(Optimal) duplication is not elementary recursive

Andrea Asperti
Dipartimento di Scienze dell'Informazione
Via di Mura Anteo Zamboni
40127 Bologna, Italy
asperti@cs.unibo.it

Paolo Coppola   Simone Martini
Dipartimento di Matematica e Informatica
Università di Udine
33100 Udine, Italy
{coppola,martini}@dimi.uniud.it

1 Introduction

In the last ten years there has been a steady interest in *optimal reduction* of λ-terms (or, more generally, of functional programs). The very story started, in fact, more than twenty years ago, with Jean-Jacques Lévy's thesis [Lév78]. The problem he attacked was to find an execution strategy for λ-terms minimizing the number of β-reductions. It was already known that no such strategy exists which reduces only one redex (i.e., a single function call) at the time. Lévy's insight was to discover that a *parallel* optimal strategy exists, the one reducing in a single step all the redexes belonging to the same *family*, a crucial notion he introduced. However, Lévy lacked the programming technology to implement his strategy. He showed it was effective, but no one knew at that time what kind of data structure could be used to dynamically maintain the families in such a way that all the redexes of a given family could be somehow shared and, therefore, reduced as a single step. The solution came in 1989, when independently Kathail [Kat90] and Lamping [Lam90] gave abstract λ-calculus machines which reduced terms as prescribed by Lévy's optimality theory. Lamping's graph rewriting approach is the one that received most interest, and after his breakthrough other variants of optimal reducers have been proposed, especially by Gonthier, Abadi and Lévy [GnAL92a] and Asperti [Asp95]. We will refer to all of them as the *optimal sharing graph approach*.

All these variants share a common core — the *abstract algorithm* in the terminology of [AM98] — and differ in the way they implement the bookkeeping work needed to maintain the families. The algorithms are described as elegant graph rewriting systems, where any rule rewrites only a pair of facing nodes (and then it can be easily implemented as a constant time operation). The abstract algorithm is responsible for the performing of the *shared β-rule* and for the incremental *duplication* of subterms. It is the job of the bookkeeping part — also called the *oracle* — to maintain in the graph enough distributed information to make the abstract algorithm correct with respect to the standard β-reduction (see Section 2, or [AG98] for a complete account).

Since all the reduction rules can be implemented as constant time operations, we may take the number of rewriting steps to reach the normal form of a term as the cost of the reduction algorithm. Lévy's theory, given a λ-term M, prescribes the number of shared β-reductions needed to reach the normal form. All optimal sharing graph algorithms perform exactly this number of β-rewritings, but the global number of reductions is greater, since the algorithms also perform both duplication (in the abstract algorithm) and bookkeeping. Is it possible to bound the total work as a (fixed) function of the number of shared β-reductions? It is this question that has been behind several contributions by Asperti, Lawall, and Mairson [Asp96, LM96, LM97], culminated in [AM98]. Define the Kalmár elementary functions $\mathbf{K}_\ell(n)$ as $\mathbf{K}_0(n) = n$ and $\mathbf{K}_{\ell+1}(n) = 2^{\mathbf{K}_\ell(n)}$. Then [AM98] shows that there are λ-terms that can be normalized with n shared β-reductions, and for which the total work needed to reach the normal form with any algorithm (and hence with any optimal sharing graph algorithm) exceeds $\mathbf{K}_\ell(n)$ for any fixed $\ell \geq 0$.

The theorem shows that optimal β reduction cannot be realized as a unit cost operation. However, it does not mean that optimal reduction, as a whole, is unfeasible. By a classical theorem of Statman [Sta79], any λ-calculus machine has to take non elementary time in the size of a term to reach the normal form. In the case of optimal reduction, the non elementary bound (as a function of the number of shared β-reductions) is just a consequence of the fact that with this sharing technique the number of redex families is surprisingly low (essentially linear in the size of the term). Hence the most part of the work — that has to be done in any case, by Statman's theorem — is devolved to duplication and bookkeeping. Still, in the case of Lamping's algorithm, it remained the problem to understand if this cost was essentially due to duplication, or to the (pretty complex) bookkeeping technique. In this paper, we prove that already the mere cost of duplication is not elementary. In other words, it is not only Lamping's bookkeeping technique to be unfeasible, but already the plain work of duplication. This is particularly interesting since, by now, there is some good evidence that the number of duplications performed by Lamping's algorithm is a lower bound to the complexity of any (sub-optimal[1]) reduction technique [LM97, AL97].

As a working tool, we use in this paper Elementary Affine

Partially supported by EC Grant ERB 4061 PL 97-0244 "LINEAR" and MURST COFIN 1998.

[1] Theoretically, there could be super-optimal implementations, where one *dynamically* looks for common subexpressions. Levy's theory of optimality does not cover these cases, which are neglected as a

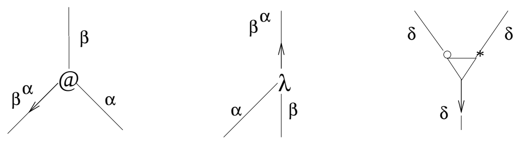


Figure 1: Lamping's (abstract) algorithm nodes.

Logic [Asp98] (EAL), a variant of Elementary Linear Logic, introduced in [Gir98]. In particular, we show that the simply typed λ-terms used in [AM98] to code a certain decision problem can be given a type also in EAL. A crucial feature of the λ-terms typeable in EAL is that their reduction does not need the bookkeeping part of the sharing graph algorithm. Therefore, we can refine the first corollary to the main theorem in [AM98] in the following manner:

> *There exists a set of λ-terms E_n :* Bool *which normalize in no more than n parallel β-steps, where the number of ~~bookkeeping~~* duplication *interactions that are required to normalize E_n using Lamping's graph reduction algorithm grows as $\Omega(\mathbf{K}_\ell(n))$ for any fixed integer $\ell \geq 0$.*

The reader may be confused by the outward paradox of typing non elementary terms in an elementary logic. There is no paradox indeed, since the complexity of reducing terms inside EAL is elementary once the depth of the term is fixed. In our case, the depth of the involved terms grows polynomially in the size of the input.

The paper is organized as follows. In Section 2 we introduce Lamping's graph reduction technology. Section 3 presents Elementary Affine Logic, analyzing the complexity of its cut-elimination and discussing the problem of type inference. Section 4 recalls Asperti and Mairson's result and sketches how to type their terms in EAL, obtaining our main result.

2 Optimal reduction

We assume the reader to have some familiarity with the simply typed λ-calculus. We only recall that the set of types is defined inductively by the grammar $\alpha ::= o | \alpha \to \alpha$, where o is a fixed base type, and the (untyped) terms are generated by the grammar $M ::= x | (M\ M) | \lambda x.M$. We assume to have an enumerable set of variables, ranged over by x. Types can be assigned to terms in the usual way. Reduction is defined by the β-rule: $(\lambda x.M\ N) \to M[N/x]$. We write $\alpha_1 \to \alpha_2 \to \alpha_3$ for $\alpha_1 \to (\alpha_2 \to \alpha_3)$ and $(M_1\ M_2\ M_3)$ for $((M_1\ M_2)\ M_3)$.

While the details of Lamping's approach for optimal reduction are complex and technical (see the book [AG98]), the few concepts needed to understand the result of this paper are easy to grasp. The first step in Lamping's (abstract) algorithm is to represent a term as a labelled graph built out of the nodes given in Figure 1. The arrow exiting a node is the *principal port* of the node. The graph is obtained from the syntax tree of the term by: (i) representing a variable by an arc (a *wire*); (ii) adding an explicit node (the triangular node, called *fan*) for the sharing of a variable occurrence; and (iii) connecting the single wire representing a variable

form of "syntactical coincidence".

$$A \vdash A \ (Ax) \qquad \frac{\Gamma \vdash A \quad A, \Delta \vdash B}{\Gamma, \Delta \vdash B}\ (Cut)$$

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\ (Perm.) \qquad \frac{\Gamma \vdash C}{\Gamma, !A \vdash C}\ (Weak.)$$

$$\frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B}\ (Contr.) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ (\multimap, r)$$

$$\frac{\Gamma \vdash A \quad B, \Delta \vdash C}{\Gamma, A \multimap B, \Delta \vdash C}\ (\multimap, l) \qquad \frac{A_1, \ldots, A_n \vdash B}{!A_1, \ldots, !A_n \vdash !B}\ (!)$$

$$\frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B}\ (\epsilon) \qquad \frac{\Gamma, !!A \vdash B}{\Gamma, !A \vdash B}\ (\delta)$$

Figure 3: Intuitionistic Linear Logic

(after sharing) to the node λ which binds the variable. Reduction of the term is represented by graph rewriting. The (typed) rules of the abstract algorithm are displayed in Figure 2. Note that two nodes are rewritten (they *interact*) only when their principal ports face each other. The first rule is the *shared β-rule*; the others, as it should be clear from their shape, perform an incremental (and controlled, to get optimality) duplication of the graph. When the duplication is over, the first of the two fan-fan rules annihilate the fans (intuitively, the two fans originate from the same node in the starting graph); otherwise, in the last rule, one fan duplicates the other. Observe, however, that this presentation of the algorithm is non deterministic, since the two fan-fan rules have the same left-hand side. To make the algorithm deterministic, one may think to label the fans in the original graph and then, throughout the reduction, apply the first fan-fan rule when the two fans have the same label; apply the other otherwise. It is one of the crucial Lamping's observations that this simple technique does not work for the simply typed (and, a fortiori, the type free) terms. A much more complex *bookkeeping* oracle machinery have to be adopted, adding more nodes (brackets, croissants) to the graphs and decorating any node with an index. The full algorithm, therefore, consists of both the abstract algorithm we just sketched and its bookkeeping part, which we omit. We will introduce in the next section a formal system to type terms for which the simple labelling of fans *is enough* to get correctness.

3 Elementary Affine Logic

3.1 The logical system

Linear Logic [Gir87] provides a logical interpretation of Lamping's algorithm as a cut-elimination process for a suitable notation for proofs (proof-nets). This was first hinted at in [GnAL92a] and then developed in [GnAL92b]. We give here an intuitive introduction to the subject, to motivate the use of EAL as a tool towards our result; see, e.g., Chapters 4 and 8 of [AG98] for a full account on the subject.

Figure 3 presents (in a somewhat non standard way) the intuitionistic fragment of Linear Logic (ILL).

It is a modal system, where the modality (!) is used to mark those formulas on which contraction and weakening may be applied. The standard implicational fragment

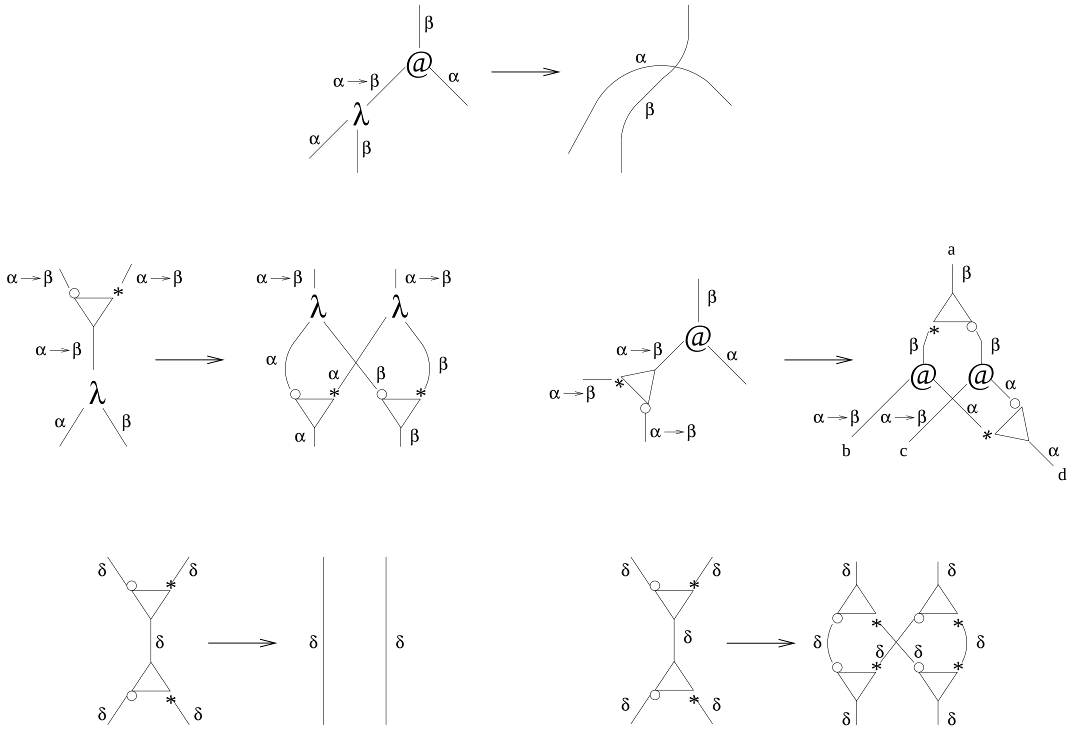
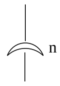
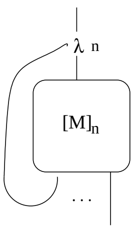


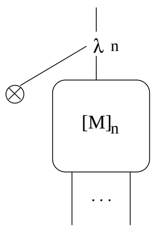
Figure 2: Rules of the abstract algorithm.

$[x]_n =$ 

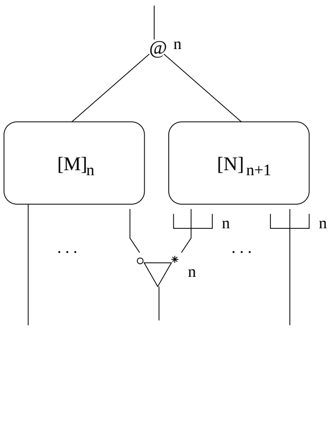
$$\dfrac{x : A \vdash x : A}{x :!A \vdash x : A}\ \epsilon$$

$[\lambda x.M]_n =$ 

$$\dfrac{\begin{array}{c}\vdots\\ !\Gamma, x :!A \vdash M : B\end{array}}{!\Gamma \vdash \lambda x.M :!A \multimap B}\ (\multimap, r)$$

$[\lambda x.M]_n =$ 

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ !\Gamma \vdash M : B\end{array}}{!\Gamma, x :!A \vdash M : B}\ (Weak.)}{!\Gamma \vdash \lambda x.M :!A \multimap B}\ (\multimap, r)$$

$[MN]_n =$ 

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ !\Theta, !\Gamma \vdash N : A\end{array}}{!!\Theta, !!\Gamma \vdash N :!A}\ !}{!\Theta, !\Gamma \vdash N :!A}\ \delta \qquad \dfrac{}{x : B \vdash x : B}\ Ax}{y :!A \multimap B, !\Theta, !\Gamma \vdash (y\ N) : B}\ (\multimap, l) \qquad \begin{array}{c}\vdots\\ !\Theta, !\Delta \vdash M :!A \multimap B\end{array}}{!\Theta, !\Theta, !\Delta, !\Gamma \vdash (M\ N) : B}\ Cut}{!\Theta, !\Delta, !\Gamma \vdash (M\ N) : B}\ Contr.$$

Figure 5: $[\cdot]_n$ translation.

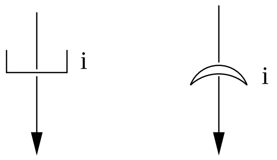


Figure 4: Bracket and croissant.

of Intuitionist Logic may be embedded into ILL via the interpretation $A \to B \equiv\, !A \multimap B$. By this, and the Curry-Howard isomorphism, for any typed λ-term we may give a corresponding ILL proof. Such proofs may be expressed as graphs (proof-nets), essentially corresponding to the syntax tree of the λ-terms but for the presence of !, which is expressed by the so-called *boxes.* Boxes are given in standard ILL proof-nets as global entities (they are certain subgraphs with specified nodes as interfaces), but they can also be described in a local way, by introducing new nodes (and node indexes) besides the ones corresponding to the connectives of the logic. These new nodes, corresponding to the rules ϵ and δ, are the nodes needed in the oracle of Lamping's algorithm—brackets and croissants, see Figure 4.

A typed term M is translated into an optimal sharing graph $[M]_0$ according to the inductive rules of Figure 5, where also the corresponding ILL derivations are given. The rightmost case corresponds to the abstraction of a variable not appearing in the body; the node $\otimes$ is called *eraser* and can be thought of as a fan with zero premises. Observe, in the application case, how the !-rule is translated into a box: the level of the translation increases from $[N]_i$ to $[N]_{i+1}$.

The rationale of this translation is the following. Since contraction (fan) may be perfomed only on !-formulas, the translation of a variable puts a ! on any variable, getting ready for a *possible* contraction. In the translation of an application, we put a box around the argument N (!-rule) and then, since there is now one more ! in the type of the free variables of N, we apply the rule δ (the bracket), finally contracting the common variables. In view of the isomorphism $A \to B \equiv\, !A \multimap B$, this technique allows the translation of any typed term (and also of untyped terms, provided we see them as having type $O \equiv\, O \multimap O$).

Restrict now the logical system, removing the rules ϵ and δ. We cannot any longer prepare to contract any variable as we did in the translation of Figure 5. The contractible variables will be only the ones that get a ! in their type by means of rule !. And we cannot any longer decrease the number of ! in the type of a variable, since rule δ is no longer there. As a reward, however, there are no brackets and croissants in the translation of any λ-term. We moved from the complete Lamping's algorithm to the abstract one. It remains only to be seen how the application of the two fan-fan rules may be discriminated. It not difficult to see that, in this framework, the level of a node never changes during reduction. Therefore, we may apply the first of the two fan-fan rules (annihilate) if the two fans have the same level; we apply the second one otherwise. In this restricted system, levels behave like labels, and these labels are sufficient to ensure correctness of reduction. If a λ-term M can be typed inside the restricted system obtained by omitting the rules ϵ and δ, then the (optimal) reduction of M can be performed without any need of the bookkeeping.

The λ-terms which can be given a type in this restricted system, therefore, are the natural choice if we want to bound the duplication work only.

What we have called so-far the restricted system, is a (fragment of) Elementary Linear Logic (ELL), sketched by Girard in [Gir98] as part of a work directed to the logical characterization of polynomial and elementary functions. ELL completely describes the elementary functions. From one side, any elementary function can be coded as a proof of ELL; from the other, normalization of ELL proofs is an elementary time procedure, once the box-nesting depth of the proof is fixed.

A more flexible syntax (also for a polynomial logic, LAL) was introduced by one of the authors in [Asp98], by allowing full weakening. The resulting system, Elementary Affine Logic (EAL), is presented in Figure 6, where also λ-terms have been added to the rules. We say that *a term M has an EAL type A under the context* Γ iff $\Gamma \vdash M : A$ is derivable in the system of Figure 6. We remark that these terms are not intended as a notation for proofs in EAL; terms for this purpose can be easily obtained from those for LAL in [Asp98]. Here we use EAL just as a typing system for pure λ-terms. The following is subject reduction.

Fact 1 *If $\vdash M : A$ and $M \to_\beta N$, then $\vdash N : A$.*

Although this may be non obvious at first, there are simply typed terms for which no EAL type can be derived. An example will be given at the end of Section 3.3.

Like ELL, also EAL characterizes the elementary functions: Any elementary function can be represented in EAL, and the normalization procedure is elementary time.

Finally, and this is the crucial issue we need in the following, we stress again that if M gets an EAL type, then M can be reduced by the abstract algorithm by a simple labelling of the fan nodes.

3.2 Complexity of reducing EAL proofs

For lack of space, we cannot enter in the details of the normalization procedure, in this paper. Let us just say that by using the technique "by rounds" introduced by Girard [Gir98] and refining a bit its argument, we obtain that the complexity of the reduction is bounded by the following function:

$$\begin{aligned} a(0) &= s \\ a(n+1) &= a(n) * 2^{a(n)} \end{aligned}$$

where s is the size of the term and n is the nesting level of boxes. The important fact is that we get a tower of exponentials whose height only depends on the nesting depth of the term. Hence normalization is elementary in the size of the input, once the depth is given.

3.3 Decorating terms

It should be clear from the rules of EAL that a derivation of a type in EAL (an *EAL-type*, from now on) for a λ-term M consists of a *skeleton* — given by the derivation of a type for M in the simple type discipline — together with a *box decoration*, introducing a suitable number of !-rules. Such modalities are needed since only !-typed variables can be contracted. Observe, moreover, that a given (simply typed) skeleton has an infinite number of possible decorations, and not all of them are instances of a single, most general one.

$$x : A \vdash x : A \ (Ax) \qquad \dfrac{\Gamma \vdash M : A \quad x : A, \Delta \vdash N : B}{\Gamma, \Delta \vdash N[M/x] : B}\ (Cut) \qquad \dfrac{\Gamma, x : A, y : B, \Delta \vdash M : C}{\Gamma, y : B, x : A, \Delta \vdash M : C}\ (Perm.)$$

$$\dfrac{\Gamma \vdash M : C}{\Gamma, x : A \vdash M : C}\ (Weak.) \qquad \dfrac{\Gamma, x :!A, x :!A \vdash M : B}{\Gamma, x :!A \vdash M : B}\ (Contr.) \qquad \dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B}\ (\multimap, r)$$

$$\dfrac{x_1 : A_1, \ldots, x_n : A_n \vdash M : B}{x_1 :!A_1, \ldots, x_n :!A_n \vdash M :!B}\ (!) \qquad \dfrac{\Gamma \vdash N : A \quad x : B, \Delta \vdash M : C}{\Gamma, f : A \multimap B, \Delta \vdash M[(f\ N)/x] : C}\ (\multimap, l)$$

Figure 6: Elementary Affine Logic

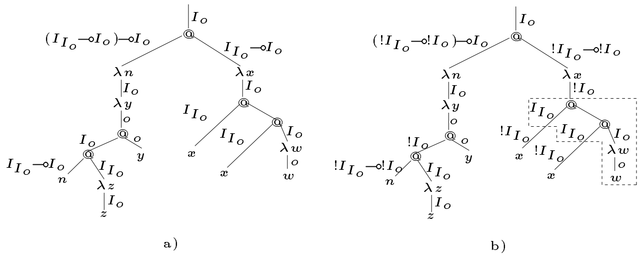


Figure 7: Type inference in EAL, I

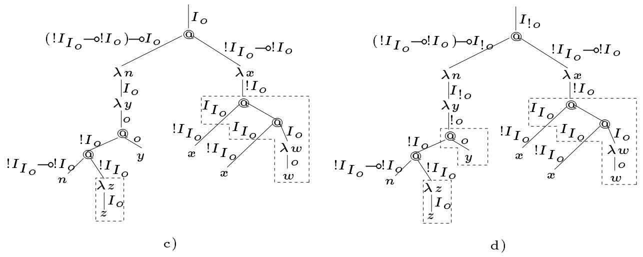


Figure 8: Type inference in EAL, II

Therefore, while the skeleton is trivially obtained, finding the right decoration is the hard part, since the introduction of a box in a portion of the proof forces other boxes to be introduced somewhere else. We conjecture type inference in EAL decidable, but we have not a complete proof of this yet. The typing of the relevant terms in Section 4 is therefore obtained using some heuristics, which we describe in this section.

We may single out three main steps in the process of type inference: "looking for contractions", "boxing arguments" and "opening boxes". We discuss these steps by going through an easy example.

Let

$$N \stackrel{def}{=} (\lambda n.\lambda y.((n\ \lambda z.z)\ y)\ \lambda x.(x\ (x\ \lambda w.w)))$$

be the simply typed term to be typed in EAL. We start from the syntax tree of the term, labelled with the types of the simple discipline (just changing $\rightarrow$ into $\multimap$) as in Figure 7 (a), where $I_\alpha \stackrel{def}{=} \alpha \multimap \alpha$ for every type α.

Now *look for contractions*. If all the variables occur only once — they are used linearly — we are done. This is not the case in our example, since x occurs twice in $M \stackrel{def}{=} \lambda x.(x\ (x\ \lambda w.w))$. As contraction in EAL is admitted only for formulas of type $!\alpha$, we need to introduce a ! before the abstraction of x. Using the usual sequent calculus notation, the *simple* type derivation of M in Figure 7 (a) corresponds to the following derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{o \vdash o}\,(Ax)}{\vdash I_o}\,(\multimap, r) \quad \dfrac{}{I_o \vdash I_o}\,(Ax)}{I_o \multimap I_o \vdash I_o}\,(\multimap, l) \quad \dfrac{}{I_o \vdash I_o}\,(Ax)}{\mathbf{I_{I_o}, I_{I_o} \vdash I_o}}\,(\multimap, l)}{\mathbf{I_{I_o} \vdash I_o}}\ \mathit{Contr?}}{\vdash I_{I_o} \multimap I_o}\,(\multimap, r)$$

which is not in EAL because the contraction rule is wrong. To obtain a correct EAL derivation we add a !-rule before contraction:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{o \vdash o}\,(Ax)}{\vdash I_o}\,(\multimap, r) \quad \dfrac{}{I_o \vdash I_o}\,(Ax)}{I_o \multimap I_o \vdash I_o}\,(\multimap, l) \quad \dfrac{}{I_o \vdash I_o}\,(Ax)}{I_{I_o}, I_{I_o} \vdash I_o}\,(\multimap, l)}{!I_{I_o}, !I_{I_o} \vdash !I_o}\,(!)}{!I_{I_o} \vdash !I_o}\,(Contr.)}{\vdash !I_{I_o} \multimap !I_o}\,(\multimap, r)$$

The corresponding typing is represented in Figure 7 (b), where the type of x inside the box is I_{I_o}, whereas it is $!I_{I_o}$ outside. The new type for M, however, needs to be propagated in the left branch of the tree for the full term N, or otherwise the topmost application would have the wrong type. As a consequence, the variable n in Figure 7 (b) gets type $!I_{I_o} \multimap !I_o$.

Observe now that the leftmost innermost application is wrong. We need to *box the argument* $\lambda z.z$, which must have type $!I_{I_o}$, as it is shown in Figure 8 (c).

Finally, in order to apply $(n\ \lambda z.z)$ of type $!I_o = !(o \multimap o)$ to y of type o, we need to *open the box*, as in Figure 8 (d). As mentioned above, types inside boxes "lose" one !, in particular $!(o \multimap o)$ becomes $o \multimap o$, allowing us to perform the application.

Observe how a single contraction inside the term M forced us to introduce boxes all over the tree.

The final decoration of Figure 8 (d) represents the following derivation in EAL:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{I_o \vdash I_o}\,(Ax)}{\vdash I_{I_o}}\,(\multimap, r)}{\vdash !I_{I_o}}\,(!) \quad \dfrac{\dfrac{\dfrac{}{o \vdash o}\,(Ax) \quad \dfrac{}{o \vdash o}\,(Ax)}{I_o, o \vdash o}\,(\multimap, l)}{!I_o, !o \vdash !o}\,(!)}{!I_{I_o} \multimap !I_o, !o \vdash !o}\,(\multimap, l)}{!I_{I_o} \multimap !I_o \vdash I_{!o}}\,(\multimap, r)}{\vdash (!I_{I_o} \multimap !I_o) \multimap I_{!o}}\,(\multimap, r) \quad \begin{array}{c} \mathcal{E} \\ \vdots \\ \vdash !I_{I_o} \multimap !I_o \end{array}}{\vdash I_{!o}}\,(Cut)$$

where $\mathcal{E}$ is the derivation given above for the subterm M.

As already mentioned, other decorations for N are possible. First, we may give N the type $!^n I_{!o}$, by adding n !-rules at the end of the derivation. Or, to be more general, we may give N the type $!^n I_{!^m o}$, if we introduce $m \geq 1$!'s before the abstraction in the derivation of M. But there are other possibilities. We may choose to introduce m !'s ("close" m boxes) in Figure 8 (d) after the abstraction of y, obtaining for N the type $!^{n+m} I_o$.

Finally we remark that type inference in EAL is not always possible. For example the simply typed λ-term

$$(\lambda n.(n\ \lambda y.(n\ \lambda z.y))\ \lambda x.(x\ (x\ y)))$$

has no EAL decoration. To see this in a simple way, write the term as a sharing graph and reduce it in the abstract algorithm by matching fans by labels. The sharing graph in normal form is *a cycle*, that is a sharing graph which does not correspond to any λ-term (least to say to y, which is the normal form of the given term). This means that the oracle *is* needed for the reduction of this term, and hence it cannot have a type in EAL.

4 Coding type theory into EAL proofs

4.1 Asperti and Mairson's result

The result of Asperti and Mairson [AM98] is obtained out of three building blocks. The first, and most novel, contribution is that any simply typed term can be "pre-compiled" in a certain way in order to drastically limit the number of its shared β-reductions.

Consider the size of types as the structural size, the size of λ-terms counting 1 for each abstraction and application and counting the size of the type for each variable and finally consider the size of sharing graphs as the number of nodes.

Let x be a variable of type σ. The *η-expansion* $\eta_\sigma(x)$ of x is defined inductively on σ as follows:

$$\begin{aligned} \eta_o(x) &\stackrel{def}{=} x \\ \eta_{\alpha_1 \to \cdots \to \alpha_n \to o}(x) &\stackrel{def}{=} \lambda y_1 : \alpha_1 . \ldots \lambda y_n : \alpha_n . \\ &\qquad (x\ \eta_{\alpha_1}(y_1)\ \ldots\ \eta_{\alpha_n}(y_n)) \end{aligned}$$

Given a simply typed lambda-term M, following the procedure in [AM98], we construct a $\beta\eta$-equivalent variant $\mathbf{or}(M)$ — the *optimal root* of M — replacing every subterm of the form $\lambda x : \sigma.E$, where $\sigma \neq o$, with $\lambda x' : \sigma.(\lambda x : \sigma.E\ \eta_\sigma(x'))$. During this process we introduce a certain number — clearly linearly bounded by the size of M — of new *preliminary β-redexes*. $\Delta(M)$ is the sharing graph obtained from $\mathbf{or}(M)$ by reducing all the new preliminary redexes and propagating all sharing nodes to the base type.

Theorem 1 *For any simply-typed λ-term M, the total number of shared β-reductions in the graph normalization of $\Delta(M)$ is limited by the size of $\Delta(M)$.*

The second ingredient is obtained from Mairson's proof [Mai92] of theorems of Statman and Meyer [Sta79, Mey74].

Define $\mathcal{D}_1 = \{\mathbf{true}, \mathbf{false}\}$, and $\mathcal{D}_{k+1} = powerset(\mathcal{D}_k)$. The decision problem for propositional calculus can be naturally generalized to higher-order types by allowing variables and quantifiers to range over values of $\mathcal{D}_k$, for $k \geq 1$. Let x^k, y^k, z^k be variables ranging over $\mathcal{D}_k$; we define *prime formulas* as $\mathbf{true}$, $\mathbf{false}$, $\mathbf{true} \in y^2$, $\mathbf{false} \in y^2$, and $x^k \in y^{k+1}$.

Finally, let a *formula* Φ be built up out of prime formulas, the propositional connectives $\wedge$, $\vee$, $\neg$, and the quantifiers $\forall$ and $\exists$. Statman showed how to reduce the truth of a higher-order formula to the reduction to a given normal form of a suitable typed term. Mairson [Mai92] showed how to simplify the proof of Statman with a different encoding based on the same basic idea — the quantifier elimination procedure — but much simpler and easy to understand for the use of *list iteration* as a quantifier elimination procedure.

Theorem 2 *A higher-order formula Φ is true if and only if its typed λ-calculus interpretation $\hat{\Phi}$: Bool is $\beta\eta$-equivalent to* `true` : Bool. *Moreover, if Φ only quantifies over universes $\mathcal{D}_i$ for $i \leq k$, then $\hat{\Phi}$ has order at most k, and $|\hat{\Phi}| = O(|\Phi|(2k)!)$.*

Finally, the last step is to show that any elementary time-bounded Turing machine can be encoded into higher-order logic, refining the proof in [Mai92].

Theorem 3 *Let M be a fixed Turing machine that accepts or rejects an input x in $\mathbf{K}_\ell(|x|)$ steps. Then there exists a formula Φ_x in higher-order logic such that M accepts x iff Φ_x is true. Moreover, Φ_x only quantifies over universes $\mathcal{D}_i$ for $i \leq (\log^* |x|) + \ell + 6$, and has length $O(|x| \log^* |x|)$.*

¿From the Theorems 1, 2, and 3 the main result of [AM98] is obtained easily. In this paper, we show that Theorems 1 and 2 hold for EAL-typed λ-terms. Then, combining these results with Theorem 3 and the observation we made at the end of Section 3.1, we derive our main result.

Theorem 4 *There exists a set of λ-terms E_n : B which normalize in at most n shared β-reductions, where the number of non β-interactions that are required to normalize E_n using Lamping's* abstract *algorithm grows as $\Omega(\mathbf{K}_\ell(n))$ for any fixed integer $\ell \geq 0$.*

4.2 Coding higher-order logic in EAL

We show in this section how higher-order logic can be coded with EAL-typed λ-terms. The (type-free) λ-terms we use are minor variants of those of [AM98][2], the main technical contribution being the type-inference inside EAL.

Define the type of Booleans as $\mathsf{B} \stackrel{def}{=} !o \multimap !o \multimap !o$. The usual λ-terms `true` and `false` giving the Church encoding of the boolean values can be given in EAL the types $!^n\mathsf{B}$, $\forall n \geq 0$. The usual connectives AND, OR and NOT can be typed, respectively, with type $!^n\mathsf{B} \multimap !^n\mathsf{B} \multimap !^n\mathsf{B}$ and $!^n\mathsf{B} \multimap !^n\mathsf{B}$.

We write L_α^τ for the EAL type of the generic lists of elements of type α:

$$L_\alpha^\tau \stackrel{def}{=} !(\alpha \multimap \tau \multimap \tau) \multimap !(\tau \multimap \tau).$$

[2] The modifications to the encoding are the following: (i) we use different terms for the encoding of equality; and (ii) the variable x_1 is not a prime formula. As a consequence of (ii), one has to adopt also a slightly different encoding of a Turing Machine in the proof of Theorem 3. In particular, define $x^1 < y^1 \stackrel{def}{=} \exists x^2.true \in x^2 \wedge false \notin x^2 \wedge y^1 \in x^2 \wedge x^1 \notin x^2$ and $x^1 = y^1 \stackrel{def}{=} \neg(x^1 < y^1 \vee y^1 < x^1)$. We believe that, with more work, also Mairson's original encoding of the Turing Machine would get a type in EAL.

Following [AM98], for any k the domain $\mathcal{D}_k$ can be encoded by a λ-term $\mathbf{D_k}$ representing $\mathcal{D}_k$ as the list of its values. Define

$$\begin{array}{rcl} \mathbf{D_1} & \stackrel{def}{=} & \lambda c.\lambda n.(c\ \mathtt{true}\ (c\ \mathtt{false}\ n)) \\ \mathbf{D_k} & \stackrel{def}{=} & (\mathit{powerset}\ \mathbf{D_{k-1}}), \end{array}$$

where

$$\begin{array}{rcl} \mathit{powerset} & \stackrel{def}{=} & \lambda A^*.(A^*\ \mathit{double}\ \lambda c.\lambda n.(c\ \lambda c'.\lambda n'.n'\ n)) \\ \mathit{double} & \stackrel{def}{=} & \lambda x.\lambda l.\lambda c.\lambda n.(l \\ & & \quad \lambda e.(c\ \lambda c'.\lambda n'.(c'\ x\ (e\ c'\ n')))\ (l\ c\ n)). \end{array}$$

Lemma 1 *For any EAL-type α, γ and τ and for $i \geq 0$:*

- *double has type* $!^{3+i}\alpha \multimap !(L^{\gamma}_{!^i L^{\tau}_{\alpha}}) \multimap !(L^{\gamma}_{!^i L^{\tau}_{\alpha}})$;
- *powerset has type* $L^{!(L^{\gamma}_{!^i L^{\tau}_{\alpha}})}_{!^{3+i}\alpha} \multimap !!(L^{\gamma}_{!^i L^{\tau}_{\alpha}})$.

Proof: See Figures 9 and 10, where the bold dashed lines stand for i boxes. $\square$

Observe how the piling up of !'s is already present at this stage of the encoding — in the type of *powerset* we have two consecutive !'s. Since in EAL is not possible to derive $!\alpha \multimap \alpha$, the number of consecutive !'s will keep increasing; intuitively, n applications of *powerset* will produce $2n$ consecutive !'s in the final type.

Given the type schema

$$\begin{array}{rcl} \Delta_0 & \stackrel{def}{=} & !^{n_0}\mathtt{B} \\ \Delta_k & \stackrel{def}{=} & !^{n_k}(L^{\tau_k}_{\Delta_{k-1}},) \end{array}$$

we can prove the following:

Lemma 2 $\forall \tau_1, \ldots, \tau_k$ *types in EAL,* $\forall n_0, \ldots, n_{k-1} \geq 0$, $\forall n_k \geq 2(k-1)$, $\mathbf{D_k}$ *has type* Δ_k.

Following [Mai92], quantifiers can be encoded by using iteration over lists. Given $n \geq 0$ and some EAL type σ, suppose to have coded with the λ-terms $\hat{Q} : L^{!^n\mathtt{B}}_{\sigma}$ the set $Q = \{e_1, \ldots, e_m\}$ of elements of type σ; suppose moreover that $\hat{\Psi} : \sigma \multimap !^n\mathtt{B}$ is a term encoding a generic formula Ψ. Then $(\hat{Q}\ \lambda z.(\mathrm{AND}\ (\hat{\Psi}\ z))\ \mathtt{true})$ is the term encoding the formula $\forall z \in Q\ \Psi$.

Prime formulas are encoded by the following terms.

$$\begin{array}{rcl} \mathrm{IFF} & \stackrel{def}{=} & \lambda b_1 :!^n\mathtt{B}.\lambda b_2 :!^n\mathtt{B}.\lambda x :!\sigma.\lambda y :!\sigma.(b_1(b_2\ x\ y) \\ & & (b_2\ y\ x)) :!^n\mathtt{B} \multimap !^n\mathtt{B} \multimap !^n\mathtt{B} \qquad \text{for } n \geq 1 \\ eq_1 & \stackrel{def}{=} & \mathrm{IFF} \qquad \text{with } n = 1 \\ \mathit{member}_k & \stackrel{def}{=} & \lambda x^{k-1}.\lambda y^k.(y^k\ \lambda y^{k-1}.(\mathrm{OR} \\ & & (eq_{k-1}\ x^{k-1}\ y^{k-1}))\ \mathtt{false}) \\ eq_k & \stackrel{def}{=} & \lambda x^k.\lambda y^k.(\lambda op.(\mathbf{D_{k-1}}\ \lambda z^{k-1}.(\mathrm{AND} \\ & & (\mathrm{IFF}\ (op\ z^{k-1}\ x^k)\ (op\ z^{k-1}\ y^k)))\ \mathtt{true}) \\ & & \mathit{member}_k) \end{array}$$

Observe that member_k is defined for $k \geq 2$.

This encoding can be understood as the λ-calculus translation of the following inductive definitions:

$$\begin{array}{rcl} x^1 =_1 y^1 & \stackrel{def}{=} & x^1 \leftrightarrow y^1 \\ x^{k-1} \in_k y^k & \stackrel{def}{=} & \exists z^{k-1} \in y^k\ z^{k-1} =_{k-1} x^{k-1} \\ x^k =_k y^k & \stackrel{def}{=} & \forall z^{k-1} \in \mathcal{D}_{k-1}\ (z^{k-1} \in_k x^k \leftrightarrow z^{k-1} \in_k y^k). \end{array}$$

The quantified formulas $\exists z^{k-1} \in y^k$ in the definition of $\in_k$, and $\forall z^{k-1} \in \mathcal{D}_{k-1}$ in the definition of $=_k$, are encoded by list iteration, as described above.

Lemma 3 *Let*

$$\begin{array}{rcl} M_1 & \stackrel{def}{=} & \mathtt{B} \\ M_k & \stackrel{def}{=} & L^{!^{2k-3}\mathtt{B}}_{!M_{k-1}}. \end{array}$$

Then member_k *has EAL-type*

$$!^{m_1}M_{k-1} \multimap !^{m_2}M_k \multimap !^{m_3}\mathtt{B}$$

for any $m_1 \geq \max\{2k-5, 2\}, m_2 \geq \max\{2k-7, 0\}, m_3 \geq \max\{4k-9, 2(k-1)\}$.

Putting together all the ingredients of the encoding, we obtain our main technical result, which establishes Theorem 2 for EAL-typed terms.

Theorem 5 *Define* $f(k) = \begin{cases} 2k-1 & 1 \leq k \leq 3 \\ 4k-9 & k \geq 4 \end{cases}$.
Let $\Psi[x_1^{k_1}, \ldots, x_n^{k_n}]$ *be the term encoding an arbitrary formula* Φ *with free variables* $x_{i+1}^{k_{i+1}}, \ldots, x_n^{k_n}$, $0 \leq i \leq n$, *and* m *quantifiers over* $\mathcal{D}_1, \ldots, \mathcal{D}_i$. *Then*

$$\exists J \subseteq \{1, \ldots, i\},\ \exists j \in \{1, \ldots, i\},\ \exists d \leq m,$$

such that in EAL

$$\Psi[x_1^{k_1}, \ldots, x_n^{k_n}] :!^{f(k_j)+d+\sum_{\ell \in J} 2(k_\ell - 1)}\mathtt{B}$$

and free variables $(i+1 \leq h \leq n)$

$$x_h^{k_h} :!^{\max\{2k_h-7,1\}+f(k_j)-f(k_h)+d+\sum_{\ell \in J} 2(k_\ell-1)}M_{k_h}.$$

Moreover the size of $\Psi[x_1^{k_1}, \ldots, x_n^{k_n}]$ *is*

$$O(|\Phi|(2k_{\mathrm{MAX}})!)$$

where k_{MAX} *is the greatest* k *such that* member_k *appears in* $\Psi[x_1^{k_1}, \ldots, x_n^{k_n}]$.

Proof: The main case of a universally quantified formula is described in Figure 11, where $p_1 = f(k_j)+d+\sum_{\ell \in J} 2(k_\ell - 1)$, $p_2 = \max\{2k_i - 7, 1\} + f(k_j) - f(k_i) + d + \sum_{\ell \in J} 2(k_\ell - 1)$, the outer bold box stands for $2(k_i - 1)$ boxes[3] and the inner bold box stands for p_1 boxes. Figure 12 is the complete coding of an example. $\square$

The following bound on the number of !'s in the type of a lambda term encoding an arbitrary formula provides a limitation also for the box-nesting depth.

[3]Notice that $\mathbf{D_{k_i}}$ has type $!^{2(k_i-1)}(L^{!^{p_1}\mathtt{B}}_{!^{p_2}M_{k_i}})$ and remember that we need to "open the boxes" before apply.

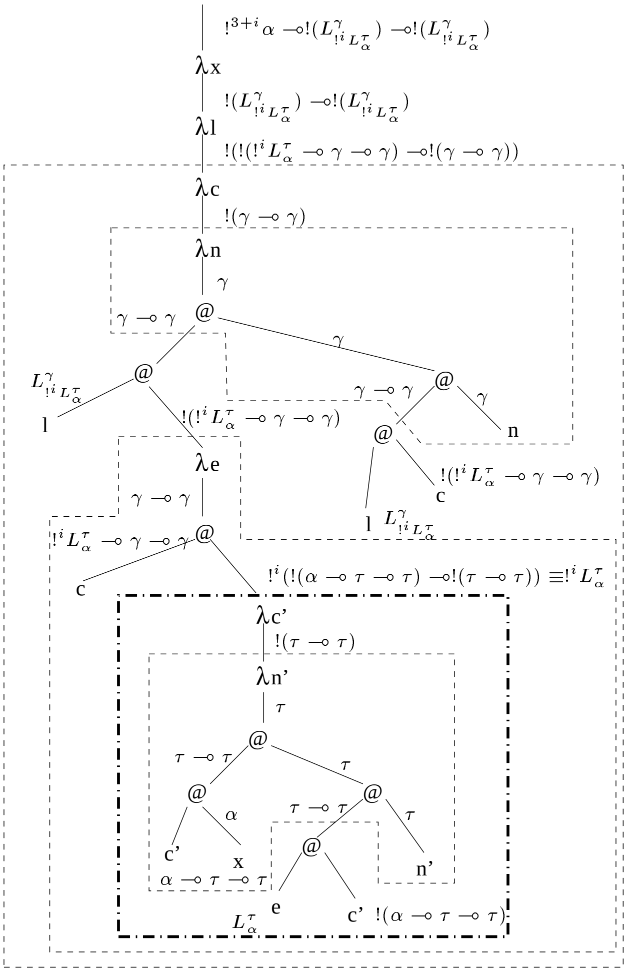


Figure 9: EAL type of *double*

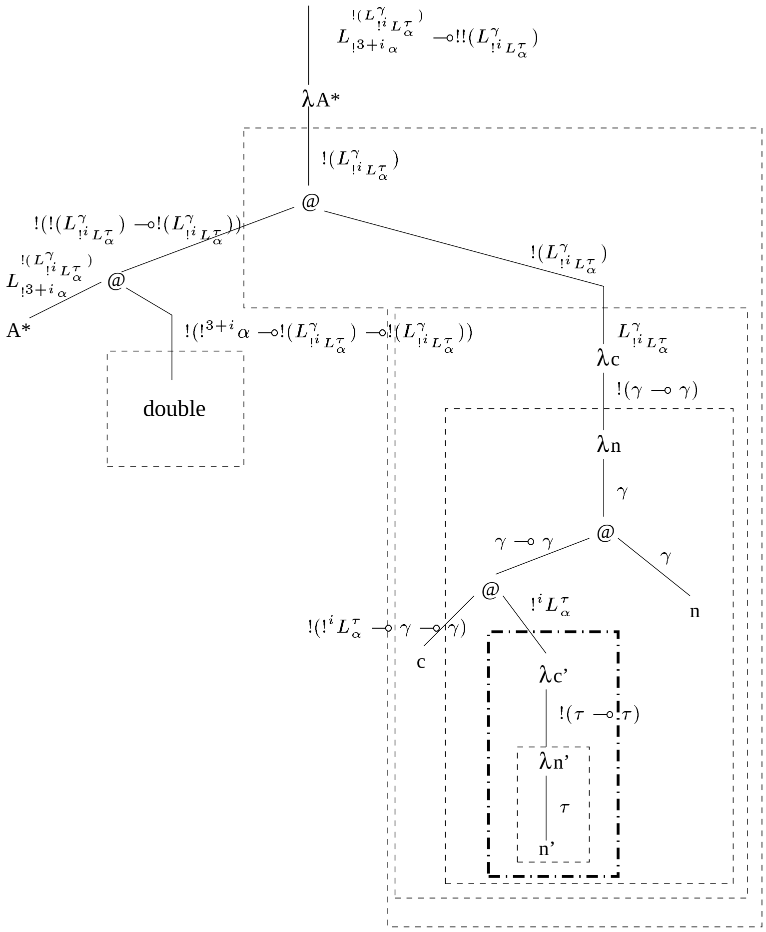


Figure 10: EAL type of *powerset*

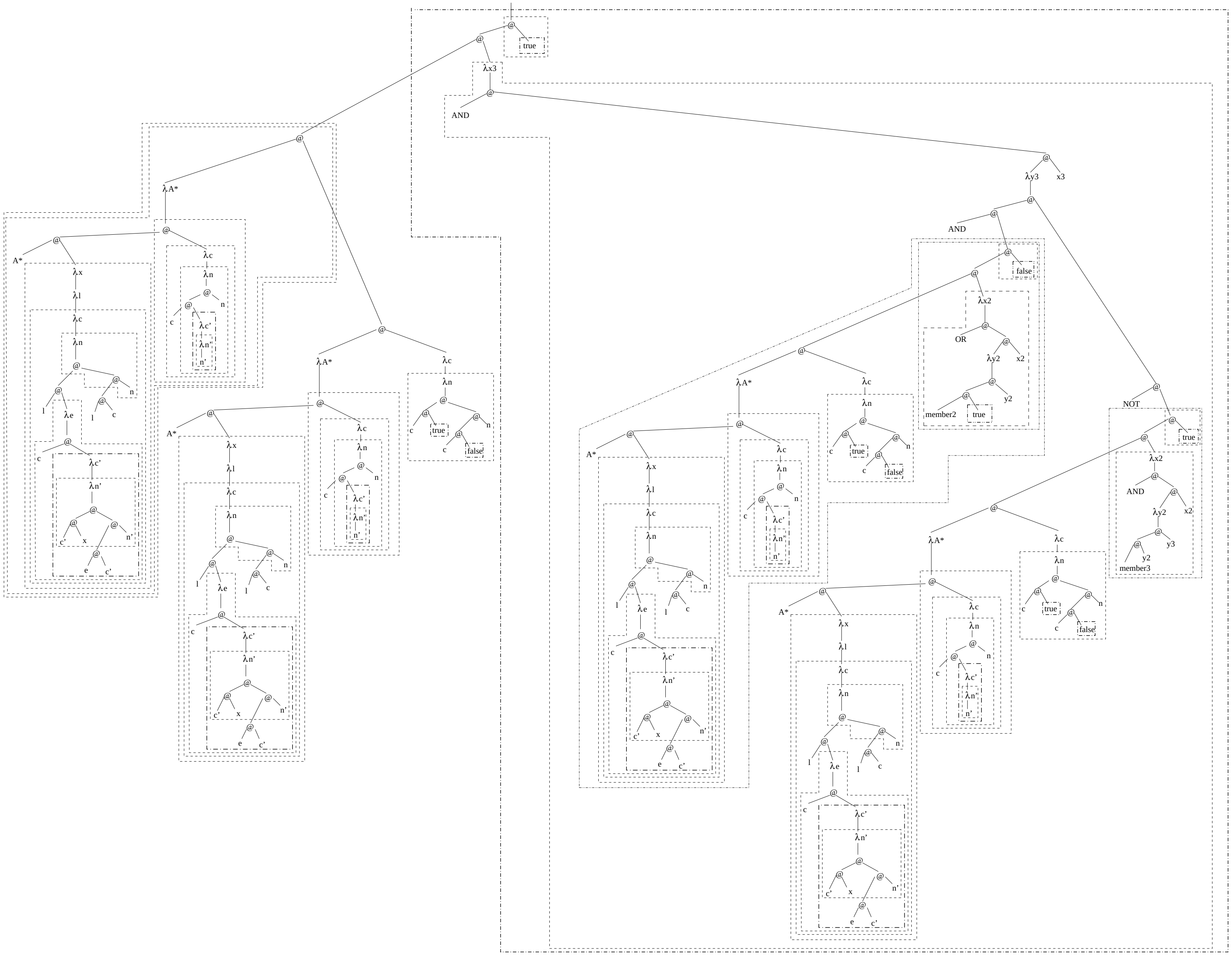


Figure 12: $\forall x^3(\exists x^2\ true \in x^2 \ \wedge\ \neg(\forall x^2\ x^2 \in x^3))$

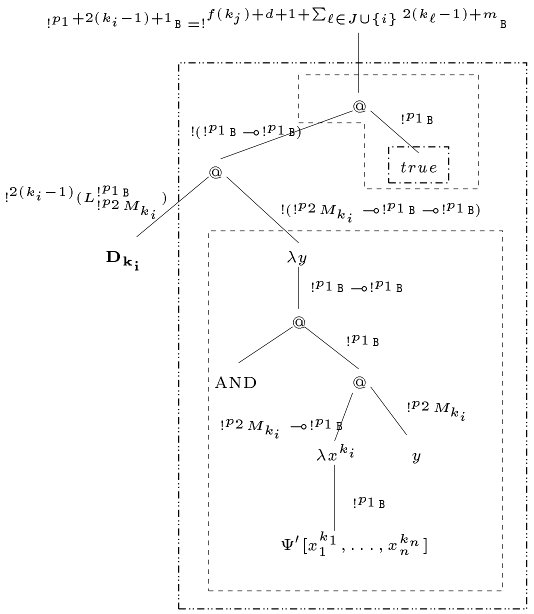


Figure 11: EAL type of an arbitrary formula

Corollary 1 *Let Ψ be a term encoding an arbitrary formula. Then Ψ has type in EAL $!^t\mathtt{B}$ with $t = O(n \cdot k_{\mathrm{MAX}})$ where k_{MAX} is the greatest k such that member$_k$ appears in Ψ and n is the number of quantifiers in Ψ.*

Proof. By the previous theorem Ψ has type $!^t\mathtt{B}$ with $t = f(k_j) + m + \sum_{\ell \in J} 2(k_\ell - 1) \leq f(k_{\mathrm{MAX}}) + n + 2n(k_{\mathrm{MAX}} - 1)$. $\square$

Theorem 6 *Let Ψ be a term encoding an arbitrary formula. Then Ψ has depth $d(\Psi) = O(n \cdot k_{\mathrm{MAX}})$ where k_{MAX} is the greatest k such that member$_k$ appears in Ψ and n is the number of quantifiers in Ψ. Moreover, we have also $d(\mathbf{or}(\Psi)) = O(n \cdot k_{\mathrm{MAX}})$.*

In order to obtain the desired result on complexity of duplication, it remains to be shown that the pre-compilation of the λ-terms given by eta-expansion can be performed inside EAL.

Theorem 7 *If M has an EAL type, so does $\mathbf{or}(M)$.*

Proof: It is sufficient to prove that the η-expansions are always typeable in EAL, as it is described in Figure 13. $\square$.

This is what is needed to obtain Theorem 1 for EAL-typed terms, and, hence, our Theorem 4.

5 Conclusions

In this paper, we have shown that the complexity result for parallel beta-reduction obtained in [AM98] can be obtained with respect to just (optimal) duplication, and not only (as one could have expected) to the complex machinery required for the management of redex-families. This casts a new and different light on Asperti and Mairson's work; in particular, it helps to understand why it does not convey any negative result for optimal reduction as a technique for the normalization of lambda-terms. In any reduction of a lambda term

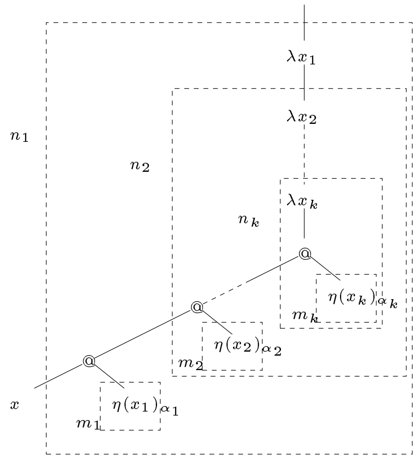


Figure 13: $\eta(x)_{!^{n_1}(!^{m_1}\alpha_1 \multimap !^{n_2}(!^{m_2}\alpha_2 \multimap \cdots !^{n_k}(!^{m_k}\alpha_k \multimap !^{m_k+1} o)\cdots))}$

there is, potentially, a non elementary work of duplication that has to be done, and even an optimal reduction technique, as parsimonious as it could be, cannot do any magic.

A lot of interesting issues are still to be investigated, starting from a theoretical comparison of the performance of Lamping's technique versus more traditional implementations. This comparison has been hindered so far by the "bookkeeping" work, whose formal investigation is just too complex with the current state of the art. But now we have a huge set of lambda terms (i.e. all the terms of EAL) that can be reduced without the need of bookkeeping, providing a main and very promising arena for a theoretical investigation of performance issues.

References

[AG98] Andrea Asperti and Stefano Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[AL97] Andrea Asperti and Cosimo Laneve. On the dynamics of sharing graphs. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 259–269, Bologna, Italy, 7–11 July 1997. Springer-Verlag.

[AM98] Andrea Asperti and Harry G. Mairson. Parallel beta reduction is not elementary recursive. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 303–315, San Diego, California, 19–21 January 1998.

[Asp95] A. Asperti. Linear logic, comonads and optimal reductions. *Fundamentae Informaticae*, 22:3–22, 1995.

[Asp96] Andrea Asperti. On the complexity of beta-reduction. In *Conference Record of POPL '96: The* $23^{\rm rd}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 110–118, St. Petersburg Beach, Florida, 21–24 January 1996.

[Asp98] Andrea Asperti. Light affine logic. In *Proc. of Symposium on Logic in Computer Science*, pages 300–308, 1998.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir98] Jean-Yves Girard. Light linear logic. *Information and Computation*, 204:143–175, 1998.

[GnAL92a] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Conference record of the 19th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 15–26, January 1992.

[GnAL92b] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. Linear logic without boxes. In *Proc. of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 223–234, June 1992.

[Kat90] Vinod K. Kathail. *Optimal Interpreters for Lambda-calculus Based Functional Programming Languages*. PhD thesis, MIT, May 1990.

[Lam90] John Lamping. An algorithm for optimal lambda calculus reduction. In ACM, editor, *POPL '90. Proceedings of the seventeenth annual ACM symposium on Principles of programming languages, January 17–19, 1990, San Francisco, CA*, pages 16–30, New York, NY, USA, 1990. ACM Press.

[Lév78] Jean-Jacques Lévy. *Réductions Correctes et Optimales dans le Lambda Calcul*. Thèse de Doctorat d'Etat, University of Paris VII, 1978.

[LM96] Julia L. Lawall and Harry G. Mairson. Optimality and inefficiency: What isn't a cost model of the lambda calculus? *ACM SIGPLAN Notices*, 31(6):92–101, June 1996.

[LM97] Julia L. Lawall and Harry G. Mairson. On global dynamics of optimal graph reduction. *ACM SIGPLAN Notices*, 32(8):188–??, August 1997.

[Mai92] Harry G. Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 103(2):387–394, September 1992.

[Mey74] Albert R. Meyer. The inherent computational complexity of theories of ordered sets. In *Proceedings of the International Congress of Mathematicians*, pages 477–482, 1974.

[Sta79] Richard Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–81, 1979.