall postpone its solution until the next section, where the concrete algorithm will be described.

Lamping's algorithm consists of a set of local graph rewriting rules. At a given stage of the computation, we can usually have several reducible configurations in the graph. In this case, the choice of next rule to apply is made non-deterministically. This does not matter that much, since, as we shall see, the graph rewriting system satisfies a one-step diamond property (which implies not only confluence but also that all reduction sequence to the normal form whenever it exists, have the same length). In particular, we shall usually choose the next rule in our example of reduction according to a didactic criterion (and sometimes for graphical convenience).

The most important of the graph rewriting rules is obviously the one corresponding to β-reduction: $((\lambda x.M)N) \to M[N/x]$. In graph reduction, substituting a variable x for a term N amounts to explicitly connecting the variable to the term N. Moreover, the value returned by the application before the redex is fired (the link above the application) becomes the instantiated body of the function (see Fig. 8).

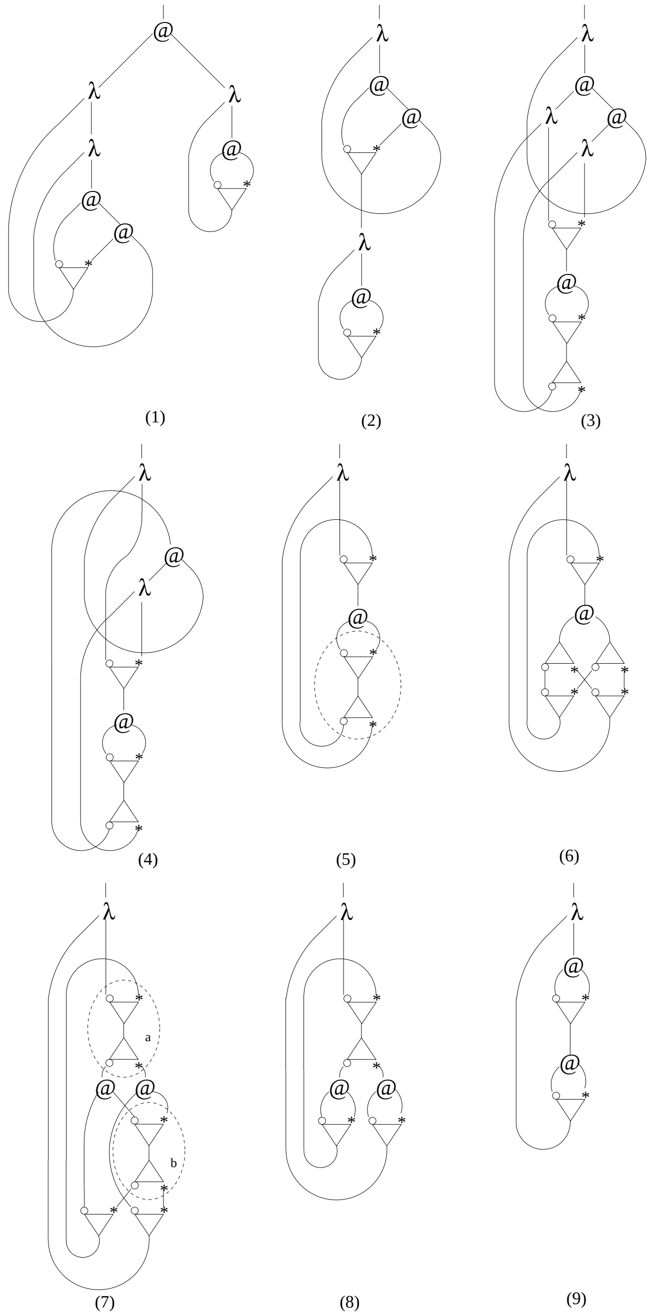


Fig. 7. Reduction of $(two\ \delta)$

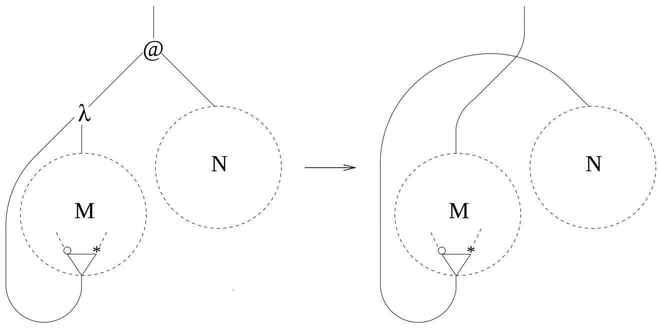


Fig. 8. β-reduction

Note that the portions of graph representing M and N do not play any rôle in the graph reduction corresponding to the β-rule. In other words, the β-reduction rule is expressed by the completely local graph rewriting rule of Fig. 9.

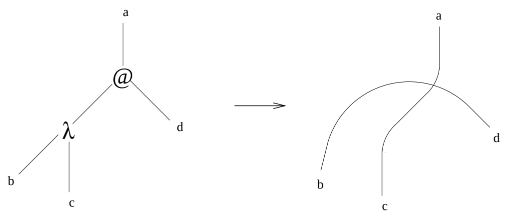


Fig. 9. the β-rule

By firing the outermost β-redex in $(two\,\delta)$, we get the graph in Fig. 7(2). Since the next redex involves a shared λ-expression, we must eventually proceed to the duplication of δ. In ordinary graph reduction, this duplication would be performed as a unique, global step on the shared piece of the graph. On the contrary, the optimal graph reduction technique proceeds in a more lazy way, duplicating the external λ but still sharing its body, as described in Fig. 10.

Note that, since the binder has been duplicated, we have been forced to introduce another fan on the edge leading from the binder to the variable. In a sense, this upside down fan works as an "unsharing" operator (fan-out)[3], which is to be "paired" with the fan(-in) sharing the body of the function. Although there is no operational distinction between a fan-in and a fan-out, their intuitive semantics is quite different; in particular, keep in mind that a fan-out is always supposed to be paired with some fan-in in the graph, delimiting its scope and annihilating its sharing effect. As we shall see, the way the correct pairing between fans is determined is a crucial point of the optimal graph reduction technique. For the

[3] A general definition of in and out nodes will be given at the end of this section.

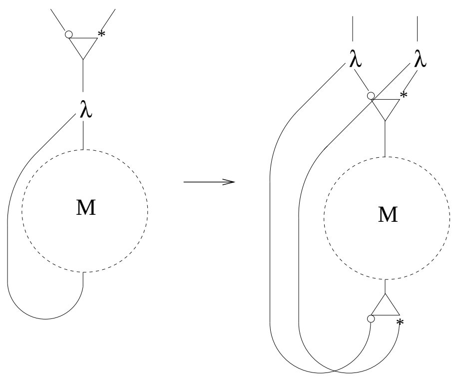


Fig. 10. The "duplication" of $\lambda x.M$

moment, we leave it to the intuition of the reader.

Since, again, the body of the function $\lambda x.M$ does not play any rôle in Fig. 10, the reduction rule can be formally expressed as a local interaction between a fan and a λ, as described in Fig. 11.

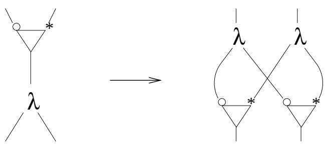


Fig. 11. Fan-λ interaction

Let us now proceed in the analysis of our example. By applying the fan-λ interaction rule, we get the graph in Fig. 7(3). Now, two β-redexes have been created, and by their firing we are led to the graph in Fig. 7(5). We have no more β-redexes in the graph, and no fan-λ interactions, so we must proceed in the duplication process, but we must be very careful here. In particular, the graph rewriting rule in Fig. 12 is strictly forbidden in the optimal implementation

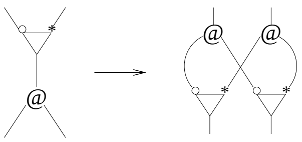


Fig. 12. Non optimal duplication of an application node

technique (although semantically correct). The intuition should be clear: since the shared application could be involved in some redex, its duplication would imply a double firing of the redex.

The only possible interaction left is between the two fans inside the dotted region. This is another crucial point of the optimal graph reduction technique. As we shall see, this interaction must be handled in a different way from the similar interactions in Fig. 7(7). Note in particular that the two fans in Fig. 7(5) are not "paired": the fan-in is a residual of the shared variable of δ, while the fan-out is a residual of the shared variable of two, in the process of duplicating δ. Since the two fans have nothing to do with each other, they must duplicate each other, according to the rule in Fig. 13(2).

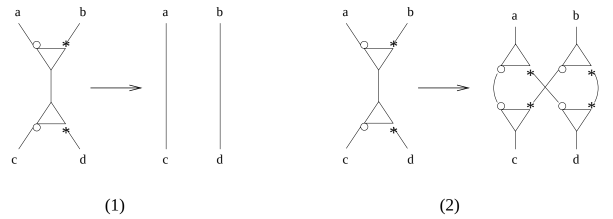


Fig. 13. fan-annihilation rule

Now (Fig. 7(6)), we have a fan-out at the function port of the application. In this case, we can apply the rule in Fig. 14. Intuitively, this rule is correct from

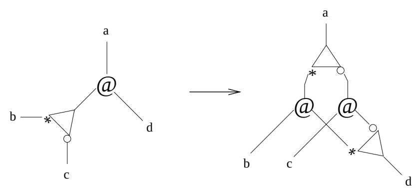


Fig. 14. fan-@ interaction

the point of view of optimal sharing since such a configuration already implies the existence of two unsharable (classes of) redexes for the application.

After applying this rule, we get the graph in Fig. 7(7). Both pairs of fans are now paired: they both belong to the same "duplication process", that has now been (locally) completed. So, the obvious rule, in this case, is to annihilate the paired fans, according to Fig. 13(1).

The problem of deciding which rule to apply when two fans meet (that is the question of how their pairing is established) is a crucial point of the optimal implementation technique. We shall postpone this complex matter for the moment.

By a double application of this rule, we get the graph in Fig. 7, which is in normal form w.r.t. Lamping's algorithm.

Fig. 15 recalls all the graph reduction rules considered so far.

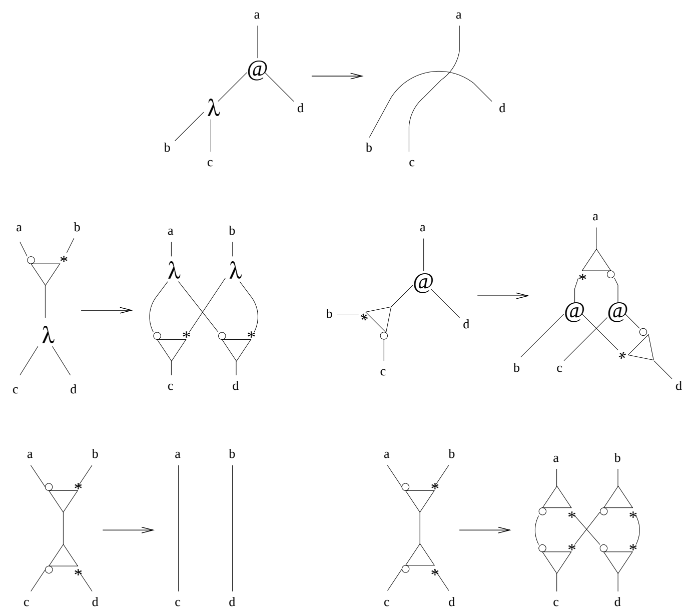


Fig. 15. All the rules of the simplified Lamping's algorithm

The main problem of the optimal graph reduction technique consists in discriminating the application of the two last rules in Fig. 15. By the previous discussion, this amounts to understanding how the fan-in and fan-out are paired in the graph. We suggest that the reader try to think of this problem by himself before proceeding any further. Remark however that the obvious labelling scheme that the reader probably has in mind does not work for the following reason: a single fan-out node can itself be shared, thus representing *two* sharings; an unshared fan-in could be paired with one of those, but not with the other. An example of this situation is given in [Lam89].

Let us also remark that all graph rewriting rules in Fig. 15 can be seen as local interactions between pairs of nodes in the graph. Such graph rewriting systems have studied by Lafont, under the name of Interaction Nets. Note in particular that each node has a unique, distinguished port where it can interact with other nodes. This port, put in evidence by an arrow in Fig. 16, will be called the *principal port* of the node. All other ports will be called *auxiliary*. An operator will be said to be *in* when its principal port is directed towards the root of the graph; is will be said to be *out* otherwise.

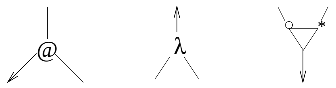


Fig. 16. Principal ports of nodes

4 Pairing fans

In order to solve the problem of correct fan pairing, we add to the bidimensional graphs presented in the previous section a local level structure. Each operator will be decorated with an integer tag which specifies the level at which it lives: two fans match if they meet at the same level; they mismatch otherwise. We furthermore add two new control operators which operate on the level structure: the *croissant*, which opens or closes a level, and the *square bracket* (or *bracket* for short), which temporarily closes a level or restores a temporarily closed one.

More precisely, sharing graphs are unoriented graphs built from the indexed nodes in Fig. 17.

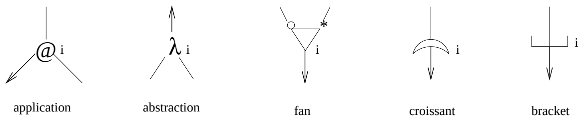


Fig. 17. Sharing graph operators

Two operators (nodes of the graph) are said to *interact* if they meet along their principal ports at the same level. The interaction rules are described in Fig. 18. The first of these is the usual β-reduction. The second one is the matching

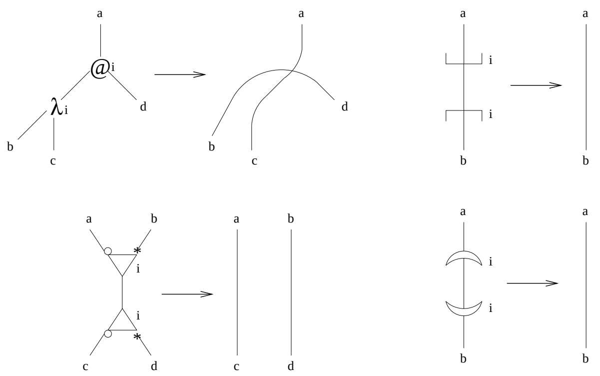


Fig. 18. Interaction rules

of fans. The latter two state respectively that opening and then immediately closing a level, or temporarily closing and then immediately reopening it, has no effect at all.

An operator at a given level can also act upon any other operator f at a higher level (reached at its principal port), according to the rules in Fig. 19 (f represents a generic operator).

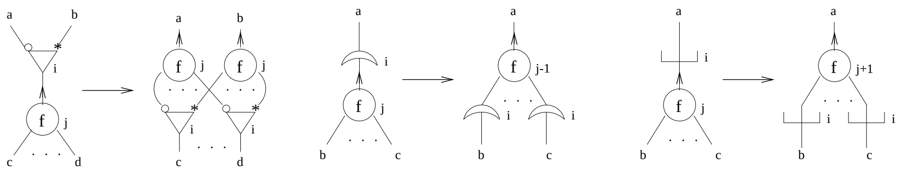


Fig. 19. Action rules

In these rules, the operators are simply propagated through each other in such a way that their effect on the level structure is left unchanged. The terminology is due to the following fact: the active operator can have an effect on the level on which the passive one is found, so the index of the latter can be changed, while the index of the former remains always the same.

Although we made a semantical distinction between action and interaction, all of the graph rewriting rules are local interactions in the terminology of Lafont. We shall often use the term interaction in this more generic sense.

5 The initial encoding of λ-terms

A λ-term N with n free variables is 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 translation is inductively defined by the rules in Fig. 20. The translation function is indexed by an integer which can be thought of as being the level at which we want the root to be; the translation starts at level 0, i.e. $[M] = [M]_0$.

Fig. 21 shows the initial encoding of the λ-term (*two I*).

6 Accumulation of control operators

If you try to reduce the λ-term (*two I*) in the previous graph rewriting system, you will end up with the first graph in Fig. 22, while one would obviously expect to arrive to the configuration (b).

Actually, these two graphs are equivalent w.r.t. the standard semantics for sharing graphs, provided by means of contexts [Lam90, GAL92a, ADLR94] (intuitively, the first one contains operators that open and then temporarily close levels that are never to be opened again). Accumulation of control operators

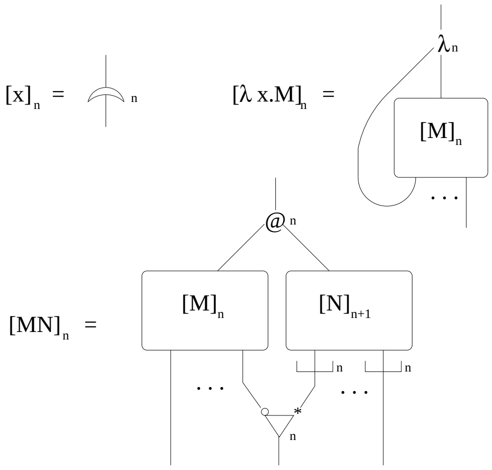


Fig. 20. Initial translation

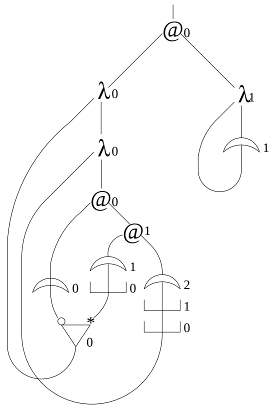


Fig. 21. Initial translation of (*two I*)

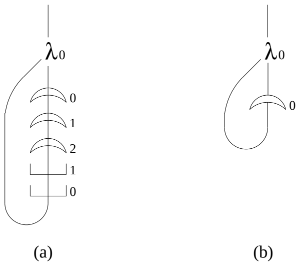


Fig. 22. Two representations of the identity

such as in the second graph is the main source of inefficiency in the optimal graph reduction technique, and we are therefore interested in eliminating as much redundant information as possible.

Let us consider for instance the configuration in Fig. 23. One immediately

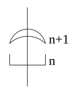


Fig. 23.

realizes two important properties of such a configuration, namely:

1. the pair of control operators can be always propagated together through other operators in the graph (that is, they can be regarded as a unique compound operator);
2. the total effect of this "compound" operator on the nodes they act upon is null (in the sense that they do not modify their level).

Recall now that the only problem of the optimal reduction technique is to guarantee the correct matching between fan-in and fan-out. Since two fans match if and only if they meet at the same level, and the "compound" configuration above has no total effect on levels, the natural idea would be to simply remove them from the graph, according to the rewriting rule in Fig. 24.

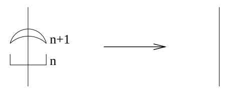


Fig. 24.

Unfortunately, this rule is not correct in general. The reason is very simple: we forgot to take into account the possible annihilation of the innermost operator, as described by the critical pair in Fig. 25. Obviously, only the first reduc-

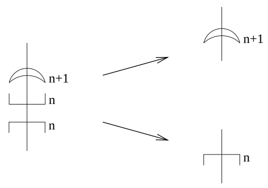


Fig. 25.

tion is correct, while the second one could easily lead to wrong and irreducible configurations.

Let us show that this can actually be a problem in practice. Consider the term

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

where I is the identity and P is an arbitrary term. In Fig. 26(a) we see the

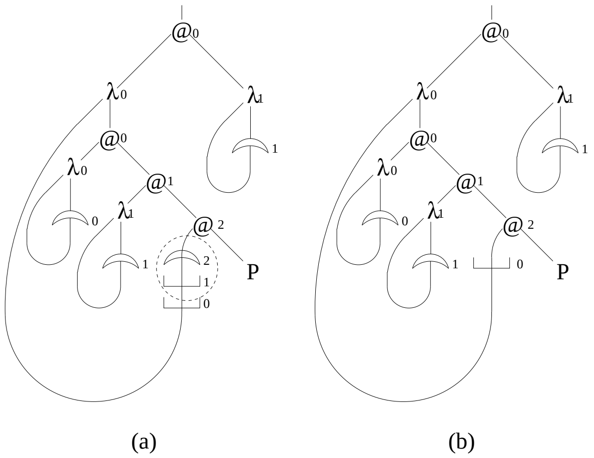


Fig. 26.

initial translation of this term; by applying the rule in Fig. 24 to this graph (see the dotted region), we obtain the graph in Fig. 26(b). As we shall see, this simplification is harmless, here. Let us now proceed in the reduction. After firing three β-redexes we get the graph in Fig. 27(a). Now, the bracket traverses the

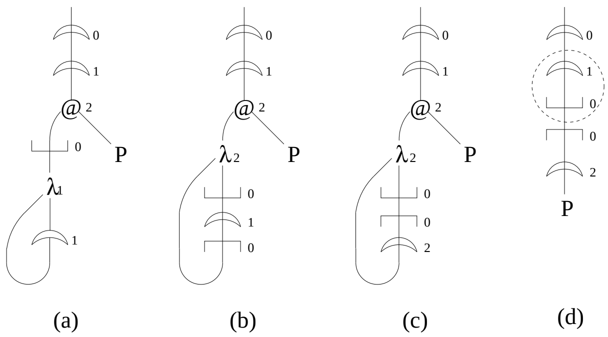


Fig. 27.

λ (b) and the lower croissant (c). Finally, by firing the last β-redex we get our critical pair. Note that if we apply the rule of Fig. 24 in the dotted region, we would get a deadlock between a bracket and a croissant at the same level.

7 Safe operators

The last section suggests that the rule in Fig. 24 cannot be applied when the bracket involved might interact at some later time. Conversely, it seems reasonable to assume that the rule *can* be applied whenever it can be made sure that the eliminated bracket will not interact if left.

Consider for instance the bracket eliminated after Fig. 27(d). Its ancestor in Fig. 27(b) was already ineligible for elimination by the rule of Fig. 24, as it threatened to interact with the bracket below it; in the terms of section 4, the lower bracket reopened a level that had been temporarily closed by the upper one, and eliminating the latter would leave the former reopening a level that had never been closed! In the sequel, we shall say in such a situation that the two brackets *match*, this notion being analogous to that of pairing in the case of fans.

The questions of interest are therefore as follows. First, we must formally define the notion of matching; we do so in section 7.1 after recalling the basic notions of context semantics [Lam89]. Then, we need a simply and efficiently computable class of brackets that can never match — those are the *safe* operators (section 7.2), which do not match by Theorem 25. Having done that, we can exhibit the new rules proposed, which we call the *safe rules*, and prove their correctness whenever the bracket involved is safe.

It is not surprising that these sections are of a more technical nature than the previous ones; the less formally-minded reader might want to skip a few of the less interesting proofs, in particular those near the middle of section 7.2.

7.1 Context semantics and matching

The goal of this section is to formally define the notion of matching; we do so using the context semantics of sharing graphs, as defined in [Lam89].

Definition 1. (Contexts) Labels, levels and contexts are inductively defined as follows:

- Labels:
  - A *label* is one of $\circ, *$.
- Levels:
  - $\Box$ (read *nil*) is a *level*;
  - if l is a label and L a level, then $l \cdot L$ is a level;
  - if L, L' are levels, then $\langle L, L' \rangle$ is a level.
- Contexts:
  - nic (read *neets*) is a *context*;

- if L is a level and C a context, then $\langle C, L\rangle$ is a context;
- furthermore, $\mathrm{nic} = \langle \mathrm{nic}, \square\rangle$.

You might want to think of $\langle\cdot,\cdot\rangle$ as a *cons* operation with its arguments reversed (i.e. $\langle b, a\rangle = \mathrm{cons}(a, b)$). $\square$ is then the empty list, and nic an infinite list of $\square$.

A level of the form $\langle A, B\rangle$ will be called a *capsule*.

Given a context $C = \langle\langle\ldots\langle\langle b, a_n\rangle, a_{n-1}\rangle\ldots a_1\rangle, a_0\rangle$, we will say that a_k is the k-th level of C; we then write $C^{n+1}[b]$ and, similarily, $C^n[\langle b, a_n\rangle]$ (the notation $C^n[b]$ means that b is the n-th tail of C).

Definition 2. (Sublevel of a level) The notion of sublevel of a level (noted $\subseteq$) is inductively defined as follows:

- $L \subseteq L$;
- $l \subseteq l \cdot L$;
- $x \subseteq l \cdot L$ if $x \subseteq L$; and
- $x \subseteq \langle a, b\rangle$ if $x \subseteq a$ or $x \subseteq b$.

Given a context C, we shall say that x is at level n in C if $x \subseteq a$ and $C^n[\langle b, a\rangle]$.

The definition is analoguous to that of a subtree of a labelled tree.

We now define the main concept in context semantics, that of *proper path* in the sense of [Lam89]; it is not to be confused with that of *consistent path*, as defined by [GAL92a]. The reader might want to think of proper paths as formalizing the concepts sketched in section 4.

Definition 3. (Proper path) A proper path in a sharing graph is a path starting at the root, in which every edge is labelled by a context, and satisfying the following properties:

- the context near the root of the graph is nic; and
- the contexts at any two adjacent edges on the path satisfy one of the relations in Fig. 28.

Two paths are said to *coincide* if they are equal after erasing of the labels. A path is said to be *looping* if it crosses the same edge more than once. We shall note $p(V)$ the label of the edge V on path p[4].

Lamping [Lam89] defines a readback procedure for sharing graphs, which is a partial function from sharing graphs to (syntax trees of) λ-terms (we do not here give a definition of the readback). A sharing graph whose readback is defined is said to be *correct*. The readback operates by associating a path starting at the root of the λ tree to any proper path starting at the root of the graph.

[4] This notation is ambiguous when p is a looping path — V might appear *twice* along p. In order to be completely precise we should use $p(q)$, where q is a prefix of p ending at edge V.

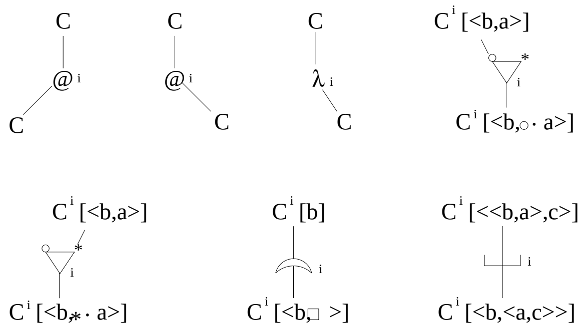


Fig. 28. Context transformations

Definition 4. (Semantics of a correct sharing graph) Given a correct sharing graph G, we call the *semantics* of G the set of proper paths in G.

Notice that there are never two proper paths that coincide in the semantics of a correct sharing graph.

In syntactic rewriting, a *residual* is what remains of a redex after a set of reductions — what remains of it when we move forward in time. Similarly, we are interested in what remains of a label or level when we move along a path; a notion of *spatial residual*.

Our method of defining the spatial residual relation between labels and levels of context *in a single graph* is inspired by the use of an underlined calculus for defining residuals of a redex in the λ-calculus. Directly dictated by the underlined calculus are the *underlined contexts*.

Definition 5. (Underlined contexts) An underlined context is a context in which any label or level can be marked (underlined). More precisely,

- Underlined levels
  - $\Box$ is an underlined level;
  - if L is an underlined level, then $\underline{L}$ is an underlined level;
  - if l is a label and L is an underlined level, then both $l \cdot L$ and $\underline{l} \cdot L$ are underlined levels;
  - if L, L' are underlined levels, then $\langle L, L' \rangle$ is an underlined level.
- Underlined contexts
  - nic is an underlined context;
  - if L is an underlined level and C an underlined context, then $\langle C, L \rangle$ is an underlined context;
  - and, again, $\mathrm{nic} = \langle \mathrm{nic}, \Box \rangle$.

An underlined proper path in a sharing graph is a path starting at the root, in which every edge is labelled by an underlined context, and satisfying the following properties:

– the context at the root of the graph is nic; and
– the contexts at any two adjacent edges on the path satisfy one of the relations in Fig. 28 or Fig. 29.

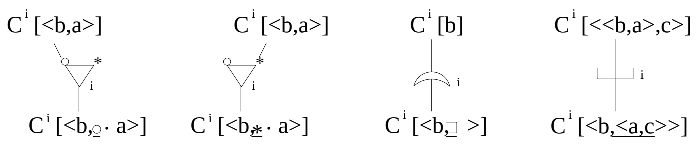


Fig. 29. Underlined context transformations

The set of the sets of proper underlined paths of the correct sharing graph G in which no two paths coincide is ordered by inclusion. A maximal element is called an *underlined semantics* of G.

Notice that the underlined semantics of a given correct graph is not unique. However, all such underlined semantics have the property of being equal to the (non-underlined) semantics of the graph modulo erasing of the underlinings.

The following definition formalizes the idea of a context being less underlined than another one but otherwise equal to it; it is obviously an order relation.

Definition 6. (Ordering on underlined contexts) We define the following order relation on underlined levels:
for any label l and underlined levels L, L', M, M',

– $L \preceq L$;
– $L \preceq \underline{L}$;
– if $L \preceq L'$ then $l \cdot L \preceq l \cdot L'$, $l \cdot L \preceq \underline{l} \cdot L'$, and $\underline{l} \cdot L \preceq \underline{l} \cdot L'$; and
– if $L \preceq L'$ and $M \preceq M'$ then $\langle M, L\rangle \preceq \langle M', L'\rangle$.

This relation is generalized to underlined contexts pointwise; in other words, if L, L' are underlined levels and C, C' are underlined contexts,

– $C \preceq C$; and
– if $C \preceq C'$ and $L \preceq L'$ then $\langle C, L\rangle \preceq \langle C', L'\rangle$.

This relation is generalized to underlined paths by

$$p \preceq p' \text{ if } \begin{cases} p \text{ and } p' \text{ coincide, and} \\ \text{for any two corresponding levels } L \text{ of } p \text{ and } L' \text{ of } p',\ L \preceq L' \end{cases}$$

It is generalized to underlined semantics by

$$S \preceq S' \text{ if for each } p \in S \text{ there exists } p' \in S' \text{ s.t. } p \preceq p'.$$

Notice that in the last case S and S' are equal up to erasing of the underlinings (this is due to the fact that no to paths coincide in an underlined semantics).

Lemma 7. *The set of underlined semantics* $\underline{\mathcal{S}}(G)$ *of a graph* G *ordered by* $\preceq$ *is a complete lattice. Its least element contains no underlined element and is the (non-underlined) semantics of* G*.*

Proof: It is easily checked that any number of underlined levels equal up to underlining have a least upper bound and a greatest lower bound. These operations generalize to underlined contexts, to underlined paths, and therefore to underlined semantics. As underlined semantics of a single graph are equal up to underlining, this implies that $(\underline{\mathcal{S}}(G), \preceq)$ is a complete lattice.□

Definition 8. (Spatial residual of a label/level) Let G be a correct graph, S its semantics and l a sublevel of a level of the context $S(V)$ of an edge V. The set of underlined semantics of G in which l is underlined is a sublattice of $\underline{\mathcal{S}}(G)$; let $\underline{S}_l(G)$ be its least element. Note that every context of an edge of a path in $\underline{S}_l(G)$ has at most one underlined element.

Then, given a path p connecting V to an edge V', let $\underline{p}$ be the unique element of $\underline{S}_l(G)$ s.t. $p \preceq \underline{p}$. The spatial residual of l at edge V' through p is the element of $p(V')$ that corresponds to the unique underlined element of $\underline{p}(V')$ if it exists.

Definition 9. (Matching brackets) Two brackets s, s' match through the path p connecting s to s' if s has at its ports contexts of the form

$$C_1^i[\langle\langle x, a\rangle, y\rangle], C_2^i[\langle x, \langle a, y\rangle\rangle]$$

s' of the form

$$C_1'^i[\langle x', \langle a', y'\rangle\rangle], C_2'^i[\langle\langle x', a'\rangle, y'\rangle]$$

and $\langle a', y'\rangle$ is a spatial residual of $\langle a, y\rangle$; in more vivid terms, s' removes the capsule inserted by s or *vice versa.*

We say that a bracket s matches if there exists a bracket s' such that s and s' match.

Notice that an out bracket can only match with an in one and *vice versa*[5]. We have the desired property of matching brackets.

Lemma 10. *Whenever two brackets are about to interact, they match. Conversely, whenever a bracket is about to act on another one, the two brackets don't match.*

A useful notion for proving that given operators do *not* match is the level of the operator at an arbitrary edge in the graph. Intuitively, it is the index that the operator would have if brought to the considered edge, but this hypothetical operation is not possible in general (think, for instance, what would happen if

[5] In and out operators are defined at the end of Section 3.

the considered operator were to meet a bracket from the secondary port). Again, this notion can be generalized to arbitrary control operators, but we shall only need it in the case of the bracket.

Definition 11. (Level of a bracket at a given edge) Consider a bracket s, an edge V and a path p connecting s to V. The contexts at the ports of s through p are of the form

$$C_1^i[\langle\langle b, a\rangle, c\rangle], C_2^i[\langle b, \langle a, c\rangle\rangle]$$

Then s has level n at V iff $p(V) = D,\ D^n[\langle x, A\rangle]$, where $a' \subseteq A$ and a' is a spatial residual of a.

We shall often omit to specify a path when speaking about spatial residuals and levels, thus speaking e.g. about the level of an operator at an edge.

The important property that relates levels with matching follows.

Lemma 12. *Given two matching brackets s, s', and given an edge V in the graph, then either (i) neither s nor s' has a level at V, or (ii) s and s' have the same level at V.*

7.2 Safe operators

We now define the class of safe operators of a set of sharing graphs. Safeness (the property of being safe) is defined so as to be computable at little cost during the graph reduction process, and, as we shall see, to guarantee that a safe bracket can never match.

Given an operator o in a graph G such that G reduces to G', an operator o' in G' will be said to be a *residual* of o if o' is "what o has become"; a formal definition of a residual will not be given, as it would be a simple but lengthy definition by cases on the reduction rules and by induction on the reduction process.

Definition 13. (Safe operator) Let $\mathcal{O}$ be the set of all the fans[6] and brackets of all the graphs obtained during the reduction of a given term. The set $\mathcal{SO}$ of *safe operators* is the *largest* subset of $\mathcal{O}$ satisfying the following properties:

- no residual of an operator not in $\mathcal{SO}$ is in $\mathcal{SO}$, and
- no residual of an operator that has been duplicated by a λ or an application is in $\mathcal{SO}$.

We now proceed to prove that safe brackets never match. We do so by considering the possible levels of brackets at particular edges in the graph (Lemmata 20 to 22) and use that information to prove that there can be no "collisions" between unrelated brackets in the graph (Lemmata 16 and 24); the main result is in Theorem 25.

[6] The authors believe that the results in this section about brackets and matching apply equally well to fans and pairing.

Lemma 14. *A safe bracket is always in.*

Proof: All the brackets in the initial graph are in, and the rules that create out brackets make them unsafe. More exactly, an out bracket b is the residual of an in one. Consider the last in bracket b_0 of which b is a residual. Then b_0 has an immediate (one step) residual b_1 which is out and such that b is a residual of b_1. An examination of the reduction rules shows that b_0 interacted with a λ node, so that b_1 is unsafe, and therefore b is unsafe.□

Lemma 15. *A non β interaction between two nodes does not modify the context at any edge not involved in the interaction.*

Proof: Consider the interacting control operators as a single node N with 2, 3 or 4 ports. During the interaction, this node is replaced by a node N' with the same arity, and such that the contexts as its ports are the same as those at the ports of N. The interaction does not, therefore, affect in any way the remainder of the graph.□

Lemma 16. *Let s be a bracket that does not match; then, after a non β interaction (the residual of) s does not match.*

Proof: A consequence of Lemma 15.□

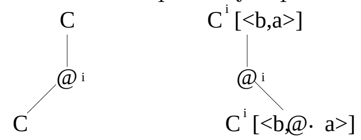
Definition 17. (λ-loop) Let p be a proper path, and L a λ node such that p enters L at its principal port, exits it through its body node, and then ends at its bound node. The p is said to be a *loop* of L if the readback of p is a path entering a λ through its principal node, exiting it through its body node, and ending at the bound port of *the same* λ.

Notice that the complexity of this definition is necessary in order to handle looping paths correctly[7].

Lemma 18. *(transparency) Let L be a λ at level n in a sharing graph, and p a loop of L. Then the first n levels of the context at the body port of L are the same as the first n levels of the context at the end of p.*

Proof: This is a standard property, already stated in [Lam89] and proved e.g. in [Lan95].□

Definition 19. (Level of an application) Consider an application node A and suppose the contexts at its nodes temporarily replaced by:



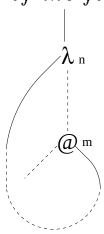
Define the level of A at an edge V as the level of the spatial residual at V of the @ inserted by A; if there is no such residual at V, A is said to have no level at V.

[7] For instance, p might cross the same λ node twice, but the two occurences would map to distinct operators in the readback.

The previous definition is clearly an *ad hoc* one (in particular, contexts so defined do not in general define a full semantics), but it is sufficient for our purpose.

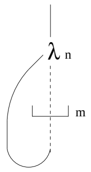
We now give a few properties about the levels of various operators in particular configurations; these will be used later to prove that some operators do not match.

Lemma 20. *Given a configuration of the form:*



the level of the application at the bound port of the λ node is n.

Lemma 21. *Given a configuration of the form:*



if the bracket is safe, then it has a level m' at the bound port of the λ, and $m' = n$.

We simultaneously prove Lemmata 20 and 21.
Proof: We shall proceed by induction on the reduction process. In the initial term, Lemma 20 is obviously verified; the configuration of Lemma 21 can only appear if the bracket is at the bound port of the λ (eventually separated from it by other in brackets or fans), its level clearly being $m' = n$.

Suppose now a reduction of a graph veryfing both properties. By Lemma 15, they are conserved by nonlocal transformations which do not change the path structures. We need therefore only consider a λ node L either during a transformation involving L or a β-reduction, and prove that the residuals of L satisfy both properties after the reduction.

Let therefore L be a λ node with index n. The interaction of L with a croissant or a bracket is transparent. If L is duplicated by a fan, both properties remain true for both residuals. If L is reduced, it has no residuals, so its residuals trivially verify the properties. If L is acted upon by a bracket, then the residuals of the bracket are unsafe, and both properties remain true.

Consider now the β-reduction of a redex, i.e. the interaction between an application node A_r and a λ node L_r, both with index m. The only case in which the reduction does not obviously have the residuals of L satisfy both properties is when A_r is in the body of L and its argument has a reference to the binding of L, as in Fig. 30.

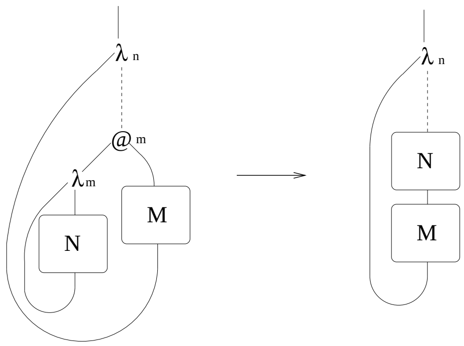
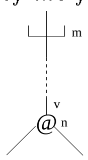


Fig. 30.

Suppose first that the loop of L_r (the one in N) contains an application node. By Lemma 20, its level at the bound port of L_r is m; therefore, its level at the edge V at which the reduction happens is still m, i.e. the same as that of A_r. By Lemma 20, it is n at the bound port of L, which proves that Lemma 20 remains verified.

Let now s be a bracket on the loop of L_r. The property of Lemma 21 being verified by L_r, the level of s at the bound port of L_r is m; after the reduction, by Lemma 20, its level at the bound port of L is n, which proves that Lemma 21 remains verified.□

Lemma 22. *Given a configuration of the form:*



if the bracket is safe and has a level m' at V, then $m' < n$.

Proof: The property is trivially verified in the initial term, as a bracket can only appear at a variable port of a lambda, and therefore never above an application. By Lemma 15, it is conserved by nonlocal non-β transformations.

Consider now a β-redex at level n and a safe bracket s. We are going to show that the property remains true for the residual of s after the reduction of the redex; in doing so, we distinguish the following cases:

(i) Suppose that the bracket is in the right son of the application node. Then, s being in, its level at a given @ does not depend on the incoming context, so the property is preserved.

(ii) Suppose that the bracket is above the application node. Then the β-reduction moves the @ nodes of its right hand son, but, by Lemma 18, the property remains true for any @ at which s has a level.

(iii) Suppose that the bracket is in the left son of the @, but not on the loop of the λ. Then the property remains trivially conserved.

(iv) Suppose now that the bracket is on the loop of the reduced λ. By Lemma 21, its level is n. Obviously, any @ node in the right son of the considered @ has a level higher than n, therefore the property is conserved.□

Lemma 23. *An out bracket always matches.*

Proof: This is a necessary condition for the context at the auxiliary port of the bracket to be defined.□

Lemma 24. *Let s be a safe bracket that does not match. Then, after a β-reduction, (the residual of) s does not match.*

Proof: By Lemma 23, s is in. We shall distinguish the following cases:

(i) s is on the right son of the @ node of the reduced redex. The β-reduction rule does not change the matching of s.

(ii) The β-reduction happens in a subgraph of the son of s. We have a transformation of the type presented in Fig. 31. s matches in M on the r.h.s. iff it

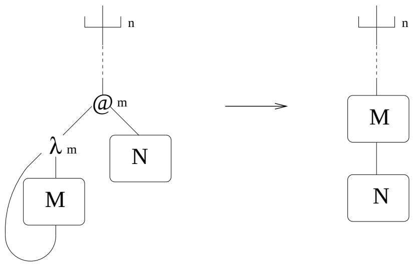


Fig. 31.

matches in M on the l.h.s. By Lemmata 18 and 22, the parts of the context important for the matching of s are not changed by M; therefore, s matches in N on the r.h.s. iff it matches on the l.h.s.

(iii) s is in the body of the reduced λ. Call m the level of the reduced λ-@ pair, and n the level of s at the bound node of the λ. Suppose that, after the reduction, (the residual of) s matches, and call s' the matching out bracket (see Fig. 32). As s does not match in the initial term, (the ancestor of) s' is in N. By Lemma 23, s' has a matching in bracket s''. As s' matches with s, it has a level at the application node, so s'' is above the latter. By Lemma 22, s'' has a level n' at the application node s.t. $n' < m$, therefore $m \neq n$, and so s and s' don't match.□

Theorem 25. *A safe bracket never matches.*

Proof: The initial graph having only in brackets, no bracket present in it can match: the property is therefore true in the initial graph. Furthermore, it is a consequence of Lemmata 16 and 24 that the property is conserved throughout the reduction process.□

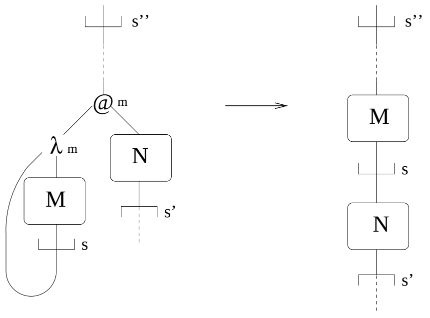
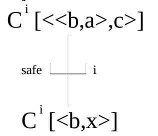


Fig. 32.

7.3 The safe rules

Using the last property, we now give a very strong semantics to safe brackets, inspired by the one of the "limited brackets" of [Lam89].

Theorem 26. *The semantics of safe brackets can be changed to*



for any level x, while retaining the correctness and the readback of the graphs encountered during the algorithm.

Proof: We prove that all the proper paths in the graph are the same under the new and the old semantics. Consider a path p under the old semantics. If it does not traverse any safe bracket, then it is obviously unaffected by the change of semantics. Suppose that it traverses a safe bracket b, so that p can be split at b into $p = p_0 \cdot p_1$. The contexts at the ports of b are of the form

$$C^i[\langle\langle b, a\rangle, c\rangle] \quad C^i[\langle b, \langle a, c\rangle\rangle]$$

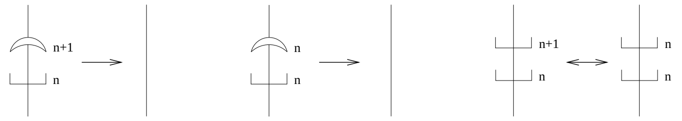
However, as b is safe, it does not match, so that the context at *any* edge along p_1 is of the form

$$C^i[\langle b', \langle a, c\rangle\rangle]$$

In more vivid terms, the information contained in the $\langle a, c\rangle$ is never used, so that it can be replaced by anything — an arbitrary level x.□

We can now exhibit the long awaited safe rules, and prove their correctness.

Theorem 27. *The following rules are correct:*



whenever the lower bracket is safe. Furthermore, in the last case, the upper bracket can become safe.

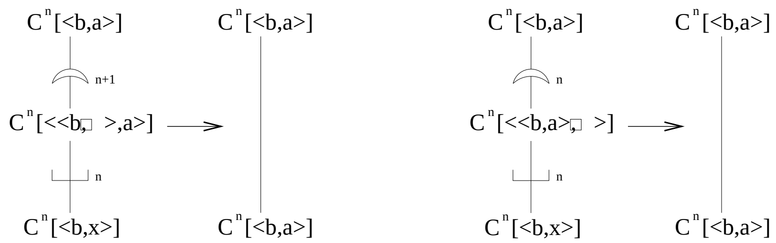


Fig. 33. The eliminated configurations, and their context transformations

Proof: The first two rules can easily be made transparent (i.e. made to satisfy Lemma 15) by applying Theorem 26 to the safe bracket with x taken to be a (see Fig. 33).

Similarily, the last rule is transparent if we take x to be $\Box$, both when the upper bracket is safe and when it is not. Furthermore, in the case where it is unsafe, the information inserted by the upper bracket is made unavailable by the lower bracket; it can not be read by any other bracket, so the upper bracket does not match — it can be made safe.$\Box$

Incidentally, the last statement of the previous theorem (the safeness of the upper bracket in the last rule) has some interesting consequences. In particular, it implies that a tower of brackets with adjacent indices the first of which is safe can be replaced by a tower of safe brackets with same index. We are currently working on a new graph rewriting system with only four operators: applications, λs, fans and a single indexed control operator representing a tower of control operators of equal index to be propagated together. A prototype implementation already exists, and has been used to benchmark the last two examples in section 2; it is described in [Chr95, Gru95].

8 Conclusions

With the addition of the "safe" rules, and with a new version of BOHM being developed at the University of Bologna, it seems that optimal reductions are to become a realistic alternative to traditional implementation techniques for functional languages. This fact is of practical relevance to functional programming.

Indeed, as seen in section 2, there are real-world examples of programming techniques (incremental modification of a functional array represented as a function from indices to values, and direct evaluation of the Denotational Semantics of an imperative language) which are completely unrealistic with traditional implementations while having a reasonable time behaviour when using BOHM. We strongly believe that the introduction of optimal techniques into mainstream functional programming might affect functional programming style.

References

[As94] A. Asperti. *Linear Logic, Comonads, and Optimal Reductions.* Fundamenta Informaticæ, Special Issue devoted to Categories in Computer Science, **22**, 1. 1994.

[As95] A. Asperti. $\delta \circ ! \epsilon = 1$*: optimizing optimal* λ*-calculus implementations.* Proc. of the Sixth International Conference on Rewriting Techniques and Applications (RTA 95), Kaiserlautern, Germany. In: Lecture Notes in Computer Science **914**, 102–116. 1995.

[AL93b] A. Asperti, C. Laneve. *Paths, Computations and Labels in the* λ*-calculus.* Theoretical Computer Science, **142**,2, 277–297. 1993.

[AL94] A. Asperti, C. Laneve. *Interaction Systems I: the theory of optimal reductions.* Mathematical Structures in Computer Science, **4**, 457–504. 1994.

[AL96] A. Asperti, C. Laneve. *Interaction Systems II: the practice of optimal reductions.* Theoretical Computer Science. To Appear.

[AL95] A. Asperti, C. Laneve. *Relating* λ*-calculus translations in sharing graphs.* Proc. of the Second International Conference on Typed Lambda Calculi and Applications (TLCA 95), Edinburgh, Scotland, 1–15. 1995.

[ADLR94] A. Asperti, V. Danos, C. Laneve, L. Regnier. *Paths in the* λ*-calculus: three years of communications without understanding.* Proc. of the 7th annual IEEE Symposium on Logic in Computer Science (LICS 94), Paris, France, 426–436. 1994.

[Chr95] J. Chroboczek. *De l'optimisation des implémentations optimales du* λ*-calcul.* Rapport de stage, DEA IMA, École Normale Supérieure, Paris. 1995.

[DR93] V. Danos, L. Regnier. *Local and asynchronous beta-reduction.* Proc. of the 8th Annual Symposium on Logic in Computer Science (LICS 93), Montréal, 296–306. 1993.

[Fie90] J. Field. *On laziness and optimality in lambda interpreters: tools for specification and analysis.* Proc. of the 17th ACM Symposium on Principles of Programming Languages (POPL 90), 1–15. 1990.

[Gi86] J. Y. Girard. *Linear Logic.* Theoretical Computer Science, **50**, 1– . 1986.

[Gi88a] J. Y. Girard. *Geometry of Interaction I: Interpretation of system F.* In Ferro, Bonotto, Valentini and Zanardo eds., Logic Colloquium 88, 221–260. North Holland. 1988.

[Gi88b] J. Y. Girard. *Geometry of Interaction II: Deadlock-free algorithms.* Proc. of the International Conference on Computer Logic (COLOG 88), Martin Löf and Mints eds. In: Lecture Notes in Computer Sciene, **417**, 76–93. Springer Verlag. 1988.

[GAL92a] G. Gonthier, M. Abadi, J.-J. Lévy. *The geometry of optimal lambda reduction.* Proc. of the 19th Symposium on Principles of Programming Languages (POPL 92), 15–26. 1992.

[GAL92b] G. Gonthier, M. Abadi, J.-J. Lévy. *Linear Logic without boxes.* Proc. of the 7th Annual IEEE Symposium on Logic in Computer Science (LICS 92), 223–234. 1992.

[Gru95] P. Gruppioni. *Sulla gestione della condivisione di espressioni riducibili in linguaggi funzionali.* Dissertazione di Laurea. Dipartimento di Matematica, Università degli Studi di Bologna. 1995.

[Laf90] Y. Lafont. *Interaction Nets.* Proc. of the 17th Symposium on Principles of Programming Languages (POPL 90), 95–108. San Francisco. 1990.

[Lam89] J. Lamping. *An algorithm for optimal lambda calculus reductions.* Internal Report, Xerox Palo-Alto Research Center (PARC). Palo-Alto. 1989.

[Lam90] J. Lamping. *An algorithm for optimal lambda calculus reductions.* Proc. of the 17th Symposium on Principles of Programming Languages (POPL 90), 16–30. San Francisco. 1990.

[Lan95] C. Laneve. *An investigation on the optimal implementation of processes.* Unpublished.

[Le78] J.-J. Lévy. *Réductions correctes et optimales dans le lambda-calcul.* Thèse de Doctorat d'État, Université de Paris VII. 1978.

[Le80] J.-J. Lévy. *Optimal reductions in the lambda-calculus.* In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry, Essays on Combinatory Logic, Lambda Calculus and Formalism*, 159–191. Academic Press. 1980.

[LM92] X. Leroy, M. Mauny. *The Caml Light system, release 0.5. Documentation and user's manual.* Technical Report, Institut National de Recherche en Informatique et Automatique (INRIA), Rocquencourt, France. September 1992.

[Re92] L. Regnier. *Lambda Calcul et Réseaux.* Thèse de Doctorat, Université de Paris VII. 1992.

[Ya94] The Yale Haskell Group. *The Yale Haskell Users Manual.* Yale University. October 1994.