Università degli Studi di Udine
Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

On the Complexity of Optimal Reduction of Functional Programming Languages

Candidate:
Paolo Coppola

Supervisor:
Simone Martini

February 8, 2002

Author's address:

Paolo Coppola
Dipartimento di Matematica e Informatica
Università degli Studi di Udine
via delle Scienze, 206 (loc. Rizzi)
I-33100 Udine, Italy

Tel: +39-0432-558457
Fax: +39-0432-558499

E-mail: `coppola@dimi.uniud.it`

Contents

"... at the time through self-study
I found out about the λ-calculus of
Curry and Church which, literally,
gave me nightmares at first."
Dana Scott

Acknowledgments

If you want to blame someone for this thesis, there are at least three persons to charge besides me. The first is my supervisor Simone Martini. It was him that made me fall in love first with formal methods, then with lambda calculus and finally with optimal reduction. I want really to thank him for the patience and time spent advising me, teaching to me and discussing with me. You have to complain mainly with him about my willing to become a researcher. Secondly I thank Andrea Asperti. The few days spent with him in Marseille were my first tasting of what Research in Computer Science could be (yes, capital letters!) and I am still bewitched about it. I am grateful with Andrea for the possibility he gave me to work with him. The third person having a responsibility in the completion of my PhD and hence in the realization of this thesis is my wife Anna. She has been the irreplaceable source of backing in these years.

Naturally my thanks go to many other people. I mention them hereafter in no particular order. I want to thank my referees Harry Mairson and Vincent van Oostrom for their willingness and their useful comments. I am obliged to Simona Ronchi della Rocca for the collaboration on EAL-principal typing. I also thank my colleagues and friends Gianluca Franco and Ivan Scagnetto for their contribution to my knowledge on types and lambda calculus. Roberto Ranon, Luca Di Gaspero and Stefano Mizzaro have been precious in their suggestions about Java and programming in general. A special thank goes to the local TEX guru and to Marino Miculan for their priceless help. Last but not least I express my deepest gratitude to my parents.

I hope to have learned at least something from the above persons and from all the others I met during my PhD. If this is the case, I consider myself lucky and satisfied.

Introduction

Why functional languages did not meet with success in real world applications? Probably there are two motivations: first actual programmers are used to imperative languages and the new comer is not so easy to learn and master. However, functional programs are often easier to check with respect to correctness and algebraic properties and this is an added value in the development of large software projects. Second, and more important, the available implementations of functional languages are not more efficient than the preexisting imperative ones. The problem is that functional languages did not play on home turf. All actual implementations run on computer with an architecture that is the one for imperative programming languages. The fetch-decode-execute cycle—the von Neumann bottleneck in [Bac78]—has nothing to do with functional languages!

So the question is: what is an efficient implementation of functional languages? Does it exist? In spite of some efforts, such implementations does not still exists or are unsatisfactory for the lack of a theory dealing with unit cost operations and independent from Turing machines. Thus, dealing the matter from another prospective: what is the inherent complexity of the evaluation of a functional programs?

This thesis is a contribution to give an answer to the last question. We can restrict the efforts in the study of functional languages to a particularly simple rewriting system that is widely accepted as the core of every language belonging to this paradigm. Such simple and powerful rewriting system is the *lambda-calculus*. The syntax we will adopt is the following:

$$M ::= x \mid \lambda x.M \mid (M\ M).$$

Optimal reduction of Lévy is the first ingredient used in this thesis. The intuitive idea behind optimal reduction is the following: if we consider the usual beta-reduction $(\lambda x.M\ N) \to_\beta M\{^N/x\}$ there is no recursive one step strategy that is optimal. Fixed a reduction strategy there always exists a lambda term for which during the normalization a redex will be duplicated forcing to do at least one beta-step more than the minimal possible amount. Lévy showed that we can define an equivalence relation between the different copies of a redex produced during a reduction and that, given a redex, it is possible to calculate its equivalent class, i.e. the set of its copies. If we have a mechanism allowing us to share all the copies of a single redex, following a call-by-need strategy we will perform the minimal number of (parallel) beta-steps in the normalization of a lambda terms. In this sense Lévy's parallel reduction is optimal: minimal number of parallel beta-steps.

The existence of a minimal number of optimal reduction steps makes them a more feasible candidate to complexity measure than the usual beta reduction steps. However, we will see that they can not be implemented as a unit cost operation.

The implementation of optimal reduction came ten years later thanks to the work of Lamping and, independently, Kathail. The beautiful graph reduction algorithm of the former is the second ingredient of the thesis. Lamping's graphs are essentially the syntax trees of lambda terms with two differences: variables are explicitly binded to the lambda abstraction and it is introduced a particular node for the sharing management. A lambda term is translated into a Lamping's graph, then the graph is reduced following a set of local rewriting rules and finally the graph in normal form is read back in order to obtain the normal form of the lambda term.

The rewriting rules are divided in two sets. The first set defines the *abstract* algorithm. It performs the optimal beta rule and duplication. In the study of optimal reduction, there are no attempts to improve the efficiency of this part of the algorithm that is on the contrary accepted to be the core of the implementation. In Chapter 2 we show that the complexity of this part of the algorithm is already non elementary. This is the first contribution of the thesis.

The result of Chapter 2 is obtained showing how it is possible to encode in Elementary Affine Logic the problem of deciding truth value of higher order formulas. This problem was shown to be non-elementary by Meyer in 1974. It is used by Statman in 1979 to show that the complexity of reduction in the simple typed lambda calculus is not elementary recursive, and by Asperti and Mairson in 1998 to state the same lower bound to the complexity of optimal beta reduction. We use it again with the addition of a particular fragment of Liner Logic of Girard to prove the non elementary bound for the complexity of the sharing implementation in the Lamping's graph reduction. In fact it is possible to normalize Elementary Affine Logic proofs with the Lamping's abstract algorithm.

From the last observation we continued our study looking for a type synthesis algorithm of lambda terms in Elementary Affine Logic. Reducing terms inside the abstract Lamping's algorithm dramatically improves the performances, then if we can prove that a lambda term has a type in EAL, we can reduce it in a more efficient way. Moreover, it is possible to define for EAL terms a notion of depth such that, fixing that, EAL terms of that depth normalize in an elementary number of steps and the type inference algorithm implicitly identifies such classes of terms. We give the positive solution to the problem of decidability of type inference in EAL in Chapter 3.

Finally, Chapter 4 introduces a java tool for the study and comparison of various implementations of optimal reducers for the lambda calculus. The application allows to analyze the traces of the execution of the optimal reducers in terms of unit cost operations. The tool comes with three different optimal reducers and others can be added.

Acknowledgments. Results on the complexity of optimal sharing exposed in Chapter 2 are due to the collaboration with Andrea Asperti and Simone Martini and have been published in [ACM00]. Results on type inference of lambda terms in EAL of Section 3.1 have been published in [CM01]. Results on the principal type for lambda terms in EAL of Section 3.2 are due to the collaboration with Simonetta Ronchi della Rocca.

I

The Complexity of Reduction

1

Optimality and Complexity

> "Currently, it is not known how to analyze the complexity of a functional program in an implementation independent way. Indeed, it is not clear that the complexity of a functional program is in any sense implementation independent. This is in marked contrast to programs written in common imperative languages. Here it is informally understood that all good implementations endow any given program with essentially the same complexity (in O-notation).
>
> Clearly, there are bad implementations of imperative languages in which the complexities of some programs are degraded below their true complexities. Thus the reason that the complexity of programs in imperative languages is well defined is that we have identified the good implementations (i.e. optimal implementations in O-terms), and we regard these implementations as defining *the* complexity of a program. Consequently, the complexity of an imperative program is implicitly implementation independent in this sense.
>
> If we wish to arrive at a similar situation in regard to the complexity of functional programs, then we need to know what an optimal implementation of functional language is."

[FS91]

In the study of efficient implementations of functional languages researchers have followed various directions. In the first one the goal is to increase the efficiency considering the underlying architecture of computers in our days. One example is the G-machine [Pey87], that avoids building graphs when an expression is purely arithmetical. Another one is to consider *eager* languages [McC62, Lan64, Mil78, Ste84, CR91] which evaluates an expression like $(F\ A)$ by first evaluating F and A (in no particular order) to, say, $F' = \lambda a. \cdots a \cdots a \cdots$ and A' and then contracting $(F'\ A')$ to $\cdots A' \cdots A' \cdots$. This evaluation strategy has advantages for the efficiency of the implementation when A is large, but its normal form A' is small, both in time and space.

A common goal is to reduce the number of function calls (beta-reduction) and in oder to attain this result researchers have developed techniques concerning reduction strategies (as in the above case) and sharing mechanism. But eager languages, although computationally complete (every computable function is lambda definable in the λI-calculus [Bar84] and the λI-calculus, the fragment of lambda calculus where all redexes are needed to reach the normal form, could be thought as the "core" of eager functional languages) have some disadvantages, as the impossibility of working with infinite objects and a minor elegance of the programming style in comparison with lazy languages [Tur76, Joh84, Tur85, vEP93, PW93]. Even more there exists lambda terms for which any order of classical reduction duplicates redexes. One example discussed by Lamping [Lam90] is

$$(\lambda g.(g(g\ \lambda x.x))\ \lambda h.(\lambda f.(f(f\ \lambda z.z))\ \lambda w.(h(w\ \lambda y.y))))$$

which has two redexes. Reducing the outer redex $(\lambda g. \ldots)$, one duplicates the inner one $(\lambda f. \ldots)$. Hence, if we do not want to duplicate redexes, the unique choice seems to contract the inner redex. But in this case we duplicate $(h(w\ \lambda y.y))$ that will be a redex as soon as a value for h is determined!

1.1 Harbingers

The first problem in the study of the complexity of reduction of functional programming languages is the choice of what to measure. Lacking a univocal answer in the literature, the first works in this field chose either to determine the intrinsic complexity for a given class of programs, following an *a priori* reasoning (Statman), or to count the most obvious thing, simply obtaining an upper bound (Schwichtenberg).

1.1.1 The typed λ-calculus is not elementary recursive

In the early attempts to study the complexity of the reduction in lambda calculus the principal result comes from Statman [Sta79] that showed that the "intrinsic" cost of normalization in the typed lambda calculus is not bounded by any elementary function.

Note
Mairson [Mai92] gave a simpler proof of the theorem of Statman using a different encoding essentially based on list iteration as quantifier elimination. We will see the encoding proposed by Mairson in Section 2.3.

Statman starts from a non elementary problem—the problem of determining the truth value of formulas in higher-order type theory. Such a problem was shown to be non elementary by Meyer in [Mey74] and Statman gives a reduction to the problem of β-conversion between simple typed lambda terms. We give Statman's encoding in the next sections.

Type theory

Consider the language of type theory, Ω, i.e. the language of set-theory where each variable has a natural number type and there are two constants $\mathbf{0}$, $\mathbf{1}$ of type o. Let the prime formulae be "stratified", i.e. they are of the form $\mathbf{0} \in x^1, \mathbf{1} \in x^1$ and $y^n \in z^{n+1}$. Arbitrary formulae are built-up from prime ones by means of the following connectives: $\neg, \wedge$ and $\forall$. The intended interpretation of Ω has $\mathbf{0}$ denoting 0, $\mathbf{1}$ denoting 1 and x^n ranging over $\mathscr{D}_n$ where $\mathscr{D}_0 = \{0, 1\}$ and $\mathscr{D}_{n+1} = powerset(\mathscr{D}_n)$.

The problem of deciding whether an arbitrary Ω-sentence is true is recursive. In fact there is a quantifier-elimination procedure for Ω-sentences (see [Hen63]).

In Ω we can define equality in the following way:

$$x^k =_k y^k \Leftrightarrow \forall z^{k-1}(z^{k-1} \in x^k \leftrightarrow z^{k-1} \in y^k).$$

Then, every element $D \in \mathscr{D}_n$ can be identified with the set of its elements, i.e. if $D = \{d_1, \dots, d_m\}$, it can be expressed in Ω as $\{x^{n-1} \mid x^{n-1} =_{n-1} d_1 \vee \cdots \vee x^{n-1} =_{n-1} d_m\}$.

Theorem 1 (Fischer and Meyer, Statman) *The problem of determining if an arbitrary Ω-sentence is true cannot be solved in elementary time.*

Proof: See [Mey74]. □

Statman's Encoding

Consider the Church numerals n of type $\mathsf{N}_0 \equiv (o \to o) \to (o \to o)$ where o is the base type, the simple typed lambda terms $\mathsf{sum}, \mathsf{mul}, \mathsf{sg}, \overline{\mathsf{sg}}$ that lambda define the homonimous arithmetic functions, and consider the type scheme $\mathsf{N}_{n+1} \equiv \mathsf{N}_n \to \mathsf{N}_0$.

In order to encode Ω-formulas we need the following simple typed terms:

$$\begin{aligned} a_1 &= \mathsf{2} : \mathsf{N}_0 \\ a_{n+1} &= (a_n\{o \to o/o\}\ a_1) : \mathsf{N}_0 \end{aligned}$$

where $\{o \to o/o\}$ indicates the type substitution. Notice that a_n lambda-defines the function $s(0) = 1$ and $s(n+1) = 2^{s(n)}$.

$$\begin{aligned} e_0 &= \lambda x.\lambda y.(\mathsf{sum}(\mathsf{mul}(\mathsf{sg}x)(\overline{\mathsf{sg}}y))(\mathsf{mul}(\mathsf{sg}y)(\overline{\mathsf{sg}}x))) : \mathsf{N}_0 \to (\mathsf{N}_0 \to \mathsf{N}_0) \\ \forall_0 &= \lambda h.(\mathsf{sum}(h\mathsf{0})(h\mathsf{1})) : \mathsf{N}_1 \to \mathsf{N}_0 \\ C &= \lambda g.(\mathsf{sum}(g(\lambda x.\mathsf{1}))(g(\lambda x.x))) : \mathsf{N}_2 \to \mathsf{N}_0 \\ p_{n+1}(x,y) &= (C\ \lambda f.(\forall_n\ \lambda w.(z\ \lambda y.(\mathsf{mul}(f(e_n wy)(xy)))))) : \mathsf{N}_0 \\ e_{n+1} &= \lambda x.\lambda y.(\forall_n\ \lambda z.(e_0(xz)(yz))) : \mathsf{N}_{n+1} \to (\mathsf{N}_{n+1} \to \mathsf{N}_0) \\ \forall_{n+1} &= \lambda y.((a_{n+1}\{\mathsf{N}_{n+2}/o\}\ \lambda z.\lambda x.p_{n+1}(x,z)\ y)\lambda w.\mathsf{1}) : \mathsf{N}_{n+2} \to \mathsf{N}_0 \end{aligned}$$

The encoding $()^*$ function from Ω-formulas to simply-typed lambda terms is the following:

$$\begin{aligned} (\mathbf{0})^* &= \mathsf{0} \\ (\mathbf{1})^* &= \mathsf{1} \\ (x^n)^* &= x : \mathsf{N}_n \\ (t_1 \in t_2)^* &= (\mathsf{sg}((t_1)^*(t_2)^*)) \\ (A \wedge B)^* &= (\mathsf{sg}(\mathsf{sum}(A)^*(B)^*)) \\ (\neg A)^* &= (\overline{\mathsf{sg}}(A)^*) \\ (\forall x^n A)^* &= (\mathsf{sg}(\forall_n\ \lambda x^{\mathsf{N}_n}.(A)^*)) \end{aligned}$$

Consider now the *definition of an object of type n* as

$$\begin{aligned} def^0(0) &= \{\mathsf{0}\} \\ def^0(1) &= \{\mathsf{1}\} \\ def^{n+1}(\alpha) &= \{\lambda y^{\mathsf{N}_n}.(\mathsf{mul}\ r_1(\cdots(\mathsf{mul}\ r_{s(n+1)}(\lambda w^{\mathsf{N}_n}.\mathsf{1}\ y))\cdots) \mid \\ &\qquad r_i = \mathsf{1} \text{ or } r_i = (e_n\ t\ y) \text{ for } t \in def^n(\beta) \\ &\qquad \text{for each } \beta \in \alpha \text{ for some } t \in def^n(\beta) \text{ there is some } i \text{ s.t. } r_i = (e_n\ t\ y)\} \end{aligned}$$

We can state the following proposition:

Proposition 2 *Suppose $\alpha, \beta \in \mathscr{D}_n$, $\gamma \in \mathscr{D}_{n+1}$, $t_1 \in def^n(\beta)$, $t_2 \in def^n(\alpha)$, and $t_3 \in def^{n+1}(\gamma)$, then*

1. $\beta = \alpha$ *iff* $(e_n\ t_1\ t_2) \to \mathsf{0}$
2. $\beta \in \gamma$ *iff* $(t_3\ t_1) \to \mathsf{0}$.

From these definitions we have the main theorem of Statman, proving that the β-reduction of simply typed terms is not elementary:

Theorem 3 *Let $A(x_1^{n_1}, \dots, x_m^{n_m})$ be an Ω-formula with free variables $\{x_1^{n_1}, \dots, x_m^{n_m}\}$, and let $\alpha_i \in \mathscr{D}_{n_i}$ and $t_i \in def^{n_i}(\alpha_i)$ for $1 \leq i \leq m$, then $A\{\alpha_1/x_1^{n_1}, \cdots, \alpha_m/x_m^{n_m}\}$ is true if and only if the simple typed lambda term $(\lambda x_1^{\mathsf{N}_{n_1}}. \cdots \lambda x_m^{\mathsf{N}_{n_m}}.(A)^*\ t_1 \cdots t_m)$ reduces to $\mathsf{0}$.*

1.1.2 Counting beta reductions in the pure typed lambda-calculus

Other works in the field of complexity of lambda reduction concern only the aspect of the number of standard beta reductions needed to reach the normal form [Sch82]. In his work, Schwichtenberg proves a lower and an upper bound to the number of standard beta reduction steps for simply typed terms.

Consider the length of a term M as the number of occurrences of variables in M except those immediately following a λ-symbol, i.e. $length(\lambda x.x) = length(\lambda x.\lambda y.y) = 1$. Let a_n and $s(n)$ as in the previous section.

Theorem 4 *Consider any sequence of reduction steps transforming a_n into its normal form, and let ℓ_n denote the total number of reduction steps in this sequence.*

$$\ell_n \geq s(n-2) - n.$$

Proof: (By [Sch82]) The length of a_n is $3n$. Note that any reduction step can at most square the length of the original term. Hence we have

$$\begin{aligned} s(n) &\leq \text{ length of the normal form of } a_n \\ &\leq (\text{length of } a_n)^{2^{\ell_n}} \\ &= (3n)^{2^{\ell_n}} \\ &\leq 2^{2^{n+\ell_n}} \qquad (\text{since } 3n \leq 2^{2^n}) \end{aligned}$$

□

In order to state the second result of Schwichtenberg, we first need to recall the following definitions:

Definition 1 *Define the following operations on functions from $\mathbb{N}^m$ to $\mathbb{N}$:*

1. composition*: given f and g, define $(f \circ g)(x) = f(g(x))$.*
2. Primitive recursion*: given f and g of the appropriate arity, define h such that $h(x, 0) = f(x)$ and $h(x, y+1) = g(x, y, h(x, y))$.*
3. Iteration*: given f, define h such that $h(x, y) = f^{[y]}(x)$, where $f^{[0]}(x) = x$ and $f^{[y+1]}(x) = f(f^{[y]}(x))$.*

4. Limited recursion*: given f, g and b, define h as in primitive recursion but only on the condition that $h(x,y) \leq b(x,y)$. Thus h is only allowed to grow as fast as another function already in the class.*

Definition 2 (The Grzegorczyk hierarchy [Grz53]) *Let $\mathcal{E}^0$ denote the smallest class containing zero, the successor function, and the projections, and which is closed under composition and limited recursion. Let $\mathcal{E}^{n+1}$ be defined similarly, except with the function E_n added to the list of initial functions, where E_n is defined as follows:*

$$\begin{aligned} E_0(x,y) &= x+y \\ E_1(x) &= x^2+2 \\ E_{n+1}(x) &= E_n^{[x]}(2) \end{aligned}$$

The third level of Grzegorczyk hierarchy is the class of elementary functions, i.e. $\mathcal{E} = \mathcal{E}_3$, moreover the union of all the levels of the Grzegorczyk hierarchy is the class $\mathcal{PR}$ of primitive recursive functions.

Consider the type level $l(\tau)$ of a type τ be defined inductively by $l(o) = 0$, $l(\sigma \to \tau) = \max\{l(\sigma)+1, l(\tau)\}$ and let $l(M)$ denotes the inner type level of M, i.e. the maximum type level of a subterm of M. Schwichtenberg shows that if during the reduction of a simply typed lambda term one searches for redexes of maximal type level, and among those one takes the rightmost one and fires it, the number of standard beta reduction steps is $\mathcal{E}^4$ upper bounded.

Theorem 5 *1. There is an $\mathcal{E}^4$ function f such that for all closed simply typed terms M the above mentioned normalization procedure terminates in a number of standard beta reduction steps $\leq f(\max\{length(M), l(M)\})$.*

2. For all m there is an elementary recursive function g_m such that for all closed simply typed terms M with $l(M) \leq m$ the above mentioned normalization procedure terminates in a number of standard beta reduction steps $\leq g_m(length(M))$.

Finally Schwichtenberg shows that combining his result with those of Gandy in [Gan80], one can obtain a universal $\mathcal{E}^4$ upper bound for the number of standard beta reduction steps with respect to any normalization procedure. This can be seen as follows: for any lambda term M of type τ, by Gandy's method one can define a close simply typed term $\overline{M}$ with the property that its numerical value is a bound on the number of standard beta reduction steps, where it does not matter in which way the reduction steps are choosen. Now to obtain a bound for the numerical value of $\overline{M}$, we first note that by the previous theorem we have an $\mathcal{E}^4$ bound on the number of standard beta reduction steps the specific normalization procedure given there will carry out to produce the normal form of $\overline{M}$; this bound is in term of $\max\{length(\overline{M}), l(\overline{M})\}$. Since by Gandy's construction of $\overline{M}$, $length(\overline{M})$ depends only linearly on $length(M)$ and $l(\overline{M}) = l(m)$, we also have an $\mathcal{E}^4$ upper bound on the number of standard beta reduction steps in terms of $\max\{length(M), l(M)\}$. Next note that any standard beta reduction step at most squares the length of the original term. So we have an $\mathcal{E}^4$ upper bound on the length (and hence on the numerical value) of the normal form of $\overline{M}$, again in terms of $\max\{length(M), l(M)\}$. This gives the desired result.

1.1.3 Optimality by Lévy

Consider the length of the reduction as the number of redexes contracted. By a classical result of Barendregt [Bar84], there is no recursive reduction strategy contracting at each step a single redex that is optimal, i.e. giving minimal length reductions for any lambda term M. However, if we allow a sharing mechanism able to give the possibility of performing parallel reductions and if we are willing to consider as a cost measure, the number of parallel reduction needed to reach the normal form, an optimal recursive strategy exists as proved by Lévy in his Ph.D. thesis [Lév78, Lév80]. In the next section we recall the concepts of *residual redexes, redex families, parallel reductions* and *optimality* leading to the definition of optimal reduction strategy in the sense of Lévy. We stress that at the time Lévy proposed his notion of optimality, he just gave the proof that the strategy is recursive, but he did not give any implementation of the optimal reduction. The problem of finding a feasible implementation has been solved ten years later by Lamping [Lam90] and independently by Kathail [Kat90]. We will see Lamping's solution in Section 1.3.

Preliminary definitions [Lév80]

In the following we introduce the basic concepts of *set of residuals* and *parallel reduction*. Consider a redex in a lambda term. During a reduction, the redex can be contracted and disappear, but also can be modified by substitution or duplicated. The set of residuals put in relation the initial redex with all its modified and duplicated "descendant" during a reduction. A parallel reduction contracts at each step a set of redexes. Thus a standard, usual, reduction is just a particular case o parallel reduction contracting singletons.

Definition 3 (Residuals) *Let R be a redex in a lambda term M and let $\rho : M \to N$ be a reduction (a sequence of redexes) from M to N.*
The set of residuals *of R by ρ is defined by induction on the length of ρ:*

$$\begin{aligned} R/0 &= \{R\} \quad \textit{where } 0 \textit{ is the empty reduction} \\ R/\rho\sigma &= \{T \mid \forall S \in R/\rho(T \in S/\sigma)\}. \end{aligned}$$

If ρ consists of a sigle redex S then R/S is defined by cases:

$$R/S = \begin{cases} \{R'\} & \text{if } R \text{ is not contained in } S.\ R' \text{ is the redex of } N \text{ which is at the same place as } R \text{ in } M. \\ \emptyset & \text{if } R \text{ coincides with } S. \\ \{R[B/x]\} & \text{if } S = (\lambda x.A\ B) \text{ and } R \text{ is contained in } A. \\ \{R_1, \ldots, R_n\} & \text{if } S = (\lambda x.A\ B),\ R \text{ is contained in } B,\ n \text{ is the number of free occurrences of } x \text{ in } A \text{ and for any } i\ R_i \text{ corresponds to } R \text{ in the } i\text{-th instance of } B \text{ in the contractum } A[B/x] \text{ of } S \text{ in } N. \end{cases}$$

In Figure 1.1, showing the set of reductions of the lambda term $M = (\Delta\ (\lambda x.(x\ y)\ I))$, it is possible to see the set of residuals of the redex $S = (\lambda x.(x\ y)\ I)$ by the reduction consisting of firing R:

$$S/R = \{S_1, S_2\}.$$

Let F be a set of (possibly nested) redexes in M. The notion of residuals is extended to F in the natural way.

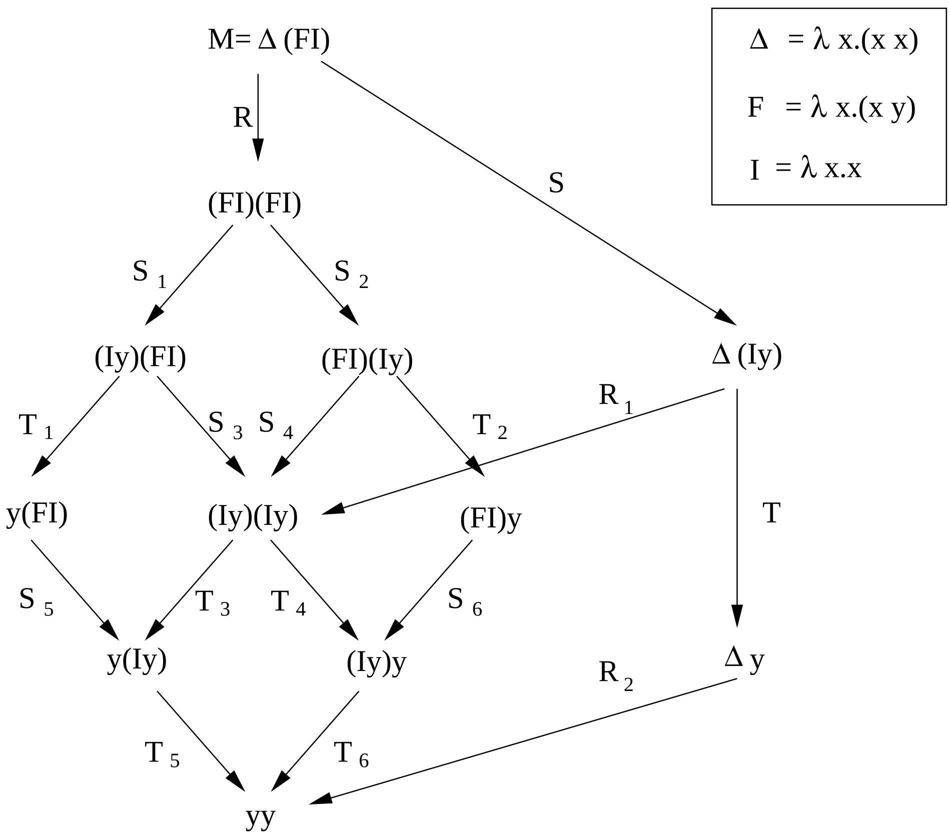


Figure 1.1: Reductions and residuals.

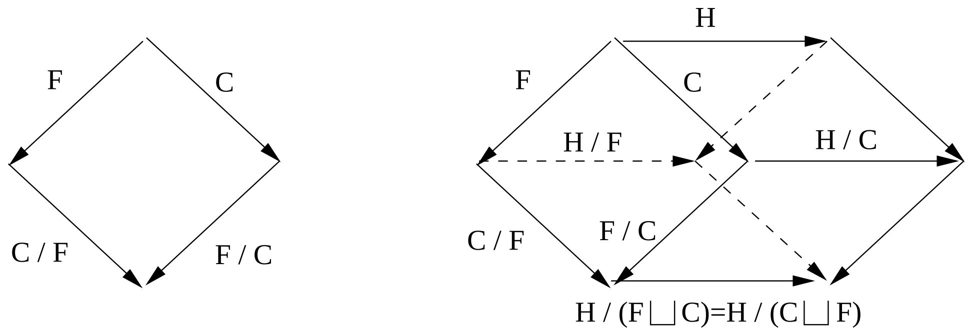


Figure 1.2: Lemma of parallel moves.

Definition 4 *A reduction* $\rho = R_1 \dots R_n \dots$ *is* relative *to* F *iff*

$$\forall n \geq 1 \quad R_n \in \mathsf{F}/(R_1 \dots R_{n-1}).$$

Moreover ρ *is a* development *of* F *iff* ρ *is relative to* F *and* $\mathsf{F}/\rho = \emptyset$.

Theorem 6 (Finite developments theorem) *Let* F *be a set of redexes in a lambda term* M*. Then*

1. *there is no infinite reduction relative to* F*,*

2. *all developments end at the same expression,*

3. *for all redex* R *in* M*, if* ρ *and* σ *are two developments of* F*, then* $R/\rho = R/\sigma$*.*

By the above theorem the order in which redexes in F are contracted is not relevant, hence we can define a *parallel reduction* as a reduction $M \xrightarrow{\mathsf{F}_1} M_1 \xrightarrow{\mathsf{F}_2} \cdots \xrightarrow{\mathsf{F}_n} M_n$ contracting some set of redexes at each step.
Notice that non parallel reductions can be seen as a particular case of parallel ones where all the sets of redexes are singleton sets.

If F and C are two sets of redexes in M, let $\mathsf{F} \bigsqcup \mathsf{C} = \mathsf{F}(\mathsf{C}/\mathsf{F})$.

Lemma 7 (Lemma of parallel moves) *Let* F *and* C *be two sets of redexes in* M*. Then*

1. $\mathsf{F} \bigsqcup \mathsf{C}$ *and* $\mathsf{C} \bigsqcup \mathsf{F}$ *end at the same expression,*

2. $\mathsf{H}/(\mathsf{F} \bigsqcup \mathsf{C}) = \mathsf{H}/(\mathsf{C} \bigsqcup \mathsf{F})$ *for any set* H *of redexes in* M*.*

Definition 5 *The* equivalence *of parallel reductions* by permutations *is the least congruence with respect to composition satisfying the lemma of parallel moves and elimination of empty steps.*

More explicitly, this relation $\equiv$ *is the least equivalence relation satisfying:*

1. $\mathsf{F} \bigsqcup \mathsf{C} \equiv \mathsf{C} \bigsqcup \mathsf{F}$ *when* F *and* C *are two sets of redexes in a same expression,*

2. $\emptyset \equiv 0$*, i.e. one immediate contraction of an empty set of redexes is equivalent to an empty parallel reduction,*

3. $\rho\sigma\tau \equiv \rho\sigma'\tau$ *if* $\sigma \equiv \sigma'$*.*

The embedding relation $\sqsubseteq$ *is defined as follows:*

$$\rho \sqsubseteq \sigma \textit{ iff } \exists\tau.\ \rho\tau \equiv \sigma.$$

Definition 6 *Let ρ and σ be two reductions which start at a same lambda term. Then the* reduction residual *σ/ρ of σ by ρ is a reduction starting at the end of ρ which is defined inductively on the sum of length of ρ and σ by:*

$$0/\rho = 0$$
$$(\sigma\mathsf{F})/\rho = (\sigma/\rho)(\mathsf{F}/(\rho/\sigma))$$

From this definition, some easy algebraic properties of residuals of reduction can be shown. For example:

$$(\sigma\tau)/\rho = (\sigma/\rho)(\tau/(\rho/\sigma))$$
$$\rho/(\sigma\tau) = (\rho/\sigma)/\tau$$
$$\rho/0 = \rho$$

Redex families

Two redexes are in the same family if one is a residual of the other. We will see that the family equivalence relation is computable and we can define *complete* parallel reductions as those reducing at each step a whole family of redexes. Moreover the number of parallel steps of any complete reduction is equal to the number of family classes of the redexes contracted during the reduction.

In order to distinguish various occurrences of the same redex R during a reduction, we refer to the reduction ρR as the redex occurrence R with *history* ρ.

Definition 7 (Copy relation $\leq$) *A redex S with history σ is a* copy *of redex R with history ρ, written $\rho R \leq \sigma S$, iff there is a reduction τ such that $\rho\tau \equiv \sigma$ and $S \in R/\tau$.*

Definition 8 (Family equivalence $\simeq$) *Two redexes R and S with histories ρ and σ are in a same* family*, written $\rho R \simeq \sigma S$, iff $\rho R \leq \sigma S$ or $\sigma S \leq \rho R$ or there is some τT such that $\rho R \simeq \tau T \simeq \sigma S$.*

Lemma 8 *$R \simeq \rho S$ iff $S \in R/\rho$.*

Theorem 9 *The family relation is computable.*

Proof: See [Lév80]. □

Let $[\rho R]$ be the equivalence class of ρR with respect to $\simeq$ and let $\rho = \mathsf{F}_1\mathsf{F}_2\ldots\mathsf{F}_n\ldots$. The set of family classes of the redexes contracted in ρ is defined in the following way:

$$\mathrm{FAM}(\rho) = \{[\mathsf{F}_1\mathsf{F}_2\ldots\mathsf{F}_{i-1}R_i] \mid R_i \in \mathsf{F}_i, i \geq 1\}$$

By Lemma 8 we can extend the definitions of *relative* reduction and *development*:

Definition 9 *ρ is* relative *to X if $\mathrm{FAM}(\rho) \subseteq \mathsf{X}$.*
A reduction ρ relative to X is a development *of X if there is no redex R such that $[\rho R] \in \mathsf{X}$.*

The finite developments theorem can be easy generalized to the definition above.

Definition 10 (Complete reduction) *A parallel reduction* $\mathsf{F}_1\mathsf{F}_2\ldots\mathsf{F}_n\ldots$ *is* complete *iff for every* $n \geq 1$, $\mathsf{F}_n \neq \emptyset$ *is a maximum set of redexes such that, for all* $R \in \mathsf{F}_n$ *and* $S \in \mathsf{F}_n$, $\mathsf{F}_1\mathsf{F}_2\ldots\mathsf{F}_{n-1}R \simeq \mathsf{F}_1\mathsf{F}_2\ldots\mathsf{F}_{n-1}S$.

Thus, at each step of a complete parallel reduction, one non-empty family class is contracted.

Lemma 10 *Any complete reduction* ρ *is a development of* $\mathrm{FAM}(\rho)$.

Lemma 11 *Let* ρ *be a complete reduction. Then* $|\rho|$ *is equal to the number of elements in* $\mathrm{FAM}(\rho)$.

Optimal reductions

A redex is *needed* if it is contracted in every normalizing reduction. A complete parallel reduction is *optimal* if it reduces at each step a whole family of a needed redex. A particular optimal strategy is the leftmost-outermost one. Notice that Lévy's optimality concept assumes the parallel beta reduction to be a unit cost operation. Unfortunately this is not the case as we will see in Section 1.5.

Definition 11
- *A parallel reduction* ρ *is* terminating *iff its final expression is in normal forms.*

Let $\rho = \mathsf{F}_1\mathsf{F}_2\ldots\mathsf{F}_n\ldots$ *and let* $\mathcal{R}(\rho)$ *be the set of redexes one of whose residuals is contracted in* ρ*:*

$$\mathcal{R}(\rho) = \{R \mid R/\mathsf{F}_1\mathsf{F}_2\ldots\mathsf{F}_{i-1} \cap \mathsf{F}_i \neq \emptyset, i \geq 1\}.$$

- *A redex* R *in a lambda term* M *is* needed *iff, for all terminating parallel reductions* ρ *starting at* M*, one has* $R \in \mathcal{R}(\rho)$.

- *A parallel reduction* $\rho = \mathsf{F}_1\mathsf{F}_2\ldots\mathsf{F}_n\ldots$ *is a* call-by-need parallel reduction *iff there is at least one needed redex in every* F_n *for* $n \geq 1$.

Theorem 12 *Let* M *have a normal form, then any call-by-need parallel reduction starting at* M *is eventually terminating.*

Definition 12 (c-complete reductions) F *with history* ρ *is a* set of copies of a single redex *iff there is one redex* S *with history* σ *such that* $\sigma S \leq \rho R$ *for every* $R \in \mathsf{F}$.

The reduction $\rho = \mathsf{F}_1\mathsf{F}_2\ldots\mathsf{F}_n\ldots$ *is* c-complete *iff, for all* $n \geq 1$*, the non-empty set* F_n *is a maximum set of copies of a single redex.*

We assume that the cost of a c-complete reduction satisfies the following equation:

$$\mathsf{cost}(\rho) = |\rho| \tag{1.1}$$

Lemma 13 *A reduction is c-complete iff it is a complete reduction.*

Theorem 14 *Any complete and call-by-need parallel reduction reaches the normal form in an optimal cost.*

Corollary 15 *The leftmost-outermost complete parallel reduction reaches the normal form in an optimal cost.*

1.2 Looking for efficient implementations of functional languages

In the study of the complexity of functional languages, we focus our attention on the lambda calculus, adopting the following thesis:

> **The Implementation Thesis**
>
> Any functional programming language may be efficiently compiled into the pure λ-calculus in such a way as to retain the possibility of equally efficient implementation.

Starting from the implementation thesis above, Frandsen and Sturtivant [FS91], proposed to consider the problem of finding an efficient implementation of pure λ-calculus in order to understand how to analyze the complexity of functional programs (see the quotation at the beginning of the chapter).

Let M be a lambda term. $\mu_\pi(M)$ is the minimum number of parallel beta reduction steps required to reduce M to its normal form (if M has no normal form $\mu_\pi(M)$ is ∞). We define also $\mu_\beta(M)$ as the minimum number of standard beta reduction steps required to reduce M (as for μ_π, if M is not reducible then $\mu_\beta(M) = \infty$), and $l_{normal}(M)$ as the number of standard beta reduction steps required to reduce M via a normal order reduction (the leftmost outermost one).

Consider the problem of finding a minimal length reduction sequence to normal form for a given lambda term M. Frandsen and Sturtivant noticed that this problem has an algorithmic solution: simply explore all reduction sequences that use up to the number of reductions which the leftmost strategy takes to reach normal form, and choose a minimal one. Clearly they did not claim that this procedure is efficient, however, its runtime is bounded by some recursive function of $l_{normal}(M)$. Notice that this is true also if M has no normal form since in that case l_{normal} is simply ∞.

Such a consideration suggests that a change of the definition of "input size" to include some measure of the length of reduction chains, may make it possible to bound the runtime of an implementation of lambda expression normal form computation by some recursive function of the new input size. Frandsen and Sturtivant proposed to consider μ_π as a component of the input size parameter. Other candidates could be μ_β and l_{normal}, but l_{normal} is not even remotely optimal for some lambda terms as the following proposition shows:

Proposition 16 *There is an infinite family $\{A_k\}$ of lambda terms such that $l_{normal}(A_k) = 2^{\Omega(\mu_\beta(A_k))}$.*

Proof: Let

$$A_0 = \lambda x.x$$
$$A_k = (\lambda x.(x\ x)\ A_{k-1})$$

Let $S(k)$ and $T(k)$ denote the number of standard beta reduction steps needed to transform A_k to normal form using normal and applicative order respectively.

It is easy to obtain the following equations:

$$\begin{aligned}S(0) &= 0\\ S(k) &= 2S(k-1)+2\\ T(0) &= 0\\ T(k) &= T(k-1)+2\end{aligned}$$

that have solutions

$$\begin{aligned}S(k) &= 2^{k+1}-2\\ T(k) &= 2k\end{aligned}$$

from which the proposition follows. □

Thus measuring the complexity of an implementation using l_{normal} can give unrealistically optimistic assessments in the sense that the predicted cost is much less than the actual cost.

μ_π is better than μ_β for a number of reasons. First, it is known that μ_π is always less then μ_β. Second, we have seen in Section 1.1.3 that an optimal strategy is known for parallel reductions. Furthermore μ_π and μ_β may be significantly different. Frandsen and Sturtivant conjectured that the ratio of standard to parallel beta reductions could be exponential[1]. Such a conjecture has been proven by Mairson and Lawall in [LM96].

Theorem 17 *There exists a family of lambda terms that normalizes in $2^{\Omega(2^n)}$ standard beta reductions, but only $\Theta(n)$ parallel beta reductions.*

Proof: Consider the term $M_n = (\lambda x.\underbrace{(2\ (\cdots(2}_{n\text{times}}\ x)\cdots))\ \lambda x.\lambda y.(x(x\ y)))$. □

Definition 13 (The complexity of an implementation of the λ-calculus)

- *A procedure $\mathcal{I}$ is an* implementation *of the lambda calculus if $\mathcal{I}$ on input a lambda term M outputs N, the normal form of M (if M has no normal form then $\mathcal{I}$ needs not halt).*
- *The* input-size *parameter $\nu : \Lambda \to \mathbb{N} \cup \{\infty\}$ is given by*

  $$\nu(M) = |M| + \mu_\pi(M) + |N|$$

  for all $M \in \Lambda$ where N is the normal form of M, and the norm denotes term size. (If M has no normal form then $\nu(M) = \infty$).
- *An implementation $\mathcal{I}$ of the lambda calculus has* (worst-case) complexity $T(\nu)$ *if $T : \mathbb{N} \to \mathbb{N}$ satisfies*

  $$T(\nu) = \max\{\mathit{run\ time\ of}\ \mathcal{I}\ \mathit{on\ input}\ M \ \mid\ M \in S_\nu\}$$

  when $S_\nu \neq \emptyset$ and where $S_\nu = \{M \in \Lambda \mid \nu(M) = \nu\}$.
- *An implementation $\mathcal{J}$ of the lambda calculus of worst case complexity $T_\mathcal{J}(\nu)$, is an* optimal *implementation if for any implementation $\mathcal{I}$ of the lambda calculus of worst case complexity $T_\mathcal{I}(\nu)$, we have $T_\mathcal{J}(\nu) = O(T_\mathcal{I}(\nu))$.*

[1] This bound is very weak. We will see a much stronger one in Section 1.5.

The complexity of an implementation is well defined, since the sets S_ν are always finite, and there is a number ν_0 such that for $\nu > \nu_0$, S_ν is always non empty.

The size of M must appear in ν because if the input is already in normal form, then this must be verified by any implementation. In general, this must take time proportional to the size of M. The length of the normal form N should be present since there are many family of lambda terms with the property that the size of the normal form is exponential in the number of reduction steps to normal form. Thus the absence of the size of N would imply an exponential lower bound on all implementations.

Strangely Frandsen and Sturtivant did not take into account the size of intermediate terms during the reduction (such a criticism was also made by Lawall and Mairson in [LM96]). Consider for example the family of terms M_n of the proof of Theorem 17. In the calculation of $\nu(M_n)$, the main factor is the size of the normal form of M_n that is exponential in the number of standard beta reductions. Consider now the lambda term $L_n = (M_n\ \lambda x.x)$. The normal form of L_n is $\lambda x.x$, hence in the calculation of $\nu(L_n)$ the main factor is the number of standard beta reduction. This may look strange according to the fact that we use M_n as a sub-program of L_n and we could expect that the input size of L_n is at least that of M_n. Thus from the Frandsen and Sturtivant definition of input size it seems they implicitly impose that an efficient implementation of the lambda calculus must have a succinct representation of intermediate terms during the reduction.

Proposition 18 *Any implementation of the lambda calculus has complexity $\Omega(\nu)$.*

> **Proof:** Frandsen and Sturtivant [FS91] give an encoding of arbitrary (imperative) computation as a lambda expression with the property that $\mu_\pi = \mu_\beta = l_{normal}$, and each step of the computation is simulated by a constant number of reductions which in turn can be implemented in such a way as to take only a bounded amount of work. Thus the existence of an implementation with complexity $o(\nu)$ would immediately give rise to a method of speeding up an arbitrary computation. $\square$

Proposition 19 *Implementations of the lambda calculus based on Turner combinators [Tur79] or Hughes super combinators [Hug82] are exponentially inefficient.*

The complexity measure proposed by Frandsen and Sturtivant is the first attempt to consider the number of optimal beta reduction instead of the standard ones. Unfortunately, it will turn out (Section 1.5) that even this measure is not adequate.

1.3 Lamping's Algorithm

At the time Lévy introduced the concepts of redex families and optimal reduction, no implementation was known. The first solution came ten years by Lamping [Lam90] and independently by Kathail [Kat90]. In this section we will see Lamping's algorithm. We claim that it is the starting point from which building efficient implementations of the lambda calculus.

The graph rewriting algorithm proposed by Lamping in [Lam90] is generally thought of as composed of two parts. The first part is responsible for the implementation of the (optimal) beta reduction and of the (partial) duplication. It is called the *abstract algorithm*. The second part, known as the *oracle*, consists of a complex machinery needed in order to maintain enough information distributed in the graph for the correct management of the shared parts.

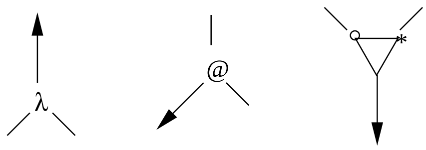


Figure 1.3: Nodes for sharing graphs.

The best way to get in confidence with Lamping's algorithm is to start with the simplified abstract version (Section 1.3.1) in order to understand the main ideas this graph rewriting technique is based on, and then to observe the cases in which the abstract algorithm fails to discriminate between different shared subgraphs. At this point we will be in the position to appreciate the full solution proposed by Lamping (Section 1.3.2).

1.3.1 Sharing Graphs

Given a lambda term M, the corresponding *sharing graph* G is essentially the syntax tree of M with two differences:

1. a specific node, the *fan*, manages the sharing;
2. variables are explicitly binded to their lambda abstraction node.

Nodes of sharing graphs are shown in Figure 1.3. The rightmost node is the *fan*. Every node has three ports. The one with an arrow is the *principal port* and it is the port through which the node interact. The other are the *auxiliary ports*. The right auxiliary port of the lambda node is the *body port*, i.e. it will be connected to the body of the function, whilst the left one is the *variable port*, and it will be connected to the variable abstracted by the lambda. The right port of the application node @ is the *argument port* and the upper port is the *result port*. Finally the two auxiliary port of the fan are distinguished just by a tag, a circle and a star in the figure.

We will introduce the algorithm, following [AG98], step by step using $(\Delta\ \lambda h.(\Delta\ (h\ I)))$, where $\Delta = \lambda x.(x\ x)$ and $I = \lambda x.x$, as an example. Figure 1.4 shows the sharing graph of the term. Notice that there is no need of specifying the name of the variable binded by the λ nodes thank to the explicit bind of variables (this is analogous to the mechanism of de Brujin's indices [Bru72]).

The shortest standard beta reduction for $(\Delta\ \lambda h.(\Delta\ (h\ I)))$ first fires the two outermost redexes and then continues with the innermost strategy:

$$
\begin{aligned}
(\Delta\ (\lambda h.(\Delta\ (h\ I))) &\to (\lambda h.(\Delta\ (h\ I))\ \lambda h.(\Delta\ (h\ I))) \\
&\to (\Delta\ (\lambda h.(\Delta\ (h\ I))\ I)) \\
&\to (\Delta\ (\Delta\ (I\ I))) \\
&\to (\Delta\ (\Delta\ I)) \\
&\to (\Delta\ (I\ I)) \\
&\to (\Delta\ I) \\
&\to (I\ I) \\
&\to I.
\end{aligned}
$$

It takes 8 beta steps to reach the normal form, but the length of the parallel reduction by families for $(\Delta\ \lambda h.(\Delta\ (h\ I)))$ is only 7. In fact the standard beta reduction, also in the

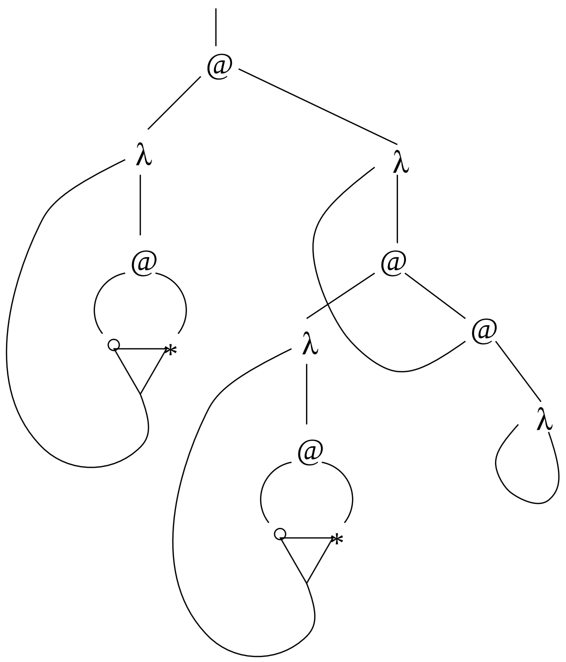


Figure 1.4: Sharing graph of $(\Delta\ \lambda h.(\Delta\ (h\ I)))$

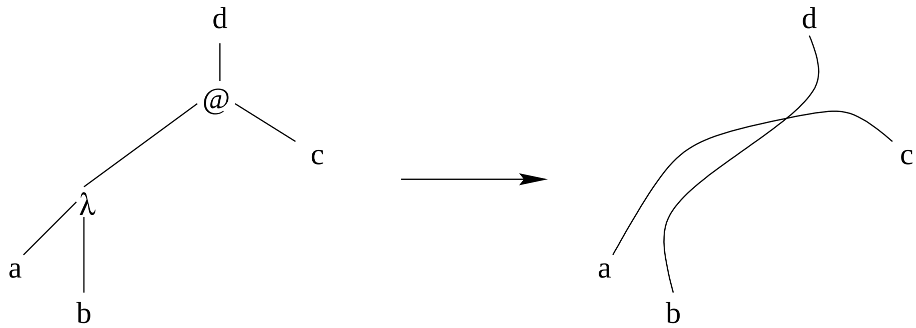


Figure 1.5: Optimal beta rule.

best case, can not avoid to duplicate one redex ($(\Delta\ (h\ I))$ in the first step). We will see how Lamping's algorithm reaches the normal form in just 7 optimal beta steps.

The algorithm consists of a set of local graph rewriting rules. Figure 1.5 shows the optimal beta rule. The body port of the lambda node is connected with the result port of the @ node and becomes the new root of the subgraph involved in the redex firing. On the other part, the substitution does not occur explicitly, but the variable port of the lambda is connected to the argument port of the @. Notice that the argument of the function may remain shared at this stage of the reduction (the variable binded by the lambda node could be shared by means of a fan node, as in Figure 1.6).

There are two beta redexes in the initial graph of $(\Delta\ \lambda h.(\Delta\ (h\ I)))$. The first two steps of graph reduction are shown in Figure 1.7, where graph are re-displayed after each step just for aesthetic reasons.

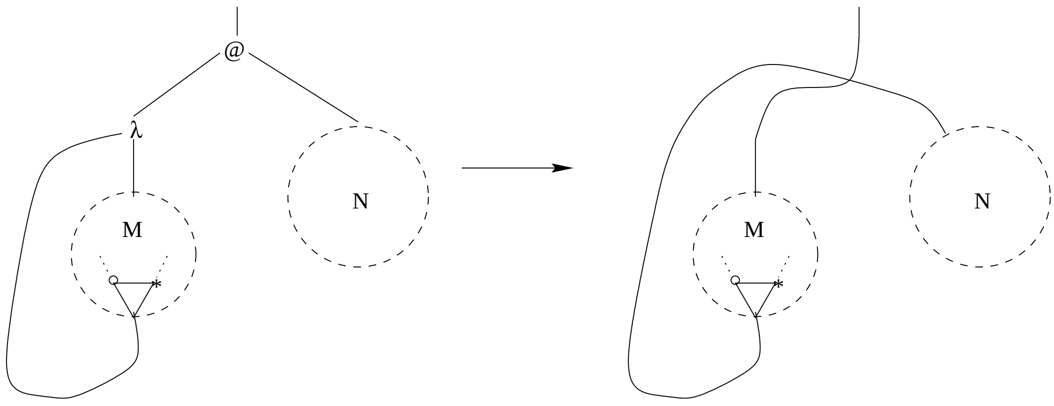
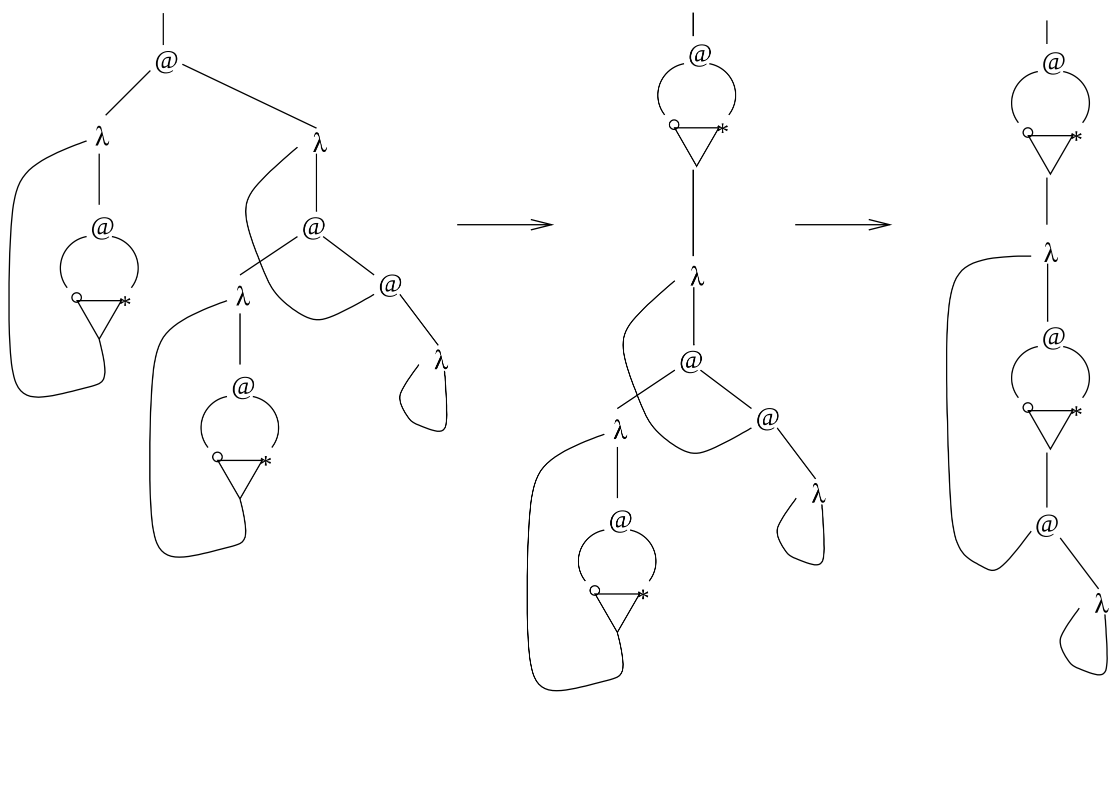


Figure 1.6: Example of optimal beta reduction.



(1) (2) (3)

Figure 1.7: First beta steps of $(\Delta\ \lambda h.(\Delta\ (h\ I)))$.

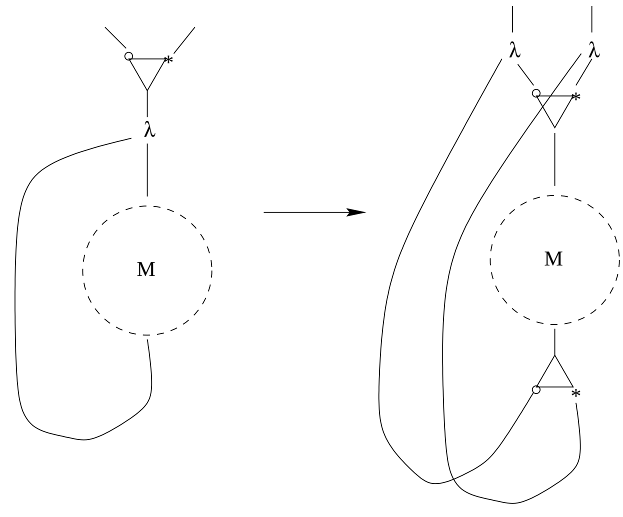


Figure 1.8: Partial duplication of a function.

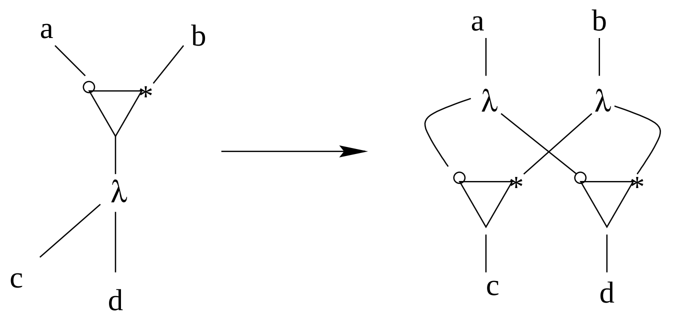


Figure 1.9: Fan-lambda interaction.

As soon as the reduction starts, the sharing graph is no more in trivial correspondence with the syntax tree of the lambda term as in the case of the initial translation. In other words, the graph in Figure 1.7(2) represents the lambda term $(\lambda h.(\Delta\ (h\ I))\ \lambda h.(\Delta\ (h\ I)))$, but is pretty different from the graph that one can expect applying the intial translation to that lambda term. The same happens for the graph in Figure 1.7(3) representing the lambda term $(\lambda h.((h\ I)(h\ I))\ \lambda h.((h\ I)(h\ I)))$ where the subterm $\lambda h. \ldots$ is shared two times and $(h\ I)$ four times. Moreover notice that the second beta step in Figure 1.7 corresponds to two standard beta reduction—the two copies of $(\Delta\ (h\ I))$ are fired simultaneously, being shared by the upper fan.

There is a redex left in $(\lambda h.((h\ I)(h\ I))\ \lambda h.((h\ I)(h\ I)))$, but at this stage of the graph reduction it is not possible to perform a beta rule. Actually the two copies of $\lambda h. \ldots$ are shared and one first needs to duplicate some parts of the graph. Lamping's technique, differently from previous implementations of shared reduction as Wadsworth's Graphs [Wad71], does not proceed to duplicate the entire subterm, but performs a *partial duplication*. Only the lambda node is duplicated in order to allow the beta reduction step, while the rest of the subgraph—the body of the function—remains shared—Figure 1.8—between two fans, the upper, called *fan-in*, denoting the start of the shared part, the lower, called *fan-out*, denoting the end of the sharing. The local rewriting rule of duplication of the lambda node is shown in Figure 1.9.

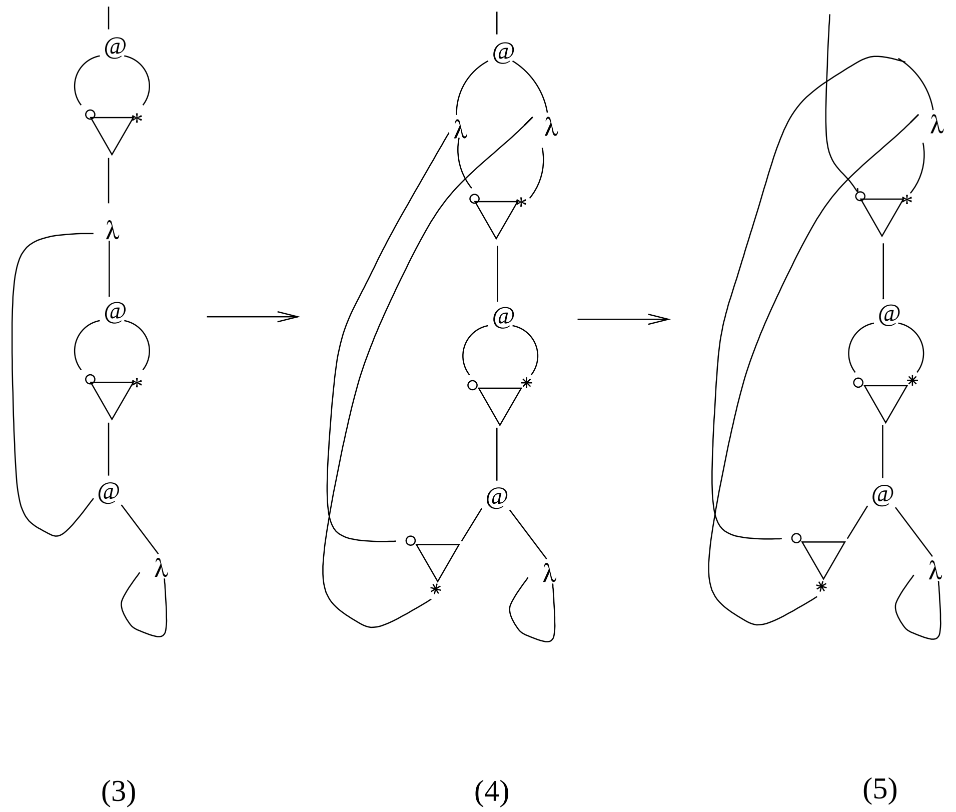


Figure 1.10: Reduction of $(\Delta\ \lambda h.(\Delta(h\ I)))$.

Let us go on with our example. After the application of the fan-λ rule—Figure 1.10(4)—a new β-redex is ready to be contracted and after this rule has been performed we get the graph of Figure 1.10(5) representing the lambda term

$$((\lambda h.((h\ I)\ (h\ I))\ I)\ (\lambda h.((h\ I)\ (h\ I))\ I)).$$

There are no more β-redexes or fan-λ-redexes left in the graph and we need to duplicate some application node. Notice that if we duplicate the outermost application—the upper one in the figure—we could loose sharing. Actually the redex $(\lambda h.((h\ I)\ (h\ I))\ I)$ would be duplicated. Thus, in general, the rule of Figure 1.11, although semantically correct, it is not permitted.

However we can duplicate the lower application node by means of the fan-out. The rule is shown in Figure 1.12. Notice that this rule preserves optimality. In fact the lower application

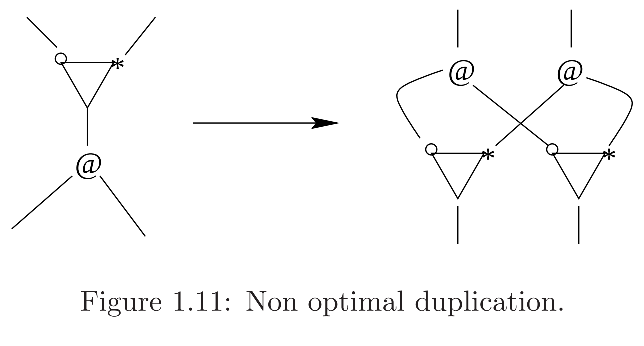


Figure 1.11: Non optimal duplication.

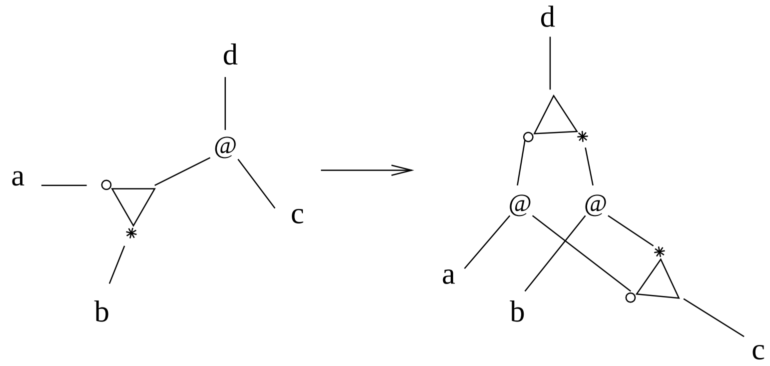


Figure 1.12: Fan-@ rule.

node corresponds to six applications in the associated lambda term: two applications for the subterms $(\lambda h.((h\ I)\ (h\ I))\ I)$ and four for subterms $(h\ I)$. In this case the graph shares the argument of the application—and the "meta" operation of application itself represented by the shared application node—but not the entire redex. Actually it is not possible to treat in the same way the six redexes above because we do not know anything about the variable h, then we can safely duplicate the application node. In general it is always correct to apply the rule of Figure 1.12, because such configuration of the graph implies the existence of two classes of non shareable redexes.

The reduction of our example proceeds according to Figure 1.13. After the fan-@ rule it is possible to perform a new fan-λ rule obtaining the graph in Figure 1.13(7) where there are two pairs of matching fans labeled with "a" and "b". This is a crucial point of the optimal reduction technique. As we will see, these two cases must be managed in two different way.

The pair of fans labeled with the "a" comes from the fan that shared the identity. Both copies refer to the same shared portion of the graph—the body of the identity—and then we can continue the duplication process annihilating the fans and connecting the corresponding links: $\circ$ with $\circ$ and $*$ with $*$, as in Figure 1.14(1).

On the other hand, in case the pair of fans labeled with "b", the fan-out refers to the shared part of the graph that starts from the upper fan-in—it was created during the duplication of the uppermost λ node. Then the fans must duplicate each other as in Figure 1.14(2).

Applying the fan-fan interaction rules of Figure 1.14, the computation proceeds as described in Figure 1.15.

The rules seen so far are enough to complete the reduction. In particular, we can start firing two β-redexes, we can then duplicate the only application left in the graph and annihilate a couple of paired fans.

The final graph in Figure 1.16 has been redrawn for the sake of clarity in Figure 1.17. Now, the identity is duplicated, its application is reduced. The resulting identity is duplicated again and by a final β-redex we obtain the expected normal form.

The normalization of this term required only seven applications of the β-rule, against eight β-reductions needed by the best strategy of standard reduction.

1.3.2 The Full Algorithm

The set of local graph rewriting rules we have seen so far is the so called *abstract Lamping's algorithm*. We have pointed out in the previous section that the crucial operation in the Lamping's graphs rewriting technique is the correct matching of fans. In other words, we

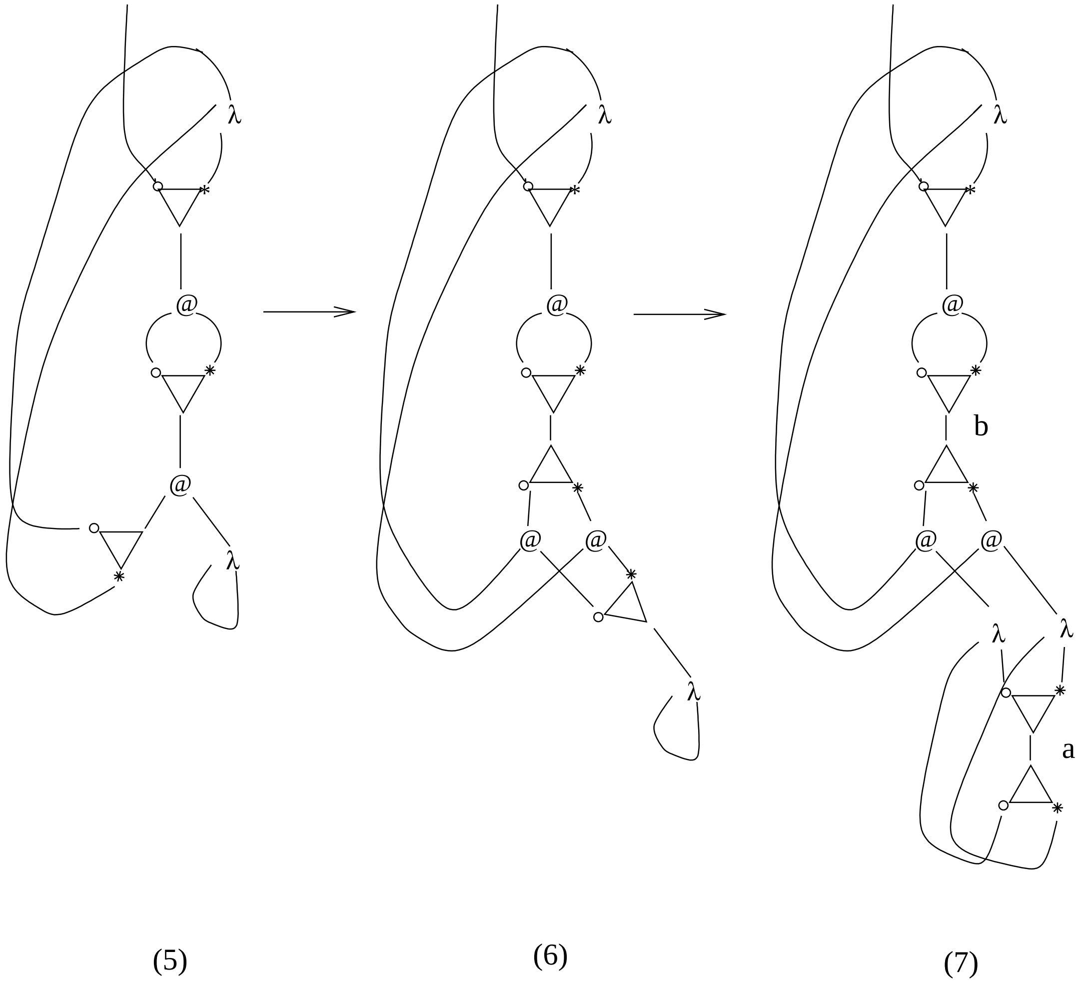


Figure 1.13: Reduction of $(\Delta\ \lambda h.(\Delta(h\ I)))$.

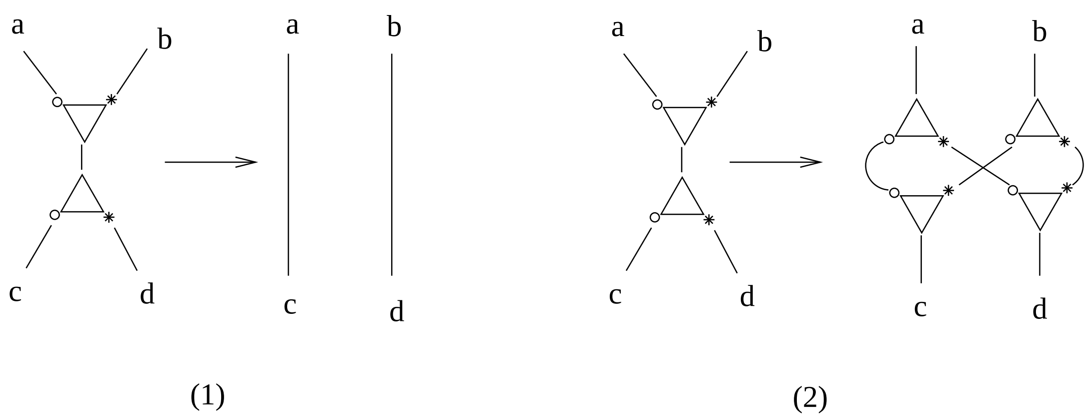
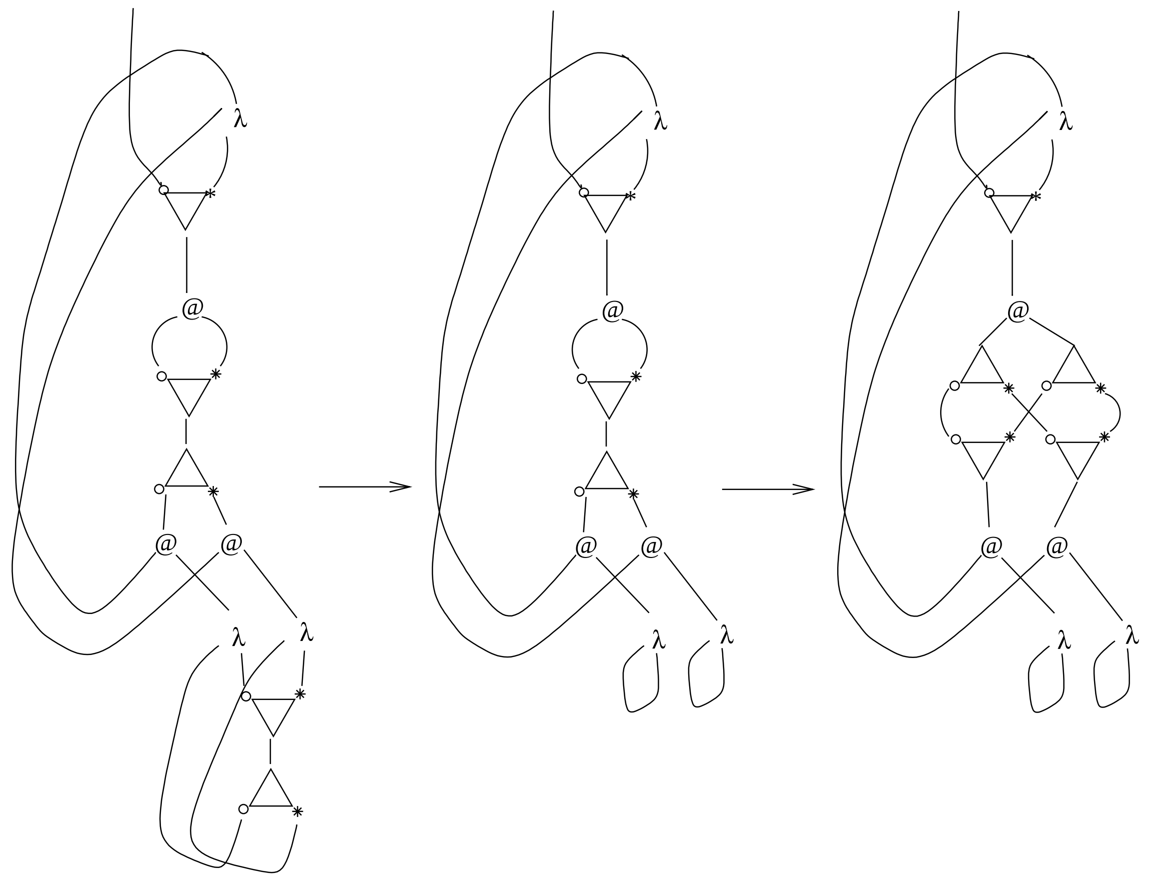
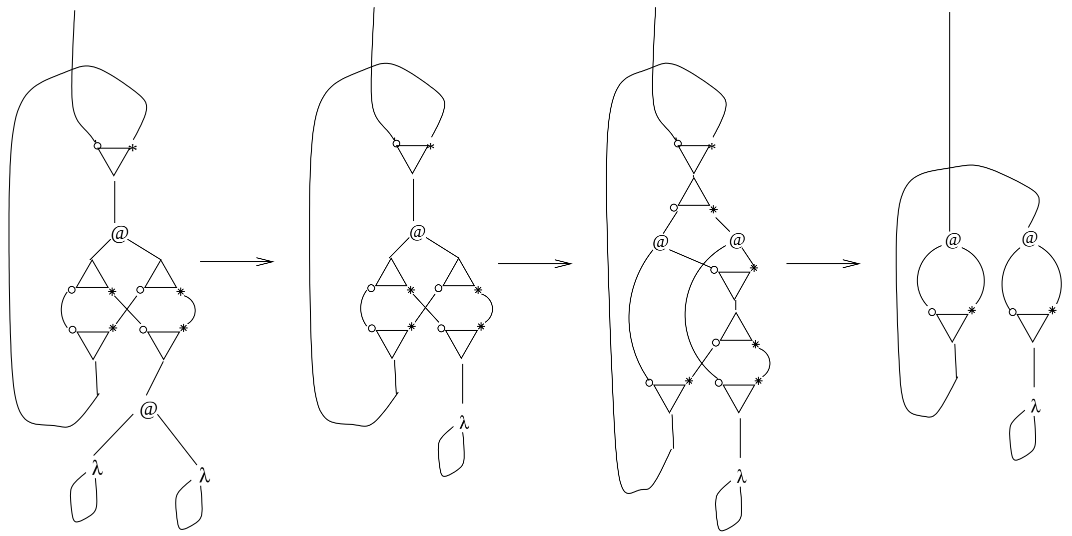


Figure 1.14: Fan-fan iterations.



(7) (8) (9)

Figure 1.15: Reduction of $(\Delta\ \lambda h.(\Delta(h\ I)))$.



(10) (11) (12) (13-14)

Figure 1.16: Reduction of $(\Delta\ \lambda h.(\Delta(h\ I)))$.

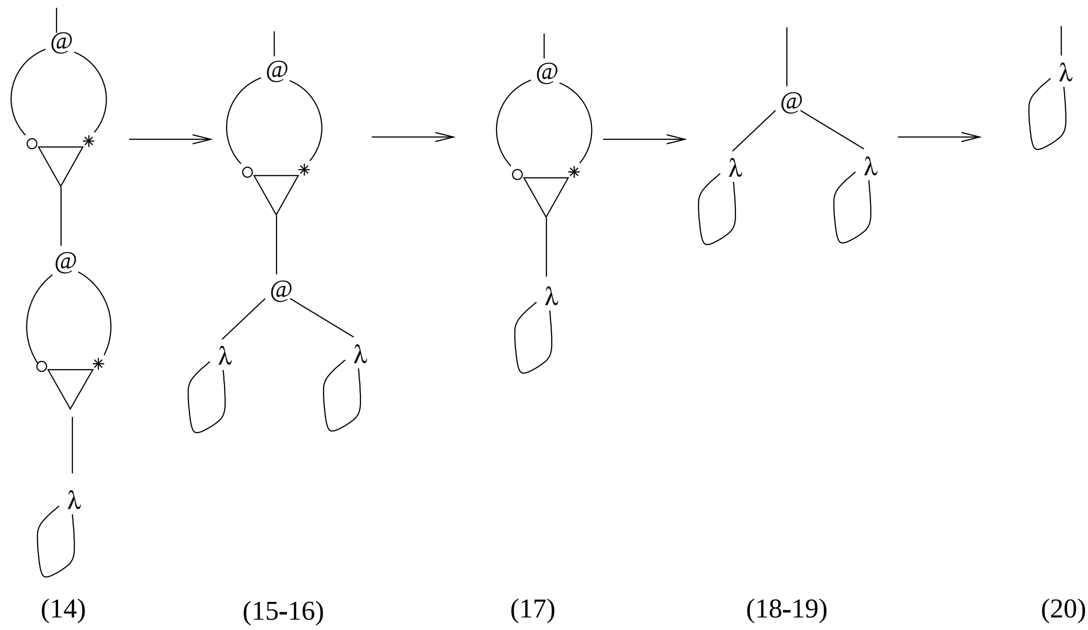


Figure 1.17: Reduction of $(\Delta\ \lambda h.(\Delta(h\ I)))$.

can separate the set of rules in two classes: the first class consists of the abstract algorithm and it is responsible for the operations of parallel beta reduction and partial duplication of lambda terms. This part of the algorithm introduce the novelty, with respect to the existent graph rewriting techniques, of fan-out nodes and hence the possibility of partially duplicate subterms. The second class—known as the *oracle*—consists of the set of rules that manage the information needed to solve the problem of pairing fans. This is the hard part of the algorithm.

Actually, the trivial solution of labeling fans is not sufficient and the graph reduction of $\Delta\Delta = (\lambda x.(x\ x)\ \lambda x.(x\ x))$ is a good counterexample.

Figure 1.18 shows some steps of the (infinite) graph reduction of $\Delta\Delta$. Initially the two fans have different labels. After the first beta step and a successive λ-fan interaction, the lower fan-in labeled with "B" and the fan-out labeled with "A" correctly duplicate each other—the graph transformation is depicted in the passage from (3) to (4) in the figure. The duplication is correct because the two fans refer to distinct shared part of the graph and in particular the "B"-labeled one was shared by the pair of the "A"-labeled.

After the duplication of the application node by means of the lower-left fan-out labeled with "A", we have two pairs of facing fans in Figure 1.18(5). Both have the same label, then they annihilate yielding the graph of Figure 1.18(6).

The shape of the graph is identical to the starting one, as we expected, but now both fans are labeled with a "B". Going on with the reduction—we can start again from the picture of Figure 1.18(1)—after one beta step and one fan-λ interaction, we get a pair of facing fans as in Figure 1.18(3), but with the same label ("B" in this case). Then the two fan will (erroneously) annihilate leading to a wrong graph that has no read-back in Λ.

Hence the simple mechanism of labeling fans is not adequate to maintain sufficient information for the correct matching of fans and we need to resort to a more complicated structure. Lamping's solution is to use a level structure and a pair of new nodes—the bracket and croissant (Figures 1.20 and 1.19)—that manage such structure, respectively increasing and decreasing levels.

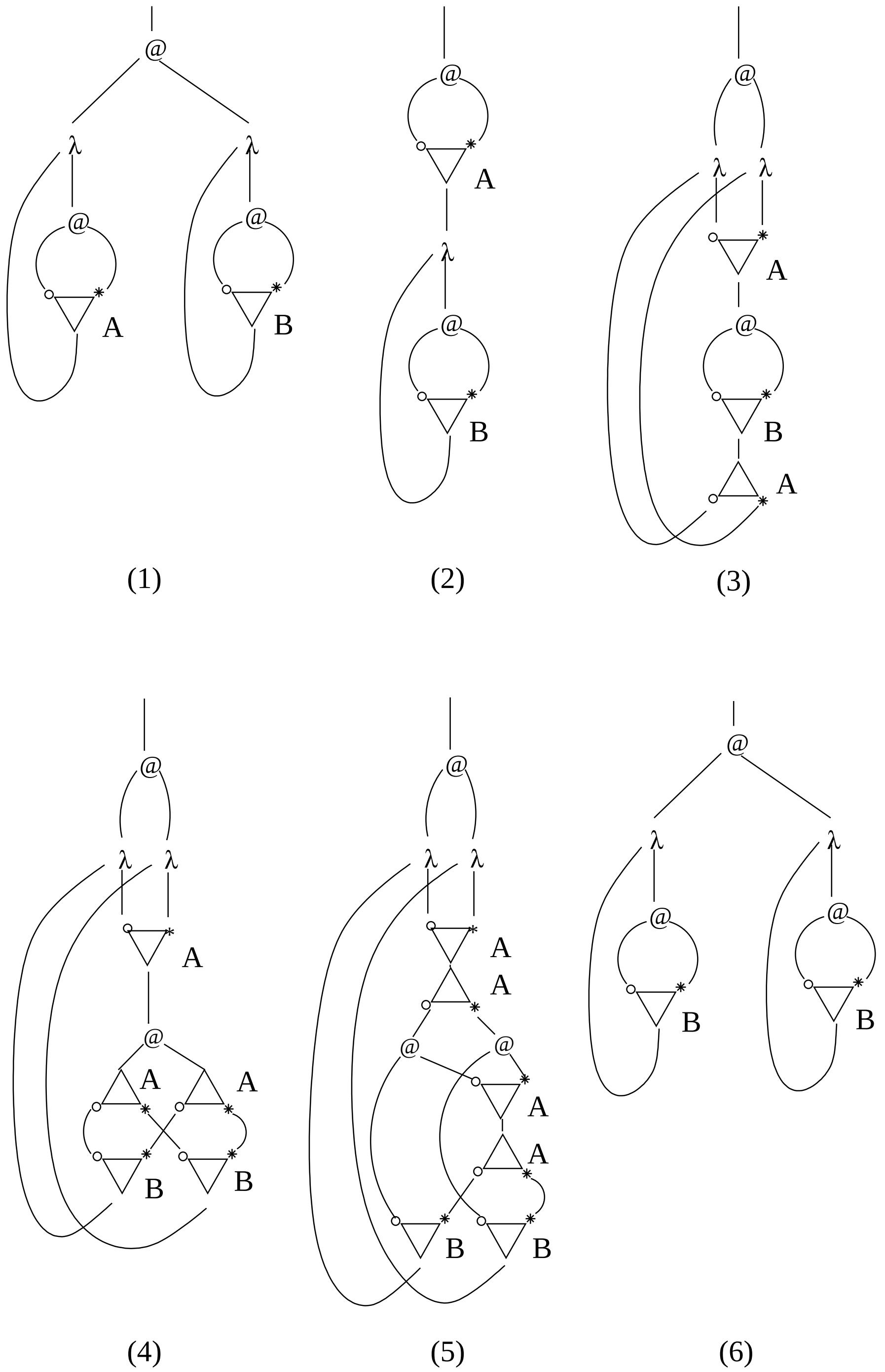


Figure 1.18: Labeling fans fails.

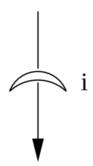


Figure 1.19: The croissant node.

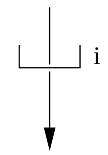


Figure 1.20: The bracket node.

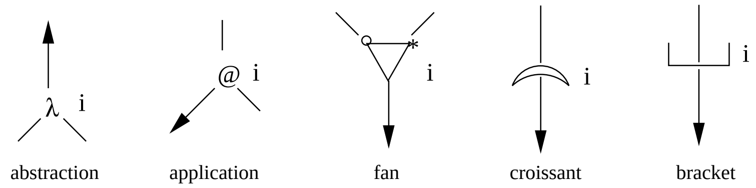


Figure 1.21: Sharing graph nodes.

In the following we present the oracle solution adopted by Asperti in [Asp94] that is quite different from Lamping's original one, but simpler to understand.

Summarizing, sharing graphs are undirected graphs built from the indexed nodes shown in Figure 1.21. Each node has a principal port—the arrow in Figure 1.21—through which it interacts with the other nodes. Figure 1.22 and 1.23 show the whole set of graph rewriting rules. In the set of rules in Figure 1.22 the facing nodes have the same level i. The first rule is the *beta-rule*. The other three are the *annihilation* of, respectively, croissants, fans and brackets. In Figure 1.23 f stands for any node and $i < j$. The node at lower level acts on the one at higher level and then, looking at the figure from left to right, a fan at level i *duplicates* any node at level greater then i, a croissant *decreases* and a bracket *increases* the level of any node at higher level it faces.

In the following we will consider (and count as)

- *fan-interactions*: all the interactions between fan nodes and application node, lambda node or fan nodes, i.e. the rule in the bottommost-leftmost picture of Figure 1.22 and the rule in the leftmost picture of Figure 1.23 where f is not a bracket or a croissant;

- *oracle-interactions*: all the interactions between brackets, croissants and other nodes, i.e. the central and rightmost rule depicted in Figure 1.23.

1.3.3 Initial Encoding

Given a lambda term M, the corresponding initial sharing graph at level 0 is inductively obtained as follows:

- if M is a variable, then the graph at level n consists of a single croissant of level n. The root of the graph is the auxiliary port of the croissant.

- If M is a lambda abstraction $\lambda x.M'$, the graph at level n is obtained from the initial graph of M' at level n whose root is connected to the body port of a new lambda node of level n. The root of the graph is the principal port of the lambda node. If x occurs free in M' then the corresponding wire is connected to the variable port of the lambda node, otherwise a new node—the *garbage node*—is created and its principal,

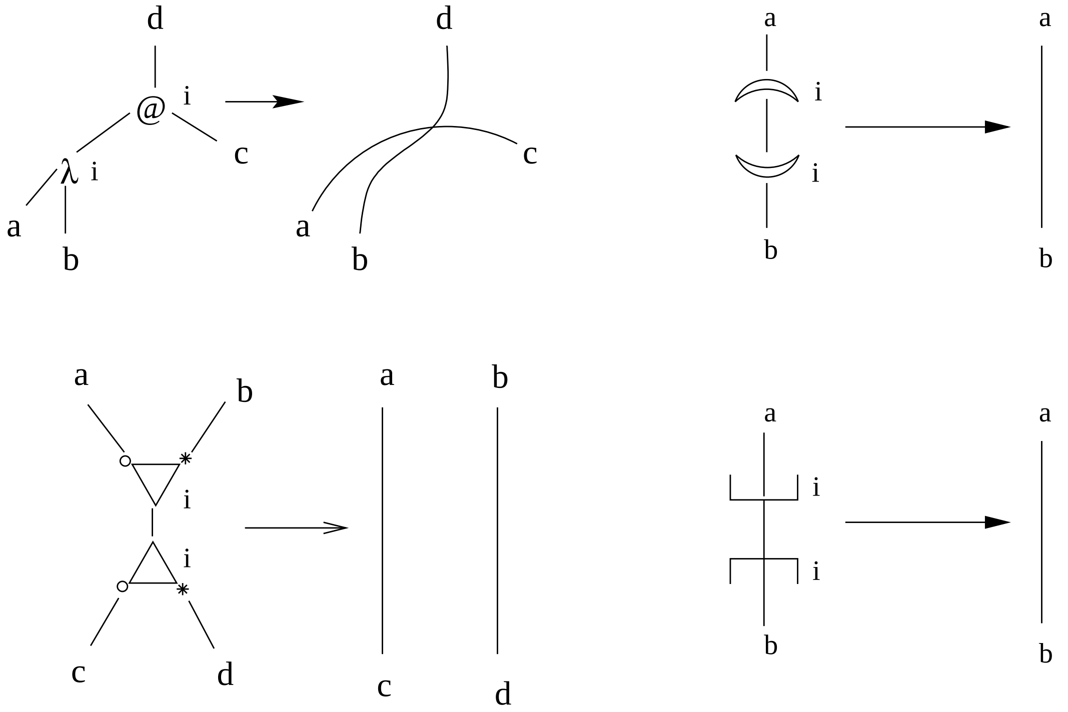


Figure 1.22: Interaction rules.

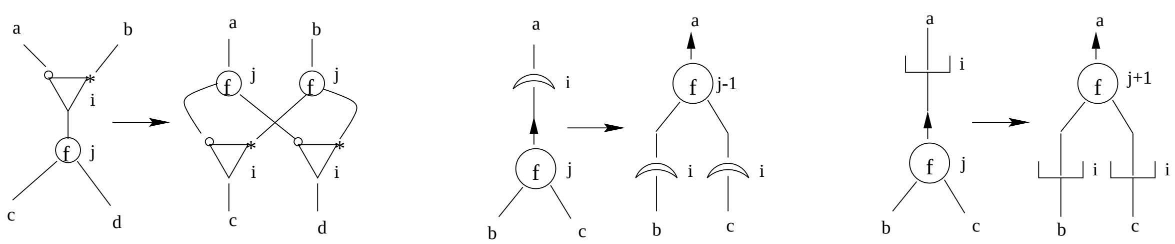


Figure 1.23: Duplication and level management.

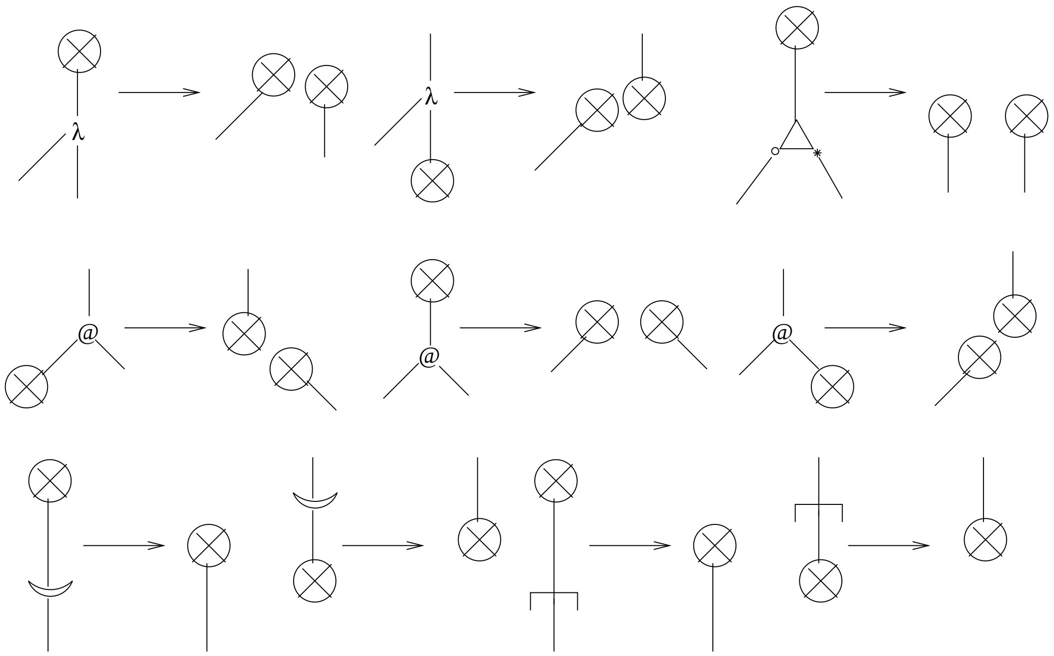


Figure 1.24: The “garbage collector”.

and unique, port is connected to the variable port of the lambda node. The garbage node acts as an eraser; the set of the interaction rules is shown in Figure 1.24.

- If M is an application $(P\ Q)$, the graph at level n is obtained from the graph of P at level i and the graph of Q at level $n+1$ where the root of the graph of P is connected to the principal port of a new application node, the root of the graph of Q is connected to the argument port of the application node and the root of the graph is the result port of the application node. All output wire of the graph of Q are connected to brackets of level n and finally all wire of the graph of P and Q that correspond to common variables are connected to fan of level n.

The initial translation function $[M]_n$ is summarized in Figure 1.25.

1.3.4 Read-back

The initial translation of a lambda term has an obvious correspondence with the syntax tree of the term, but as far as the reduction starts, as we have seen, the correspondence is lost. In order to recover the lambda term from the sharing graph we need to visit the graph recording the auxiliary port through which we enter in a fan-in to be able to choose the correct auxiliary port of the corresponding fan-out.

The level structure of the sharing graphs lead us to the use of a leveled data structure to record the auxiliary ports passed through. A *context* C is a list $\langle\langle\cdots\langle C', a_{n-1}\rangle, \cdots, a_1\rangle, a_0\rangle$ whose elements could be lists again. With $C^n[\cdot]$ we indicate the context whose first n elements are the ones of C and the tail is not specified, i.e.:

$$C^n[\cdot] = \langle\langle\cdots\langle\cdot, a_{n-1}\rangle, \cdots, a_1\rangle, a_0\rangle.$$

Putting a context C'' in $[\cdot]$ we obtain:

$$C^n[C''] = \langle\langle\cdots\langle C'', a_{n-1}\rangle, \cdots, a_1\rangle, a_0\rangle.$$

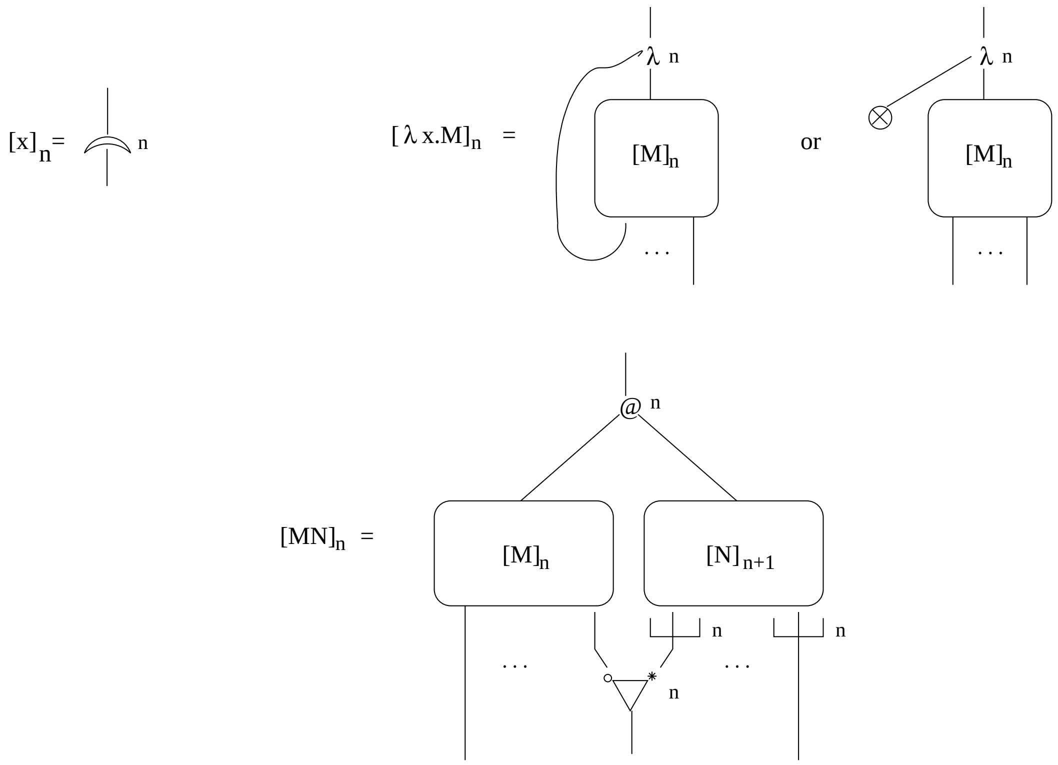
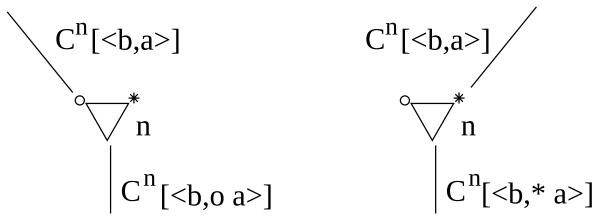
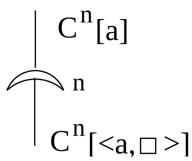


Figure 1.25: Initial translation.

When, during the read-back we pass through a fan-in of level n, we have to modify the context $C^n[\langle b, a\rangle]$ into $C^n[\langle b, \circ a\rangle]$ if entering from the $\circ$ auxiliary port, $C^n[\langle b, *a\rangle]$ otherwise. Conversely, passing through a fan out, we will exit from the $\circ$ port if the context is $C^n[\langle b, \circ a\rangle]$, from the $*$ port if the context is $C^n[\langle b, *a\rangle]$. Figure below summarizes all cases exposed so far.

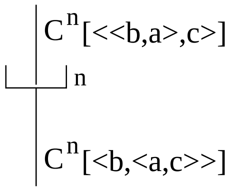


Passing through a croissant of level n from the auxiliary port to the principal port the context at level n is shifted at level $n+1$ and an empty context $\Box$ is added at level n. The process is reversible and then passing from the principal to the auxiliary port the (empty) context at level n is discarded and all context above decreased.



Regarding context transformations operated by bracket, traversing such node from auxiliary to principal port, we have to "save" the context at level $n+1$, i.e. the context

$C^n[\langle\langle b, a\rangle, c\rangle]$ becomes $C^n[\langle b, \langle a, c\rangle\rangle]$. Traversing in the opposite direction, we have to operate the inverse transformation.



Finally the lambda and application nodes does not transform contexts. Let us see an example: in Figure 1.26 is drawn a sharing graph obtained during the reduction of $((\lambda x.\lambda y.(x(x\ y))\ lamx.(x\ x))\lambda x.x)$.

Starting the read-back from the root of the graph we encounter first a croissant of level 0 and we add an empty context. Then we enter from the $\circ$ auxiliary port of a fan node of level 0 and we record this port in the context $\langle\cdot, \circ\rangle$. After the fan we encounter an application node, then we understand that the lambda term corresponding to the graph we are reading-back is an application $(M_1\ M_2)$. We start to read-back M_1 continuing the visit of the graph along the principal port of the application node. We traverse a croissant of level 1 adding an empty context, then we traverse a fan-in of level 1 from its $\circ$ auxiliary port obtaining the context $\langle\langle\cdot, \circ\rangle, \circ\rangle$. At this point we enter from the principal port of a fan-out of level 0 and we have to choose from which auxiliary port to exit. The context at level 0 is $\circ$ then we exit from that auxiliary port with the remaining context $\langle\langle\cdot, \circ\rangle, \square\rangle$. Then we meet a croissant of level 0 from its principal port, then we discard the empty context of level 0 and shift the higher contexts. After we traverse the upper fan-in in the figure from the $*$ auxiliary port obtaining the context $\langle\cdot, *\circ\rangle$ and coming to an application node. Hence the M_1 sub-term is again an application $(M_{1_1}\ M_{1_2})$ and we can go on with the read-back of the functional part M_{1_1} and continue the visit passing through the principal port of the application node. We traverse the croissant of level 1 adding an empty context and the fan-in of level 1 from the $\circ$ port, then we have to decide again from which auxiliary port of the fan-out of level 0 to exit. Now the context at level 0 is $*\circ$ and then we exit from the $*$ port with context $\langle\langle\cdot, \circ\rangle, \circ\rangle$. At this stage we meet a lambda node, thus the sub-term M_{1_1} has the form $\lambda x.M'_{1_1}$. We continue with the read-back of the body of the function and we encounter a bracket of level 0 form its auxiliary port. Then we traverse it saving the context at level 1, obtaining $\langle\cdot, \langle\circ, \circ\rangle\rangle$. After we add an empty context for the traversed croissant of level 1 and we restore context of level 1 traversing a bracket of level 0 from its principal port. Finally we get to a lambda node from it variable port, then the sub-term M'_{1_1} is a variable x.

The lambda term we have read-back so far is $((\lambda x.x\ M_{1_2})M_2)$ where M_{1_2} and M_2 are still unknown. In order continue the read-back we have to restart the process from the application node with the context $\langle\cdot, *\circ\rangle$ and exit from the argument port for reading-back M_{1_2}. In the same way, in order to read-back M_2 we have to exit from the argument port of the application node with the context $\langle\cdot, \circ\rangle$ and continue the visit. The final read-back of the graph in Figure 1.26 is $((\lambda x.x\ \lambda x.x)(\lambda x.x\ \lambda x.x))$.

1.4 The complexity of Lamping's algorithm

We have seen that Lamping's algorithm correctly implements the optimal beta reduction in the sense of Lévy. However, we do not know if the implementation proposed is efficient. In [LM96] we find a first attempt to study the efficiency of optimal reducers.

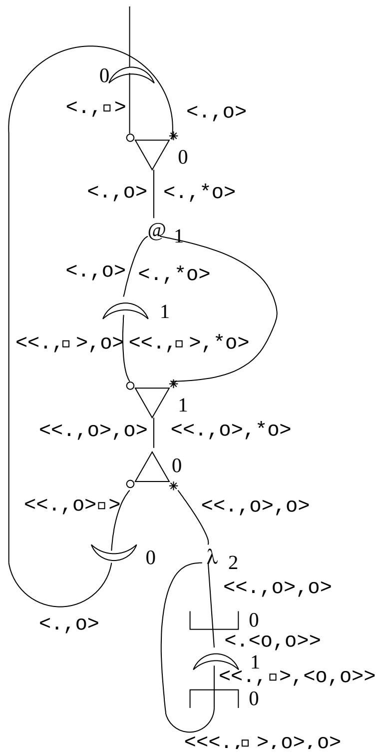


Figure 1.26: Read-back of $((\lambda x.x \ldots$

Consider the term

$$C_n = \lambda x.\overbrace{(2\ (2\ (\cdots (2}^{n \text{ times}}\ \ x)\cdots))$$

Since $(2\ 1)$ reduces to 1 in four standard β-steps, the term $E = (C_n\ 1)$ normalizes in $\Theta(n)$ standard β-steps. Looking at the sharing graph corresponding to E it is possible to prove that it normalizes in $\Theta(n)$ parallel β-steps and fan-interactions, but $\Omega(2^n)$ oracle interactions.

Theorem 20 *Lamping's algorithm take $\Omega(2^n)$ oracle interactions to effect n parallel β-steps.*

Moreover, if we slightly change the term E in $(C_n\ \lambda x.\lambda y.(y\ x))$ and we start the graph reduction, we obtain an intermediate sharing graph whose read-back is a lambda term in normal form, but the graph requires and exponential number of (useful) fan interactions to normalize.

Lawall and Mairson proved that Lamping's algorithm is inefficient in the cost model proposed by Frandsen and Sturtivant.

Theorem 21 *If $\nu(M)$ is the Frandsen-Sturtivant cost metric for evaluating a lambda term M, then the number of fan interactions required by Lamping's optimal evaluator grows as $\Omega(2^{\nu(M)})$.*

Proof: Consider the term $M = ((C_n\ \lambda x.\lambda y.(x\ (x\ y)))\ \lambda w.w)$. M has length $\Theta(N)$, reduces to $\lambda w.w$ of length $\Theta(1)$ and can be reduced in $5n+1$ parallel beta steps, then its cost $\nu(M)$ is $\Theta(n)$. However, looking at the graph reduction, the fan connecting the two occurrences of x in $\lambda x.\lambda y.(x\ (x\ y))$ is duplicated exponentially. $\square$

An analogous result was obtained by Asperti in [Asp96] with the following constructions:

$$\begin{aligned}
two' &= \lambda x.\lambda y.(\lambda z.(z\ (z\ y))\ \lambda w.(x\ w)) \\
\delta &= \lambda x.(x\ x) \\
g &= \lambda n.((((n\ \delta)\ two')\ \lambda x.x)\ q) \\
h &= \lambda n.((((n\ two')\ two')\ \lambda w.w)\ q)
\end{aligned}$$

For any Church integer n the number of fan interactions in the reduction of $(g\ \mathsf{n})$ and $(h\ \mathsf{n})$ is exponential in the number of family reductions.

Asperti [Asp96] proposed as new complexity measure the number of optimal beta reduction steps plus the number of annihilation rules between fans. The measure proposed seems to be polynomially related to the effective cost of the calculation, but, as observed by Lawall and Mairson, it seems undesirable to base the inherent difficulty of reducing a term on an implementation.

On the other hand Lawall and Mairson proposed two cost models based on *labelled lambda-calculus*. Such a variant of the lambda-calculus is obtained annotating each subterm of the initial lambda-term with a unique label. As reduction occurs, labels are concatenated according to certain rules, so that the labels encode the history of the computation. When $(\lambda x.M\ N)$ is reduced, the label on each free x in M is (reverse) concatenated with a unique new label, associated one to one with the label of the function, and the label of the argument; the label of N is concatenated with the new label and the label of the redex:

$$((\lambda x.M)^{\ell}\ N) \to M^{\underline{\ell}}[N^{\underline{\ell}}/x]$$

For example

$$\begin{aligned}
&((\lambda x.(x^1\ x^2)^3)^4\ (\lambda y.((\lambda z.(y^5\ z^6)^7)^8\ (\lambda w.w^9)^{10})^{11})^{12})^{13} \to \\
&\quad((\lambda y.((\lambda z.(y^5\ z^6)^7)^8\ (\lambda w.w^9)^{10})^{11})^{12\underline{4}1}\ (\lambda y.((\lambda z.(y^5\ z^6)^7)^8\ (\lambda w.w^9)^{10})^{11})^{12\underline{4}1})^{3\underline{4}13} \to \\
&\quad((\lambda z.((\lambda y.((\lambda z.(y^5\ z^6)^7)^8\ (\lambda w.w^9)^{10})^{11})^{12\underline{4}2\underline{12\underline{4}1}5}\ z^6)^7)^8\ (\lambda w.w^9)^{10})^{11\underline{12\underline{4}1}3\underline{4}13}
\end{aligned}$$

Lévy's original idea of optimality is that all redexes in the entire computation having identical labels on the functional part are reduced in one parallel beta step (notice that in the example the two copies of $(\lambda z.\dots)$ have the same label and hence should be reduced at the same time although they are nested, as discussed above).

The first cost model proposed by Lawall and Mairson in [LM96] relates the cost of reducing a term to the number of unique labels generated in the reduction. The second, more liberal, considers the sum of the length of the labels generated. They conjectured that the total number of interactions, including the oracle, is polynomial in the sum of the lengths of the labels. In [LM97] they proved that Lamping's abstract algorithm satisfies the first cost model proposed.

Theorem 22 *The total number of fan interactions needed to reach the normal form is $O(n^{10})$, where n is the sum of the number of labels generated during reduction, and the number of lambda and application nodes in the final graph.*

Results in [Asp96, LM96], even though they are related to a particular implementation of the optimal reduction—the Lamping's one—, gave a first indication that the number of optimal beta reduction steps is not an adequate measure of the complexity of the reduction. In the next section we will see that this is indeed the case. The measure proposed by Frandsen and Sturtivant should be discarded and the one indicated in [LM97] seems to be more promising. Unfortunately the conjecture of Mairson and Lawall is still unproved and the weight of the bookkeeping interactions in the reduction is an open question.

1.5 Inherent complexity of implementing optimal reduction

Is the number of redex families of a lambda term M (equivalent to the number of optimal beta reduction steps needed to reach the normal form of M) a good measure of the complexity of the reduction of M? In 1998 Asperti and Mairson [AM98] gave the definitive negative answer to this question.

1.5.1 The general case

Define the Kalmár elementary functions $\mathbf{K}_\ell(n)$ as $\mathbf{K}_0(n) = n$ and $\mathbf{K}_{\ell+1}(n) = 2^{\mathbf{K}_\ell(n)}$. We have already seen in Section 1.1.1 that the problem of determining the truth value of formulas in higher-order type theory is $\Omega(\mathbf{K}_\ell(n))$. Moreover Statman [Sta79] proved that it is possible to encode such non elementary problem in the simply typed lambda calculus.

The *eta-expansion* technique in [AM98] allows to reduce any simple typed term M in a polynomial number of optimal beta steps.

Definition 14 *Let σ be a simple type generated by the grammar*

$$\sigma ::= o \mid \sigma \to \sigma.$$

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

$$\eta_o(x) = x$$
$$\eta_{\alpha_1 \to \cdots \to \alpha_n \to o}(x) = \lambda y_1 : \alpha_1.\ \ldots\ \lambda y_n : \alpha_n.(x\ \eta_{\alpha_1}(y_1)\ \ldots\ \eta_{\alpha_n}(y_n))$$

If we label sharing graphs, we can notice a nice property: when a fan interacts with a lambda or an application node the residual fans have type labels on their wires strictly simpler than before the interaction occurred. All other interactions of fans do not affect the type labels. This means that when a fan has the base type labeling its wires it will never interact with—i.e. it will never duplicate—a lambda or an application node.

Definition 15 *Let M be a simply typed lambda term. The* optimal root $\mathbf{or}(M)$ *of M is derived by replacing every subterm of the form $\lambda x : \sigma.E$ with $\lambda x' : \sigma.(\lambda x.E\ \eta_\sigma(x'))$, where $\sigma \neq o$ and x occurs more than once in E. We refer to the new beta redexes introduced by this transformation as* preliminary *redexes.*

If we connect a fan to the root of an eta expansion $\eta_\sigma(x)$, after a linear number in the size of $\eta_\sigma(x)$ of fan-lambda and fan-application interactions, all residual fans in the sharing graph have base type labels on their wires.

Definition 16 *We define $\Delta(M)$ to be the sharing graph obtained from the graph of $\mathbf{or}(M)$ by reducing all of the preliminary redexes, and propagating all fan nodes to the base type.*

Notice that the number of optimal beta reductions needed to transform $\mathbf{or}(M)$ in $\Delta(M)$—the number of preliminary redexes—is bounded by the number of lambda in M. Define the size $\|\sigma\|$ of the simple type σ as the size of its tree representation and the size $|M|$ of the simply typed lambda term M as the number of its lambda and application plus the size of the type of its variables.

Theorem 23 *Let M be a simply typed lambda term. Then all sharing nodes in $\Delta(M)$ have atomic types and the number of nodes of $\Delta(M)$ is $\leq |M|$.*

Hence in the normalization of $\Delta(M)$ there will never be a lambda or an application duplication.

Theorem 24 *The total number of optimal beta reductions in the normalization of $\Delta(M)$ cannot exceed its initial size.*

The eta expansion technique allows to reduce every simply typed lambda term M in a polynomial number of optimal beta reductions with respect to the size $|M|$. Asperti and Mairson [AM98] gave an encoding of higher-order formulas over the finite base type $\mathbf{D_1} = \{true, false\}$ in the simply typed lambda calculus such that any formula F is true if and only if its encoding $\underline{F}$ reduces to $\lambda x.\lambda y.x$. Moreover if F quantifies over universes $\mathbf{D_i}$ for $i \leq k$, the size of its encoding is $|\underline{F}| = O(|F|(2k)!)$.

Finally they state the following theorem:

Theorem 25 *Let* TM *be a fixed Turing Machine that accepts of rejects an input x in $\mathbf{K}_\ell(|x|)$ steps. Then there exists a formula F_x in higher-order logic such that* TM *accepts x if and only if F_x is true. Moreover F_x only quantifies over universes $\mathbf{D_i}$ for $i \leq (\log^* |x|) + \ell + 6$ and has length $O(|x| \log^* |x|)$.*[2]

Hence for any elementary Turing Machine TM we can build a simply typed lambda term of size $O(|x| \log^c |x|)$ for any integer $c > 0$ such that TM accepts its input if and only if its encoding in the simply typed lambda calculus reduces to $\lambda x.\lambda y.x$. Combining this result with the eta expansion technique we have the main theorem of [AM98]:

Theorem 26 *There exists a set of λ-terms M_n where $|M| = O(|n| \log^c |n|)$ for any integer $c > 0$, such that M_n normalizes in $O(|M_n|)$ parallel beta steps, and the time needed to implement the parallel beta steps, on any first-class machine model, grows as $\Omega(\mathbf{K}_\ell(n))$ for any fixed integer $\ell \geq 0$.*

1.5.2 The polynomial case

The result of Asperti and Mairson can not be considered negatively with respect to optimal evaluators and in particular Lamping's algorithm. The non elementary bound is due to the specific complexity of the problem taken into account. The sharing mechanism is simply so powerful to dramatically reduce the number of redex families, but it can not do the "miracle" of reducing the complexity of the problem of deciding truth of higher-order formulas.

As far as we know there are no results proving the inefficiency of optimal evaluators. On the contrary it is possible to prove that Lamping's algorithm is polynomial for PTime-complete problems, as we proved in [Cop97] (essentially the same result was shown, among others, in [Sch01]).

Definition 17 *A* boolean circuit *is a graph $C = (V, E)$ such that:*

- *$V = \{1, \ldots, n\}$ is the set of gates;*
- *C is direct, acyclic and connected;*
- *if (i, j) is an edge in E then $i < j$;*
- *any gate $i \in V$ has a label $lab(i) \in \{true, false, and, or\}$;*

[2] The $\log^*$ factor can be removed as it is shown in the journal version of the paper of Asperti and Mairson appearing in Volume 170, Number 1 of Information and Computation.

- *the number of incoming edges of the gate i is defined as follows:*

$$indegree(i) = \begin{cases} 0 & \textit{if } lab(i) \in \{true, false\} \\ 2 & \textit{if } lab(i) = and \textit{ or } lab(i) = or; \end{cases}$$

- *the gates with $indegree(i) = 0$ are called* input gates*;*

- *gate n has no outgoing edges and is called* output gate*.*

The circuit consisting of a single gate labeled with $true$ ($false$) has boolean value true (false). The value of the circuit with output gate labeled with *and* (*or*) is calculated from the value of the sub-circuits connected to its two incoming edges.

The Circuit-Value problem is P-Complete with respect to the $\log n$-space reductions [Pap94]. Given a boolean circuit $C = (V, E)$ it is possible to encode it in the lambda calculus. The encoding $[lab(i)]$ of labels is the following:

$$\begin{aligned} [true] &= \lambda x.\lambda y.x \\ [false] &= \lambda x.\lambda y.y \\ [and] &= \lambda b_1.\lambda b_2.((b_1\ b_2)\ [false]) \\ [or] &= \lambda b_1.\lambda b_2.((b_1\ [true])\ b_2). \end{aligned}$$

The encoding algorithm first build the set $V' = \{(i, lab(i), outdegree(i)) | i \in V\}$. Then it performs a depth first visit of the graph following the decreasing order of gates writing in the output buffer:

$$\begin{cases} (([lab(i)] & \text{the first time it encounters } i \text{ with } lab(i) \in \{and, or\} \\ [lab(i)] & \text{if } lab(i) \in \{true, false\} \\ ) & \text{the second and third time it encounters } i \text{ with } lab(i) \in \{and, or\} \end{cases}$$

If the algorithm reaches a gate j with $outdegree(j) > 1$—an output gate for a shared sub-graph—it defers the visit in depth of the sub-graph and writes in the output buffer the corresponding variable x_j. When the firs visit ends the algorithm scans the set V' looking for a gate i with $outdegree(i) > 1$. When it reaches such a gate, it writes on the top of the buffer "(λx_i.", recursively encode the shared sub-graph writing the encoding at the end of the buffer and finally writes ")". The procedure ends when all V' has been scanned.

The complexity in space of the encoding algorithm is $O(\log n)$, being necessary just the space for the encoding of labels, a pointer to the processing gate, a pointer to the next gate, the number of gates and the number of edges. Moreover looking at the sharing graphs generated by the encoding algorithm it is easy to prove the following lemma:

Lemma 27 *Given a lambda term $\widehat{C}$ encoding a boolean circuit C, in the graph normalization of the sharing graph corresponding to $\widehat{C}$ there are only fan-lambda interactions between the lambda nodes of terms true and $false$ and only fan-fan annihilations in the body of them. No other fan-interactions occur (in particular there are no duplications of fans or applications).*

Theorem 28 *Every instance C of* Circuit Value *can be reduced in* $\log n$ *space to a sharing graph G_C normalizing in a polynomial number of interactions.*

1.6 Conclusions

Neither the number of standard beta reduction steps nor the number of optimal beta reductions could be considered as a good measure of the complexity of the reduction of lambda terms. On the other hand, the optimal reduction of Lévy and its implementation given by Lamping seem to be useful tools for investigating complexity questions inherent to the reduction. In particular we know that Lamping's algorithm is polynomial for PTime-complete problems and we can not blame the algorithm for the non exponential complexity obtained in the reduction of the terms of Asperti and Mairson because the problem encoded has that very inherent complexity as proved by Statman.

Results seen so far do not explain if, in the case of Lamping's algorithm, the non elementary overhead for the terms of Asperti and Mairson is due to the sharing management or to the oracle. We will answer to this question in the next chapter. We will see that under certain conditions it is possible to reduce lambda terms by using only the abstract Lamping's algorithm, i.e. with oracle of complexity $O(1)$.

We agree with Frandsen and Sturtivant when they say that we should include some measure of the length of reduction chains. However we have to forsake the number of beta reduction steps and we think that the sum of the length of the labels generated during the reduction (the proposal of [LM97]) is a good candidate in this direction.

2

Complexity of optimal sharing

Deciding truth value of higher-order formulas is a non elementary problem that can be encoded in the simple typed lambda calculus in such a way that the number of parallel beta reductions performed by any optimal reducers are just polynomial. The non elementary computation, in the case of Lamping's reducer, should be performed either by the oracle or by the sharing management.

We will see that already the sharing is non elementary. We will use Elementary Affine Logic as a tool. The proofs of this logic identify a subset of the simple typed lambda calculus for which the abstract Lamping's algorithm is correct, i.e. the naive solution of using labels for the correct matching of fans is sufficient. We will show that the non elementary problem taken into account can be encoded into this logic and thus we will obtain a non elementary lower bound to the complexity of sharing in the optimal reduction.

An alternative way to get the same result is to analyze optimal sharing reductions for the full MELL calculus, and prove that the number of interactions involving croissants and brackets is at most exponential in the number of other graph operations. Harry Mairson acquaint me in a personal communication in January 2000 with this analysis performed by him and Julia Lawall. The fan normal form and canonical beta reduction introduced by Mairson and Lawall in [LM97] allow to prove the exponential bound for oracle interactions. The analysis is a confirmation to the result of this Chapter. However, our encoding also show that there exists a large class of programs that can be implemented with the abstract Lamping's algorithm, i.e. the elementary Turing Machines. For an intuition on the improvement that can be achieved reducing terms with the abstract algorithm instead of the full version we refer the reader to the end of Section 4.1.

2.1 Linear Logic

Intuitionistic Linear Logic is the logical system defined by the following rules[1]:

$$\frac{\Gamma, x : A, y : B, \Delta \vdash_{\mathsf{LL}} M : C}{\Gamma, y : B, x : A, \Delta \vdash_{\mathsf{LL}} M : C}\ Ex$$

[1] $M\{^N/x\}$ denotes the usual substitution.

$$\frac{}{x : A \vdash_{\mathsf{LL}} x : A}\ Ax \qquad \frac{\Gamma \vdash_{\mathsf{LL}} M : A \quad \Delta, x : A \vdash_{\mathsf{LL}} N : B}{\Gamma, \Delta \vdash_{\mathsf{LL}} N\{^M/x\} : B}\ Cut$$

$$\frac{\Gamma \vdash_{\mathsf{LL}} M : A \quad x : B, \Delta \vdash_{\mathsf{LL}} N : C}{\Gamma, y : A \multimap B, \Delta \vdash_{\mathsf{LL}} N\{^{(y\ M)}/x\} : C}\ (\multimap, l) \qquad \frac{\Gamma, x : A \vdash_{\mathsf{LL}} M : B}{\Gamma \vdash_{\mathsf{LL}} \lambda x.M : A \multimap B}\ (\multimap, r)$$

Clearly, this fragment of Linear Logic corresponds to the linear lambda calculus via the Curry-Howard isomorphism. In order to exit from the linear framework and to recover the full whole power of intuitionistic logical systems, Linear Logic is equipped by a suitable modality '!' (read *of course* or *bang*). The modality applies to formulas, expressing their non linear nature. So, a formula of the kind $!A$ can be freely duplicated and erased. From the logical point of view, this means that we allow contraction and weakening on these formulas:

$$\frac{x :!A, y :!A, \Gamma \vdash_{\mathsf{LL}} M : B}{z :!A, \Gamma \vdash_{\mathsf{LL}} M\{^z/x, ^z/y\} : B}\ Contr$$

$$\frac{\Gamma \vdash_{\mathsf{LL}} M : B}{\Gamma, x :!A \vdash_{\mathsf{LL}} M : B}\ Weak$$

The other logical rules dealing with the modality are those allowing left and right introduction:

$$\frac{x : A, \Gamma \vdash_{\mathsf{LL}} M : B}{x :!A, \Gamma \vdash_{\mathsf{LL}} M : B}\ \epsilon \qquad \frac{!\Gamma \vdash_{\mathsf{LL}} M : B}{!\Gamma \vdash_{\mathsf{LL}} M :!B}\ (!, r)$$

where if $\Gamma = x_1 : A_1, \ldots, x_k : A_k$ then $!\Gamma = x_1 :!A_1, \ldots, x_k :!A_k$. It is possible to split the $(!, r)$ rule into two more elementary rules, obtaining an equivalent system:

$$\frac{\Gamma \vdash_{\mathsf{LL}} M : B}{!\Gamma \vdash_{\mathsf{LL}} M :!B}\ ! \qquad \frac{x :!!A, \Gamma \vdash_{\mathsf{LL}} M : B}{x :!A, \Gamma \vdash_{\mathsf{LL}} M : B}\ \delta$$

Intuitionistic logic can be embedded into Linear Logic. Take for example the translation function $(A \to B)^* =!(A)^* \multimap (B)^*$.

Definition 18 (Erasure) *Given the LL-formula A, $\overline{A}$ is the intuitionistic formula obtained from A erasing every ! and converting $\multimap$ into $\to$.*

Linear Logic proofs can be represented in a graphical way by means of Proof Nets [Gir87] that abstract from the syntactical bureaucracy due to the arbitrary sequentialization of inference rules. Proof Net links are depicted in Figure 2.1. A proof of a sequent $A_1, \ldots, A_n \vdash_{\mathsf{LL}} B$ is represented as a net with $n+1$ distinguished nodes, called *conclusions*: n negative conclusions for $A_1, \ldots, A_n$ and one positive conclusion for B. Positive and negative signs in Figures 2.1–2.9 respect such a convention.

Definition 19 (Proof Nets) *Proof nets are inductively defined as follows:*

- *an axiom link is a proof net. The left (right) node is the negative (positive) conclusion of the net;*
- *if N is a proof net with negative conclusion A and positive conclusion B, then the net in Figure 2.2 is a proof net with positive conclusion $A \multimap B$ and with all negative conclusions of N but A;*

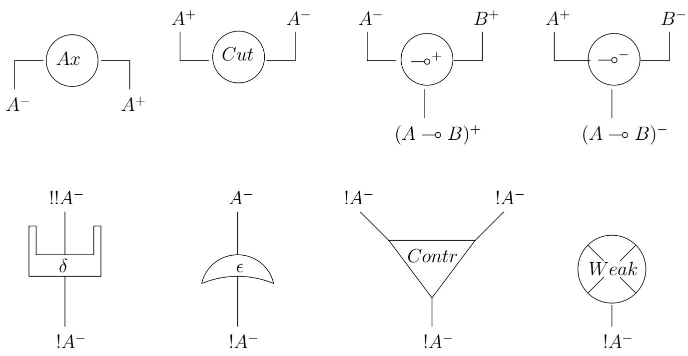


Figure 2.1: Proof Net links.

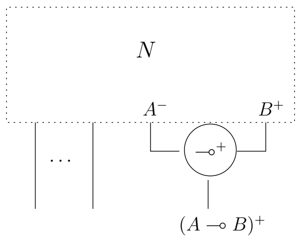


Figure 2.2: Positive linear implication.

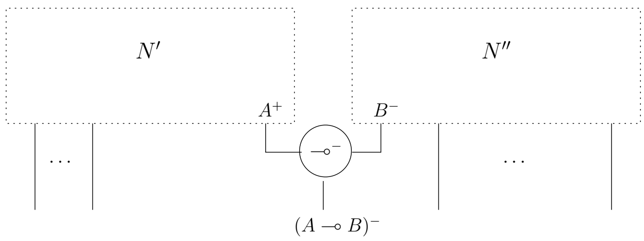


Figure 2.3: Negative linear implication.

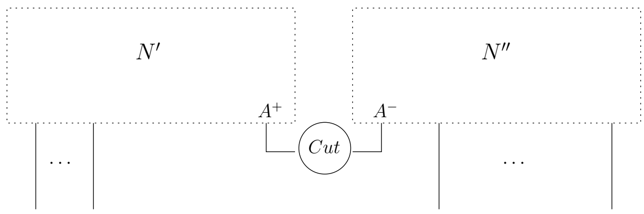


Figure 2.4: Cut.

- *if N' is a proof net with positive conclusion A and N'' is a proof net with negative conclusion B, then the net in Figure 2.3 is a proof net. The positive conclusion of N'' is the positive conclusion of the new net. $A \multimap B$ and all the negative conclusions of N' and N'' but B are negative conclusions of the new proof net;*

- *if N' is a proof net with positive conclusion A and N'' is a proof net with negative conclusion A, then the net in Figure 2.4 is a proof net. The positive conclusion of N'' is the new positive conclusion. All negative conclusions of N' and N'' but A are negative conclusions of the new proof net;*

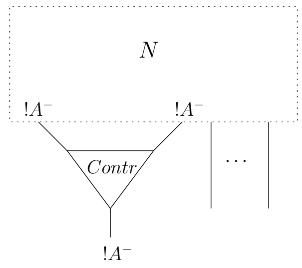


Figure 2.5: Contraction.

- *if N is a proof net with two negative conclusions $!A$, then the net in Figure 2.5 is a proof net with the same conclusions of N but one $!A$;*

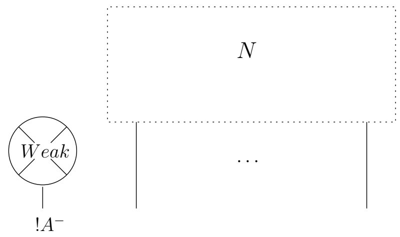


Figure 2.6: Weakening.

- *if N is a proof net, then the net in Figure 2.6 is a proof net with the same conclusions of N plus the negative conclusion $!A$;*

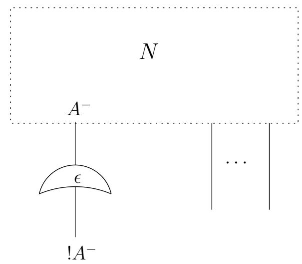


Figure 2.7: Left introduction of !.

- *if N is a proof net with negative conclusion A then the net in Figure 2.7 is a proof net with the same conclusions of N but A that is changed in $!A$;*

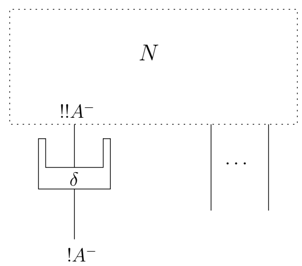


Figure 2.8: Delta rule.

- *if N is a proof net with negative conclusion $!!A$ then the net in Figure 2.8 is a proof net with the same conclusions of N but $!!A$ that is changed in $!A$;*

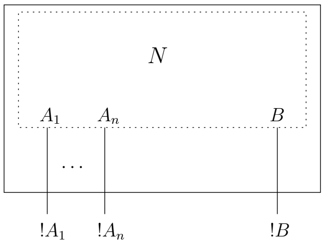


Figure 2.9: Box.

- *finally, if N is a proof net with negative conclusions $A_1, \ldots, A_n$ and positive conclusion B, then the net in Figure 2.9 is a proof net with negative conclusions $!A_1, \ldots, !A_n$ and positive conclusion $!B$.*

The cut elimination process can be performed in the proof net framework following the rewriting rules of Figure 2.10.

In 1992 Gonthier, Abadi and Lévy [GAL92] put in evidence a strong connection between proof nets and sharing graphs for optimal reduction. In fact it is possible to perform all rewriting of boxes in Figure 2.10 in a local way using the rewriting rules of Section 1.3.2, where $Contr$ behaves like a fan, ϵ like a croissant, δ like a bracket and $Weak$ like an eraser, and where boxes are substituted by levels. Moreover the initial translation of Section 1.3.3 of lambda terms in sharing graphs, corresponds to the use of the recursive type $D =!D \multimap D$.

In the next sections we briefly recall some fragment of Linear Logic with particular complexity property. We use these logics as tools in the study of the complexity of optimal reduction.

2.1.1 Elementary Affine Logic [Gir98, Asp98]

Removing ϵ and δ from the set of rules of Linear Logic and adding full weakening, we obtain a restricted system for which the cut elimination process, once the box-nesting depth of the proof is fixed, has elementary complexity in the size of the proof itself:

$$\frac{}{A \vdash_{\mathsf{EAL}} A}\,Ax \qquad \frac{\Gamma \vdash_{\mathsf{EAL}} A \quad A, \Delta \vdash_{\mathsf{EAL}} B}{\Gamma, \Delta \vdash_{\mathsf{EAL}} B}\,Cut \qquad \frac{\Gamma, A, B, \Delta \vdash_{\mathsf{EAL}} C}{\Gamma, B, A, \Delta \vdash_{\mathsf{EAL}} C}\,Ex$$

$$\frac{\Gamma \vdash_{\mathsf{EAL}} A \quad B, \Delta \vdash_{\mathsf{EAL}} C}{\Gamma, A \multimap B, \Delta \vdash_{\mathsf{EAL}} C}\,(\multimap, l) \qquad \frac{\Gamma, A \vdash_{\mathsf{EAL}} B}{\Gamma \vdash_{\mathsf{EAL}} A \multimap B}\,(\multimap, r)$$

$$\frac{\Gamma \vdash_{\mathsf{EAL}} C}{\Gamma, A \vdash_{\mathsf{EAL}} C}\,Weak \qquad \frac{\Gamma, !A, !A \vdash_{\mathsf{EAL}} B}{\Gamma, !A \vdash_{\mathsf{EAL}} B}\,Contr \qquad \frac{A_1, \ldots, A_n \vdash_{\mathsf{EAL}} B}{!A_1, \ldots, !A_n \vdash_{\mathsf{EAL}} !B}\,!$$

The complexity bound for the cut elimination process can be obtained considering a normalization strategy "by level". Starting from level 0—the part of the proof net outside every boxes—first reduce all linear cuts at that level, than perform all duplications and box fusions, again at level 0. After this process we can consider the next level and perform the same operations. Looking at Figure 2.10 we see that all rules of normalization of proof nets do not change the box nesting except for that involving δ and ϵ. Since those operators are not

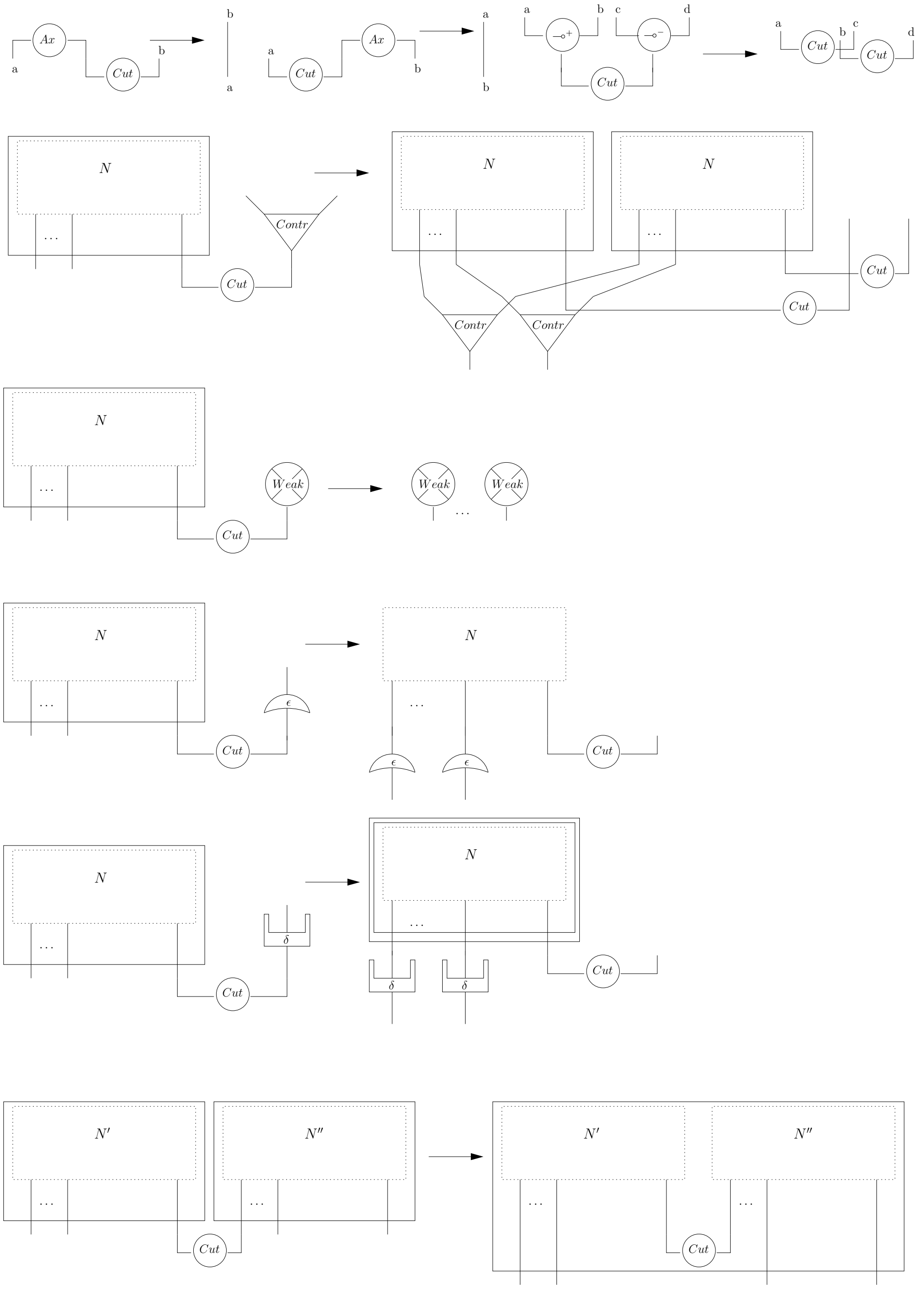


Figure 2.10: Cut elimination.

present in EAL, it is easy to see that the any normalization strategy and in particular the normalization by level can not change the levels of the proof net. Moreover no reduction at level i can create a new redex at level $j < i$. Then we are assured that the strategy by level is terminating. Now consider the costs: if the proof net has size n at level i, the number of graph reductions is at most n hence the cost of the graph reduction is at that level is at most $O(n^2)$. Moreover, the proof net increases the size to at most an exponential. Than the normal form of a proof net of size n and depth d is obtained with cost $O(\mathbf{K}_d(n))$.

Any elementary function can be coded as a proof of EAL. Considering the result in [GAL92] it is easy to see that the normalization of proofs of EAL can be implemented using the Lamping's abstract algorithm. In fact, as we have already observed, no operators for δ and ϵ and no rules changing the level of (parts of) the graph are present, hence we may simply label each fan with its level in the starting graph and then "match them by label", that is when two fans face they duplicate each other if they have different level, annihilate otherwise.

Finally notice that EAL is decidable as it has been proved in [Lag01].

2.1.2 Light Affine Logic [Gir98, Asp98]

If we add a second modality to EAL and split the ! rule into three new rules, the first and the second allowing to add a ! on the right hand side of the turnstile if there is at most one assumption on the left hand side, the third coping with the general case with a context of any dimension, but introducing the new modality §, we obtain a polynomial logic. Intuitively, if we allow to duplicate (contract) only formulas of type ! and we allow to !-box only proof nets with at most one negative conclusion, we force the normalization process to follow a particular order of reduction that, once the box-nesting depth of the proof is fixed, turns out to be polynomial in the size of the proof. Moreover any polynomial function can be coded as a proof of Light Affine Logic.

$$\frac{}{A \vdash_{\mathsf{LAL}} A}\, Ax \qquad \frac{\Gamma \vdash_{\mathsf{LAL}} A \quad A, \Delta \vdash_{\mathsf{LAL}} B}{\Gamma, \Delta \vdash_{\mathsf{LAL}} B}\, Cut \qquad \frac{\Gamma, A, B, \Delta \vdash_{\mathsf{LAL}} C}{\Gamma, B, A, \Delta \vdash_{\mathsf{LAL}} C}\, Ex$$

$$\frac{\Gamma \vdash_{\mathsf{LAL}} A \quad B, \Delta \vdash_{\mathsf{LAL}} C}{\Gamma, A \multimap B, \Delta \vdash_{\mathsf{LAL}} C}\, (\multimap, l) \qquad \frac{\Gamma, A \vdash_{\mathsf{LAL}} B}{\Gamma \vdash_{\mathsf{LAL}} A \multimap B}\, (\multimap, r)$$

$$\frac{\Gamma \vdash_{\mathsf{LAL}} C}{\Gamma, A \vdash_{\mathsf{LAL}} C}\, Weak \qquad \frac{\Gamma, !A, !A \vdash_{\mathsf{LAL}} B}{\Gamma, !A \vdash_{\mathsf{LAL}} B}\, Contr \qquad \frac{\vdash_{\mathsf{LAL}} B}{\vdash_{\mathsf{LAL}} !B}\, !_0 \qquad \frac{A \vdash_{\mathsf{LAL}} B}{!A \vdash_{\mathsf{LAL}} !B}\, !_1$$

$$\frac{A_1, \ldots, A_n, A_{n+1}, \ldots, A_{n+k} \vdash_{\mathsf{LAL}} B}{!A_1, \ldots, !A_n, \S A_{n+1}, \ldots, \S A_{n+k} \vdash_{\mathsf{LAL}} \S B}\, \S$$

2.1.3 Soft Linear Logic [Laf01]

Recently Yves Lafont has showed a simpler logic with the same property of LAL, i.e. any polynomial function can be coded as a proof of it and the normalization procedure, once the box-nesting depth of the proof is fixed, has polynomial complexity in the size of the proof. Soft Linear Logic is obtained from EAL simply removing weakening[2] and contraction and

[2]The removal of weakening is not necessary and the affine version of SLL still has the polynomial properties of the original one.

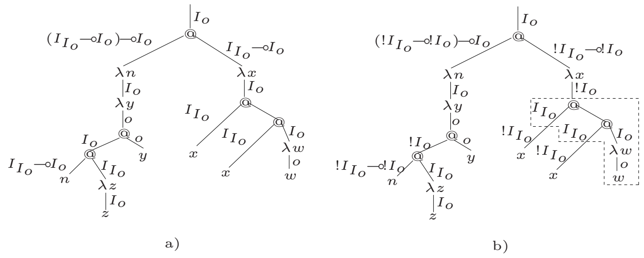


Figure 2.11: Type inference in EAL, I

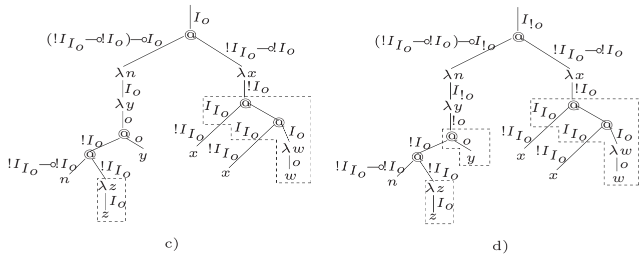


Figure 2.12: Type inference in EAL, II

substituting them with a sort of general contraction

$$\frac{\Gamma, A^{(n)} \vdash_{\mathsf{SLL}} B}{\Gamma, !A \vdash_{\mathsf{SLL}} B}\ Contr$$

where $A^{(n)}$ indicates zero or more copies of the formula A. The whole set of rule for Soft Linear Logic is the following:

$$\frac{}{A \vdash_{\mathsf{SLL}} A}\ Ax \qquad \frac{\Gamma \vdash_{\mathsf{SLL}} A \quad A, \Delta \vdash_{\mathsf{SLL}} B}{\Gamma, \Delta \vdash_{\mathsf{SLL}} B}\ Cut \qquad \frac{\Gamma, A, B, \Delta \vdash_{\mathsf{SLL}} C}{\Gamma, B, A, \Delta \vdash_{\mathsf{SLL}} C}\ Ex$$

$$\frac{\Gamma \vdash_{\mathsf{SLL}} A \quad B, \Delta \vdash_{\mathsf{SLL}} C}{\Gamma, A \multimap B, \Delta \vdash_{\mathsf{SLL}} C}\ (\multimap, l) \qquad \frac{\Gamma, A \vdash_{\mathsf{SLL}} B}{\Gamma \vdash_{\mathsf{SLL}} A \multimap B}\ (\multimap, r)$$

$$\frac{\Gamma, A^{(n)} \vdash_{\mathsf{SLL}} B}{\Gamma, !A \vdash_{\mathsf{SLL}} B}\ Contr \qquad \frac{A_1, \ldots, A_n \vdash_{\mathsf{SLL}} B}{!A_1, \ldots, !A_n \vdash_{\mathsf{SLL}} !B}\ !$$

2.2 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.

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. In [CM01] it has been given a type inference algorithm for λ-terms in EAL, however the typing of the relevant terms in Section 2.3 is obtained using some heuristics, which we describe in this section, as we need to type *family of terms* and not just terms.

We may single out three main steps in the process of type inference[3]: “looking for contractions”, “boxing arguments” and “opening boxes”. We discuss these steps by going through an easy example.

Let

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

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 2.11 (a), where $I_\alpha{=}\alpha \multimap \alpha$ 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 $M{=}\lambda x.(x\ (x\ \lambda w.w))$. 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 2.11 (a) corresponds to the following derivation

$$\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$o \vdash o$}
\RightLabel{$(\multimap, r)$}
\UnaryInfC{$\vdash I_o$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$I_o \vdash I_o$}
\RightLabel{$(\multimap, l)$}
\BinaryInfC{$I_o \multimap I_o \vdash I_o$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$I_o \vdash I_o$}
\RightLabel{$(\multimap, l)$}
\BinaryInfC{$\mathbf{I_{I_o}, I_{I_o} \vdash I_o}$}
\RightLabel{$Contr?$}
\UnaryInfC{$\mathbf{I_{I_o} \vdash I_o}$}
\RightLabel{$(\multimap, r)$}
\UnaryInfC{$\vdash I_{I_o} \multimap I_o$}
\end{prooftree}$$

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

$$\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$o \vdash o$}
\RightLabel{$(\multimap, r)$}
\UnaryInfC{$\vdash I_o$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$I_o \vdash I_o$}
\RightLabel{$(\multimap, l)$}
\BinaryInfC{$I_o \multimap I_o \vdash I_o$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$I_o \vdash I_o$}
\RightLabel{$(\multimap, l)$}
\BinaryInfC{$I_{I_o}, I_{I_o} \vdash I_o$}
\RightLabel{$(!)$}
\UnaryInfC{$!I_{I_o}, !I_{I_o} \vdash !I_o$}
\RightLabel{$(Contr.)$}
\UnaryInfC{$!I_{I_o} \vdash !I_o$}
\RightLabel{$(\multimap, r)$}
\UnaryInfC{$\vdash !I_{I_o} \multimap !I_o$}
\end{prooftree}$$

The corresponding typing is represented in Figure 2.11 (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 2.11 (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 2.12 (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 2.12 (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.

[3]These steps are ad hoc heuristics. A complete approach to type inference is given in the next chapter.

The final decoration of Figure 2.12 (d) represents the following derivation in EAL:

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

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 2.12 (d) after the abstraction of y, obtaining for N the type $!^{n+m} I_o$.

Finally we remark that there exists simply typed terms without any EAL types. 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 (see Figure 2.13 where the redexes fired at every step are indicated by a dashed oval). 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. For a formal proof we refer the reader to Example 1 in Chapter 3.

2.3 Coding type theory into EAL proofs

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], the main technical contribution being the type-inference inside EAL.

Remark 20 *The modifications to the encoding of [AM98] 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 25. In particular, define $x^1 < y^1 = \exists x^2.true \in x^2 \wedge false \notin x^2 \wedge y^1 \in x^2 \wedge x^1 \notin x^2$ and $x^1 = y^1 = \neg(x^1 < y^1 \vee y^1 < x^1)$.*

Define the type of Booleans as $\mathtt{B}{=}!o \multimap !o \multimap !o$; write L^τ_α for the EAL type of the generic lists of elements of type α:

$$L^\tau_\alpha = !(\alpha \multimap \tau \multimap \tau) \multimap !(\tau \multimap \tau).$$

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-term $\hat{Q} : L^{!^n\mathtt{B}}_\sigma$ 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 $(\hat{Q}\ \lambda z.(\mathtt{AND}\ (\hat{\Psi}\ z))\ \mathtt{true})$ is the term encoding the formula $\forall z \in Q\ \Psi$.

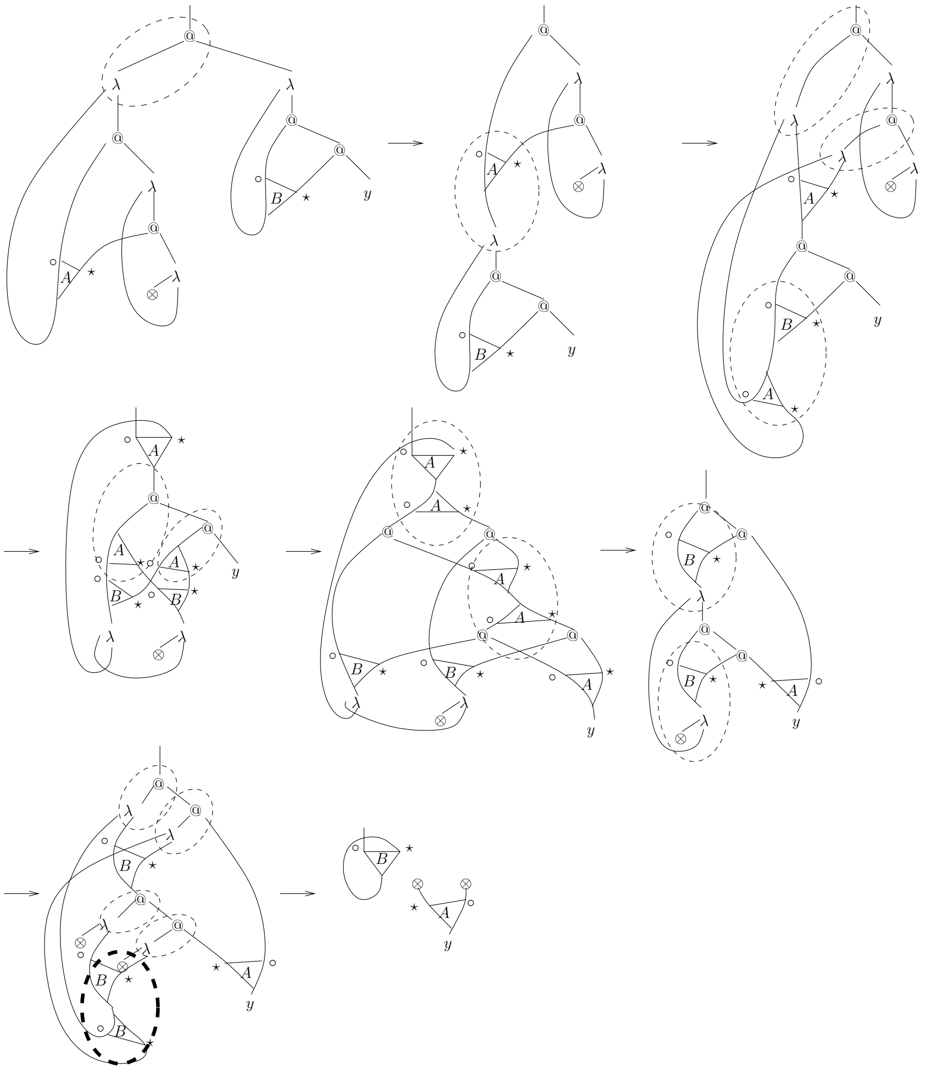


Figure 2.13: Incorrect reduction of $(\lambda n.(n\ \lambda y.(n\ \lambda z.y))\ \lambda x.(x\ (x\ y)))$.

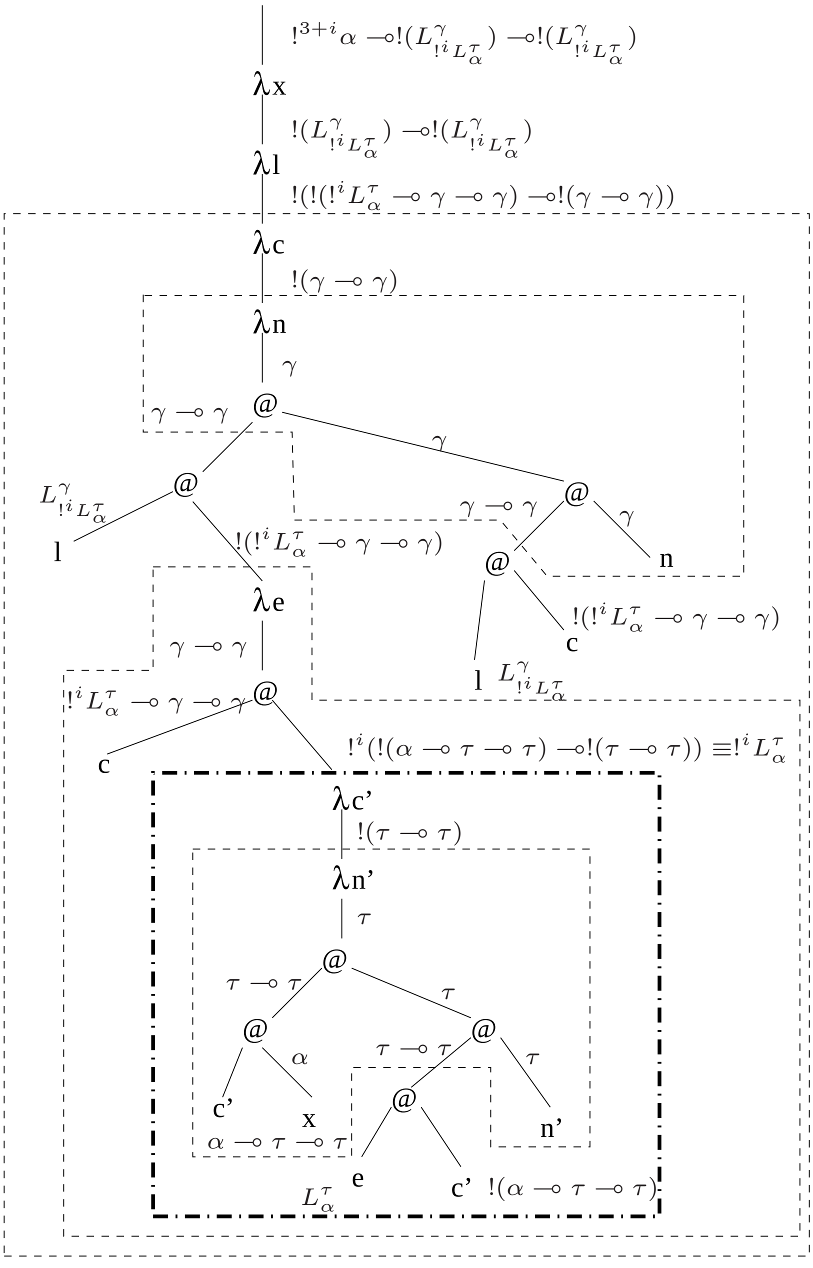


Figure 2.14: EAL type of *double*

Encoding of prime formulas can be understood as the λ-calculus translation of the following inductive definitions:

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

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.

Definition 21 (Erasure) *For A EAL-type, $(A)^*$ is the simple type obtained from A by stripping all !'s and changing all $\multimap$ into $\rightarrow$.*

Definition 22 (Size of terms)

- *For σ simple type,* size $||\sigma||$ *is defined inductively as follows:*

$$\begin{aligned} ||o|| &= 1 \\ ||\alpha \rightarrow \beta|| &= 1 + ||\alpha|| + ||\beta||. \end{aligned}$$

- *For $M : \sigma$ simply typed term,* size $|M|$ *is defined inductively as follows:*

$$\begin{aligned} |x| &= ||\sigma|| \textit{ if } x \textit{ has type } \sigma \\ |\lambda x.M| &= 1 + |M| \\ |(M\ N)| &= 1 + |M| + |N|. \end{aligned}$$

- *For $M : A$ EAL-typed term,* size $|M|$ *is defined inductively as follows:*

$$\begin{aligned} |x| &= ||(A)^*|| \textit{ if } x \textit{ has type } A \\ |\lambda x.M| &= 1 + |M| \\ |(M\ N)| &= 1 + |M| + |N|. \end{aligned}$$

Definition 23 (Depth)

- *For A EAL type,* depth $d(A)$ *is defined inductively as follows:*

$$\begin{aligned} d(o) &= 0 \\ d(A \multimap B) &= \max\{d(A), d(B)\} \\ d(!A) &= 1 + d(A). \end{aligned}$$

- *For $M : A$ EAL-typed term,* depth $d(M : A)$ *is the maximum number of nested boxes in the EAL-derivation of $M : A$.*

The full encoding with EAL-types, is summarized in Table 2.1. The rest of the section will prove all the statements about types, size and depth of EAL-derivations.

Note to Table 2.1

(*) The size of *double* and *powerset* depends on types α, γ, τ and are relevant only to the calculus of $\mathbf{D_k}$. Depth of eq_k is relevant only to the calculus of depth of $member_k$.

(**) Δ_1 and Δ_k are defined in Definition 24. M_k is defined in Lemma 36. E^k is defined in the proof of Lemma 36. d is the constant of Theorem 4.1 in [AM98]. The function f is defined in Theorem 37.

Table 2.1: Encoding.

Formula	Term	Type	Size	Depth
$\mathtt{true}$	$\lambda x\ y.x$	$\forall n \geq 0.\ !^n\mathtt{B}$	3	n
$\mathtt{false}$	$\lambda x\ y.y$	$\forall n \geq 0.\ !^n\mathtt{B}$	3	n
$\mathtt{NOT}$	$\lambda b\ x\ y.(b\ y\ x)$	$\forall n \geq 0.\ !^n\mathtt{B} \multimap !^n\mathtt{B}$	12	n
$\mathtt{AND}$	$\lambda b_1\ b_2\ x\ y.(b_1\ (b_2\ x\ y)\ y)$	$\forall n \geq 0.\ !^n\mathtt{B} \multimap !^n\mathtt{B} \multimap !^n\mathtt{B}$	21	n
$\mathtt{OR}$	$\lambda b_1\ b_2\ x\ y.(b_1\ x\ (b_2\ x\ y))$	$\forall n \geq 0.\ !^n\mathtt{B} \multimap !^n\mathtt{B} \multimap !^n\mathtt{B}$	21	n
$\mathtt{IFF}$	$\lambda b_1\ b_2\ x\ y.(b_1$ $(b_2\ x\ y)(b_2\ y\ x))$	$\forall n \geq 1.\ !^n\mathtt{B} \multimap !^n\mathtt{B} \multimap !^n\mathtt{B}$	29	n
double	$\lambda x\ l\ c\ n.(l\ \lambda e.(c\ \lambda c'\ n'.$ $(c'\ x(e\ c'\ n')))(l\ c\ n))$	$\forall \alpha, \gamma, \tau, \forall i \geq 0.$ $!^{3+i}\alpha \multimap !L^{\gamma}_{!^i L^{\tau}_{\alpha}} \multimap !L^{\gamma}_{!^i L^{\tau}_{\alpha}}$	(*)	$3+i$
powerset	$\lambda A^*.(A^*\ double \lambda c\ n.$ $(c\ \lambda c'\ n'.n'\ n))$	$\forall \alpha, \gamma, \tau, \forall i \geq 0.$ $L^{!L^{\gamma}_{!^i L^{\tau}_{\alpha}}}_{!^{3+i}\alpha} \multimap !^2 L^{\gamma}_{!^i L^{\tau}_{\alpha}}$	(*)	$4+i$
$\mathbf{D_1}$	$\lambda c\ n.(c\ \mathtt{true}\ (c\ \mathtt{false}\ n))$	$\forall \tau_1, \forall n_0, n_1 \geq 0.\ \Delta_1$ (**)	$\leq 2d$ (**)	$n_0 + n_1 + 1$
$\mathbf{D_k}$	$(powerset\ \mathbf{D_{k-1}})$	$\forall \tau_1, \dots, \tau_k,$ $\forall n_0, \dots, n_{k-1} \geq 0,$ $\forall n_k \geq 2(k-1).\ \Delta_k$ (**)	$\leq d(2k)!$	$k + \sum_{i=0}^{k} n_i$
eq_1	$\mathtt{IFF}$	$\forall n \geq 1.\ !^n\mathtt{B} \multimap !^n\mathtt{B} \multimap !^n\mathtt{B}$	29	n
$member_k$	$\lambda x^{k-1}.\lambda y^k.(y^k\ \lambda y^{k-1}.(\mathtt{OR}$ $(eq_{k-1}\ x^{k-1}\ y^{k-1}))\ \mathtt{false})$	$\forall n \geq 0.$ $!^{n+\max\{2k-5,2\}}M_{k-1}$ $\multimap !^{n+\max\{2k-7,0\}}M_k$ $\multimap !^{n+\max\{4k-9,2(k-1)\}}\mathtt{B}$ (**)	$\leq c * k^2 + d(2(k-1))!$	$n + 2(k-1) + \max\{2k-7, 0\}$
eq_k	$\lambda x^k.\lambda y^k.(\lambda op.(\mathbf{D_{k-1}}\ \lambda z^{k-1}.(\mathtt{AND}$ $(\mathtt{IFF}\ (op\ z^{k-1}\ x^k)\ (op\ z^{k-1}\ y^k)))$ $\mathtt{true})\ member_k)$	E^k (**)	$\leq c * k^2 + d(2(k-1))!$	(*)
Higher-order logic formulas				
$\mathbf{true} \in x^2$	$(member_2\ \mathtt{true}\ x^2)$	$!^{f(2)}\mathtt{B}$ (**)	$\leq 4c + 2d$	$f(2)$
$\mathbf{false} \in x^2$	$(member_2\ \mathtt{false}\ x^2)$	$!^{f(2)}\mathtt{B}$	$\leq 4c + 2d$	$f(2)$
$x^{k-1} \in x^k$	$(member_k\ x^{k-1}\ x^k)$	$!^{f(k)}\mathtt{B}$	$\leq c * k^2 + d(2(k-1))!$	$f(k)$
$\neg\Phi'$	$(\mathtt{NOT}\ \Psi')$	$!^{f(k_j)+h+\sum_{\ell \in J} 2(k_\ell - 1)}\mathtt{B}$	$O(\|\neg\Phi'\|d(2k_{\mathrm{MAX}})!)$	$f(k_j) + h + \sum_{\ell \in J} 2(k_\ell - 1)$
$\Phi_1 \wedge \Phi_2$	$(\mathtt{AND}\ \Psi_1\ \Psi_2)$	$!^{f(k_j)+h+\sum_{\ell \in J} 2(k_\ell - 1)}\mathtt{B}$	$O(\|\Phi_1 \wedge \Phi_2\|d(2k_{\mathrm{MAX}})!)$	$f(k_j) + h + \sum_{\ell \in J} 2(k_\ell - 1)$
$\Phi_1 \vee \Phi_2$	$(\mathtt{OR}\ \Psi_1\ \Psi_2)$	$!^{f(k_j)+h+\sum_{\ell \in J} 2(k_\ell - 1)}\mathtt{B}$	$O(\|\Phi_1 \vee \Phi_2\|d(2k_{\mathrm{MAX}})!)$	$f(k_j) + h + \sum_{\ell \in J} 2(k_\ell - 1)$
$\forall y^{k_i}\Phi'$	$(\mathbf{D_{k_i}}\ \lambda y^{k_i}.(\mathtt{AND}\ (\lambda x^{k_i}.\Psi'\ y^{k_i}))\ \mathtt{true})$	$!^{f(k_j)+h+\sum_{\ell \in J} 2(k_\ell - 1)}\mathtt{B}$	$O(\|\forall y^{k_i}\Phi'\|d(2k_{\mathrm{MAX}})!)$	$f(k_j) + h + \sum_{\ell \in J} 2(k_\ell - 1)$
$\exists y^{k_i}\Phi'$	$(\mathbf{D_{k_i}}\ \lambda y^{k_i}.(\mathtt{OR}\ (\lambda x^{k_i}.\Psi'\ y^{k_i}))\ \mathtt{false})$	$!^{f(k_j)+h+\sum_{\ell \in J} 2(k_\ell - 1)}\mathtt{B}$	$O(\|\exists y^{k_i}\Phi'\|d(2k_{\mathrm{MAX}})!)$	$f(k_j) + h + \sum_{\ell \in J} 2(k_\ell - 1)$

Lemma 29 (Booleans) *We encode boolean values in EAL as follows:*

1. $\mathtt{true} = \lambda x\ y.x :!^n\mathtt{B}$ *for any* $n \geq 0$, $|\mathtt{true}| = 3$, $d(\mathtt{true} :!^n\mathtt{B}) = n$ *and the deepest type that appears in the EAL-derivation of* $\mathtt{true} :!^n\mathtt{B}$ *is* $!^n\mathtt{B}$, *of depth* $d(!^n\mathtt{B}) = n + 1$;
2. $\mathtt{false} = \lambda x\ y.y :!^n\mathtt{B}$ *for any* $n \geq 0$, $|\mathtt{false}| = 3$, $d(\mathtt{false} :!^n\mathtt{B}) = n$, *deepest type* $!^n\mathtt{B}$ *of depth* $d(!^n\mathtt{B}) = n + 1$;
3. $\mathtt{NOT} = \lambda b\ x\ y.(b\ y\ x) :!^n\mathtt{B} \multimap !^n\mathtt{B}$ *for any* $n \geq 0$, $|\mathtt{NOT}| = 12$, $d(\mathtt{NOT} :!^n\mathtt{B} \multimap \cdots) = n$, *deepest type* $!^n\mathtt{B}$ *of depth* $d(!^n\mathtt{B}) = n + 1$;
4. $\mathtt{AND} = \lambda b_1\ b_2\ x\ y.(b_1\ (b_2\ x\ y)\ y) :!^n\mathtt{B} \multimap !^n\mathtt{B} \multimap !^n\mathtt{B}$ *for any* $n \geq 0$, $|\mathtt{AND}| = 21$, $d(\mathtt{AND} : !^n\mathtt{B} \multimap \cdots) = n$, *deepest type* $!^n\mathtt{B}$ *of depth* $d(!^n\mathtt{B}) = n + 1$;
5. $\mathtt{OR} = \lambda b_1\ b_2\ x\ y.(b_1\ x\ (b_2\ x\ y)) :!^n\mathtt{B} \multimap !^n\mathtt{B} \multimap !^n\mathtt{B}$ *for any* $n \geq 0$, $|\mathtt{OR}| = 21$, $d(\mathtt{OR} :!^n\mathtt{B} \multimap \cdots) = n$, *deepest type* $!^n\mathtt{B}$ *of depth* $d(!^n\mathtt{B}) = n + 1$;
6. $\mathtt{IFF} = \lambda b_1\ b_2\ x\ y.(b_1\ (b_2\ x\ y)\ (b_2\ y\ x)) :!^n\mathtt{B} \multimap !^n\mathtt{B} \multimap !^n\mathtt{B}$ *for any* $n \geq 1$, $|\mathtt{IFF}| = 29$, $d(\mathtt{IFF} :!^n\mathtt{B} \multimap \cdots) = n$, *deepest type* $!^n\mathtt{B}$ *of depth* $d(!^n\mathtt{B}) = n + 1$;

Proof: Simple inspection of the EAL-derivations. □

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

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

where

$$\begin{aligned} \mathit{powerset} &= \lambda A^*.(A^*\ \mathit{double}\ \lambda c.\lambda n.(c\ \lambda c'.\lambda n'.n'\ n)) \\ \mathit{double} &= \lambda x.\lambda l.\lambda c.\lambda n.(l \\ &\qquad \lambda e.(c\ \lambda c'.\lambda n'.(c'\ x\ (e\ c'\ n')))\ (l\ c\ n)). \end{aligned}$$

Lemma 30 *For any EAL-type* α, γ *and* τ *and for* $i \geq 0$:

- *double has type* $!^{3+i}\alpha \multimap !(L^\gamma_{!^i L^\tau_\alpha}) \multimap !(L^\gamma_{!^i L^\tau_\alpha})$; $d(\mathit{double} :!^{3+i}\alpha \multimap \cdots) = 3 + i$ *and the deepest type in the EAL-type derivation of double is* $!(L^\gamma_{!^i L^\tau_\alpha})$ *of depth* $2 + \max\{d(\gamma), 1 + i + d(\tau), 1 + i + d(\alpha)\}$.
- *powerset has type* $L^{!(L^\gamma_{!^i L^\tau_\alpha})}_{!^{3+i}\alpha} \multimap !!(L^\gamma_{!^i L^\tau_\alpha})$. $d(\mathit{powerset} : L^{!(L^\gamma_{!^i L^\tau_\alpha})}_{\ldots} \multimap \cdots) = 4 + i$ *and the deepest type in the EAL-type derivation of powerset is* $!!(L^\gamma_{!^i L^\tau_\alpha})$ *of depth* $3 + \max\{d(\gamma), 1 + i + d(\tau), 1 + i + d(\alpha)\}$.

Proof: As for types, see figures 2.14 and 2.15, where the bold dashed lines stand for i boxes.

The deepest type in the derivation of *double* $:!^{3+i}\alpha \multimap !(L^\gamma_{!^i L^\tau_\alpha}) \multimap !(L^\gamma_{!^i L^\tau_\alpha})$ is the final type.

$$\begin{aligned} &d(!^{3+i}\alpha \multimap !(L^\gamma_{!^i L^\tau_\alpha}) \multimap !(L^\gamma_{!^i L^\tau_\alpha})) = \\ &\quad = \max\{3 + i + d(\alpha), 1 + \max\{1 + d(\gamma), 1 + i + \max\{1 + d(\tau), 1 + d(\alpha)\}\}\} \\ &\qquad\qquad = d(!(L^\gamma_{!^i L^\tau_\alpha})). \end{aligned}$$

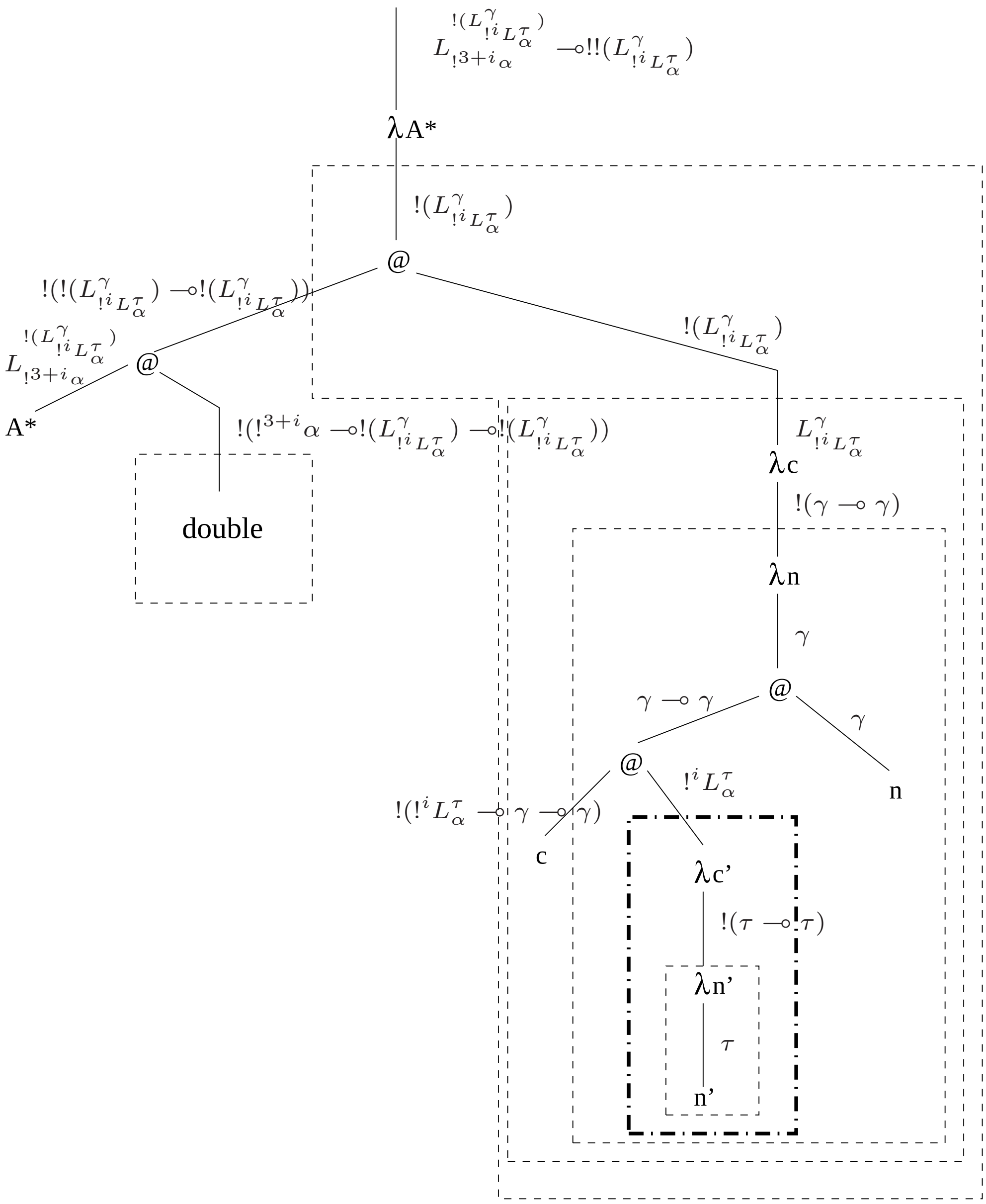


Figure 2.15: EAL type of *powerset*

Analogously, for $powerset$: $L_{!^{3+i}\alpha}^{!(L^{\gamma}_{!^i L^{\tau}_{\alpha}})} \multimap !!(L^{\gamma}_{!^i L^{\tau}_{\alpha}})$:

$$d(L_{!^{3+i}\alpha}^{!(L^{\gamma}_{!^i L^{\tau}_{\alpha}})} \multimap !!(L^{\gamma}_{!^i L^{\tau}_{\alpha}})) =$$

$$= \max \left\{ \begin{array}{r} \max \left\{ \begin{array}{r} 1+3+i+d(\alpha), \\ 1+1+\max\{1+d(\gamma), 1+i+\max\{1+d(\tau), 1+d(\alpha)\}\} \end{array} \right\}, \\ 2+\max\{1+d(\gamma), 1+i+\max\{1+d(\tau), 1+d(\alpha)\}\} \end{array} \right\}$$
$$= d(!!(L^{\gamma}_{!^i L^{\tau}_{\alpha}})).$$

□

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

Definition 24

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

we can prove the following:

Lemma 31

1. *$\forall \tau_1, \ldots, \tau_k$ 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.*

2. *$d(\mathbf{D_k} : \Delta_k) = k + \sum_{i=0}^{k} n_i$.*

3. *The deepest type in the derivation of $\mathbf{D_k}$ is Δ_k of depth $d(\Delta_k) = \max_{0 \leq j \leq k} \left\{ \sum_{i=j}^{k} (1+n_i) + d(\tau_j) \right\}$.*

4. *$|\mathbf{D_k}| \leq d(2k)!$ where d is a fixed constant.*

Proof: For the proof of the size of $\mathbf{D_k}$ see [AM98].

By induction on k:

$k = 1$ then

1. $\forall \tau_1$ type in EAL, $\forall n_0 \geq 0$ and $\forall n_1 \geq 2(1-1)$, D_1 has type $\Delta_1 = !^{n_1}(L^{\tau_1}_{!^{n_0}\mathtt{B}})$. The proof is shown in Figure 2.16.
2. Looking at Figure 2.16 is easy to see that the depth of $\mathbf{D_1}$ is $1+n_0+n_1$.
3. Again, looking at Figure 2.16, the deepest type is $!^{n_1}(L^{\tau}_{!^{n_0}\mathtt{B}}) = \Delta_1$ and

$$d(!^{n_1} L^{\tau_1}_{!^{n_0}\mathtt{B}}) = n_1 + 1 + \max\{d(\tau_1), n_0 + 1\}.$$

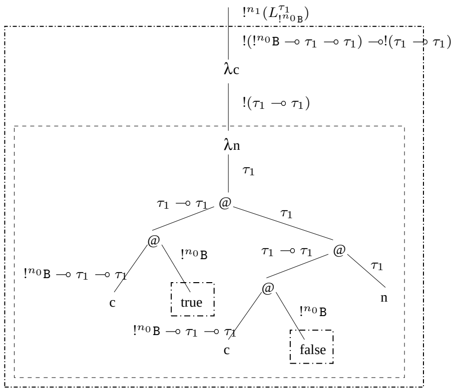


Figure 2.16: EAL type of $\mathbf{D_1}$

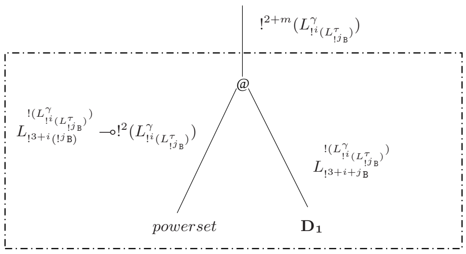


Figure 2.17: EAL type of $\mathbf{D_2}$

$k \geq 2$ 1. • if $k = 2$ by Lemma 30, *powerset* has type $L_{!^{3+i}\alpha}^{!(L^{\gamma}_{!^i L^{\tau}_{\alpha}})} \multimap !!(L^{\gamma}_{!^i L^{\tau}_{\alpha}})$ for any type α, γ and τ and for $i \geq 0$, and, in particular, when $\alpha = !^j\mathtt{B}, j \geq 0$, *powerset* has type:

$$L_{!^{3+i}(!^j\mathtt{B})}^{!(L^{\gamma}_{!^i L^{\tau}_{!^j\mathtt{B}}})} \multimap !!(L^{\gamma}_{!^i L^{\tau}_{!^j\mathtt{B}}})$$

By the previous point $\mathbf{D_1}$ has type $!^{n_1}(L^{\tau_1}_{!^{n_0}\mathtt{B}})$ $\forall n_0, n_1 \geq 0$ and for any type τ_1. Hence, in particular, for $n_1 = 0, n_0 = 3 + i + j$ and $\tau_1 = !(L^{\gamma}_{!^i(L^{\tau}_{!^j\mathtt{B}})})$, $\mathbf{D_1}$ has type

$$L_{!^{3+i+j}\mathtt{B}}^{!(L^{\gamma}_{!^i L^{\tau}_{!^j\mathtt{B}}})}$$

Hence $\mathbf{D_2} = (powerset\ \mathbf{D_1})$, $\forall m \geq 0$ has type

$$!^{2+m}(L^{\gamma}_{!^i L^{\tau}_{!^j\mathtt{B}}})$$

see Figure 2.17 where the bold box represents m "normal" boxes. Then $\forall \tau_1, \tau_2, \forall n_0, n_1 \geq 0, \forall n_2 \geq 2, \mathbf{D_2}$ has type

$$\Delta_2 = !^{n_2}(L^{\tau_2}_{!^{n_1}(L^{\tau_1}_{!^{n_0}\mathtt{B}})}).$$

• if $k > 2$, $\mathbf{D_k} = (powerset\ \mathbf{D_{k-1}})$ by definition.
By inductive hypothesis $\forall \tau_1, \ldots, \tau_{k-1}$ types in EAL, $\forall n_0, \ldots, n_{k-2} \geq$

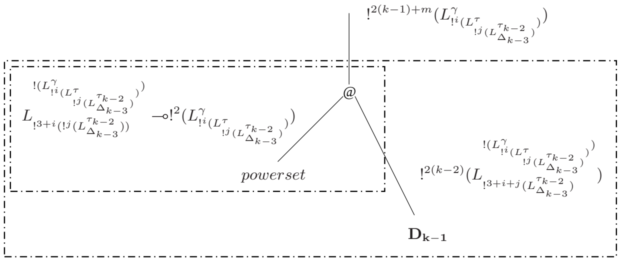


Figure 2.18: EAL type of $\mathbf{D_k}$

0, $\forall n_{k-1} \geq 2(k-2)$, $\mathbf{D_{k-1}}$ has type $\Delta_{k-1} = !^{n_{k-1}}(L^{\tau_{k-1}}_{!^{n_{k-2}}(L^{\tau_{k-2}}_{\Delta_{k-3}})})$.
Similar to the previous case, for $\alpha = !^j(L^{\tau_{k-2}}_{\Delta_{k-3}})$, $n_{k-1} = 2(k-2)$, $n_{k-2} = 3+i+j$ and $\tau_{k-1} = !(L^{\gamma}_{!^i(L^{\tau}_{!^j(L^{\tau_{k-2}}_{\Delta_{k-3}})})})$ one has $\forall m \geq 0$, $\mathbf{D_k}$ of type

$$!^{2(k-1)+m}(L^{\gamma}_{!^i(L^{\tau}_{!^j(L^{\tau_{k-2}}_{\Delta_{k-3}})})})$$

(see Figure 2.18, where the inner bold box represents $2(k-2)$ normal boxes and the outer one represents m normal boxes). Hence $\forall \tau_1, \ldots, \tau_k, \forall n_0, \ldots, n_{k-1} \geq 0, \forall n_k \geq 2(k-1), \mathbf{D_k}$ has type

$$\Delta_k = !^{n_k}(L^{\tau_k}_{!^{n_{k-1}}(L^{\tau_{k-1}}_{!^{n_{k-2}}(L^{\tau_{k-2}}_{\Delta_{k-3}})})}).$$

2. For $k \geq 2$ one has:

$$\begin{aligned}\mathbf{D_k} = (&\mathit{powerset}_{k-1}^{L^{\cdots}_{!^{3+n_{k-1}}(!^{n_{k-2}}(L^{\cdots}_{\Delta_{k-3}}))}} \\ &(\mathit{powerset}_{k-2}^{L^{\cdots}_{!^{3+(3+n_{k-1}+n_{k-2})}(!^{n_{k-3}}(L^{\cdots}_{\Delta_{k-4}}))}} \\ &\qquad\cdots \\ &(\mathit{powerset}_1^{L^{\cdots}_{!^{3+(3+\cdots(3+n_{k-1}+n_{k-2})\cdots+n_1)}(!^{n_0}\mathtt{B})}} \\ &\qquad \mathbf{D_1}^{L^{\cdots}_{!^{3+(3+\cdots(3+n_{k-1}+n_{k-2})\cdots+n_1)+n_0}\mathtt{B}}})\cdots))\end{aligned}$$

where the subscripts at every *powerset* simply distinguish various instances of the same term (with different types).
Looking at Figure 2.18, every application of $\mathit{powerset}_j$ in the construction above is inside $2(j-1)$ boxes, hence depth of $\mathbf{D_k}$ is

$$\max\left\{\begin{array}{l} d(\mathbf{D_1} : L^{\cdots}_{!^{3(k-1)+\sum_{i=0}^{k-1} n_i}\mathtt{B}}), \\ \max_{1\leq j\leq k-1}\{d(\mathit{powerset}_j : L^{\cdots}_{!^{3(k-j)+\sum_{i=j-1}^{k-1} n_i}\cdots}) + 2(j-1)\}\end{array}\right\}+m$$

but

$$d(powerset_j : L^{\cdots}_{!^{3(k-j)+\sum_{i=j-1}^{k-1} n_i}\cdots}) + 2(j-1) \\ > \\ d(powerset_{j+1} : L^{\cdots}_{!^{3(k-j-1)+\sum_{i=j}^{k-1} n_i}\cdots}) + 2j$$

indeed

$$1 + 3(k-j) + \sum_{i=j-1}^{k-1} n_i + 2(j-1) > 1 + 3(k-j-1) + \sum_{i=j}^{k-1} n_i + 2j \\ \Updownarrow \\ -1 + n_{j-1} > -2$$

and by hypothesis every n_i is greater than zero. Hence depth of $\mathbf{D_k}$ is

$$\begin{aligned} &\max \left\{ \begin{array}{l} d(\mathbf{D_1} : L^{\cdots}_{!^{3(k-1)+\sum_{i=0}^{k-1} n_i}\mathtt{B}}), \\ d(powerset_1 : L^{\cdots}_{!^{3(k-1)+\sum_{i=0}^{k-1} n_i}\cdots}) \end{array} \right\} + m \\ &= \max \left\{ \begin{array}{l} 1 + 3(k-1) + \sum_{i=0}^{k-1} n_i, \\ 1 + 3(k-1) + \sum_{i=0}^{k-1} n_i \end{array} \right\} + m \\ &= k + \sum_{i=0}^{k} n_i. \end{aligned}$$

3. for $\mathbf{D_k} : !^{2(k-1)+m} L^{\tau_k}_{\Delta_{k-1}}$, looking at Figure 2.18, we prove first that the deepest type is $!^{2(k-1)+m} L^{\tau_k}_{\Delta_{k-1}}$, indeed

$$\begin{aligned} &d(!^{2(k-1)} L^{\tau_k}_{!^{n_{k-1}} L^{\tau_{k-1}}_{!^{n_{k-2}} L^{\tau_{k-2}}_{\Delta_{k-3}}}}) = \\ &= 2(k-1) + 1 + \\ &\quad + \max \left\{ \begin{array}{c} d(\tau_k), \\ n_{k-1} + 1 + \max \left\{ \begin{array}{c} n_{k-2} + 1 + \max\{d(\tau_{k-2}), d(\Delta_{k-3})\}, \\ d(\tau_{k-1}) \end{array} \right\} \end{array} \right\} \\ &= 2(k-2) + 1 + \max \left\{ \begin{array}{c} 1 + 1 + \max \left\{ \begin{array}{c} d(\tau_k), \\ d(\Delta_{k-1}) \end{array} \right\} \\ 3 + n_{k-1} + n_{k-2} + 1 + \max \left\{ \begin{array}{c} d(\tau_{k-2}), \\ d(\Delta_{k-3}) \end{array} \right\} \end{array} \right\} \\ &= d(!^{2(k-2)} L^{!L^{\tau_k}_{\Delta_{k-1}}}_{!^{3+n_{k-1}+n_{k-2}} L^{\tau_{k-2}}_{\Delta_{k-3}}}) \end{aligned}$$

Then we have depth of type of $\mathbf{D_{k-i}}$ equals to depth of type of $\mathbf{D_{k-i-1}}$ in the derivation of $\mathbf{D_k}$. Moreover depth of type of $\mathbf{D_{k-i}}$ is greater than depth of type of $powerset_{k-i}$, hence the deepest type is the type of $\mathbf{D_k}$. Finally, assuming $\tau_0 = o$, we prove:

$$d(\Delta_k) = \max_{0 \le j \le k} \left\{ \sum_{i=j}^{k} (1 + n_i) + d(\tau_j) \right\}$$

By induction on k:
$d(\Delta_0) = d(!^{n_0}\mathtt{B}) = n_0 + 1$;

$$
\begin{aligned}
d(\Delta_k) &= 1 + n_k + \max\{d(\tau_k), d(\Delta_{k-1})\}\\
&\overset{\text{IH}}{=} \max\left\{1 + n_k + d(\tau_k), 1 + n_k + \max_{0\le j\le k-1}\left\{\sum_{i=j}^{k-1}(1+n_i) + d(\tau_j)\right\}\right\}\\
&= \max_{0\le j\le k}\left\{\sum_{i=j}^{k}(1+n_i) + d(\tau_j)\right\}.
\end{aligned}
$$

□

Prime formulas are encoded by the following terms.

$$
\begin{aligned}
eq_1 &= \mathtt{IFF} \qquad \text{with } n = 1\\
member_k &= \lambda x^{k-1}.\lambda y^k.(y^k\ \lambda y^{k-1}.(\mathtt{OR}\\
&\qquad (eq_{k-1}\ x^{k-1}\ y^{k-1}))\ \mathtt{false})\\
eq_k &= \lambda x^k.\lambda y^k.(\lambda op.(\mathbf{D_{k-1}}\ \lambda z^{k-1}.(\mathtt{AND}\\
&\qquad (\mathtt{IFF}\ (op\ z^{k-1}\ x^k)\ (op\ z^{k-1}\ y^k)))\ \mathtt{true})\\
&\qquad\quad member_k)
\end{aligned}
$$

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

Lemma 32 *Let*

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

Then

1. *$member_k$ has EAL-type*

$$!^{m+\max\{2k-5,2\}}M_{k-1} \multimap !^{m+\max\{2k-7,0\}}M_k \multimap !^{m+\max\{4k-9,2(k-1)\}}\mathtt{B}$$

   for any $m \geq 0$.

2. $\exists c \geq 0.\exists d \geq 0.\quad |member_k| \leq c * k^2 + d(2(k-1))!$.
3. $d(member_k : !^{m+\max\{2k-5,2\}}M_{k-1} \multimap \cdots) = m + 2(k-1) + \max\{2k-7, 0\}$.
4. *The deepest type in $member_k$ is $!^{m+\max\{4k-9,2(k-1)\}}\mathtt{B}$ of depth* $\max\{4k-8, 2k-1\}$.

We will prove Lemma 32 after a detour regarding some *open* terms whose instances yield $member_k$ and eq_k. For $k \geq 2$, define

$$
\begin{aligned}
eq'_1 &= \mathtt{IFF}\\
member'_k &= \lambda x^{k-1}.\lambda y^k.(y^k\ \lambda y^{k-1}.(\mathtt{OR}\ (eq'_{k-1}\ x^{k-1}\ y^{k-1}))\ \mathtt{false})\\
eq'_k &= \lambda x^k.\lambda y^k.(\lambda op.(d_{k-1}\ \lambda z^{k-1}.(\mathtt{AND}\ (\mathtt{IFF}\ (op\ z^{k-1}\ x^k)\\
&\qquad (op\ z^{k-1}\ y^k)))\ \mathtt{true})\ member'_k)
\end{aligned}
$$

Note that $member'_k$, for $k \geq 3$ and eq'_k, for $k \geq 2$, are not closed terms, due to the presence of the unbound variable d_{k-1} in the definition of eq'_k. In particular, for $k \geq 3$, $FV(member'_k) = \{d_1, \ldots, d_{k-2}\}$ and, for $k \geq 2$, $FV(eq'_k) = \{d_1, \ldots, d_{k-1}\}$.

Moreover notice that:

$$\begin{array}{rcl} member_k & = & member'_k[\mathbf{D_1}/d_1, \ldots, \mathbf{D_{k-2}}/d_{k-2}] \\ eq_k & = & eq'_k[\mathbf{D_1}/d_1, \ldots, \mathbf{D_{k-1}}/d_{k-1}]. \end{array}$$

Definition 25 *We define types $E^k = E^k_1 \multimap E^k_1 \multimap E^k_3$ of eq'_k, for $k \geq 1$, and $M^k = M^k_1 \multimap M^k_2 \multimap M^k_3$ of $member'_k$, for $k \geq 2$ as follows:*

$$\begin{array}{rclcrclcrcl} E^1_1 & = & !\mathtt{B} & & & & & & E^1_3 & = & !\mathtt{B} \\ M^k_1 & = & !E^{k-1}_1 & & M^k_2 & = & L^{E^{k-1}_3}_{E^{k-1}_1} & & M^k_3 & = & !E^{k-1}_3 \\ E^k_1 & = & !M^k_2 & & & & & & E^k_3 & = & !M^k_3 \end{array}$$

Hence

$$\begin{array}{rcl} E^1 & = & !\mathtt{B} \multimap !\mathtt{B} \multimap !\mathtt{B} \\ M^k & = & !E^{k-1}_1 \multimap L^{E^{k-1}_3}_{E^{k-1}_1} \multimap !E^{k-1}_3 \\ E^k & = & !M^k_2 \multimap !M^k_2 \multimap !M^k_3 \end{array}$$

Lemma 33 $\forall n \geq 0\ \forall k \geq 1\ \exists n_0, \ldots, n_k \geq 0\ \exists \tau_1, \ldots, \tau_k$ *s.t.* $\Delta_k = !^n(L^{E^k_3}_{E^k_1})$

Lemma 34 $\forall k \geq 1\ \exists n_0, \ldots, n_{k-1} \geq 0\ \exists \tau_1, \ldots, \tau_k$ *such that* $\forall n_k \geq 0\ \Delta_k = !^{n_k}(L^{M^{k+1}_3}_{M^{k+1}_1})$.

Lemma 35 *1. For any $k \geq 2$, $member'_k$ has type M^k in EAL, with $FV(member'_k) = \{d_1, \ldots, d_{k-2}\}$, d_i of type $!^{2(k-2-i)+1}(L^{M^{i+1}_3}_{M^{i+1}_1})$; moreover $d(member'_k : M^k) = 2(k-1)$;*

2. for any $k \geq 1$, eq'_k has type E^k in EAL, with $FV(eq'_k) = \{d_1, \ldots, d_{k-1}\}$, d_i of type $!^{2(k-1-i)}(L^{M^{i+1}_3}_{M^{i+1}_1})$ for $k \geq 1$; moreover $d(eq'_k : E^k) = 2(k-1)+1$.

Proof: By mutual induction on k.
First of all, note that $\forall k \geq 2 \quad M^k_3 \equiv !^{2(k-1)}\mathtt{B}$ and $\forall k \geq 1 \quad E^k_3 \equiv !^{2(k-1)+1}\mathtt{B}$.

The proof for $member'_k$ is shown in Figure 2.19, where `OR` has type

$$!^{2(k-2)+1}\mathtt{B} \multimap !^{2(k-2)+1}\mathtt{B} \multimap !^{2(k-2)+1}\mathtt{B}$$

for the observation above, and the bold box represents $2(k-2)+1$ normal boxes. Note that eq'_{k-1} has free variables $d_1 : !^{2(k-3)}(L^{M^2_3}_{M^2_1}), \ldots, d_{k-2} : L^{M^{k-1}_3}_{M^{k-1}_1}$ and it is inside a box; hence $member'_k$ has the same free variables with the same types, but with one more !, that is $d_1 : !^{2(k-3)+1}(L^{M^2_3}_{M^2_1}), \ldots, d_{k-2} : !(L^{M^{k-1}_3}_{M^{k-1}_1})$. The proof for eq'_k is shown in Figure 2.20. Note that $member'_k$ has free variables $d_1 : !^{2(k-3)+1}(L^{M^2_3}_{M^2_1}), \ldots, d_{k-2} : !(L^{M^{k-1}_3}_{M^{k-1}_1})$ and it is inside a box therefore, the free

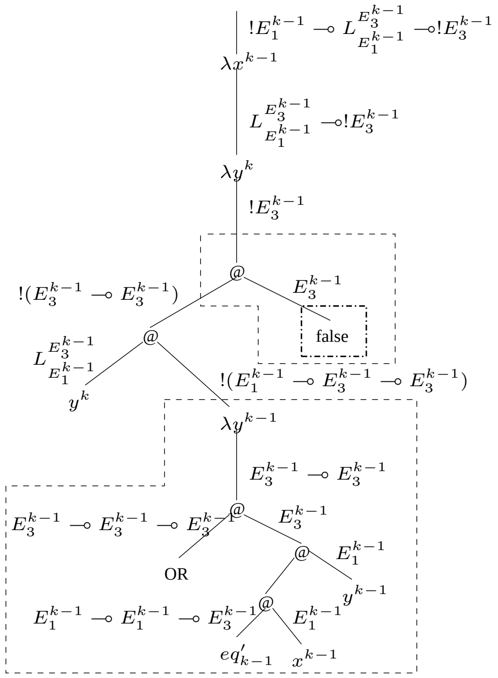


Figure 2.19: EAL type of $member'_k$

variables of eq'_k are $d_{k-1} : L_{M_1^k}^{M_3^k}$ and the variables of $member'_k$ with the same type with one more !, that is $d_1 :!^{2(k-2)}(L_{M_1^2}^{M_3^2}), \ldots, d_{k-1} : L_{M_1^k}^{M_3^k}$.

Looking at figures 2.20 and 2.19,

$$d(eq'_1 : E^1) = 1$$

$$\begin{aligned} d(member'_k : M^k) &= \max\{d(\texttt{false} : E_3^{k-1}), d(\texttt{OR}{:}E_3^{k-1}{\multimap}\cdots), d(eq'_{k-1}{:}E^{k-1})\} + 1 \\ &= \max\{2(k-2)+1, d(eq'_{k-1} : E^{k-1})\} + 1 \\ &\overset{\text{IH}}{=} \max\{2(k-2)+1, 2(k-2)+1\} + 1 \ = \ 2(k-1) \end{aligned}$$

$$\begin{aligned} d(eq'_k : E^k) &= \max\{2(k-1), d(member'_k : M^k)\} + 1 \\ &\overset{\text{IH}}{=} \max\{2(k-1), 2(k-1)\} + 1 \\ &= 2(k-1)+1. \end{aligned}$$

□

Lemma 36 $\forall k \geq 2 \quad \forall n \geq \max\{2k-7, 0\}$ *member*$_k$ *has type* $!^n M_1^k \multimap !^n M_2^k \multimap !^n M_3^k$ *in EAL.*

Proof: By the previous lemma $member'_k$ has type M^k in EAL, with free variables $d_1 :!^{2(k-3)+1}(L_{M_1^2}^{M_3^2}), \ldots, d_{k-2} :!(L_{M_1^{k-1}}^{M_3^{k-1}})$, and by Lemma 31, $\forall \tau_1, \ldots, \tau_j$ types

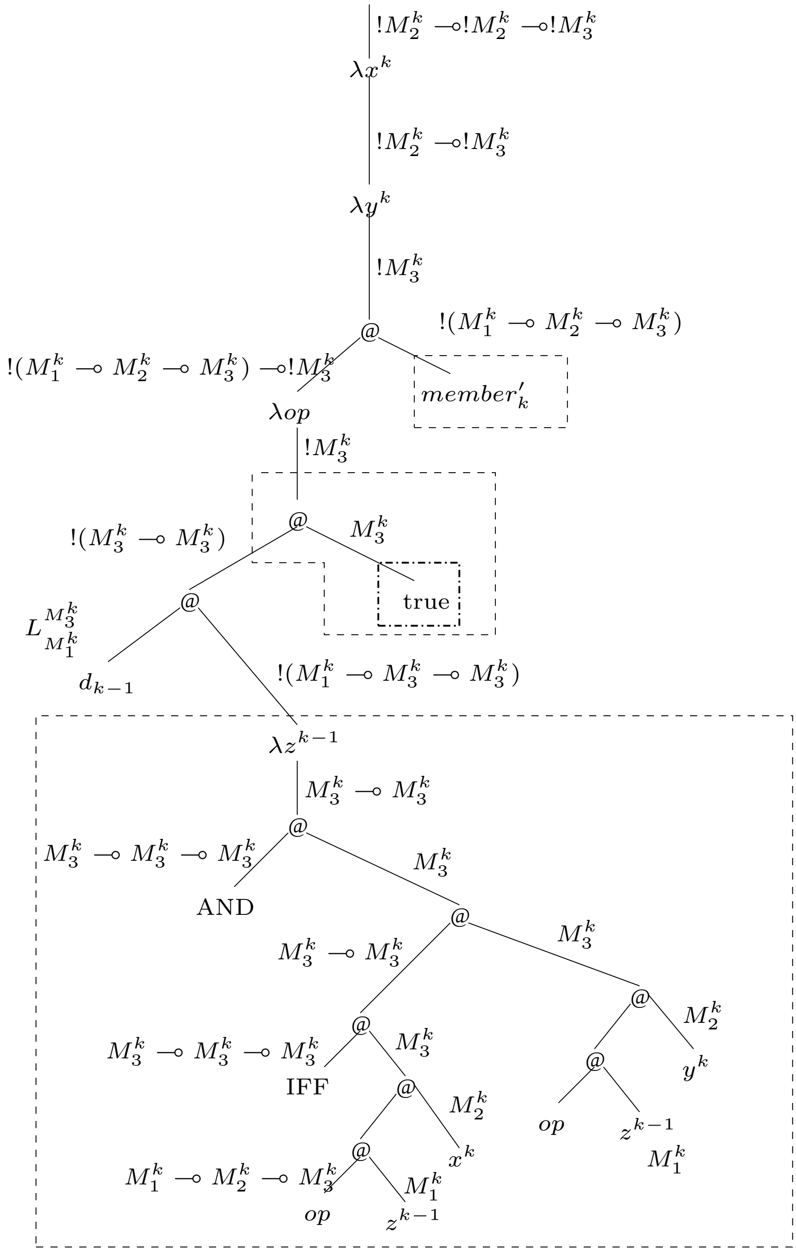


Figure 2.20: EAL type of eq'_k

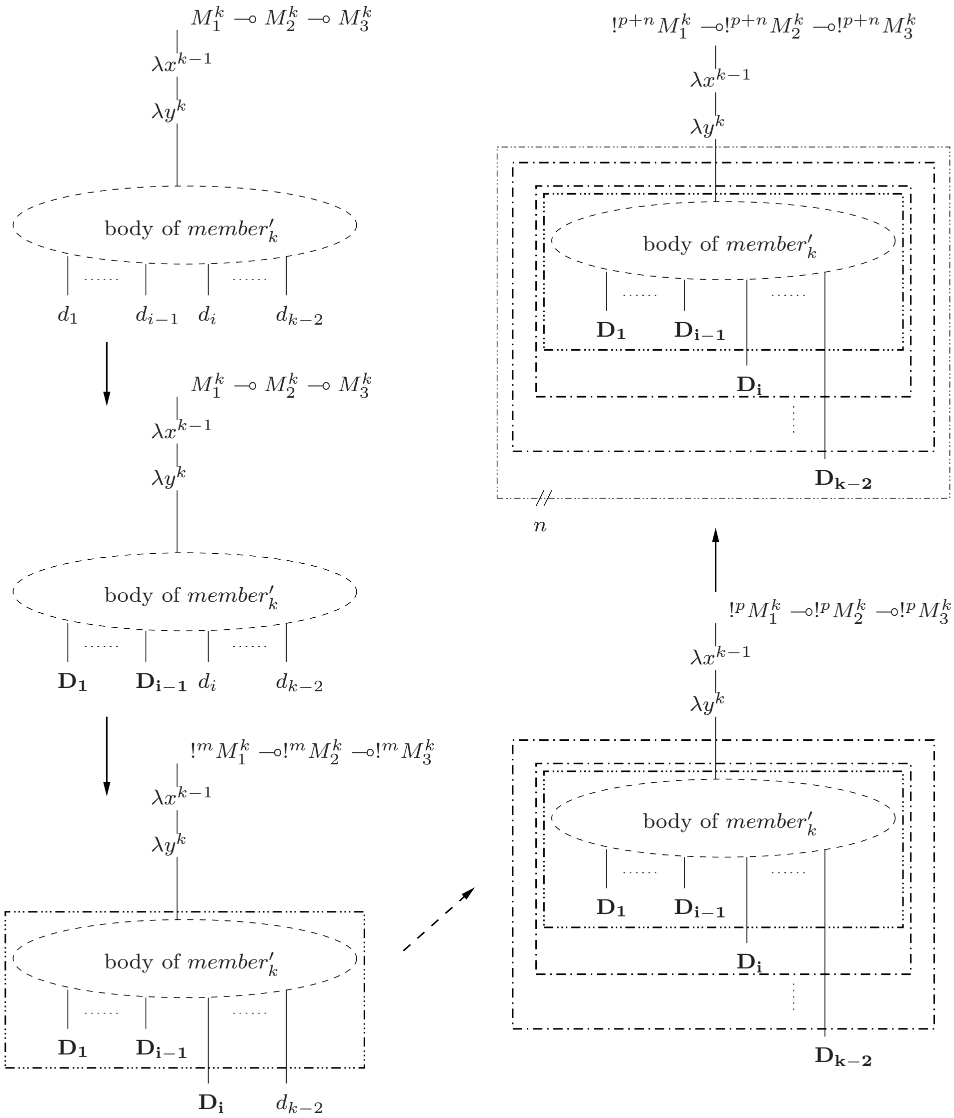


Figure 2.21: EAL type of $member_k$

in EAL, $\forall n_0, \ldots, n_{j-1} \geq 0$, $\forall n_j \geq 2(j-1)$, $\mathbf{D_j}$ has type Δ_j, in particular, by Lemma 34, $\mathbf{D_j}$ has type $!^{\max\{2(j-1), 2(k-2-j)+1\}}(L^{M_3^{j+1}}_{M_1^{j+1}})$. The proof is shown in Figure 2.21, where

1. $i = \lceil \frac{k}{2} \rceil$ is the lowest i s.t. $2(i-1) > 2(k-2-i)+1$,
2. $$\begin{array}{rcl} \mathbf{D_1} & : & !^{2(k-3)+1}(L^{M_3^2}_{M_1^2}) \\ & & \vdots \\ \mathbf{D_{i-1}} & : & !^{2(k-2-i+1)+1}(L^{M_3^i}_{M_1^i}) \\ \mathbf{D_i} & : & !^{2(i-1)}(L^{M_3^{i+1}}_{M_1^{i+1}}) \\ & & \vdots \\ \mathbf{D_{k-2}} & : & !^{2(k-3)}(L^{M_3^{k-1}}_{M_1^{k-1}}) \end{array}$$
3. the bold box inserted at step three represent $m = \begin{cases} 1 & k \text{ even} \\ 3 & k \text{ odd} \end{cases}$ boxes,
4. $p = \max\{2k-7, 0\}$.

The other bold boxes represent four normal boxes. $\square$

Proof: [of Lemma 32]

1. In view of Lemma 36, we need to prove that $\forall k \geq 2$, $\forall n \geq \max\{2k-7, 0\}$, $\exists m \geq 0$, such that

   $$\begin{aligned} &!^n M_1^k \multimap !^n M_2^k \multimap !^n M_3^k = \\ &\quad = !^{m+\max\{2k-5,2\}} M_{k-1} \multimap !^{m+\max\{2k-7,0\}} M_k \multimap !^{m+\max\{4k-9,2(k-1)\}} \mathtt{B}. \end{aligned}$$

   and vice versa $\forall m \geq 0$, $\exists n \geq \max\{2k-7, 0\}$ such that the same equivalence holds.
   (a) $!^n M_3^k = !^{m+\max\{2k-7,0\}+2(k-1)} \mathtt{B} = !^{m+\max\{4k-9,2(k-1)\}} \mathtt{B}$;
   (b) by induction on $k \geq 2$ we prove $M_2^k = M_k$:
   $(k=2)$ $M_2^2 = L^{!\mathtt{B}}_{!\mathtt{B}} = L^{!^{4-3}\mathtt{B}}_{!M_1} = M_2$;
   $(k>2)$ $M_2^k = L^{!^{2(k-2)+1}\mathtt{B}}_{!M_2^{k-1}} \stackrel{\text{IH}}{=} L^{!^{2k-3}\mathtt{B}}_{!M_{k-1}} = M_k$.
   Hence trivially $!^n M_2^k = !^{m+\max\{2k-7,0\}} M_k$;
   (c) $!^{m+\max\{2k-7,0\}} M_1^k = !^{m+\max\{2k-7,0\}} !! M_2^{k-1} = !^{m+\max\{2k-5,2\}} M_{k-1}$.
2. First notice the following:

   $$\begin{aligned} (E_1^1)^* &= (\Delta_0)^* = (\mathtt{B})^* \\ (E_1^k)^* &= (L^{\mathtt{B}}_{E_1^{k-1}})^* = (L^{\mathtt{B}}_{\Delta_{k-2}})^* = (\Delta^{\mathtt{B}}_{k-1})^* \\ (M_1^2)^* &= (\Delta_0)^* = (\mathtt{B})^* \\ (M_1^k)^* &= (L^{\mathtt{B}}_{E_1^{k-2}})^* = (L^{\mathtt{B}}_{\Delta_{k-3}})^* = (\Delta^{\mathtt{B}}_{k-2})^* \end{aligned}$$

and

$$\begin{aligned}
||(\Delta_k^\tau)^*|| &= k(4 + 4||(\tau)^*||) + 5\\
||(\mathtt{B})^*|| &= ||(\Delta_0^o)^*|| = ||(\Delta_0)^*|| = 5\\
||(\Delta_k^{\mathtt{B}})^*|| &= 24k + 5 \le 24(k+1)
\end{aligned}$$

hence

$$\begin{aligned}
|member_k| &= 8 + |\mathtt{OR}| + |eq_{k-1}| + |\mathtt{false}|\\
&\qquad + |y^k| + |x^{k-1}| + |y^{k-1}|\\
&= 32 + |eq_{k-1}| + ||(\Delta_{k-1}^{\mathtt{B}})^*|| + 2||(\Delta_{k-2}^{\mathtt{B}})^*||\\
&= 32 + 24(k-1) + 5 + 48(k-2) + 10 + |eq_{k-1}|\\
&= 72k - 73 + |eq_{k-1}|
\end{aligned}$$

$$\begin{aligned}
|eq_k| &= 14 + |\mathtt{AND}| + |\mathtt{IFF}| + |\mathtt{true}| + |\mathbf{D_{k-1}}|\\
&\qquad + |member_k| + |op| + 2|z^{k-1}| + |x^k| + |y^k|\\
&= 14 + 21 + 29 + 3 + |\mathbf{D_{k-1}}| + |member_k|\\
&\qquad + ||(\Delta_{k-2}^{\mathtt{B}} \multimap \Delta_{k-1}^{\mathtt{B}} \multimap \mathtt{B})^*|| + 2||(\Delta_{k-2}^{\mathtt{B}})^*|| + 2||(\Delta_{k-1}^{\mathtt{B}})^*||\\
&= 144k - 112 + |\mathbf{D_{k-1}}| + |member_k|\\
&\le 144k - 112 + d(2(k-1))! + |member_k|
\end{aligned}$$

then

$$\begin{aligned}
|member_2| &= 72 * 2 - 73 + 29 = 100\\
|member_k| &\le c_k * k + d(2(k-2))! + |member_{k-1}|\\
&\le \sum_{i=3}^{k} c_i i + \sum_{i=1}^{k-2} d(2i)! \le c * k^2 + d(2(k-1))!
\end{aligned}$$

3. Looking at Figure 2.21, by Lemma 36 we have depth

$$\max \left\{ \begin{array}{l} d(member'_k : M^k) + \max\{2k-7, 0\},\\ \max_{1 \le j < \lceil k/2 \rceil}\{d(\mathbf{D_j} :!^{2(k-2-j)+1} L^{\cdots}_{M_1^{j+1}})\} + \max\{2k-7, 0\}\\ \max_{\lceil k/2 \rceil \le j \le k-2}\{d(\mathbf{D_j} :!^{2(j-1)} L^{\cdots}_{M_1^{j+1}}) + 4(k-2-j)\} \end{array} \right\} + m$$

First we need to show that for all $k \ge 1$ $E_1^k = \Delta_{k-1}$ with $n_{k-1} = \cdots = n_0 = 1$ (we use notation $\Delta_{k-1}^{(1,\dots)}$) and for all $k \ge 2$ $M_1^k = \Delta_{k-2}$ with $n_{k-2} = 2$ and $n_{k-3} = \cdots = n_0 = 1$ (we use $\Delta_{k-2}^{(2,1,\dots)}$). By induction on k:

$$\begin{aligned}
E_1^1 &= !\mathtt{B} = \Delta_0^{(1,\dots)}\\
E_1^k &= !L_{E_1^{k-1}} \overset{\text{IH}}{=} !L_{\Delta_{k-2}^{(1,\dots)}} = \Delta_{k-1}^{(1,\dots)}
\end{aligned}$$

then, using the result above:

$$\begin{aligned}
M_1^2 &= !E_1^1 = !^2\mathtt{B} = \Delta_0^{(2)}\\
M_1^k &= !E_1^{k-1} = !^2 M_2^{k-1} = !^2 L_{E_1^{k-2}} = !^2 L_{\Delta_{k-3}^{(1,\dots)}} = \Delta_{k-2}^{(2,1,\dots)}.
\end{aligned}$$

therefore, in the calculus of $d(\mathbf{D_j})$ for $d(member_k)$, we have $\sum_{\ell=0}^{j-1} n_\ell = j+1$. Hence

$$d(member_k :!^{\max\{2k-5,2\}+n} M_{k-1} \multimap \cdots) =$$
$$= \max \left\{ \begin{array}{l} d(member'_k : M^k) + \max\{2k-7,0\}, \\ \max_{1 \leq j < \lceil k/2 \rceil} \{d(\mathbf{D_j} :!^{2(k-2-j)+1} L^{\cdots}_{\Delta^{(2,1,\ldots)}_{j-1}})\} + \max\{2k-7,0\} \\ \max_{\lceil k/2 \rceil \leq j \leq k-2} \{d(\mathbf{D_j} :!^{2(j-1)} L^{\cdots}_{\Delta^{(2,1,\ldots)}_{j-1}}) + 4(k-2-j)\} \end{array} \right\} + m$$
$$= m + 2(k-1) + \max\{2k-7,0\}$$

4. In the derivation of $member_k$ the deepest type is the deeper between the type of $member'_k$ and the types of the various $\mathbf{D_i}$s.
   First notice that $M_k = M_2^k = \Delta_{k-1}$ with $n_{k-1} = 0$ and $n_{k-2} = \cdots = n_0 = 1$ and $\tau_{k_i} =!^{2k_i-3}\mathtt{B}$, then

$$d(M_k) = d(M_2^k) = d(\Delta_{k-1}^{\substack{(0,1,\ldots) \\ \tau_{k_i} =!^{2k_i-3}\mathtt{B}}}) =$$
$$= \max_{0 \leq j \leq k-1} \left\{ \sum_{i=j}^{k-1} (1+n_i) + d(\tau_j) \right\} = 2k-1.$$

   Hence depth of type of $member_k$ is:

$$d(!^{\max\{2k-5,2\}} M_{k-1} \multimap !^{\max\{2k-7,0\}} M_k \multimap !^{\max\{4k-9,2(k-1)\}}\mathtt{B}) =$$
$$= \max\{4k-8, 2k-1\} = d(!^{\max\{4k-9,2(k-1)\}}\mathtt{B})$$

   By Lemma 36, the $\mathbf{D_i}$s have type $!^{\max\{2(i-1),2(k-2-i)+1\}}(L^{M_3^{i+1}}_{M_1^{i+1}})$ for $1 \leq i \leq k-2$.

$$d(!^{\max\{2(i-1),2(k-2-i)+1\}}(L^{M_3^{i+1}}_{M_1^{i+1}})) = 2k-1.$$

   Hence the deepest type is $!^{\max\{4k-9,2(k-1)\}}\mathtt{B}$, of depth $\max\{4k-8, 2k-1\}$.

□

Putting together all the ingredients of the encoding, we obtain our main technical result.

Theorem 37 *Define* $f(k) = \left\{ \begin{array}{ll} 2k-1 & 1 \leq k \leq 3 \\ 4k-9 & k \geq 4 \end{array} \right.$.
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*

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

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

1. *$d(\Psi[x_1^{k_1},\dots,x_n^{k_n}] :!^{f(k_j)+d+\sum_{\ell\in J}2(k_\ell-1)}\mathtt{B}) = f(k_j)+d+\sum_{\ell\in J}2(k_\ell-1)$;*
2. *the deepest type in the EAL-derivation of $\Psi[x_1^{k_1},\dots,x_n^{k_n}]$ is the final one, of depth $\leq f(k_j)+d+\sum_{\ell\in J}2(k_\ell-1)+1$;*
3. *the size of $\Psi[x_1^{k_1},\dots,x_n^{k_n}]$ is $O(|\Phi|(2k_{\mathrm{MAX}})!)$ where k_{MAX} is the greatest k such that $member_k$ appears in $\Psi[x_1^{k_1},\dots,x_n^{k_n}]$.*

Proof: By induction on Ψ.

base : all base cases have no quantifiers, hence $m=d=0$ and $J=\emptyset$. Then the thesis becomes $\Psi :!^{f(k_j)}\mathtt{B}$ with free variables $x_i^{k_i} :!^{\max\{2k_i-7,1\}+f(k_j)-f(k_i)}M_{k_i}$ and

1. $d(\Psi)=f(k_j)$;
2. the deepest type is the final one of depth $\leq f(k_j)+1$;
3. size of Ψ is $O(|\Phi|(2k)!)$.

We show the proof for the general case $\Psi[x^{k-1},x^k]=(member_k\ x^{k-1}\ x^k)$, leaving the cases $\Psi[x^2]=(member_2\ \mathtt{true}\ x^2)$ and $\Psi[x^2]=(member_2\ \mathtt{false}\ x^2)$ to the reader.
By Lemma 32 $\Psi[x^{k-1},x^k] :!^{m_3}\mathtt{B}$ for every $m_3\geq\max\{4k-9,2(k-1)\}$. Now, $f(k)\geq\max\{4k-9,2(k-1)\}$ hence $\Psi[x^{k-1},x^k] :!^{f(k)}\mathtt{B}$.
Again by Lemma 32 $x^k :!^{m_2}M_k$ for any $m_2\geq\max\{2k-7,0\}$, and in particular $x^k :!^{\max\{2k-7,1\}+f(k)-f(k)}M_k$, and $x^{k-1} :!^{m_1}M_{k-1}$ for any $m_1\geq\max\{2k-5,2\}$, and in particular $\max\{2(k-1)-7,1\}+f(k)-f(k-1)\geq\max\{2k-5,2\}$.

1. Regarding the depth of the formula we have:

$$
\begin{aligned}
d(\Psi[x^{k-1},x^k] :!^{f(k)}\mathtt{B}) &= d(member_k :!^{\max\{2k-5,3\}}M_{k-1})\\
&= \left\{\begin{array}{ll} 2(k-1)+1 & k\leq 3\\ 4k-9 & k\geq 4\end{array}\right. = f(k)
\end{aligned}
$$

Than the thesis holds with $k_j=k$. Moreover,

2. considering the EAL-derivations, the deepest type is the type of $member_k$ of depth $\max\{4k-8,2k-1\}\leq f(k)+1$ by Lemma 36.
3. Finally, by Lemma 36, size of Ψ is $O(|\Phi|(2k)!)$.

inductive step :

if $\Psi[x_1^{k_1},\dots,x_n^{k_n}]=(\mathtt{NOT}\ \Psi'[x_1^{k_1},\dots,x_n^{k_n}])$ **then** the thesis holds by inductive hypothesis. Moreover

1. about depth we have:

$$
\begin{aligned}
&d(\Psi[x_1^{k_1}\dots x_n^{k_n}] :!^{f(k_j)+d+\sum_{\ell\in J}2(k_\ell-1)}\mathtt{B}) =\\
&\quad=\max\{d(\mathtt{NOT} :!^{f(k_j)+d+\sum_{\ell\in J}2(k_\ell-1)}\mathtt{B}), d(\Psi' :!^{f(k_j)+d+\sum_{\ell\in J}2(k_\ell-1)}\mathtt{B})\}\\
&\quad\stackrel{\mathrm{IH}}{=}\max\{f(k_j)+d+\sum_{\ell\in J}2(k_\ell-1), f(k_j)+d+\sum_{\ell\in J}2(k_\ell-1)\}\\
&\quad= f(k_j)+d+\sum_{\ell\in J}2(k_\ell-1)
\end{aligned}
$$

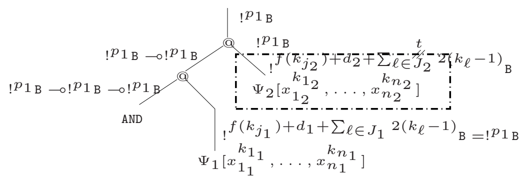


Figure 2.22: EAL type of an AND formula

2. It is easy to see that the deepest type is the final one, of depth

$$d(!^{f(k_j)+d+\sum_{\ell\in J}2(k_\ell-1)}\mathtt{B}) = f(k_j)+d+\sum_{\ell\in J}2(k_\ell-1)+1.$$

3. By inductive hypothesis $|\Psi'[x_1^{k_1},\ldots,x_n^{k_n}]|$ is $O(|\Phi'|(2k_{\text{MAX}})!)$ and by Lemma 29 $|\mathtt{NOT}| = 12$ hence the thesis holds.

if $\Psi[x_1^{k_1},\ldots,x_{i_2}^{k_{i_2}},\ldots,x_{i_1}^{k_{i_1}},\ldots,x_n^{k_n}] = (\mathtt{AND}\ \Psi_1[x_1^{k_1},\ldots,x_{i_1}^{k_{i_1}}]\ \Psi_2[x_{i_2}^{k_{i_2}},\ldots,x_n^{k_n}])$ then by inductive hypothesis $\exists J_1 \subseteq \{1,\ldots,i\}$, $\exists d_1 \leq m$, $\exists j_1 \in \{1,\ldots,n\}$ such that $\Psi_1 :!^{f(k_{j_1})+d_1+\sum_{\ell\in J_1}2(k_\ell-1)}\mathtt{B}$ and

$$x_h^{k_h} :!^{\max\{2k_h-7,1\}+f(k_{j_1})-f(k_h)+d_1+\sum_{\ell\in J_1}2(k_\ell-1)}M_{k_h}$$

for $1 \leq h \leq i_1$, and exists $J_2 \subseteq \{1,\ldots,i\}$, $\exists d_2 \leq m$, $\exists j_2 \in \{1,\ldots,n\}$ such that $\Psi_2 :!^{f(k_{j_2})+d_2+\sum_{\ell\in J_2}2(k_\ell-1)}\mathtt{B}$ and

$$x_h^{k_h} :!^{\max\{2k_h-7,1\}+f(k_{j_2})-f(k_h)+d_2+\sum_{\ell\in J_2}2(k_\ell-1)}M_{k_h}$$

for $i_2 \leq h \leq n$. Without loss of generality suppose

$$f(k_{j_1})+d_1+\sum_{\ell\in J_1}2(k_\ell-1) \geq f(k_{j_2})+d_2+\sum_{\ell\in J_2}2(k_\ell-1)$$

then we can construct

$$f(k_{j_1})+d_1+\sum_{\ell\in J_1}2(k_\ell-1) - (f(k_{j_2})+d_2+\sum_{\ell\in J_2}2(k_\ell-1))$$

boxes[4] around Ψ_2 (t boxes in Figure 2.22). Then

$$\Psi_2 :!^{\left(\begin{array}{c} f(k_{j_2})+d_2+\sum_{\ell\in J_2}2(k_\ell-1)+ \\ +f(k_{j_1})+d_1+\sum_{\ell\in J_1}2(k_\ell-1)- \\ -(f(k_{j_2})+d_2+\sum_{\ell\in J_2}2(k_\ell-1)) \end{array}\right)}\mathtt{B}$$

hence

$$\Psi_2 :!^{f(k_{j_1})+d_1+\sum_{\ell\in J_1}2(k_\ell-1)}\mathtt{B}$$

For free variables is analogous.

[4]Remember that "constructing n boxes around a term" is always possible in EAL, for the presence of the !-introduction rule.

1. As above, suppose

$$f(k_{j_1}) + d_1 + \sum_{\ell \in J_1} 2(k_\ell - 1) \geq f(k_{j_2}) + d_2 + \sum_{\ell \in J_2} 2(k_\ell - 1)$$

and that we need to add $f(k_{j_1})+d_1+\sum_{\ell \in J_1} 2(k_\ell-1)-(f(k_{j_2})+d_2+\sum_{\ell \in J_2} 2(k_\ell - 1))$ boxes around $\Psi_2[x_{i_2}^{k_{i_2}}, \ldots, x_n^{k_n}]$ as in Figure 2.22. Then

$$\begin{array}{l} d(\Psi[x_1^{k_1} \ldots x_n^{k_n}] :!^{f(k_{j_1})+d_1+\sum_{\ell \in J_1} 2(k_\ell-1)}\mathtt{B}) = \\ = \max \left\{ \begin{array}{l} d(\mathtt{AND} :!^{f(k_{j_1})+d_1+\sum_{\ell \in J_1} 2(k_\ell-1)}\mathtt{B}), \\ d(\Psi_1[x_1^{k_1}, \ldots, x_{i_1}^{k_{i_1}}] :!^{f(k_{j_1})+d_1+\sum_{\ell \in J_1} 2(k_\ell-1)}\mathtt{B}), \\ \left(\begin{array}{l} d(\Psi_2[x_{i_2}^{k_{i_2}}, \ldots, x_n^{k_n}] :!^{f(k_{j_2})+d_2+\sum_{\ell \in J_2} 2(k_\ell-1)}\mathtt{B}) + \\ \quad +f(k_{j_1}) + d_1 + \sum_{\ell \in J_1} 2(k_\ell - 1) - \\ \quad -(f(k_{j_2}) + d_2 + \sum_{\ell \in J_2} 2(k_\ell - 1)) \end{array} \right) \end{array} \right\} \\ = f(k_{j_1}) + d_1 + \sum_{\ell \in J_1} 2(k_\ell - 1) \end{array}$$

2. It is easy to see that the deepest type is the final one, of depth

$$d(!^{f(k_j)+d+\sum_{\ell \in J} 2(k_\ell-1)}\mathtt{B}) = f(k_j) + d + \sum_{\ell \in J} 2(k_\ell - 1) + 1.$$

3. By inductive hypothesis $|\Psi'[x_1^{k_1}, \ldots, x_n^{k_n}]|$ is $O(|\Phi'|(2k_{\text{MAX}})!)$ and by Lemma 29 $|\mathtt{AND}| = 21$ hence the thesis holds.

If $\Psi[x_1^{k_1}, \ldots, x_{i_2}^{k_{i_2}}, \ldots, x_{i_1}^{k_{i_1}}, \ldots, x_n^{k_n}] = (\mathtt{OR}\ \Psi_1[x_1^{k_1}, \ldots, x_{i_1}^{k_{i_1}}]\ \Psi_2[x_{i_2}^{k_{i_2}}, \ldots, x_n^{k_n}])$ as above.

If $\Psi[x_1^{k_1}, \ldots, x_n^{k_n}] = (\mathbf{D_{k_i}}\ \lambda y^{k_i}.(\mathtt{AND}\ (\lambda x^{k_i}.\Psi'[x_1^{k_1}, \ldots, x_n^{k_n}]\ y^{k_i}))\mathtt{true})$ **then** by inductive hypothesis Ψ' has m' quantifiers over $\mathcal{D}_1, \ldots, \mathcal{D}_{i'}$ and there exist

$$J' \subseteq \{1, \ldots, i'\} \ \wedge \ j' \in \{1, \ldots, n\} \ \wedge \ d' \leq m'$$

such that

$$\Psi' :!^{f(k_{j'})+d'+\sum_{\ell \in J'} 2(k_\ell-1)}\mathtt{B}$$

with free variables

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

Now Ψ has $m = m' + 1$ quantifiers over $\mathcal{D}_1, \ldots, \mathcal{D}_{i'}, \mathcal{D}_i$, then we can type Ψ in EAL adding $2(k_i - 1)$ boxes as in Figure 2.23 where the bold box represents $2(k_i - 1)$ boxes, $p_1 = f(k_j) + d' + \sum_{\ell \in J'} 2(k_\ell - 1)$ and $p_2 = \max\{2k_i - 7, 1\} + f(k_j) - f(k_i) + d' + \sum_{\ell \in J'} 2(k_\ell - 1)$. Hence the thesis holds with $J = J' \cup \{i\}, j = j'$ and $d = d' + 1$.

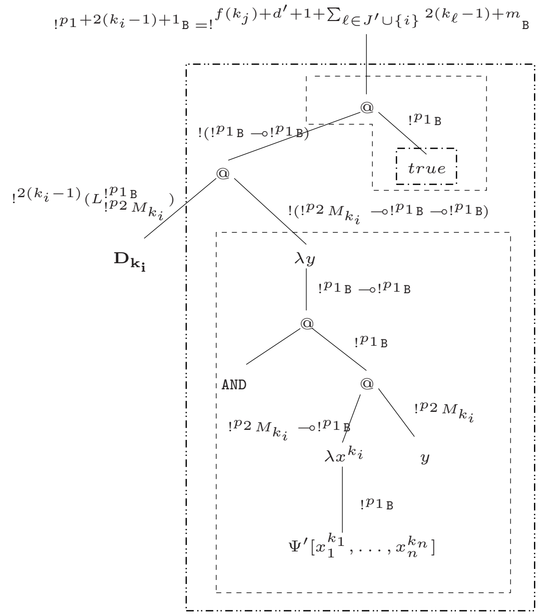


Figure 2.23: EAL type of an arbitrary formula

1. About depth we have the following:

$$
\begin{aligned}
&d(\Psi[x_1^{k_1}\ldots x_n^{k_n}] :!^{f(k_{j'})+(d'+1)+\sum_{\ell\in J'\cup\{h\}} 2(k_\ell-1)}\mathtt{B}) = \\
&= \max\left\{\begin{array}{l}
d(\mathtt{true} :!^{f(k_{j'})+d'+\sum_{\ell\in J'} 2(k_\ell-1)}\mathtt{B}) + 2(k_h-1)+1,\\
d(\mathtt{AND} :!^{f(k_{j'})+d'+\sum_{\ell\in J'} 2(k_\ell-1)}\mathtt{B}) + 2(k_h-1)+1,\\
d(\Psi'[x_1^{k_1}\ldots x_n^{k_n}] :!^{f(k_{j'})+d'+\sum_{\ell\in J'} 2(k_\ell-1)}\mathtt{B}) + 2(k_h-1)+1,\\
d(\mathbf{D_{k_h}} :!^{2(k_h-1)} L^{\cdots}_{!^{\max\{2k_h-7,1\}+f(k_{j'})-f(k_h)+d'+\sum_{\ell\in J'} 2(k_\ell-1)} M_{k_h}})
\end{array}\right\}\\
&= \max\left\{\begin{array}{l}
f(k_{j'}) + d' + \sum_{\ell\in J'} 2(k_\ell-1) + 2(k_h-1)+1,\\
f(k_{j'}) + d' + \sum_{\ell\in J'} 2(k_\ell-1) + 2(k_h-1)+1,\\
f(k_{j'}) + d' + \sum_{\ell\in J'} 2(k_\ell-1) + 2(k_h-1)+1,\\
1+3(k_h-1)+\sum_{\ell=0}^{k_h-1} n_\ell
\end{array}\right\}
\end{aligned}
$$

but

$$
\begin{aligned}
&!^{\max\{2k_h-7,1\}+f(k_{j'})-f(k_h)+d'+\sum_{\ell\in J'} 2(k_\ell-1)} M_{k_h} = \\
&\qquad = \Delta_{k_h-1}^{(\max\{2k_h-7,1\}+f(k_{j'})-f(k_h)+d'+\sum_{\ell\in J'} 2(k_\ell-1),1,\ldots)}
\end{aligned}
$$

hence

$$
\begin{aligned}
&= \max\left\{\begin{array}{l}
f(k_{j'}) + (d'+1) + \sum_{\ell\in J'\cup\{h\}} 2(k_\ell-1),\\
1+3(k_h-1)+\max\{2k_h-7,1\}+f(k_{j'})-\\
\qquad -f(k_h)+d'+\sum_{\ell\in J'} 2(k_\ell-1)+(k_h-1)
\end{array}\right\}\\
&= \max\left\{\begin{array}{l}
f(k_{j'}) + (d'+1) + \sum_{\ell\in J'\cup\{h\}} 2(k_\ell-1),\\
f(k_{j'}) + (d'+1) + \sum_{\ell\in J'\cup\{h\}} 2(k_\ell-1)+\\
\qquad +2(k_h-1)+\max\{2k_h-7,1\}-f(k_h)
\end{array}\right\}
\end{aligned}
$$

$$2(k_h-1)+\max\{2k_h-7,1\}-f(k_h)=0 \text{ then}$$

$$= f(k_{j'}) + (d'+1) + \sum_{\ell\in J'\cup\{h\}} 2(k_\ell-1)$$

2. Looking at Figure 2.23, the deepest type is again the final one, indeed

$$
\begin{aligned}
& d(!^{2(k_i-1)}(L^{!^{p_1}\mathtt{B}}_{!^{p_2}M_{k_i}})) = \\
&= 2(k_i-1)+1+\max\left\{\begin{array}{c} f(k_j)+d'+\sum_{\ell\in J'}2(k_\ell-1)+1, \\ \max\{2k_i-7,1\}+f(k_j)-f(k_i)+d'+ \\ +\sum_{\ell\in J'}2(k_\ell-1)+2k_i-1 \end{array}\right\} \\
&\qquad \text{but } f(k_i)=2(k_i-1)+\max\{2k_i-7,1\}, \text{ then} \\
&= 2(k_i-1)+1+\max\left\{\begin{array}{c} f(k_j)+d'+\sum_{\ell\in J'}2(k_\ell-1)+1 \\ f(k_j)+d'+\sum_{\ell\in J'}2(k_\ell-1)+1 \end{array}\right\} \\
&= f(k_j)+(d'+1)+\sum_{\ell\in J'\cup\{i\}}2(k_\ell-1)+1 \\
&= d(!^{f(k_j)+d+\sum_{\ell\in J}2(k_\ell-1)}\mathtt{B})
\end{aligned}
$$

3. An arbitrary formula Φ is built up from `AND`, `OR`, `true`, `false`, `IFF`, `NOT`, $\mathbf{D_i}$, $member_j$ all bounded by $O(d(2k_{\mathrm{MAX}})!)$, then the size of the term encoding Φ is $O(|\Phi|d(2k_{\mathrm{MAX}})!)$.

The case $\Psi[x_1^{k_1},\ldots,x_n^{k_n}] = (\mathbf{D_{k_i}}\ \lambda y^{k_i}.(\mathtt{OR}\ (\lambda x^{k_i}.\Psi'[x_1^{k_1},\ldots,x_n^{k_n}]\ y^{k_i}))\ \mathtt{false})$ is analogous. □

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.

Corollary 38 *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 Theorem 37 Ψ has type $!^t\mathtt{B}$ with $t = f(k_j)+m+\sum_{\ell\in J}2(k_\ell-1) \leq f(k_{\mathrm{MAX}})+n+2n(k_{\mathrm{MAX}}-1)$. □

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

Proof: It is sufficient to prove that for any EAL type σ the η-expansion $\eta(x)_\sigma$ is always typeable in EAL, as it is described in Figure 2.24. □

Theorem 40 *Let Ψ a term encoding an arbitrary formula. Then $d(\mathbf{or}(\Psi)) = O(n\cdot k_{\mathrm{MAX}})$, where n is the number of quantifiers in Ψ and k_{MAX} is the greatest k such that $member_k$ appears in Ψ.*

Proof: It is sufficient to investigate depth of types in the derivation of Ψ. Indeed, depth of $\mathbf{or}(\Psi)$ cannot exceed depth of Ψ plus the maximal depth of a type in the derivation of Ψ, because, in the worst case, we can η-expand the deepest variable of deepest type in Ψ and $d(\eta(x)_\sigma)$ is trivially $d(\sigma)$. □

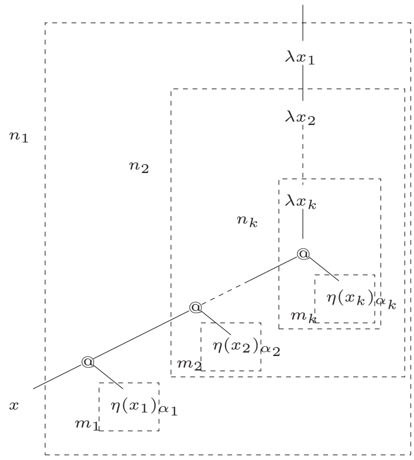


Figure 2.24: $\eta(x)_{!^{n_1}(!^{m_1}\alpha_1 \multimap !^{n_2}(!^{m_2}\alpha_2 \multimap \cdots !^{n_k}(!^{m_k}\alpha_k \multimap !^{m_{k+1}}o)\cdots))}$

This is what is needed to obtain Theorem 24 for EAL-typed terms, and, hence, our main result:

Theorem 41 *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 $\ell \geq 0$.*

2.4 Conclusions

The complexity of optimal sharing is not elementary. This does not mean that the sharing mechanism introduced by Lamping is unfeasible. The non elementary complexity is inherent to the problem of deciding truth of higher order formulas encoded in this Chapter. For that reason the sum of unit cost operations performed by Lamping's algorithm must be non elementary. We have shown that the problem can be solved with the abstract Lamping's algorithm and then only beta and sharing reductions are needed. Since the number of optimal beta reduction steps is polynomial, as it is shown by Asperti, our result says that the majority of the computation is performed by the sharing that should be, for this reason, non elementary.

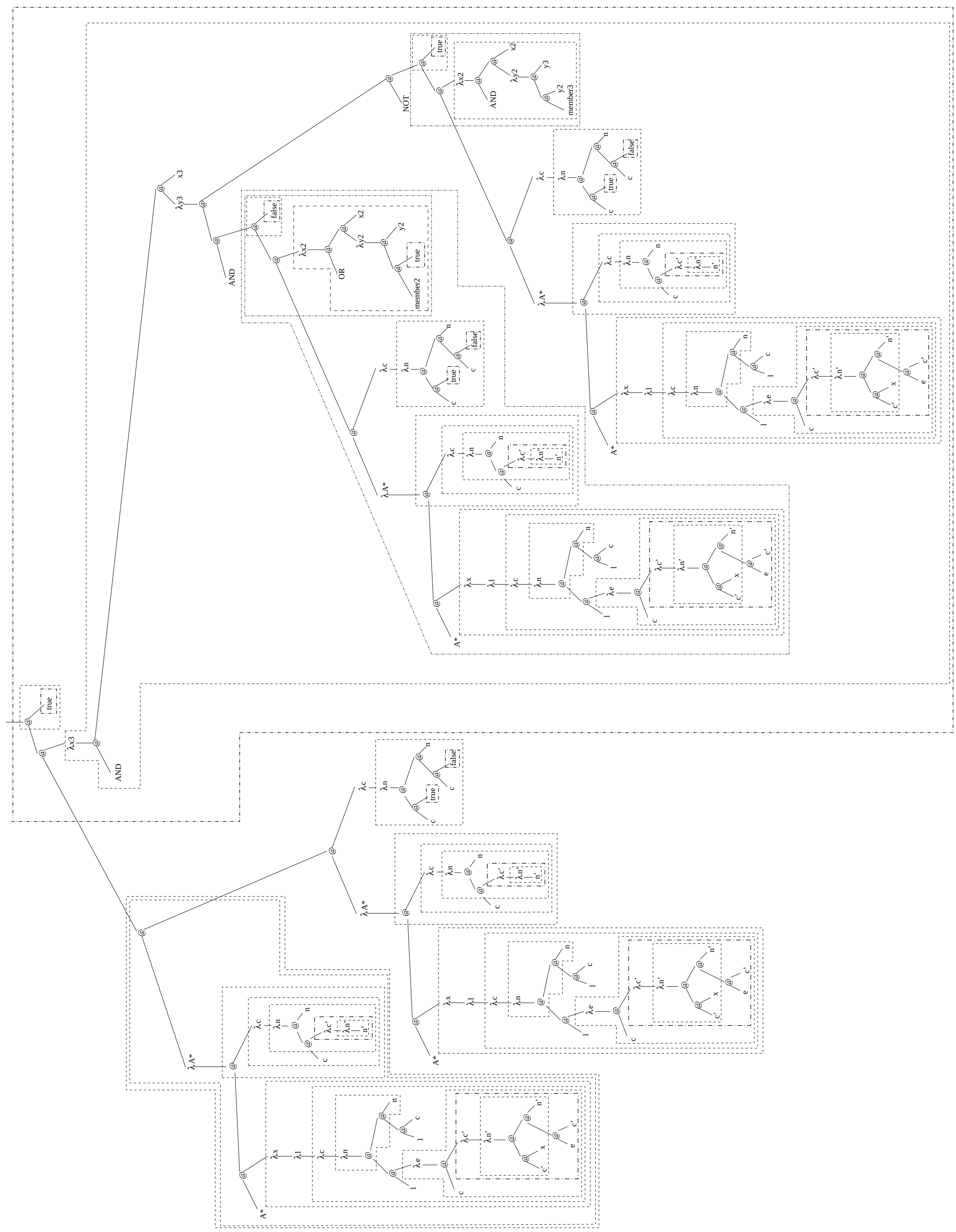


Figure 2.25: $\forall x^3 (\exists x^2 \; true \in x^2 \;\; \wedge \;\; \neg(\forall x^2 \; x^2 \in x^3))$

3

EAL-typing

In the previous Chapter we have seen that the complexity of optimal sharing is not elementary recursive. We have shown that a particular family of lambda terms is typeable in EAL and then we have proved the thesis. As we have already mentioned, we used an ad hoc technique. In the current Chapter we prove that type inference in EAL is decidable. After the study in [Sch94, DJS95] on linear decorations, we propose a simple idea for producing all possible decorations in EAL of a given intuitionistic derivation. Then we give a type inference algorithm in EAL for simply typed lambda terms. Given a simply typed lambda term, our algorithm produces an EAL-type with a parametric number of ! and a set of linear constraints such that for any solution of the set of constraints it is possible to build an EAL-type for the starting lambda term. Finally we prove the existence of a principal type for terms of EAL. We give a procedure that given a pure lambda term generates a finite set of "schemata" such that any EAL-type for the lambda term can be obtained by substitutions.

3.1 Type inference in EAL

A simple inspection of the rules of EAL shows that any λ-term with an EAL-type has also a simple type[1]. Indeed, the simple type (and the corresponding derivation) is obtained by forgetting the exponentials, which must be present in an EAL derivation because of contraction.

The idea underlying our type inference algorithm is simple:

1. finding all "maximal decorations";

2. solving sets of linear constraints.

We informally present the main point with an example on the term $two \equiv \lambda xy.(x(x\ y))$. One (sequent) simple type derivation for two is:

[1] However there are simply typed terms not typeable in EAL as we have already shown in Section 2.2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{w{:}\alpha\vdash w{:}\alpha}\quad\dfrac{}{y{:}\alpha\vdash y{:}\alpha}}{x{:}\alpha\to\alpha,y{:}\alpha\vdash(x\ y){:}\alpha}\quad\dfrac{}{z{:}\alpha\vdash z{:}\alpha}}{x{:}\alpha\to\alpha,x{:}\alpha\to\alpha,y{:}\alpha\vdash(x(x\ y)){:}\alpha}}{x{:}\alpha\to\alpha,x{:}\alpha\to\alpha\vdash\lambda y.(x(x\ y)){:}\alpha\to\alpha}}{x{:}\alpha\to\alpha\vdash\lambda y.(x(x\ y)){:}\alpha\to\alpha}}{\vdash\lambda xy.(x(x\ y)){:}(\alpha\to\alpha)\to\alpha\to\alpha}$$

If we change every $\to$ in $\multimap$, the previous derivation can be viewed as the skeleton of an EAL derivation. To obtain a full EAL derivation (provided it exists), we need to decorate this skeleton with exponentials, and to check that the contraction is performed only on exponential formulas.

Let's produce first a *maximal decoration* of the skeleton, interleaving n ! introduction rules after each logical rule. For example

$$\dfrac{\dfrac{}{w{:}\alpha\vdash_{\mathsf{EAL}} w{:}\alpha}\quad\dfrac{}{y{:}\alpha\vdash_{\mathsf{EAL}} y{:}\alpha}}{x{:}\alpha\multimap\alpha,y{:}\alpha\vdash_{\mathsf{EAL}}(x\ y){:}\alpha}$$

becomes

$$\dfrac{\dfrac{\dfrac{}{w{:}\alpha\vdash_{\mathsf{EAL}} w{:}\alpha}}{!^{n_1}w{:}\alpha\vdash_{\mathsf{EAL}} !^{n_1}w{:}\alpha}\;!^{n_1}\qquad\dfrac{\dfrac{}{y{:}\alpha\vdash_{\mathsf{EAL}} y{:}\alpha}}{!^{n_2}y{:}\alpha\vdash_{\mathsf{EAL}} !^{n_2}y{:}\alpha}\;!^{n_2}}{x{:}!^{n_2}\alpha\multimap!^{n_1}\alpha,y{:}!^{n_2}\alpha\vdash_{\mathsf{EAL}}(x\ y){:}!^{n_1}\alpha}$$

where n_1 and n_2 are fresh variables. We obtain in this way a meta-derivation representing all EAL derivations with $n_1, n_2 \in \mathbb{N}$.

Continuing to decorate the skeleton of *two* (i.e. to interleave n_i ! rules) we obtain

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{w{:}\alpha\vdash_{\mathsf{EAL}} w{:}\alpha}}{w{:}!^{n_1}\alpha\vdash_{\mathsf{EAL}} w{:}!^{n_1}\alpha}\;!^{n_1}\qquad\dfrac{\dfrac{}{y{:}\alpha\vdash_{\mathsf{EAL}} y{:}\alpha}}{y{:}!^{n_2}\alpha\vdash_{\mathsf{EAL}} y{:}!^{n_2}\alpha}\;!^{n_2}}{x{:}!^{n_2}\alpha\multimap!^{n_1}\alpha,y{:}!^{n_2}\alpha\vdash_{\mathsf{EAL}}(x\ y){:}!^{n_1}\alpha}}{x{:}!^{n_3}(!^{n_2}\alpha\multimap!^{n_1}\alpha),y{:}!^{n_2+n_3}\alpha\vdash_{\mathsf{EAL}}(x\ y){:}!^{n_1+n_3}\alpha}\;!^{n_3}\qquad\dfrac{\dfrac{}{z{:}\alpha\vdash_{\mathsf{EAL}} z{:}\alpha}}{z{:}!^{n_4}\alpha\vdash_{\mathsf{EAL}} z{:}!^{n_4}\alpha}\;!^{n_4}}{x{:}!^{n_1+n_3}\alpha\multimap!^{n_4}\alpha,x{:}!^{n_3}(!^{n_2}\alpha\multimap!^{n_1}\alpha),y{:}!^{n_2+n_3}\alpha\vdash_{\mathsf{EAL}}(x(x\ y)){:}!^{n_4}\alpha}}{x{:}!^{n_5}(!^{n_1+n_3}\alpha\multimap!^{n_4}\alpha),x{:}!^{n_3+n_5}(!^{n_2}\alpha\multimap!^{n_1}\alpha),y{:}!^{n_2+n_3+n_5}\alpha\vdash_{\mathsf{EAL}}(x(x\ y)){:}!^{n_4+n_5}\alpha}\;!^{n_5}}{x{:}!^{n_5}(!^{n_1+n_3}\alpha\multimap!^{n_4}\alpha),x{:}!^{n_3+n_5}(!^{n_2}\alpha\multimap!^{n_1}\alpha)\vdash_{\mathsf{EAL}}\lambda y.(x(x\ y)){:}!^{n_2+n_3+n_5}\alpha\multimap!^{n_4+n_5}\alpha}}{x{:}!^{n_5+n_6}(!^{n_1+n_3}\alpha\multimap!^{n_4}\alpha),x{:}!^{n_3+n_5+n_6}(!^{n_2}\alpha\multimap!^{n_1}\alpha)\vdash_{\mathsf{EAL}}\lambda y.(x(x\ y)){:}!^{n_6}(!^{n_2+n_3+n_5}\alpha\multimap!^{n_4+n_5}\alpha)}\;!^{n_6}}{x{:}!^{n_5+n_6}(!^{n_1+n_3}\alpha\multimap!^{n_4}\alpha)\vdash_{\mathsf{EAL}}\lambda y.(x(x\ y)){:}!^{n_6}(!^{n_2+n_3+n_5}\alpha\multimap!^{n_4+n_5}\alpha)}$$

The last rule—contraction—is correct in EAL iff the types of x are unifiable and banged. In other words iff the following constraints are satisfied:

$$n_1,n_2,n_3,n_4,n_5,n_6\in\mathbb{N}\quad\wedge\quad n_5=n_3+n_5\quad\wedge\quad n_1+n_3=n_2\quad\wedge\quad n_4=n_1\quad\wedge\quad n_5+n_6\geq 1.$$

The second, third and fourth of these constraints come from unification; the last one from the fact that contraction is allowed only on exponential formulas. These constraints are equivalent to

$$n_1,n_5,n_6\in\mathbb{N}\quad\wedge\quad n_3=0\quad\wedge\quad n_1=n_2=n_4\quad\wedge\quad n_5+n_6\geq 1.$$

Since clearly these constraints admit solutions, we conclude the decoration procedure obtaining

$$\dfrac{\dfrac{\vdots}{x{:}!^{n_5+n_6}(!^{n_1}\alpha\multimap!^{n_1}\alpha)\vdash_{\mathsf{EAL}}\lambda y.(x(x\ y)){:}!^{n_6}(!^{n_1+n_5}\alpha\multimap!^{n_1+n_5}\alpha)}}{\vdash_{\mathsf{EAL}}\lambda xy.(x(x\ y)){:}!^{n_5+n_6}(!^{n_1}\alpha\multimap!^{n_1}\alpha)\multimap!^{n_6}(!^{n_1+n_5}\alpha\multimap!^{n_1+n_5}\alpha)}$$

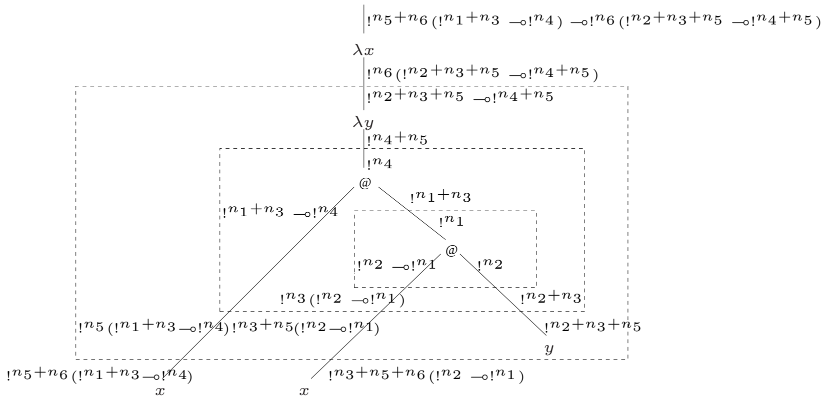


Figure 3.1: Meta EAL type derivation of *two*.

Thus *two* has EAL types $!^{n_5+n_6}(!^{n_1}\alpha \multimap !^{n_1}\alpha) \multimap !^{n_6}(!^{n_1+n_5}\alpha \multimap !^{n_1+n_5}\alpha)$, for any n_1, n_5, n_6 solutions of

$$n_1,n_5,n_6 \in \mathbb{N} \quad \wedge \quad n_5+n_6 \geq 1.$$

We may display the full derivation in a more manageable way, representing the skeleton with the syntax tree of the lambda term with edges labelled with types and adding boxes representing the ! introduction rules, as in Figure 3.1.

Finally notice that at the beginning of this section, we started with "*one (sequent) derivation*" for *two* (there are other derivations, building in a different way the application $x(x\ y)))$. If that derivation had produced an unsolvable set of constraint, the procedure should restart with another derivation. To avoid this problem, our search for maximal decorations (i.e., the collection of constraints) is not performed on sequent derivations, but on the syntax tree of the term. For this reason we introduce NEAL in Section 3.1.1. However, the fact that multiple derivations for a term and principal type scheme are possible, will surface again. It may happen that a solution to a set of constraints corresponds to more than one derivation (a *superposition of derivations*), with non compatible box-assignments. In this case, Lemma 59 ensures that compatible box assignments may be found.

3.1.1 NEAL

The natural deduction calculus (NEAL) for EAL in given in Figure 3.2, after [BBdPH93, Asp98, Rov98].

Lemma 42 (Weakening) *If* $\Gamma \vdash_{\mathsf{NEAL}} A$ *then* $B, \Gamma \vdash_{\mathsf{NEAL}} A$.

> **Proof:** Trivially by induction for *contr*, $(\multimap\ I)$ and $(\multimap\ E)$ rules and by definition for *ax* and ! rules. $\square$

To annotate NEAL derivations, we use terms generated by the following grammar (*elementary affine terms* Λ^{EA}):

$$M ::= x \mid \lambda x.M \mid (M\ M) \mid !(M)\,[M/x, \ldots, M/x] \mid \|M\|^{M}_{x,x}$$

Observe that in $!(M)\,[M/x, \ldots, M/x]$, the $[M/x]$ is a kind of explicit substitution. To define ordinary substitution, define first the set of free variables of a term M, $\mathtt{FV}(M)$, inductively as follows:

$$\frac{}{\Gamma, A \vdash_{\mathsf{NEAL}} A}\ ax \qquad \frac{\Gamma \vdash_{\mathsf{NEAL}} !A \quad \Delta, !A, !A \vdash_{\mathsf{NEAL}} B}{\Gamma, \Delta \vdash_{\mathsf{NEAL}} B}\ contr$$

$$\frac{\Gamma, A \vdash_{\mathsf{NEAL}} B}{\Gamma \vdash_{\mathsf{NEAL}} A \multimap B}\ (\multimap I) \qquad \frac{\Gamma \vdash_{\mathsf{NEAL}} A \multimap B \quad \Delta \vdash_{\mathsf{NEAL}} A}{\Gamma, \Delta \vdash_{\mathsf{NEAL}} B}\ (\multimap E)$$

$$\frac{\Delta_1 \vdash_{\mathsf{NEAL}} !A_1 \cdots \Delta_n \vdash_{\mathsf{NEAL}} !A_n \quad A_1, \ldots, A_n \vdash_{\mathsf{NEAL}} B}{\Gamma, \Delta_1, \ldots, \Delta_n \vdash_{\mathsf{NEAL}} !B}\ !$$

Figure 3.2: Natural Elementary Affine Logic in sequent style notation

- $\mathtt{FV}(x) = \{x\}$
- $\mathtt{FV}(\lambda x.M) = \mathtt{FV}(M) \setminus \{x\}$
- $\mathtt{FV}(M_1\ M_2) = \mathtt{FV}(M_1) \cup \mathtt{FV}(M_2)$
- $\mathtt{FV}(!\,(M)\,[M_1/x_1, \ldots, M_n/x_n]) = \bigcup_{i=1}^{n} \mathtt{FV}(M_i) \cup \mathtt{FV}(M) \setminus \{x_1, \ldots, x_n\}$
- $\mathtt{FV}(\|M\|^{N}_{x_1,x_2}) = (\mathtt{FV}(M) \setminus \{x_1, x_2\}) \cup \mathtt{FV}(N)$

Ordinary substitution $N\{M/x\}$ of a term M for the free occurrences of x in N, is defined in the obvious way:

1. $x\{M/x\} = M$;
2. $y\{M/x\} = y$ if $y \neq x$;
3. $\lambda x.N\{M/x\} = \lambda x.N$;
4. $\lambda y.N\{M/x\} = \lambda z.(N\{z/y\}\{M/x\})$ where z is a fresh variable;
5. $(N\ P)\{M/x\} = (N\{M/x\}\ P\{M/x\})$;
6. $!\,(N)\,[P_1/x_1, \ldots, P_n/x_n]\,\{M/x\} =$
   $!\,(N\{y_1/x_1\} \cdots \{y_n/x_n\}\{M/x\})\,[P_1\{M/x\}/y_1, \ldots, P_n\{M/x\}/y_n]$
   if $x \notin \{x_1, \ldots, x_n\}$, where $y_1, \ldots, y_n$ are all fresh variables;
7. $!\,(N)\,[P_1/x_1, \ldots, P_n/x_n]\,\{M/x\} = !\,(N)\,[P_1\{M/x\}/x_1, \ldots, P_n\{M/x\}/x_n]$
   if $\exists i$ s.t. $x_i = x$;
8. $\|N\|^{P}_{y,z}\,\{M/x\} = \|N\{y'/y\}\{z'/z\}\{M/x\}\|^{P\{M/x\}}_{y',z'}$ if $x \notin \{y, z\}$, where y', z' are fresh variables;
9. $\|N\|^{P}_{y,z}\,\{M/x\} = \|N\|^{P\{M/x\}}_{y,z}$ if $x \in \{y, z\}$.

Elementary terms may be mapped to λ-terms, by forgetting the exponential structure:

- $x^* = x$
- $(\lambda x.M)^* = \lambda x.M^*$

$$\frac{}{\Gamma, x : A \vdash_{\mathsf{NEAL}} x : A}\ ax \qquad \frac{\Gamma \vdash_{\mathsf{NEAL}} M :!A \quad \Delta, x :!A, y :!A \vdash_{\mathsf{NEAL}} N : B}{\Gamma, \Delta \vdash_{\mathsf{NEAL}} \|N\|^{M}_{x,y} : B}\ contr$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{NEAL}} M : B}{\Gamma \vdash_{\mathsf{NEAL}} \lambda x.M : A \multimap B}\ (\multimap I) \qquad \frac{\Gamma \vdash_{\mathsf{NEAL}} M : A \multimap B \quad \Delta \vdash_{\mathsf{NEAL}} N : A}{\Gamma, \Delta \vdash_{\mathsf{NEAL}} (M\ N) : B}\ (\multimap E)$$

$$\frac{\Delta_1 \vdash_{\mathsf{NEAL}} M_1 :!A_1 \cdots \Delta_n \vdash_{\mathsf{NEAL}} M_n :!A_n \quad x_1 : A_1, \ldots, x_n : A_n \vdash_{\mathsf{NEAL}} N : B}{\Gamma, \Delta_1, \ldots, \Delta_n \vdash_{\mathsf{NEAL}} !\,(N)\,[M_1/x_1, \ldots, M_n/x_n] :!B}\ !$$

Figure 3.3: Term Assignment System for Natural Elementary Affine Logic

- $(M_1\ M_2)^* = (M_1^*\ M_2^*)$
- $(!\,(M)\,[M_1/x_1, \ldots, M_n/x_n])^* = M^*\{M_1^*/x_1, \ldots, M_n^*/x_n\}$
- $(\|M\|^{N}_{x_1,x_2})^* = M^*\{N^*/x_1, N^*/x_2\}$

Definition 26 (Legal elementary terms) *The elementary terms are* legal *under the following conditions:*

1. *x is legal;*
2. *$\lambda x.M$ is legal iff M is legal;*
3. *$(M_1\ M_2)$ is legal iff M_1 and M_2 are both legal and $\mathtt{FV}(M_1) \cap \mathtt{FV}(M_2) = \emptyset$;*
4. *$!\,(M)\,[M_1/x_1, \ldots, M_n/x_n]$ is legal iff M and M_i are legal for any $i \quad 1 \le i \le n$ and $\mathtt{FV}(M) = \{x_1, \ldots, x_n\}$ and $(i \neq j \Rightarrow \mathtt{FV}(M_i) \cap \mathtt{FV}(M_j) = \emptyset)$;*
5. *$\|M\|^{N}_{x,y}$ is legal iff M and N are both legal and $\mathtt{FV}(M) \cap \mathtt{FV}(N) = \emptyset$.*

Proposition 43 *If M is a legal term, then every free variable $x \in \mathtt{FV}(M)$ is linear in M.*

> **Proof:** By trivial induction on the structure of M using definitions of legal terms and FV. □

Note
From now on we will consider only legal terms.

NOTATION
Let $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$ be a basis. $dom(\Gamma) = \{x_1, \ldots, x_n\}$; $\Gamma(x_i) = A_i$; $\Gamma \restriction V = \{x : A | x \in V \wedge A = \Gamma(x)\}$.

The term assignment system is shown in Figure 3.3, where all bases in the premises of the contraction, $\multimap$ elimination and !-rule, have domains with empty intersection.

Lemma 44

1. *If $\Gamma \vdash_{\mathsf{NEAL}} M : A$ then $\mathtt{FV}(M) \subseteq dom(\Gamma)$;*

2. *if* $\Gamma \vdash_{\mathsf{NEAL}} M : A$ *then* $\Gamma \restriction \mathtt{FV}(M) \vdash_{\mathsf{NEAL}} M : A$.

Lemma 45 (Substitution) *If* $\Gamma, x : A \vdash_{\mathsf{NEAL}} M : B$ *and* $\Delta \vdash_{\mathsf{NEAL}} N : A$ *and* $dom(\Gamma) \cap dom(\Delta) = \emptyset$ *then* $\Gamma, \Delta \vdash_{\mathsf{NEAL}} M\{N/x\} : B$.

Proof: Recalling that both M and N are legal terms, by easy induction on the structure of M. □

Theorem 46 (Equivalence) $\Gamma \vdash_{EAL} A$ *if and only if* $\Gamma \vdash_{NEAL} A$.

Lemma 47 (Unique Derivation) *For any legal term M and formula A, if there is a valid derivation of the form $\Gamma \vdash_{\mathsf{NEAL}} M : A$, then such derivation is unique (up to weakening).*

A notion of reduction is needed to state and obtain completeness of the type inference algorithm. We have first two *logical* reductions ($\to_\beta$ and $\to_{\mathsf{dup}}$) corresponding to the elimination of principal cuts in EAL. The other five reductions are permutation rules, allowing contraction to be moved out of a term.

$$(\lambda x.M\ N) \quad \to_\beta \quad M\{N/x\}$$

$$\|N\|_{x,y}^{!(M)[M_1/x_1,\ldots,M_n/x_n]} \quad \to_{\mathsf{dup}}$$
$$\left\|\cdots\left\|N\{{}^{!(M)\left[x_1'/x_1,\ldots,x_n'/x_n\right]}/x\}\{{}^{!(M')\left[y_1'/y_1,\ldots,y_n'/y_n\right]}/y\}\right\|_{x_1',y_1'}^{M_1}\cdots\right\|_{x_n',y_n'}^{M_n}$$

$$!(M)[{}^{M_1}/x_1,\cdots,{}^{!(N)[P_1/y_1,\ldots,P_m/y_m]}/x_i,\cdots,{}^{M_n}/x_n] \quad \to_{!-!}$$
$$!(M\{N/x_i\})[{}^{M_1}/x_1,\cdots,{}^{P_1}/y_1,\cdots,{}^{P_m}/y_m,\cdots{}^{M_n}/x_n]$$

$$(\|M\|_{x_1,x_2}^{M_1}\ N) \quad \to_{@-\mathsf{c}} \quad \|(M\{x_1'/x_1,x_2'/x_2\}\ N)\|_{x_1',x_2'}^{M_1}$$

$$(M\ \ \|N\|_{x_1,x_2}^{N_1}) \quad \to_{@-\mathsf{c}} \quad \|(M\ N\{x_1'/x_1,x_2'/x_2\})\|_{x_1',x_2'}^{N_1}$$

$$!(M)[{}^{M_1}/x_1,\cdots,{}^{\|M_i\|_{y,z}^N}/x_i,\cdots,{}^{M_n}/x_n] \quad \to_{!-\mathsf{c}}$$
$$\left\|!(M)[{}^{M_1}/x_1,\cdots,{}^{M_i\{y'/y,z'/z\}}/x_i,\cdots,{}^{M_n}/x_n]\right\|_{y',z'}^{N}$$

$$\|M\|_{x_1,x_2}^{\|N\|_{y_1,y_2}^P} \quad \to_{\mathsf{c}-\mathsf{c}} \quad \left\|\|M\|_{x_1,x_2}^{N\{y_1'/y_1,y_2'/y_2\}}\right\|_{y_1',y_2'}^{P}$$

$$\lambda x.\,\|M\|_{y,z}^N \quad \to_{\lambda-\mathsf{c}} \quad \|\lambda x.M\|_{y,z}^N \text{ where } x \notin \mathtt{FV}(N)$$

where M' in the $\to_{\mathsf{dup}}$-rule is obtained from M replacing all its free variables with fresh ones (x_i is replaced with y_i); x_1' and x_2' in the $\to_{@-\mathsf{c}}$-rule, y' and z' in the $\to_{!-c}$-rule and y_1', y_2' in the $\to_{\mathsf{c}-\mathsf{c}}$-rule are fresh variables.

Definition 27 *The reduction relation on legal terms $\rightsquigarrow$ is defined as the reflexive and transitive closure of the union of $\to_\beta, \to_{\mathsf{dup}}, \to_{!-!}, \to_{@-\mathsf{c}}, \to_{!-\mathsf{c}}, \to_{\mathsf{c}-\mathsf{c}}, \to_{\lambda-\mathsf{c}}$.*

Proposition 48 *Let $M \rightsquigarrow N$ and M be a legal term, then N is a legal term.*

Proposition 49 *Let $M \to_r N$ where r is not $\to_\beta$, then $M^* = N^*$.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{n : I_{I_o} \to I_o \vdash_{\mathsf{EAL}} n : I_{I_o} \to I_o \quad \dfrac{z : I_o \vdash_{\mathsf{EAL}} z : I_o}{\vdash_{\mathsf{EAL}} \lambda z.z : I_{I_o}}}{n : I_{I_o} \to I_o \vdash_{\mathsf{EAL}} (n\ \lambda z.z) : I_o} \quad y : o \vdash_{\mathsf{EAL}} y : o}{n : I_{I_o} \to I_o, y : o \vdash_{\mathsf{EAL}} ((n\ \lambda z.z)\ y) : o}}{n : I_{I_o} \to I_o \vdash_{\mathsf{EAL}} \lambda y.((n\ \lambda z.z)\ y) : I_o}}{\vdash_{\mathsf{EAL}} \lambda n.\lambda y.((n\ \lambda z.z)\ y) : (I_{I_o} \to I_o) \to I_o} \quad \dfrac{\dfrac{x : I_{I_o} \vdash_{\mathsf{EAL}} x : I_{I_o} \quad \dfrac{x : I_{I_o} \vdash_{\mathsf{EAL}} x : I_{I_o} \quad \dfrac{w : o \vdash_{\mathsf{EAL}} w : o}{\vdash_{\mathsf{EAL}} \lambda w.w : I_o}}{x : I_{I_o} \vdash_{\mathsf{EAL}} (x\ \lambda w.w) : I_o}}{x : I_{I_o} \vdash_{\mathsf{EAL}} (x\ (x\ \lambda w.w)) : I_o}}{\vdash_{\mathsf{EAL}} \lambda x.(x\ (x\ \lambda w.w)) : I_{I_o} \to I_o}}{\vdash_{\mathsf{EAL}} (\lambda n.\lambda y.((n\ \lambda z.z)\ y)\ \lambda x.(x\ (x\ \lambda w.w))) : o \to o}$$

Figure 3.4: Simple type derivation of $(\lambda n.\lambda y.((n\ \lambda z.z)\ y)\ \lambda x.(x\ (x\ \lambda w.w))) : o \to o$

Lemma 50 *Let M be a well typed term in $\{\mathsf{dup}, !\text{–}!, @ - \mathsf{c}, ! - \mathsf{c}, \mathsf{c} - \mathsf{c}, \lambda - \mathsf{c}\}$-normal form, then*

1. *if $R = \|N\|^{P}_{x,y}$ is a subterm of M, then either $P = (P_1\ P_2)$ or P is a variable;*
2. *if $R =! (N) [P_1/x_1, \ldots, P_k/x_k]$ is a subterm of M, then for any $i \in \{1, \ldots, k\}$ either $P_i = (Q_i\ S_i)$ or P_i is a variable.*

Theorem 51 (Subject Reduction) *Let $\Gamma \vdash_{\mathsf{NEAL}} M : A$ and $M \leadsto N$, then $\Gamma \vdash_{\mathsf{NEAL}} N : A$.*

3.1.2 Example of type inference

The type inference algorithm is given as a set of inference rules, specifying several functions. The complete set of rules is given in Section 3.1.4; the properties of the algorithm will be stated and proved in Section 3.1.3. We start here with the detailed discussion of an example, which will also introduce the various rules and the problems they face.

A class of types for an EAL-typeable term can be seen as a decoration of a simple type with a suitable number of boxes.

Definition 28 *A* general EAL-type Θ *is generated by the following grammar:*

$$\Theta ::=!^{n_1+\cdots+n_k} o |!^{n_1+\cdots+n_k} (\Theta \multimap \Theta)$$

where $k \geq 0$ and $n_1, \ldots, n_k$ are variables ranging on $\mathbb{N}$.

We shall illustrate our algorithm on the term $(\lambda n.\lambda y.((n\ \lambda z.z)\ y)\ \lambda x.(x\ (x\ \lambda w.w))) : o \to o$, whose *simple* type derivation is given in Figure 3.4 (I_α stands for $\alpha \to \alpha$).

The algorithms searches for the leftmost innermost subterm which is not already EAL-typed. In this case, it is the variable

$$n : (((o \to o) \to (o \to o)) \to (o \to o))\,.$$

Its most general EAL-type is obtained from its simple type by adding p_i modalities wherever possible. This is the rôle of the function $\mathscr{P}$:

$$\overline{\mathscr{P}(o) =!^{p} o} \tag{3.1}$$

$$\dfrac{\mathscr{P}(\sigma) = \Theta \quad \mathscr{P}(\tau) = \Gamma}{\mathscr{P}(\sigma \to \tau) =!^{p} (\Theta \multimap \Gamma)}. \tag{3.2}$$

The main function of the algorithm—the type synthesis function $\mathcal{S}$—may now be applied. In the case of a variable x of simple type σ the rule is:

$$\dfrac{\mathscr{P}(\sigma) = \Theta}{\mathcal{S}(x : \sigma) = \langle \Theta, \{x : \Theta\}, \emptyset, \emptyset \rangle} \tag{3.3}$$

Observe that, given a term M of simple type σ, $\mathcal{S}(M : \sigma)$ returns a quadruple:

$$\langle \text{general EAL-type, base}^2\ \{x_i : \Theta_i\}_i \text{ of pairs (variable:general EAL-type), set of linear constraints, critical points}^3\rangle.$$

In our example we obtain:

$$n :!^{p_1} (!^{p_2} (!^{p_3} (!^{p_4} o \multimap !^{p_5} o) \multimap !^{p_6} (!^{p_7} o \multimap !^{p_8} o)) \multimap !^{p_9} (!^{p_{10}} o \multimap !^{p_{11}} o)) . \tag{3.4}$$

for any $p_i \in \mathbb{N}, 1 \leq i \leq 11$. In the following we will not explicit the "$\in \mathbb{N}$" for any variables we will introduce, being this constraint implicated by Definition 28.

Notation
From now on, we will write $(n \multimap m)$ instead of $(!^n o \multimap !^m o)$, for a better reading.

Analogously, $z : (o \rightarrow o)$ is typed

$$z : p_{12}(p_{13} \multimap p_{14}) \tag{3.5}$$

It is now the turn of the subterm $\lambda z.z$. The type synthesis rule for an abstraction $\lambda x.M$, where x occurs in M, takes the following steps:

1. infer the EAL-type for M;
2. add *all possible boxes* around M (function $\mathbb{B}$, which will be described later); the algorithm tries to build all possible decorations[4] that in the case of an abstraction $\lambda x.M$ are the decorations of all subterms of M, already build by inductive hypothesis, plus all possible box-decorations of the whole M, performed at this stage of the inference by function $\mathbb{B}$, plus all possible box decorations of $\lambda x.M$, eventually performed at the next step of the inference procedure;
3. contract all the types of abstracted variable x (function $\mathcal{C}$, which will be described later).

The rule is the following:

$$\frac{\begin{array}{l} \mathcal{C}(\Theta_1, \ldots, \Theta_h) = A_3 \\ \mathbb{B}\left(M, B_1, \Gamma_1, cpts \cup \begin{Bmatrix} sl_1(x) \\ \vdots \\ sl_k(x) \end{Bmatrix}, A_1\right) = \left\langle B \cup \begin{Bmatrix} x : \Theta_1 \\ \vdots \\ x : \Theta_h \end{Bmatrix}, \Gamma, A_2 \right\rangle \\ \mathcal{S}(M : \tau) = \langle \Gamma_1, B_1, A_1, cpts \cup \{sl_1(x), \ldots, sl_k(x)\}\rangle \end{array}}{\mathcal{S}(\lambda x.M : \sigma \rightarrow \tau) = \left\langle \Theta_1 \multimap \Gamma, B, \left\{ \begin{array}{c} A_2 \\ A_3 \end{array}\right., cpts \right\rangle} \tag{3.6}$$

In our example, there is only one occurrence of z and therefore the contraction function $\mathcal{C}$ is called with only one type and does not produce any constraint. Also the boxing function $\mathbb{B}$ produce no result, being called on a variable, i.e. it acts as the identity returning a triple with the same base, type and (empty, in this case) set of constraints:

$$\mathbb{B}(z, \{z : p_{12}(p_{13}o \multimap p_{14}o)\}, p_{12}(p_{13}o \multimap p_{14}o), \emptyset, \emptyset) = \langle\{z : p_{12}(p_{13}o \multimap p_{14}o)\}, p_{12}(p_{13}o \multimap p_{14}o), \emptyset\rangle.$$

[2]A base here is a *multiset*, i.e. there could be serveral copies of $x : \Theta$.

[3]We will discuss critical points in a moment.

[4]More precisely it builds all possible decorations without exponential cuts and with some other properties listed in Theorem 58.

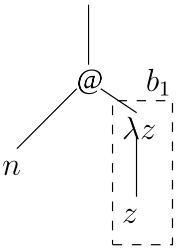


Figure 3.5: Decoration of $(n\ \lambda z.z)$.

The rôle of *cpts* and *sl* will be discussed in the context of the critical points, below. Coming back to our example, for $\lambda z.z : ((o \to o) \to (o \to o))$ we infer the EAL-type

$$\lambda z.z : p_{12}(p_{13} \multimap p_{14}) \multimap p_{12}(p_{13} \multimap p_{14}) \tag{3.7}$$

When the algorithm infers the EAL-type for $(n\ \lambda z.z) : (o \to o)$, it:

1. adds all possible boxes around the argument $\lambda z.z$ with the boxing function, that in this case adds b_1 boxes around $\lambda z.z$ returning a triple with the same base, b_1 banged type and unmodified set of (again empty) constraints:

$$\begin{aligned}\mathbb{B}(\lambda z.z, \emptyset, p_{12}(p_{13} \multimap p_{14}) \multimap p_{12}(p_{13} \multimap p_{14}), \emptyset, \emptyset) =& \\ = \langle \emptyset, b_1(p_{12}(p_{13} \multimap p_{14}) \multimap p_{12}(p_{13} \multimap p_{14})), \emptyset \rangle&\end{aligned}$$

2. imposes the EAL-type of n to be functional, i.e. the constraint

$$\boxed{p_1 = 0} \tag{3.8}$$

3. unifies the EAL-type of the boxed $\lambda z.z$ with the argument part of the EAL-type of n:

$$\mathscr{U}\left(\begin{array}{c} b_1(p_{12}(p_{13} \multimap p_{14}) \multimap p_{12}(p_{13} \multimap p_{14})), \\ p_2(p_3(p_4 \multimap p_5) \multimap p_6(p_7 \multimap p_8)) \end{array}\right).$$

Observe that the implicational structure of the types is already correct, since we start from a simple type derivation. Therefore, unification only produces a set of contraints on the variables used to indicate boxes. In our example, we get the constraints:

$$\left\{\begin{array}{ll} b_1 & = p_2 \\ p_{12} & = p_3 \\ p_{13} & = p_4 \\ p_{14} & = p_5 \\ p_{12} & = p_6 \\ p_{13} & = p_7 \\ p_{14} & = p_8 \end{array}\right. \Leftrightarrow \boxed{\left\{\begin{array}{lll} b_1 & = p_2 & \\ p_3 & = p_6 & = p_{12} \\ p_4 & = p_7 & = p_{13} \\ p_5 & = p_8 & = p_{14}. \end{array}\right.} \tag{3.9}$$

The type synthesis rule[5] for an application, provided that M and N are not applications

[5]We will explain $\mathbb{U}$ later

themselves, is:

$$\frac{\begin{array}{l}\mathscr{U}(\Theta_1,\Theta_3)=A_4\\ \mathbb{B}(N,B_2,\Theta_2,cpts_2,A_2)=\langle B_3,\Theta_3,A_3\rangle\\ \mathcal{S}(N:\sigma)=\langle\Theta_2,B_2,A_2,cpts_2\rangle\\ \mathcal{S}(M:\sigma\to\tau)=\langle !^{\sum n_i}(\Theta_1\multimap\Gamma),B_1,A_1,cpts_1\rangle\end{array}}{\mathcal{S}((M\ N):\tau)=\left\langle\Gamma,B_1\cup B_3,\left\{\begin{array}{c}A_1\\ A_3\\ A_4\\ \sum n_i=0\end{array}\right.,cpts_1\Cup cpts_2\right\rangle} \tag{3.10}$$

Figure 3.5 shows the decoration obtained so far for:

$$n:b_1(p_3(p_4\multimap p_5)\multimap p_3(p_4\multimap p_5))\multimap p_9(p_{10}\multimap p_{11})\vdash(n\ \lambda z.z):p_9(p_{10}\multimap p_{11}). \tag{3.11}$$

Next step is the inference of a general EAL-type p_{15} for $y:o$. Then the algorithm starts to process $((n\ \lambda z.z)\ y):o$. As before, the algorithm

1. applies $\mathbb{B}$ to the argument y (a void operation here, since the boxing function does nothing for variables);

2. imposes the EAL-type of $(n\ \lambda z.z)$ to be functional:

$$p_9=0. \tag{3.12}$$

3. unifies the EAL-types, to make type-correct the application:

$$\mathscr{U}(p_{10},p_{15})=\quad\boxed{p_{10}=p_{15}.} \tag{3.13}$$

However, the present case is more delicate than the application we treated before, since the function part is already an application. Two consecutive applications in $((n\ \lambda z.z)\ y)$ indicates that more than one decoration is possible. Indeed, there can be several *sequent* derivations building the same term, that can be differently decorated. The issue is better appreciated if we look ahead for a moment and we consider the term $\lambda y.((n\ \lambda z.z)\ y)$. There are two (simple) sequent derivations for this term, both starting with the term $(x\ y):o$, for $x:o\to o, y:o$. The first derivation, via a left $\to$-rule, obtains $((n\ \lambda z.z)\ y):o$; then it bounds y, giving $\lambda y.((n\ \lambda z.z)\ y):o\to(o\to o)$. The second derivation permutes the rules: it starts by binding y, obtaining $\lambda y.(x\ y)$ and only at this point substitues $(n\ \lambda z.z)$ for x, via the left $\to$-rule. When we add boxes to the two derivations, we see this is a *critical* situation. Indeed, in the first derivation we may box $(x\ y)$, then $((n\ \lambda z.z)\ y)$ and finally $\lambda y.((n\ \lambda z.z)\ y)$. In the second, we box $(x\ y)$, then $\lambda y.(x\ y)$ and finally the whole term. The two (incompatible) decorations are depicted in the two bottom trees of Figure 3.9. The critical edge—where the boxing radically differs—is the root of the subtree for $((n\ \lambda z.z)\ y)$, corresponding to the x that is substituted for in the left $\to$-rule. Let us then resume the discussion of the type inference for this term. At this stage we collect the *critical point*, marked with a star in Figure 3.6, indicating the presence of two possible derivations. When, in the future, it will be possible to add boxes, for example b_2 in Figure 3.6 during the type inference of $\lambda y.((n\ \lambda z.z)\ y)$, the algorithm will consider the critical point as one of the closing points of such boxes, c_2 in Figure 3.6, eventually modifying the constraint in Equation (3.12) that impose type of $(n\ \lambda z.z)$ to be functional and not exponential. Indeed,

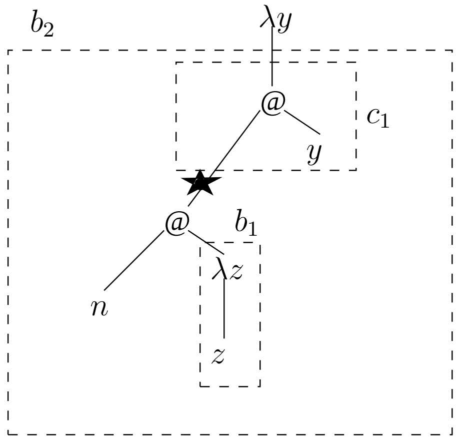


Figure 3.6: Critical point in the decoration of $\lambda y.((n\ \lambda z.z)y)$.

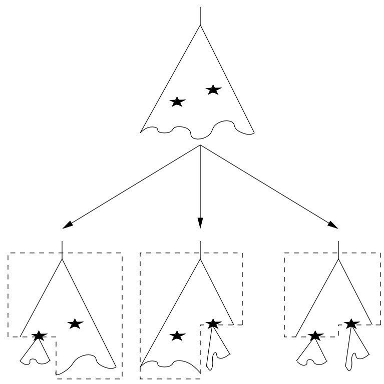


Figure 3.7: Combinations of two critical points.

for completeness, the algorithm must take into account all possible derivations. When there will be more than one critical point, at every stage of the type inference, when it is possible to apply a ! rule, the algorithm will compute all possible combinations of the critical points (see Figure 3.7, showing a schematic example with two critical points) eventually modifying some constraints. We call *slices*[6] such combinations of critical points; they are the data maintained by the algorithm and indicated in the rules as *cpts*. The task of combining the two lists of slices collected during the type inference of the function and argument part of an application is performed by $\Cup$ whose rules are given in Section 3.1.4.

Definition 29 *The list of* free variable occurrences *of a lambda term M is defined in the following way:*

(a) $\texttt{FVO}(x) = [x]$*;*

(b) $\texttt{FVO}(\lambda x.M) = \texttt{FVO}(M) - x$*;*

(c) $\texttt{FVO}((M_1\ M_2)) = \texttt{FVO}(M_1) :: \texttt{FVO}(M_2)$ *(the concatenation of lists).*

[6] We thank Philippe Dague for useful discussions and suggestions on the calculation of critical points.

Definition 30 *A* slice *is a set of pairs (constraint, list of free variable occurrences) as in the following*[7]*:*

$$sl = \{(A^{j_1}, [y_{1_1}, \ldots, y_{1_h}]), \ldots, (A^{j_k}, [y_{k_1}, \ldots, y_{k_h}])\}$$

A slice corresponds to a combination of critical points.

In our example the algorithm collects the slice $(p_9 = 0, [n])$. Notice that a slice partitions the set of free variable occurrences in a derivation: it marks the set of variable occurrences whose types should not be modified when the box is added. This is the intuitive meaning of the set of free variable occurrences in the data structure we use.

NOTATION

- $sl(x)$ indicates a slice having x as an element of every list of variables in it.
- $x \in sl$ if and only if there exists one element of sl whose list of variables contains x.
- $A^j \in sl$ if and only if there exists one element of sl whose constraint is A^j.
- Being A^j the constraint $\pm n_{j_1} \pm \cdots \pm n_{j_k} = 0$, $A^j - n$ corresponds to the constraint $\pm n_{j_1} \pm \cdots \pm n_{j_k} - n = 0$.

The general type inference rule for the application we are considering now, i.e. $((M_1\ M_2)\ N)$ when N is not an application, is the following:

$$\frac{\begin{array}{l} cpts = (cpts_1 \cup \{(\sum n_i = 0, \mathtt{FVO}((M_1\ M_2)))\}) \uplus cpts_2 \\ \mathscr{U}(\Theta_1, \Theta_3) = A_4 \\ \mathbb{B}(N, B_2, \Theta_2, cpts_2, A_2) = \langle B_3, \Theta_3, A_3 \rangle \\ \mathcal{S}(N : \sigma) = \langle \Theta_2, B_2, A_2, cpts_2 \rangle \\ \mathcal{S}((M_1\ M_2) : \sigma \to \tau) = \langle !^{\sum n_i}(\Theta_1 \multimap \Gamma), B_1, A_1, cpts_1 \rangle \end{array}}{\mathcal{S}((M_1\ M_2)\ N) : \tau) = \left\langle \Gamma, B_1 \cup B_3, \left\{ \begin{array}{c} A_1 \\ A_3 \\ A_4 \\ \sum n_i = 0 \end{array} \right., cpts \right\rangle} \tag{3.14}$$

In the example case we obtain:

$$\left\{ \begin{array}{lcl} n & : & b_1(p_3(p_4 \multimap p_5) \multimap p_3(p_4 \multimap p_5)) \\ & & \qquad\qquad \multimap p_9(p_{10} \multimap p_{11}), \\ y & : & p_{10} \end{array} \right\} \vdash ((n\ \lambda z.z)\ y) : p_{11} \tag{3.15}$$

and critical points $cpts = \{(p_9 = 0, [n])\}$.

Typing $\lambda y.((n\ \lambda z.z)\ y) : o \to o$ involves rule (3.6), the same we used for $\lambda z.z$, but now the boxing procedure $\mathbb{B}$ is called on a subterm that is not a single variable. The complete set of rules for $\mathbb{B}$ is the following:

$$\frac{}{\mathbb{B}(x, B, \Gamma, cpts, A) = \langle B, \Gamma, A \rangle} \tag{3.16}$$

[7] A^j means the j-th row of the matrix A, i.e. the j-th constraint.

Boxing of a variable produces no changes in the base, type and set of constraints.

$$\frac{\mathscr{B}(B,\Gamma,cpts,A)=\langle B_1,\Gamma_1,A_1\rangle}{\mathbb{B}(M,B,\Gamma,cpts,A)=\langle !^b B_1,!^b\Gamma_1,A_1\rangle} \tag{3.17}$$

$\mathscr{B}$ takes care of the list of critical points, by adding boxes "inside" the term as in Figure 3.7; at the end, $\mathbb{B}$ adds b boxes "around" the term.

$$\frac{}{\mathscr{B}(B,\Gamma,\emptyset,A)=\langle B,\Gamma,A\rangle} \tag{3.18}$$

$\mathscr{B}$ with no critical points produces no changes.

$$\frac{\begin{array}{l}\mathscr{B}\,(B_1,!^c\Gamma,cpts,A_2)=\langle B,\Delta,A_1\rangle\\ B_1=\left\{x_i:\left\{\begin{array}{ll}!^c\Theta_i & x_i\notin sl\\ \Theta_i & x_i\in sl\end{array}\right.\right\}_i\\ A_2=\left(\left\{\begin{array}{ll}A^j & A^j\notin sl\\ A^j-c & A^j\in sl\end{array}\right.\right)_j\end{array}}{\mathscr{B}(\{x_i:\Theta_i\}_i,\Gamma,\{sl\}\cup cpts,A)=\langle B,\Delta,A_1\rangle} \tag{3.19}$$

Therefore, rule (3.6) gives in our case:

$$\mathcal{S}(\lambda y.((n\ \lambda z.z)\ y):o\to o)=$$
$$\left\langle\begin{array}{c} b_2+c_1+p_{10}\multimap b_2+c_1+p_{11},\\ \{n:b_2(b_1(p_3(p_4\multimap p_5)\multimap p_3(p_4\multimap p_5))\multimap p_9(p_{10}\multimap p_{11}))\}\,,\\ \left\{\begin{array}{c}\vdots\\ p_9-c_1=0\\ \vdots\end{array}\right.,\\ \{(p_9-c_1=0,[n])\}\end{array}\right\rangle \tag{3.20}$$

where $p_9-c_1=0$ is the unique constraint (Equation (3.12)) modified by $\mathbb{B}$. The decoration obtained is shown in Figure 3.6. Observe that, at this stage, the presence of incompatible derivations does not show up yet. It will be taken into account as soon as we will try to box a superterm of the one we just processed. If $\lambda y.((n\ \lambda z.z)\ y)$ would be the whole term, on the contrary, an additional call to the function $\mathbb{B}$ would be performed, see the rule (3.59) for function $\mathscr{S}$.

When the algorithm processes $\lambda n.\lambda y.((n\ \lambda z.z)\ y):(((o\to o)\to(o\to o))\to(o\to o))\to(o\to o)$ it applies again rule (3.6). It adds c_2 boxes passing through the critical point and b_3 boxes around the term, obtaining:

$$\mathcal{S}(\lambda n.\lambda y.((n\ \lambda z.z)\ y):(((o\to o)\to(o\to o))\to(o\to o))\to(o\to o))=$$
$$\left\langle\begin{array}{c} b_3+b_2(b_1(p_3(p_4\multimap p_5)\multimap p_3(p_4\multimap p_5))\multimap p_9(p_{10}\multimap p_{11}))\\ \multimap b_3+c_2(b_2+c_1+p_{10}\multimap b_2+c_1+p_{11})\end{array},\ \begin{array}{c}\emptyset,\\ \left\{\begin{array}{c}\vdots\\ \boxed{p_9-c_1-c_2=0}\\ \vdots\end{array}\right.,\\ \emptyset\end{array}\right\rangle \tag{3.21}$$

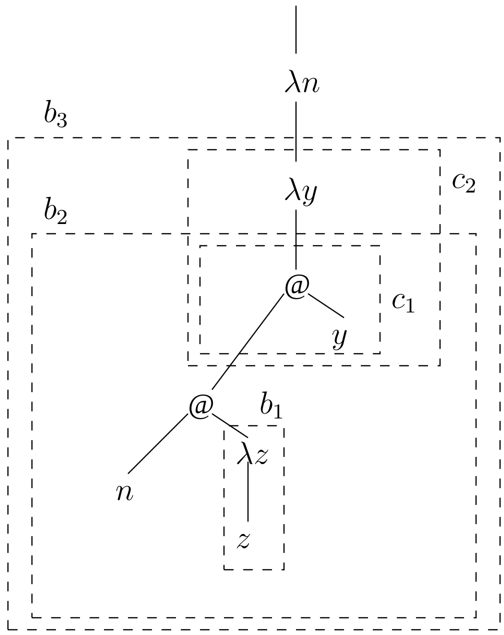


Figure 3.8:

where $p_9 - c_1 - c_2 = 0$ is the uniqe constraints modified at this stage of the synthesis.

The critical point $(p_9 - c_2 - c_2 = 0, [n])$ is removed. In fact, to bound n, the substitution of $n(\lambda z.z)$ for x has to be already performed. It does not make sense to derive first $\lambda n.\lambda y.(x\ y)$, add boxes, and then substitute $n(\lambda z.z)$ for x, since this would be a free-variable catching substitution.

Figure 3.8 shows the decoration obtained. Notice that boxes c_2 and b_2 belong to the two *incompatible EAL-derivations* we already discussed before. The algorithm maintains at the same time these derivations guaranteeing (see Lemma 59) that if the final solution instantiates two incompatible derivations, we can always calculate an equivalent EAL-derivation (Figure 3.9 shows the two possible derivations for our example).

Going on with the type synthesis, the algorithm starts processing the leftmost occurrence of x in $(x\ (x\ \lambda w.w))$. We use superscripts (1) and (2) to discriminate the right and left occurrence, respectively. For the leftmost—$x^{(2)} : (o \to o) \to (o \to o)$—we infer the EAL-type

$$p_{16}(p_{17}(p_{18} \multimap p_{19}) \multimap p_{20}(p_{21} \multimap p_{22})); \tag{3.22}$$

analogously, for the rightmost $x^{(1)} : (o \to o) \to (o \to o)$ we get the EAL-type

$$p_{23}(p_{24}(p_{25} \multimap p_{26}) \multimap p_{27}(p_{28} \multimap p_{29})). \tag{3.23}$$

The EAL-type of $w : o$ is p_{30} and then $\lambda w.w : o \to o$ is typeable in EAL with type $p_{30} \multimap p_{30}$.

The innermost application $(x^{(1)}\ \lambda w.w)$ is typed $p_{27}(p_{28} \multimap p_{29})$, once we have imposed

$$\boxed{p_{23} = 0,} \tag{3.24}$$

we have boxed $\lambda w.w$ with b_4 boxes, and we have unified the types

$$\mathscr{U}(p_{24}(p_{25} \multimap p_{26}), b_4(p_{30} \multimap p_{30})) = \boxed{\begin{cases} b_4 = p_{24} \\ p_{25} = p_{26} = p_{30}. \end{cases}} \tag{3.25}$$

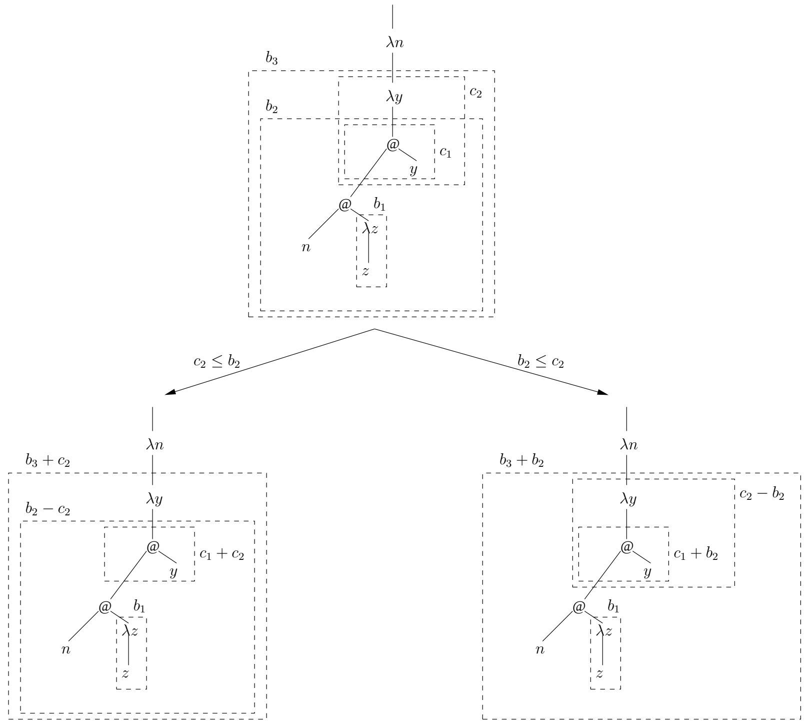


Figure 3.9: Superimposed derivations.

When the algorithm processes $(x^{(2)}\ (x^{(1)}\ \lambda w.w))$, it adds b_5 boxes around the argument, imposes

$$\boxed{p_{16} = 0} \tag{3.26}$$

and unifies the types

$$\mathscr{U}\,(p_{17}(p_{18} \multimap p_{19}), b_5 + p_{27}(p_{28} \multimap p_{29})) = \left\{ \begin{array}{l} p_{17} = b_5 + p_{27} \\ \boxed{\begin{array}{l} p_{18} = p_{28} \\ p_{19} = p_{29}. \end{array}} \end{array} \right. \tag{3.27}$$

Moreover, the presence of two consecutive applications makes the algorithm collect a new critical point $(p_{17} = b_5 + p_{27}, [x^{(1)}])$. The derivation obtained is:

$$\left\{ \begin{array}{r} x^{(1)} : b_5(b_4(p_{25} \multimap p_{25}) \multimap p_{27}(p_{18} \multimap p_{19})), \\ x^{(2)} : p_{17}(p_{18} \multimap p_{19}) \multimap p_{20}(p_{21} \multimap p_{22}) \end{array} \right\} \vdash (x^{(2)}\ (x^{(1)}\ \lambda w.w)) : p_{20}(p_{21} \multimap p_{22}) \tag{3.28}$$

and its decoration is shown in Figure 3.10.

For the type inference of $\lambda x.(x^{(2)}\ (x^{(1)}\ \lambda w.w)) : ((o \to o) \to (o \to o)) \to (o \to o)$, the algorithm applies the usual rule for abstractions seen above (3.6), but in this case there are two instances of the bound variable x. Here comes to work the function $\mathcal{C}$, whose rules are the following.

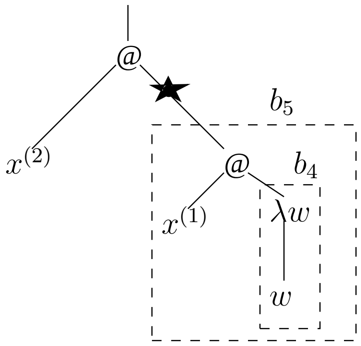


Figure 3.10:

$$\overline{\mathcal{C}(\Theta) = \emptyset} \tag{3.29}$$

$$\frac{\mathscr{U}\,(!^{n_1+\cdots+n_h}\Theta_1, \Theta_2, \ldots, \Theta_k) = A}{\mathcal{C}(!^{n_1+\cdots+n_h}\Theta_1, \ldots, \Theta_k) = \left\{ \begin{array}{c} n_1 + \cdots + n_h \geq 1 \\ A \end{array} \right.} \tag{3.30}$$

Therefore the contraction of k general EAL-types is obtained by unification and the constraint that the contracted types have at least one "!" (since in EAL contraction is allowed only for exponential formulas).

Coming back to our example, the algorithm adds c_3 boxes passing through the critical point and b_6 boxes around the body of the abstraction. The $\mathscr{B}$ function modifies the first constraint in Equation (3.27):

$$\boxed{p_{17} = b_5 + p_{27} - c_3}. \tag{3.31}$$

Then the algorithm contracts the types of x:

$$\begin{aligned} &\mathcal{C}\left(\begin{array}{l} b_6 + b_5(b_4(p_{25} \multimap p_{25}) \multimap p_{27}(p_{18} \multimap p_{19})), \\ b_6 + c_3(p_{17}(p_{18} \multimap p_{19}) \multimap p_{20}(p_{21} \multimap p_{22})) \end{array} \right) = \\ &\qquad = \boxed{\left\{ \begin{array}{l} b_6 + b + 5 \geq 1 \\ b_5 = c_3 \\ b_4 = p_{17} \\ p_{18} = p_{19} = p_{21} = p_{22} = p_{25} \\ p_{20} = p_{27} \end{array} \right.} \end{aligned} \tag{3.32}$$

Finally it removes the critical point $(p_{17} = b_5 + p_{27} - c_3, [x^{(1)}])$.

The derivation obtained, whose decoration is shown in Figure 3.11, is:

$$\begin{aligned} \vdash \lambda x.(x\ (x\ \lambda w.w)) : b_6 + b_5(b_4(p_{18} \multimap p_{18}) \multimap p_{20}(p_{18} \multimap p_{18})) \\ \multimap b_6 + b_5 + p_{20}(p_{18} \multimap p_{18}). \end{aligned} \tag{3.33}$$

The algorithm process now the whole term $(\lambda n.\lambda y.((n\ \lambda z.z)\ y)\ \lambda x.(x\ (x\ \lambda w.w))) : o \to o$. It adds b_7 boxes around the argument of the application and unifies the EAL-types for the

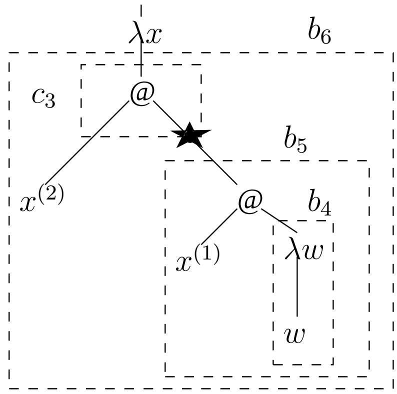


Figure 3.11:

correct application:

$$\mathscr{U}\begin{pmatrix} b_3 + b_2(b_1(p_3(p_4 \multimap p_5) \multimap p_3(p_4 \multimap p_5)) \multimap p_9(p_{10} \multimap p_{11}), \\ b_7(b_6 + b_5(b_4(p_{18} \multimap p_{18}) \multimap p_{20}(p_{18} \multimap p_{18})) \multimap b_6 + b_5 + p_{20}(p_{18} \multimap p_{18}) \end{pmatrix} =$$

$$= \boxed{\begin{cases} b_7 = b_3 + b_2 \\ b_1 = b_6 + b_5 \\ b_4 = p_3 = p_{20} \\ p_4 = p_5 = p_{10} = p_{11} = p_{18} \\ p_9 = b_6 + b_5 + p_{20} \end{cases}} \tag{3.34}$$

Since this is the complete term, the final step of the algorithm is a single call to the function $\mathscr{S}$, which in this case simply adds b_8 boxes around the term. Therefore, the simply typed lambda term

$$(\lambda n.\lambda y.((n\ \lambda z.z)\ y)\ \lambda x.(x\ (x\ \lambda w.w))) : o \to o \tag{3.35}$$

has EAL-type

$$!^{b_8+b_3+c_2}(!^{b_2+c_1+p_4}o \multimap !^{b_2+c_1+p_4}o) \tag{3.36}$$

for any $p_1, \dots, p_{30}, b_1, \dots, b_8, c_1, c_2, c_3 \in \mathbb{N}$ solutions of the set of constrains[8] in equations (3.8)–(3.34):

$$\begin{cases} b_6 + b_5 \geq 1 \\ b_7 = b_3 + b_2 \\ b_1 = p_2 = b_6 + b_5 \\ b_5 = c_3 \\ p_1 = p_{16} = p_{23} = 0 \\ p_9 = c_1 + c_2 = b_6 + b_5 + b_4 \\ p_{17} = b_5 + p_{27} - c_3 \\ b_4 = p_3 = p_6 = p_{12} = p_{17} = p_{20} = p_{24} = p_{27} \\ p_4 = p_5 = p_7 = p_8 = p_{10} = p_{11} = p_{13} = p_{14} = p_{15} = p_{18} \\ p_4 = p_{19} = p_{21} = p_{22} = p_{25} = p_{26} = p_{28} = p_{29} = p_{30}. \end{cases} \tag{3.37}$$

The final decoration is shown in Figure 3.12. Considering the set of constraints in Equa-

[8] We have boxed the constraints which were not modified by $\mathscr{B}$ until the end of the type inference process in the exposition above. They are now all collected in the set of constraints below.

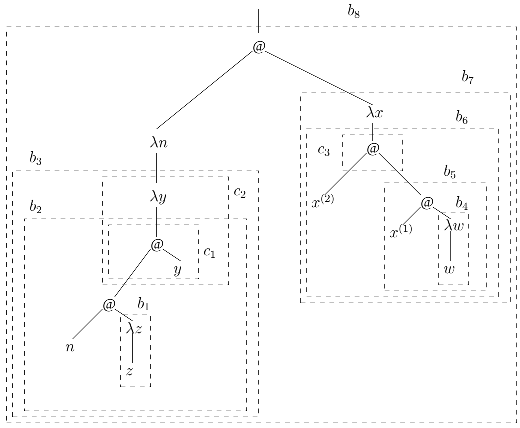


Figure 3.12: Final superimposed decoration.

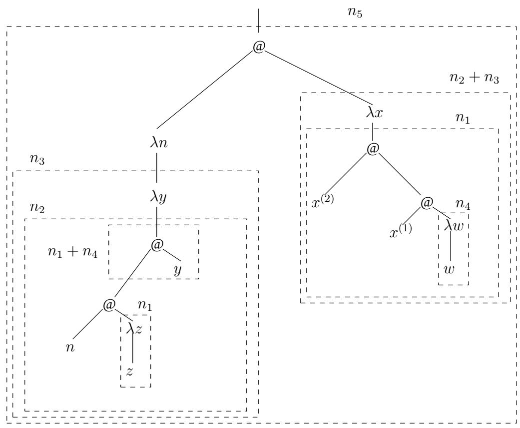


Figure 3.13: Final decoration.

tion (3.37) and the incompatibility of c_2 and b_2 stated above, the simply typed term

$$(\lambda n.\lambda y.((n\ \lambda z.z)\ y)\ \lambda x.(x\ (x\ \lambda w.w))) : o \to o$$

can be typed in EAL either:

1. for any $n_1, \ldots, n_6 \in \mathbb{N}, n_1 \geq 1$ with EAL-type $!^{n_3+n_5}(!^{n_1+n_2+n_4+n_6}o \multimap !^{n_1+n_2+n_4+n_6}o)$ and decoration shown in Figure 3.13, or

2. for any $m_1, \ldots, m_7 \in \mathbb{N}, m_1 \geq 1\ \wedge\ m_2 + m_3 = m_1 + m_5$ with EAL-type $!^{m_3+m_4+m_6}(!^{m_2+m_7}o \multimap !^{m_2+m_7}o)$ and decoration shown in Figure 3.14.

3.1.3 Type Inference

A class of types for an EAL-typeable term can be seen as a decoration of a simple type with a suitable number of boxes. We propose an algorithm collecting integer constraints whose solutions corresponds to proper box assignments.

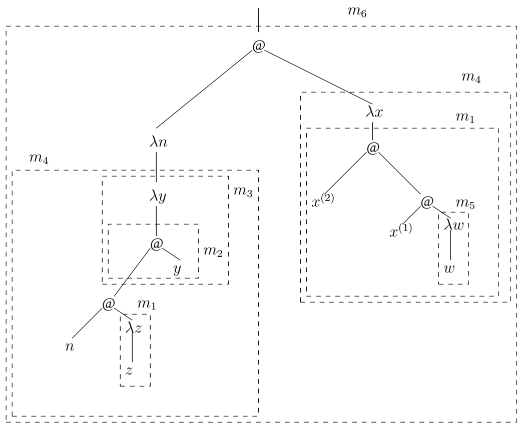


Figure 3.14: Another possible final decoration.

Definition 31 (Type Synthesis Algorithm) *Given a simply typeable lambda term and its principal type scheme $M : \sigma$, the* type synthesis algorithm $\mathscr{S}(M : \sigma)$ *returns a triple $\langle \Theta, B, A \rangle$, where Θ is a general EAL-type, B is a base (i.e. a multi-set of pairs variable, general EAL-type) and A is a set of linear constraints.*

The algorithm $\mathscr{S}(M : \sigma)$ is defined in the Section 3.1.4. One of the crucial issues is the localization of the points where derivations may differ for the presence or absence of boxes around some subterms. This is the role of *critical points*, managed by the boxing procedure, $\mathscr{B}$ (see Section 3.1.4).

Proposition 52 (Termination) *Let M be a simply typed term and let σ be its most general type. $\mathscr{S}(M : \sigma)$ always terminates with a triple $\langle \Theta, B, A \rangle$.*

The algorithm is exponential in the size of the λ-term, because to investigate all possible derivations we need to (try to) box all possible combinations of critical points (see the clauses for the product union, $\Cup$, in Section 3.1.4) that are roughly bounded by the size of the term.

Correctness and completeness of $\mathscr{S}$ are much simpler if, instead of EAL, we formulate proofs and results with reference to an equivalent natural deduction formulation.

3.1.4 The full algorithm

In the following n, n_1, n_2 are always fresh variables, o is the base type. Moreover, we consider $!^{n_1}(!^{n_2}\Theta)$ syntactically equivalent to $!^{n_1+n_2}\Theta$.

NOTATION
Given a set of linear constraints A and a solution X of A, for any general EAL-type Θ and for any base $B = \{x_1 : \Theta_1, \ldots, x_n : \Theta_n\}$, we denote with $X(\Theta)$ the instantiation of Θ with X and with $X(B)$ the instantiation of B with X, i.e. $X(B) = \{x_1 : X(\Theta_1), \ldots, x_n : X(\Theta_n)\}$.

Unification: $\mathscr{U}$

Unification takes a set of $h \geq 2$ general EAL-types having the same underlying intuitionistic shape and returns a set of linear equations A such that for any solution X of A, the instantiations of the h general EAL-types are syntactically identical.

$$\frac{}{\mathscr{U}(!^{\sum n_{i_1}} o, \dots !^{\sum n_{i_h}} o) = \begin{cases} \sum n_{i_1} - \sum n_{i_2} = 0 \\ \vdots \\ \sum n_{i_{h-1}} - \sum n_{i_h} = 0 \end{cases}} \tag{3.38}$$

$$\frac{\mathscr{U}(\Theta_{1_1}, \dots, \Theta_{1_h}) = A_1 \qquad \mathscr{U}(\Theta_{2_1}, \dots, \Theta_{2_h}) = A_2}{\mathscr{U}\left(\begin{array}{l} !^{\sum n_{i_1}}(\Theta_{1_1} \multimap \Theta_{2_1}), \\ \vdots, \\ !^{\sum n_{i_h}}(\Theta_{1_h} \multimap \Theta_{2_h}) \end{array} \right) = \begin{cases} \sum n_{i_1} - \sum n_{i_2} = 0 \\ \vdots \\ \sum n_{i_{h-1}} - \sum n_{i_h} = 0 \\ A_1 \\ A_2 \end{cases}} \tag{3.39}$$

Contraction ($\mathcal{C}$) and Type Processing ($\mathscr{P}$)

Contraction in EAL is allowed only for exponential formulas. Thus, given k general EAL-types, $\mathcal{C}$ returns a the same set of constraints of $\mathscr{U}$ with the additional constraint that the number of external ! must be greater than zero.

$$\frac{}{\mathcal{C}(\Theta) = \emptyset} \tag{3.40}$$

$$\frac{\mathscr{U}(!^{n_1+\cdots+n_h}\Theta_1, \Theta_2, \dots, \Theta_k) = A}{\mathcal{C}(!^{n_1+\cdots+n_h}\Theta_1, \dots, \Theta_k) = \begin{cases} n_1 + \cdots + n_h \geq 1 \\ A \end{cases}} \tag{3.41}$$

Given a simple type τ, $\mathscr{P}$ returns the most general EAL-type whose cancellation is τ simply adding everywhere p exponentials (every p is a fresh variable).

$$\frac{}{\mathscr{P}(o) = !^p o} \tag{3.42}$$

$$\frac{\mathscr{P}(\sigma) = \Theta \quad \mathscr{P}(\tau) = \Gamma}{\mathscr{P}(\sigma \to \tau) = !^p(\Theta \multimap \Gamma)} \tag{3.43}$$

Boxing: $\mathscr{B}$ and $\mathbb{B}$

$$\frac{}{\mathscr{B}(B, \Gamma, \emptyset, A) = \langle B, \Gamma, A \rangle} \tag{3.44}$$

$$\frac{\begin{array}{l} \mathscr{B}(B_1, !^c\Gamma, cpts, A_2) = \langle B, \Delta, A_1 \rangle \\ B_1 = \left\{ x_i : \begin{cases} !^c\Theta_i & x_i \notin sl \\ \Theta_i & x_i \in sl \end{cases} \right\}_i \\ A_2 = \left(\begin{cases} A^j & A^j \notin sl \\ A^j - c & A^j \in sl \end{cases} \right)_j \end{array}}{\mathscr{B}(\{x_i : \Theta_i\}_i, \Gamma, \{sl\} \cup cpts, A) = \langle B, \Delta, A_1 \rangle} \tag{3.45}$$

$$\frac{}{\mathbb{B}(x, B, \Gamma, cpts, A) = \langle B, \Gamma, A\rangle} \tag{3.46}$$

$$\frac{\mathscr{B}(B, \Gamma, cpts, A) = \langle B_1, \Gamma_1, A_1\rangle}{\mathbb{B}(M, B, \Gamma, cpts, A) = \langle !^b B_1, !^b\Gamma_1, A_1\rangle} \tag{3.47}$$

Proposition 53 *Let $b, c_1, \ldots, c_k$ be the fresh variables introduced by $\mathbb{B}(M, B, \Gamma, cpts, A) = \langle !^b B_1, !^b\Gamma_1, A_1\rangle$ and let X be a solution of A, then*

1. *$X_1 = (X, b = 0, c_1 = 0, \ldots, c_k = 0)$ is a solution of A_1;*
2. *$X_1(\Gamma_1) = X(\Gamma)$;*
3. *$X_1(B_1) = X(B)$.*

Product union: $\Cup$

$$\frac{}{\emptyset \Cup cpts = cpts} \tag{3.48}$$

$$\frac{}{cpts \Cup \emptyset = cpts} \tag{3.49}$$

$$\frac{\begin{Bmatrix} sl_{2_1} \\ \vdots \\ sl_{n_1} \end{Bmatrix} \Cup \begin{Bmatrix} sl_{1_2} \\ \vdots \\ sl_{n_2} \end{Bmatrix} = cpts}{\begin{Bmatrix} sl_{1_1} \\ \vdots \\ sl_{n_1} \end{Bmatrix} \Cup \begin{Bmatrix} sl_{1_2} \\ \vdots \\ sl_{n_2} \end{Bmatrix} = \{sl_{1_1}, sl_{1_1} \cup sl_{1_2}, \ldots, sl_{1_1} \cup sl_{n_2}\} \cup cpts} \tag{3.50}$$

Type synthesis: $\mathcal{S}$

Let $\neg app(M) \leftrightarrow \nexists M_1, M_2$ s.t. $M = (M_1\ M_2)$.

$$\frac{\mathscr{P}(\sigma) = \Theta}{\mathcal{S}(x : \sigma) = \langle \Theta, \{x : \Theta\}, \emptyset, \emptyset\rangle} \tag{3.51}$$

$$\frac{\begin{array}{l} h \geq 1 \\ \mathcal{C}(\Theta_1, \ldots, \Theta_h) = A_3 \\ \mathbb{B}\left(M, B_1, \Gamma_1, cpts \cup \begin{Bmatrix} sl_1(x) \\ \vdots \\ sl_k(x) \end{Bmatrix}, A_1\right) = \left\langle B \cup \begin{Bmatrix} x : \Theta_1 \\ \vdots \\ x : \Theta_h \end{Bmatrix}, \Gamma, A_2 \right\rangle \\ \mathcal{S}(M : \tau) = \langle \Gamma_1, B_1, A_1, cpts \cup \{sl_1(x), \ldots, sl_k(x)\}\rangle \end{array}}{\mathcal{S}(\lambda x.M : \sigma \to \tau) = \left\langle \Theta_1 \multimap \Gamma, B, \left\{ \begin{array}{c} A_2 \\ A_3 \end{array} \right., cpts \right\rangle} \tag{3.52}$$

$$\frac{\begin{array}{l} x \notin \texttt{FV}((M_1\ M_2)) \\ cpts = cpts_1 \cup \{(\sum n_i - n = 0, \texttt{FVO}(M_1\ M_2))\} \\ \mathscr{P}(\sigma) = \Theta \\ \mathbb{B}\left((M_1\ M_2), B_1, \Gamma_1, cpts_1, A_1\right) = \left\langle B, !^{\sum n_i}\Gamma, A\right\rangle \\ \mathcal{S}((M_1\ M_2) : \tau) = \langle \Gamma_1, B_1, A_1, cpts_1\rangle \end{array}}{\mathcal{S}(\lambda x.(M_1\ M_2) : \sigma \to \tau) = \left\langle \Theta \multimap !^n\Gamma, B, \left\{ \begin{array}{c} A \\ \sum n_i - n = 0 \end{array} \right., cpts \right\rangle} \tag{3.53}$$

$$\frac{\begin{array}{l} \neg app(M) \\ x \notin \texttt{FV}(M) \\ \mathscr{P}(\sigma) = \Theta \\ \mathbb{B}\left(M, B_1, \Gamma_1, cpts, A_1\right) = \langle B, \Gamma, A\rangle \\ \mathcal{S}(M : \tau) = \langle \Gamma_1, B_1, A_1, cpts\rangle \end{array}}{\mathcal{S}(\lambda x.M : \sigma \to \tau) = \langle \Theta \multimap \Gamma, B, A, cpts\rangle} \tag{3.54}$$

$$\frac{\begin{array}{l} \neg app(M)\ \wedge\ \neg app(N) \\ \mathscr{U}(\Theta_1, \Theta_3) = A_4 \\ \mathbb{B}(N, B_2, \Theta_2, cpts_2, A_2) = \langle B_3, \Theta_3, A_3\rangle \\ \mathcal{S}(N : \sigma) = \langle \Theta_2, B_2, A_2, cpts_2\rangle \\ \mathcal{S}(M : \sigma \to \tau) = \langle !^{\sum n_i}(\Theta_1 \multimap \Gamma), B_1, A_1, cpts_1\rangle \end{array}}{\mathcal{S}((M\ N) : \tau) = \left\langle \Gamma, B_1 \cup B_3, \left\{ \begin{array}{c} A_1 \\ A_3 \\ A_4 \\ \sum n_i = 0 \end{array} \right., cpts_1 \Cup cpts_2 \right\rangle} \tag{3.55}$$

$$\frac{\begin{array}{l} \neg app(M) \\ cpts = cpts_1 \Cup \left(cpts_2 \cup \{(A_4^1, \texttt{FVO}((N_1\ N_2)))\}\right) \\ \mathscr{U}(\Theta_3, \Theta_1) = A_4 \\ \mathbb{B}((N_1\ N_2), B_2, \Theta_2, cpts_2, A_2) = \langle B_3, \Theta_3, A_3\rangle \\ \mathcal{S}((N_1\ N_2) : \sigma) = \langle \Theta_2, B_2, A_2, cpts_2\rangle \\ \mathcal{S}(M : \sigma \to \tau) = \langle !^{\sum n_i}(\Theta_1 \multimap \Gamma), B_1, A_1, cpts_1\rangle \end{array}}{\mathcal{S}((M\ (N_1\ N_2)) : \tau) = \left\langle \Gamma, B_1 \cup B_3, \left\{ \begin{array}{c} A_1 \\ A_3 \\ A_4 \\ \sum n_i = 0 \end{array} \right., cpts \right\rangle} \tag{3.56}$$

Notice that A_4^1 indicates the equality constraints between the outermost number of ! in the type of $(N_1\ N_2)$ and in the function part of the type of M.

$$\frac{\begin{array}{l} \neg app(N) \\ cpts = (cpts_1 \cup \{(\sum n_i = 0, \texttt{FVO}((M_1\ M_2)))\}) \Cup cpts_2 \\ \mathscr{U}(\Theta_1, \Theta_3) = A_4 \\ \mathbb{B}(N, B_2, \Theta_2, cpts_2, A_2) = \langle B_3, \Theta_3, A_3\rangle \\ \mathcal{S}(N : \sigma) = \langle \Theta_2, B_2, A_2, cpts_2\rangle \\ \mathcal{S}((M_1\ M_2) : \sigma \to \tau) = \langle !^{\sum n_i}(\Theta_1 \multimap \Gamma), B_1, A_1, cpts_1\rangle \end{array}}{\mathcal{S}((M_1\ M_2)\ N) : \tau) = \left\langle \Gamma, B_1 \cup B_3, \left\{ \begin{array}{c} A_1 \\ A_3 \\ A_4 \\ \sum n_i = 0 \end{array} \right., cpts \right\rangle} \tag{3.57}$$

$$
\frac{\begin{array}{l}
cpts_4 = cpts_2 \cup \{(A_4^1, \texttt{FVO}((N_1\ N_2)))\} \\
cpts_3 = cpts_1 \cup \{(\sum n_i = 0, \texttt{FVO}((M_1\ M_2)))\} \\
\mathscr{U}(\Theta_3, \Theta_1) = A_4 \\
\mathbb{B}((N_1\ N_2), B_2, \Theta_2, cpts_2, A_2) = \langle B_3, \Theta_3, A_3 \rangle \\
\mathcal{S}((N_1\ N_2) : \sigma) = \langle \Theta_2, B_2, A_2, cpts_2 \rangle \\
\mathcal{S}((M_1\ M_2) : \sigma \to \tau) = \langle !^{\sum n_i}(\Theta_1 \multimap \Gamma), B_1, A_1, cpts_1 \rangle
\end{array}}{\mathcal{S}((M_1\ M_2)\ (N_1\ N_2)) : \tau) = \left\langle \Gamma, B_1 \cup B_3, \left\{ \begin{array}{c} A_1 \\ A_3 \\ A_4 \\ \sum n_i = 0 \end{array} \right. , cpts_3 \Cup cpts_4 \right\rangle} \tag{3.58}
$$

Type synthesis algorithm: $\mathscr{S}$

$\mathscr{S}$ simply box the term, forgets the set of critical points and eventually contracts common variables in the base.

$$
\frac{\begin{array}{l}
\mathcal{C}(\Theta_{1_1}, \ldots, \Theta_{k_1}) = A_1 \qquad \ldots \qquad \mathcal{C}(\Theta_{1_h}, \ldots, \Theta_{k_h}) = A_h \\
\mathbb{B}(M, B_1, \Theta_1, cpts, A') = \left\langle \left\{ \begin{array}{c} x_1 : \Theta_{1_1}, \ldots, x_1 : \Theta_{k_1}, \\ \vdots \\ x_h : \Theta_{1_h}, \ldots, x_h : \Theta_{k_h} \end{array} \right\}, \Theta, A \right\rangle \\
\mathcal{S}(M : \sigma) = \langle \Theta_1, B_1, A', cpts \rangle
\end{array}}{\mathscr{S}(M : \sigma) = \left\langle \Theta, \{x_1 : \Theta_{1_1}, x_2 : \Theta_{1_2}, \ldots, x_h : \Theta_{1_h}\}, \left\{ \begin{array}{c} A \\ A_1 \\ \vdots \\ A_h \end{array} \right. \right\rangle} \tag{3.59}
$$

Example 1 *We have already seen in Chapter 2 that the simply typed lambda term*

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

is not typeable in EAL. Here follows the trace of the type synthesis algorithm executed on such a term:

$\mathcal{S}(\lambda n.(n\ \lambda y.(n\ \lambda z.y))\ \lambda x.(x\ (x\ y))) : o)$

$\mathcal{S}(\lambda n.(n\ \lambda y.(n\ \lambda z.y)) : ((o \to o) \to o) \to o)$

$\mathcal{S}((n\ \lambda y.(n\ \lambda z.y)) : o)$

$\mathcal{S}(n : (o \to o) \to o)$

$\mathscr{P}((o \to o) \to o) = p_1(p_2(p_3 \multimap p_4) \multimap p_5)$

$= \langle p_1(p_2(p_3 \multimap p_4) \multimap p_5), \{n : p_1(p_2(p_3 \multimap p_4) \multimap p_5)\}, \emptyset, \emptyset \rangle$

$\mathcal{S}(\lambda y.(n\lambda z.y) : o \to o)$

$\mathcal{S}((n\ \lambda z.y) : o)$

$\mathcal{S}(n : (o \to o) \to o)$

$$\mathscr{P}((o \to o) \to o) = p_6(p_7(p_8 \multimap p_9) \multimap p_{10})$$
$$= \langle p_6(p_7(p_8 \multimap p_9) \multimap p_{10}), \{n : p_6(p_7(p_8 \multimap p_9) \multimap p_{10})\}, \emptyset, \emptyset\rangle$$

$$\mathcal{S}(\lambda z.y : o \to o)$$
$$\mathcal{S}(y : o)$$
$$\mathscr{P}(o) = p_{11}$$
$$= \langle p_{11}, \{y : p_{11}\}, \emptyset, \emptyset\rangle$$
$$\mathbb{B}(y, \{y : p_{11}\}, p_{11}, \emptyset, \emptyset) = \langle \{y : p_{11}\}, p_{11}, \emptyset\rangle$$
$$\mathscr{P}(\alpha) = p_{12}$$
$$= \langle p_{12} \multimap p_{11}, \{y : p_{11}\}, \emptyset, \emptyset\rangle$$
$$\mathbb{B}(\lambda z.y, \{y : p_{11}\}, p_{12} \multimap p_{11}, \emptyset, \emptyset)$$
$$\mathscr{B}(\{y : p_{11}\}, p_{12} \multimap p_{11}, \emptyset, \emptyset) = \langle \{y : p_{11}\}, p_{12} \multimap p_{11}, \emptyset\rangle$$
$$= \langle \{y : b_1 + p_{11}\}, b_1(p_{12} \multimap p_{11}), \emptyset\rangle$$
$$\mathscr{U}(p_7(p_8 \multimap p_9), b_1(p_{12} \multimap p_{11})) = \begin{cases} p_7 = b_1 \\ p_8 = p_{12} \\ p_9 = p_{11} \end{cases}$$
$$= \left\langle p_{10}, \{n : p_6(p_7(p_8 \multimap p_9) \multimap p_{10}), y : b_1 + p_{11}\}, \begin{cases} p_7 = b_1 \\ p_8 = p_{12} \\ p_9 = p_{11} \\ p_6 = 0 \end{cases}, \emptyset \right\rangle$$
$$= \langle p_{10}, \{n : b_1(p_8 \multimap p_9) \multimap p_{10}, y : b_1 + p_9\}, \emptyset, \emptyset\rangle$$
$$\mathbb{B}((n\ \lambda z.y), \{n : b_1(p_8 \multimap p_9) \multimap p_{10}, y : b_1 + p_9\}, p_{10}, \emptyset, \emptyset)$$
$$= \langle \{n : b_2(b_1(p_8 \multimap p_9) \multimap p_{10}), y : b_2 + b_1 + p_9\}, b_2 + p_{10}, \emptyset\rangle$$
$$\mathcal{C}(b_2 + b_1 + p_9) = \emptyset$$
$$= \langle b_2 + b_1 + p_9 \multimap b_2 + p_{10}, \{n : b_2(b_1(p_8 \multimap p_9) \multimap p_{10})\}, \emptyset, \emptyset\rangle$$
$$\mathbb{B}(\lambda y.(n\ \lambda z.y), \{n : b_2(b_1(p_8 \multimap p_9) \multimap p_{10})\}, b_2 + b_1 + p_9 \multimap b_2 + p_{10}, \emptyset, \emptyset)$$
$$= \langle \{n : b_3 + b_2(b_1(p_8 \multimap p_9) \multimap p_{10})\}, b_3(b_2 + b_1 + p_9 \multimap b_2 + p_{10}), \emptyset\rangle$$
$$\mathscr{U}(p_2(p_3 \multimap p_4), b_3(b_2 + b_1 + p_9 \multimap b_2 + p_{10})) = \begin{cases} p_2 = b_3 \\ p_3 = b_2 + b_1 + p_9 \\ p_4 = b_2 + p_{10} \end{cases}$$
$$= \left\langle p_5, \left\{ \begin{array}{l} n : p_1(p_2(p_3 \multimap p_4) \multimap p_5), \\ n : b_3 + b_2(b_1(p_8 \multimap p_9) \multimap p_{10}) \end{array} \right\}, \begin{cases} p_2 = b_3 \\ p_3 = b_2 + b_1 + p_9 \\ p_4 = b_2 + p_{10} \\ p_1 = 0 \end{cases}, \emptyset \right\rangle$$
$$= \left\langle p_5, \left\{ \begin{array}{l} n : b_3(b_2 + b_1 + p_9 \multimap b_2 + p_{10}) \multimap p_5, \\ n : b_3 + b_2(b_1(p_8 \multimap p_9) \multimap p_{10}) \end{array} \right\}, \emptyset, \emptyset \right\rangle$$
$$\mathbb{B}\left((n\ \lambda y.(n\ \lambda z.y)), \left\{ \begin{array}{l} n : b_3(b_2 + b_1 + p_9 \multimap b_2 + p_{10}) \multimap p_5, \\ n : b_3 + b_2(b_1(p_8 \multimap p_9) \multimap p_{10}) \end{array} \right\}, p_5, \emptyset, \emptyset \right)$$
$$= \left\langle \left\{ \begin{array}{l} n : b_4(b_3(b_2 + b_1 + p_9 \multimap b_2 + p_{10}) \multimap p_5), \\ n : b_4 + b_3 + b_2(b_1(p_8 \multimap p_9) \multimap p_{10}) \end{array} \right\}, b_4 + p_5, \emptyset \right\rangle$$
$$\mathcal{C}(b_4(b_3(b_2 + b_1 + p_9 \multimap b_2 + p_{10}) \multimap p_5), b_4 + b_3 + b_2(b_1(p_8 \multimap p_9) \multimap p_{10}))$$
$$= \begin{cases} b_4 \geq 1 \\ b_4 = b_4 + b_3 + b_2 \\ b_3 = b_1 \\ b_2 + b_1 + p_9 = p_8 \\ b_2 + p_{10} = p_9 \\ p_5 = p_{10} \end{cases} = \begin{cases} b_4 \geq 1 \\ b_3 = 0 \\ b_2 = 0 \\ b_1 = 0 \\ p_8 = p_5 \\ p_9 = p_5 \\ p_{10} = p_5 \end{cases}$$
$$= \langle b_4((p_5 \multimap p_5) \multimap p_5) \multimap b_4 + p_5, \emptyset, \{b_4 \geq 1\}, \emptyset\rangle$$

$$
\begin{array}{l}
\mathcal{S}(\lambda x.(x\ (x\ y)) : (o \to o) \to o) \\
\quad \mathcal{S}((x\ (x\ y)) : o) \\
\quad\quad \mathcal{S}(x : o \to o) \\
\quad\quad\quad = \langle p_1(p_2 \multimap p_3), \{x : p_1(p_2 \multimap p_3)\}, \emptyset, \emptyset\rangle \\
\quad\quad \mathcal{S}((x\ y) : o) \\
\quad\quad\quad \mathcal{S}(x : o \to o) \\
\quad\quad\quad\quad = \langle p_4(p_6 \multimap p_7), \{x : p_4(p_6 \multimap p_7)\}, \emptyset, \emptyset\rangle \\
\quad\quad\quad \mathcal{S}(y : o) \\
\quad\quad\quad\quad = \langle p_8, \{y : p_8\}, \emptyset, \emptyset\rangle \\
\quad\quad\quad \mathscr{U}(p_6, p_8) = \{p_6 = p_8\} \\
\quad\quad\quad = \langle p_7, \{x : p_6 \multimap p_7, y : p_6\}, \emptyset, \emptyset\rangle \\
\quad\quad \mathbb{B}((x\ y), \{x : p_6 \multimap p_7, y : p_6\}, p_7, \emptyset, \emptyset) \\
\quad\quad\quad = \langle \{x : b_1(p_6 \multimap p_7), y : b_1 + p_6\}, b_1 + p_7, \emptyset\rangle \\
\quad\quad \mathscr{U}(b_1 + p_7, p_2) = \{b_1 + p_7 - p_2 = 0\} \\
\quad\quad cpts = \left\{ \left(b_1 + p_7 - p_2 = 0, \left\{ \begin{array}{l} x : b_1(p_6 \multimap p_7), \\ y : b_1 + p_6 \end{array} \right\} \right) \right\} \\
\quad\quad = \left\langle p_3, \left\{ \begin{array}{l} x : p_2 \multimap p_3, \\ x : b_1(p_6 \multimap p_7), \\ y : b_1 + p_6 \end{array} \right\}, \{b_1 + p_7 - p_2 = 0\}, cpts \right\rangle \\
\quad \mathbb{B}\left((x\ (x\ y)), \left\{ \begin{array}{l} x : p_2 \multimap p_3, \\ x : b_1(p_6 \multimap p_7), \\ y : b_1 + p_6 \end{array} \right\}, p_3, cpts, \{b_1 + p_7 - p_2 = 0\} \right) \\
\quad\quad \mathscr{B}\left(\left\{ \begin{array}{l} x : p_2 \multimap p_3, \\ x : b_1(p_6 \multimap p_7), \\ y : b_1 + p_6 \end{array} \right\}, p_3, cpts, \{b_1 + p_7 - p_2 = 0\} \right) \\
\quad\quad\quad = \left\langle \left\{ \begin{array}{l} x : b_2(p_2 \multimap p_3), \\ x : b_1(p_6 \multimap p_7), \\ y : b_1 + p_6 \end{array} \right\}, b_2 + p_3, \{b_1 + p_7 - p_2 - b_2 = 0\} \right\rangle \\
\quad\quad = \left\langle \left\{ \begin{array}{l} x : b_3 + b_2(p_2 \multimap p_3), \\ x : b_3 + b_1(p_6 \multimap p_7), \\ y : b_3 + b_1 + p_6 \end{array} \right\}, b_3 + b_2 + p_3, \{b_1 + p_7 - p_2 - b_2 = 0\} \right\rangle \\
\quad \mathcal{C}(b_3 + b_2(p_2 \multimap p_3), b_3 + b_1(p_6 \multimap p_7)) = \left\{ \begin{array}{l} b_3 + b_2 \geq 1 \\ b_3 + b_2 = b_3 + b_1 \\ p_2 = p_6 \\ p_3 = p_7 \end{array} \right. = \left\{ \begin{array}{l} b_3 + b_2 \geq 1 \\ b_2 = b_1 \\ p_2 = p_6 \\ p_3 = p_7 \end{array} \right. \\
\quad = \left\langle \begin{array}{lll} b_3 + b_2(p_2 \multimap p_3) \multimap b_3 + b_2 + p_3, & \{y : b_3 + b_1 + p_6\}, & \\ \left\{ \begin{array}{l} b_1 + p_7 - p_2 - b_2 = 0 \\ b_3 + b_2 \geq 1 \\ b_2 = b_1 \\ p_2 = p_6 \\ p_3 = p_7 \end{array} \right. & , & \emptyset \end{array} \right\rangle \\
\quad = \langle b_3 + b_1(p_2 \multimap p_2) \multimap b_3 + b_1 + p_2, \{y : b_3 + b_1 + p_2\}, \{b_3 + b_1 \geq 1\}, \emptyset\rangle \\
\mathbb{B}(\lambda x.(x\ (x\ y)), \{y : b_3 + b_1 + p_2\}, b_3 + b_1(p_2 \multimap p_2) \multimap b_3 + b_1 + p_2, \emptyset, \{b_3 + b_1 \geq 1\}) \\
\quad = \langle \{y : b_2 + b_3 + b_1 + p_2\}, b_2(b_3 + b_1(p_2 \multimap p_2) \multimap b_3 + b_1 + p_2), \{b_3 + b_1 \geq 1\}\rangle \\
\mathscr{U}(b_4((p_5 \multimap p_5) \multimap p_5, b_2(b_3 + b_1(p_2 \multimap p_2) \multimap b_3 + b_1 + p_2)) \\
\quad = \left\{ \begin{array}{l} b_4 = b_2 \\ p_5 = p_2 \\ p_5 = b_3 + b_1 + p_2 \end{array} \right. = \left\{ \begin{array}{l} b_4 = b_2 \\ p_5 = p_2 \\ b_3 + b_1 = 0 \end{array} \right.
\end{array}
$$

Notice that the last constraint $b_3 + b_1 = 0$ *is incompatible with the previous* $b_3 + b_1 \geq 1$ *hence the set of solutions is empty.*

3.1.5 Properties of the Type Syntesis Algorithm

The following Lemma states that any slice in the set of critical points bars the rest of the term.

Lemma 54 *Let* $\mathcal{S}(M : \sigma) = \langle \Theta, B, A, cpts \rangle$*. For any slice* sl *in* $cpts$*,* $sl = \{cpt_1, \ldots, cpt_k\}$*, for every path from the root of the syntax tree of* M *to any leaf, there exists at least one* cpt_i *in the path.*

> **Proof:** By induction on M. The unique interesting case is $M = (M_1\ M_2)$. The thesis holds by inductive hypothesis and by a simple inspection of rules for $\mathcal{S}$ and for the product union. □

The following lemma illustrates the relation between the set of critical points calculated by the algorithm for a given term M and a particular class of decompositions of M.

Lemma 55 *Let* $\mathcal{S}(M : \sigma) = \langle \Theta, B, A, cpts \rangle$*.*

1. $\forall \{cpt_1, \ldots, cpt_k\} = sl \in cpts$ *there exist* $P, (N_{1_1}\ N_{2_1}), \ldots, (N_{1_k}\ N_{2_k})$ *such that* P *is not a variable,* $x_1, \ldots, x_k \in \mathtt{FV}(P)$ *and* $M = P\{(N_{1_1}\ N_{2_1})/x_1, \ldots, (N_{1_k}\ N_{2_k})/x_k\}$*;*
2. $\forall P, (N_{1_1}\ N_{2_1}), \ldots, (N_{1_k}\ N_{2_k})$ *such that* P *is not a variable,* $x_1, \ldots, x_k \in \mathtt{FV}(P)$ *and* $M = P\{(N_{1_1}\ N_{2_1})/x_1, \ldots, (N_{1_k}\ N_{2_k})/x_k\}$*, there exists* $\{cpt_1, \ldots, cpt_k\} = sl \in cpts$ *such that* cpt_i *is the critical point at the root of* $(N_{1_i}\ N_{2_i})$*.*

> **Proof:** By structural induction on M.
>
> 1. If M is a variable, the thesis trivially holds being $cpts = \emptyset$. If $M = \lambda x.M'$, either sl consists of a single critical point corresponding to the root of M', then $P = \lambda x.y$, or sl is a slice of M', then by inductive hypothesis there exists P' s.t. the thesis holds for M'. We take $P = \lambda x.P'$. Finally if $M = (M_1\ M_2)$, if in sl there is a critical point cpt_i corresponding to the root of M_1 then by Lemma 54 all the other critical points in sl belong to M_2 or there is only one critical point corresponding to the root of M_2. In the first case by inductive hypothesis there exists P_2 s.t. the thesis holds for M_2 and sl without cpt_i. Then we take $P = (y\ P_2)$. The other cases are analogous.
> 2. If M is a variable then $\not\exists P$ and the thesis trivially holds. If $M = \lambda x.M'$ then $P = \lambda x.P'$. If P' is a variable, then the slice to consider is the one containing only the critical point corresponding to the root of M'. Such a slice has been added to $cpts$ in the rule for $\mathcal{S}(\lambda x.(M_1\ M_2) : \sigma)$ where $x \notin \mathtt{FV}((M_1\ M_2))$. Otherwise the thesis holds by inductive hypothesis. Finally if $M = (M_1\ M_2)$, then $P = (P_1\ P_2)$. If both P_1 and P_2 are not a variable, then by inductive hypothesis there exists sl_1 and sl_2. Then the thesis holds by definition of product union. The other cases are analogous.
>
> □

Consider the *length* $L(M)$ of an EAL-term M defined inductively:

$$\begin{aligned}
L(x) &= 0\\
L(\lambda x.M) &= 1 + L(M)\\
L((M\ N)) &= 1 + L(M) + L(N)\\
L(!\,(M)\,[M_1/x_1, \ldots, M_n/x_n]) &= L(M) + \sum_{i=1}^{n} L(M_i)\\
L(\|M\|^N_{x,y}) &= L(M) + L(N).
\end{aligned}$$

Definition 32 *An EAL-term M is* simple *if and only if*

1. *M has no subterm of the form $\|M_1\|^{M_2}_{x,y}$ where $(M_2)^*$ is not a variable,*
2. *$L(M) = L((M)^*)$*

Fact 56 *A simple EAL-term contracts at most variables.*

Definition 33 *The set of* candidate EAL-terms *is the set of all EAL-terms P such that*

1. *P is in $\{!\text{–}!, @ - \mathsf{c}, ! - \mathsf{c}, \mathsf{c} - \mathsf{c}, \lambda - \mathsf{c}, \mathsf{dup}\}$-normal form;*
2. *P is simple;*
3. *if $\|R\|^Q_{x,y}$ is a subterm of P, then $x, y \in \mathtt{FV}(R)$;*
4. *if $!\,(R)\,[Q_1/x_1, \ldots, Q_k/x_k]$ is a subterm of P, then R is not a variable.*

Lemma 57 *For any Θ general EAL-type there exists X s.t. $X(\mathscr{P}(\overline{\Theta})) = \Theta$.*

Theorem 58 (Completeness) *Let $\Gamma \vdash_{\mathsf{NEAL}} P : \Psi$ and let P be a candidate EAL-term. Let $\mathscr{S}(P^* : \overline{\Psi}) = \langle \Theta, B, A\rangle$, then there exists X integer solution of A such that $X(B) \subseteq \Gamma$, $\Psi = X(\Theta)$ and $X(B) \vdash_{\mathsf{NEAL}} P : X(\Theta)$.*

> **Proof:** The request on the $\{!\text{–}!, @ - \mathsf{c}, ! - \mathsf{c}, \mathsf{c} - \mathsf{c}, \lambda - \mathsf{c}, \mathsf{dup}\}$-normal form is not a loss of generality, for the subject reduction lemma and Proposition 49. By Lemma 50, the only restriction induced by the request of contracting at most variable is the exclusion of elementary terms with subterms of the form $\|R\|^{(Q_1\ Q_2)}_{x,y}$ or $!(R)[^{P_1}/x_1, \cdots, {}^{(Q_1\ Q_2)}/x, \cdots, {}^{P_n}/x_n]$ with $\|S\|^x_{y,z}$ subterm of R. In a sense, these terms "contract too much". Indeed, it could be the case that a term P is elementary thanks to the sharing of a β-redex (inside $(Q_1\ Q_2)$). However, the corresponding λ-term P^*, cannot share any redex—there is no sufficient syntax for this in the λ-calculus—hence P^* could be not elementary. We also do not take into account elementary affine terms with "false contractions". This is not a limitation by Lemma 42 and Theorem 51. Finally we discard term such $!(x)[M/x]$. Again this is not a limitation, in fact $(!(x)[M/x])^* = M^*$ and $\Gamma \vdash_{\mathsf{NEAL}} !(x)[M/x] :!\Psi$ if and only if $\Gamma \vdash_{\mathsf{NEAL}} M :!\Psi$. Our aim is to identify λ-terms that are reducible using optimal reduction without the *oracle* needed for the correct matching of fans. The NEAL terms excluded corresponds to EAL proof nets which are not (the initial encoding of) λ-terms, since they either contract an application or contract a variable introduced by a weakening or contains a superfluous number of exponentials.
>
> By induction on P.

- If $\Gamma, x : \Psi \vdash_{\mathsf{NEAL}} x : \Psi$ then $\mathcal{S}(x : \overline{\Psi}) = \langle \mathscr{P}(\overline{\Psi}), \{x : \mathscr{P}(\overline{\Psi})\}, \emptyset\rangle$ and the thesis holds by Lemma 57 being any X solution of the empty set of constraints.
- If the type derivation ends with

$$\frac{\Gamma \vdash_{\mathsf{NEAL}} x :!\Phi \quad \Delta, y :!\Phi, z :!\Phi \vdash_{\mathsf{NEAL}} N : \Psi}{\Gamma, \Delta \vdash_{\mathsf{NEAL}} \|N\|^{x}_{y,z} : \Psi}$$

then the thesis holds by inductive hypothesis on $\Delta, y :!\Phi, z :!\Phi \vdash_{\mathsf{NEAL}} N : \Psi$.

- If P is an abstraction then the type derivation is

$$\frac{\Gamma, x : \Psi \vdash_{\mathsf{NEAL}} M : \Phi}{\Gamma \vdash_{\mathsf{NEAL}} \lambda x.M : \Psi \multimap \Phi}$$

The thesis holds by inductive hypothesis. Notice that the solution X instantiates all variables introduced by the $\mathbb{B}$ call of the rule for $\mathcal{S}$ to 0. It is easy to see looking at the rules for $\mathbb{B}$ that if in the solution X there is one variable introduced by $\mathbb{B}$ that is not set to zero, then the type is exponential and $\Psi \multimap \Phi$ is not.

- If P is an application

$$\frac{\Gamma \vdash_{\mathsf{NEAL}} M : \Phi \multimap \Psi \quad \Delta \vdash_{\mathsf{NEAL}} N : \Phi}{\Gamma, \Delta \vdash_{\mathsf{NEAL}} (M\ N) : \Psi}$$

By inductive hypothesis there are solutions X_1 for M and X_2 for N. Now, by the same considerations of the previous point, X_1 sets all variables introduced by the last $\mathbb{B}$ call to 0. Thus the constraint $\sum n_j = 0$ of the rule for $\mathcal{S}$ is satisfied. Moreover X_1, X_2 satisfies the constraints for the unification of types, because they are identical by hypothesis. Hence the thesis holds.

- Finally, if the derivation is

$$\frac{\Delta_1 \vdash_{\mathsf{NEAL}} M_1 :!\Phi_1 \cdots \Delta_n \vdash_{\mathsf{NEAL}} M_n :!\Phi_n \quad x_1 : \Phi_1, \ldots, x_n : \Phi_n \vdash_{\mathsf{NEAL}} N : \Psi}{\Gamma, \Delta_1, \ldots, \Delta_n \vdash_{\mathsf{NEAL}} !\,(N)\,[M_1/x_1, \ldots, M_n/x_n] :!\Psi}$$

then by Lemma 50 either M_i is a variable or an application. If all M_i are variables, then the thesis holds getting the solution of the inductive hypothesis and increasing the variable b introduced by $\mathbb{B}$ by one.

If there is an M_i that is an application, then by Lemma 55 there is a critical point collected by the algorithm at the root of M_i. Then we take as solution X the union of the solutions obtained by inductive hypothesis with the variable introduced by $\mathbb{B}$ for the critical point corresponding to M_i increased by one.

□

Notation
We use

$$\frac{\Gamma \vdash M :!^n A \quad x : A \vdash N : B}{\Gamma \vdash !^n(N)[^M/x] :!^n B}$$

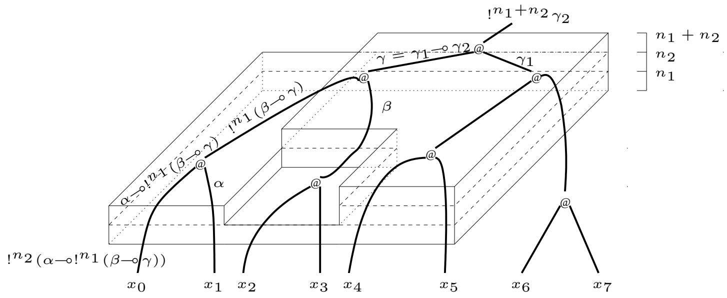


Figure 3.15: Boxes as levels.

as a shorthand for

$$\dfrac{\Gamma \vdash M : \overbrace{!\cdots!}^{n} A \qquad \dfrac{x_2 :!!A \vdash x_2 :!!A \qquad \dfrac{x_1 :!A \vdash x_1 :!A \quad x : A \vdash N : B}{x_1 :!A \vdash !(N)[^{x_1}/x] :!B}}{\begin{array}{c}\vdots\\ x_{n-1} : \overbrace{!\cdots!}^{n-1} : A \vdash \overbrace{!(\cdots!(}^{n-1} N)[^{x_1}/x]\cdots)[^{x_{n-1}}/x_{n-2}] : \overbrace{!\cdots!}^{n-1} B\end{array}}}{\Gamma \vdash \overbrace{!(\cdots!(}^{n} N)[^{x_1}/x]\cdots)[^{M}/x_{n-1}] : \overbrace{!\cdots!}^{n} B}$$

Lemma 59 (Superimposing of derivations) *Let $\mathscr{S}(M : \sigma) = \langle \Theta, B, A \rangle$ and let A be solvable. If there is a solution X_1 of A that instantiates two boxes belonging to two superimposed derivations that are not compatible, then there exists another solution X_2 where all the instantiated boxes belong to the same derivation.*

Moreover $X_1(\Theta) = X_2(\Theta)$ and $X_1(B) = X_2(B)$.

Proof: The proof of the lemma can be easily understood if we follow the intuition explained below with an example.

We may think of boxes as levels; boxing a subterm can then be seen as raising that subterm, as in Figure 3.15, where also some types label the edges of the syntax tree of a simple term. In particular, the edge starting from the @-node and ending in x_0 has label $!^{n_2}(\alpha \multimap !^{n_1}(\beta \multimap \gamma))$ at level 0 (nearest to x_0) and has label $(\alpha \multimap !^{n_1}(\beta \multimap \gamma))$ at level n_2. This is the graphical counterpart of the !-rule

$$\dfrac{\ldots, x_0 : T, \ldots \vdash \ldots}{\ldots, x_0 :!^{n_2}T, \ldots \vdash \ldots}\ !^{n_2}$$

The complete decoration of Figure 3.15 can be produced in NEAL in two ways: by the instantiation of

$$!^{n_2}\left((((x_0\ x_1)y)((x_4\ x_5)w))\right)[(x_2\ x_3)/y, (x_6\ x_7)/w]$$

and[9]

$$!^{n_1}\left(((z(x_2\ x_3))((x_4\ x_5)w))\right)[(x_0\ x_1)/z, (x_6\ x_7)/w],$$

[9]The correct legal terms should have all free variable inside the square brackets. We omit to write variables when they are just renamed, for readability reasons (compare the first elementary term above with the (fussy) correct one $!^{n_2}\left((((x_0\ x_1)y)((x_4\ x_5)w))\right)[x_0'/x_0, x_1'/x_1, (x_2\ x_3)/y, x_4'/x_4, x_5'/x_5, (x_6\ x_7)/w]$).

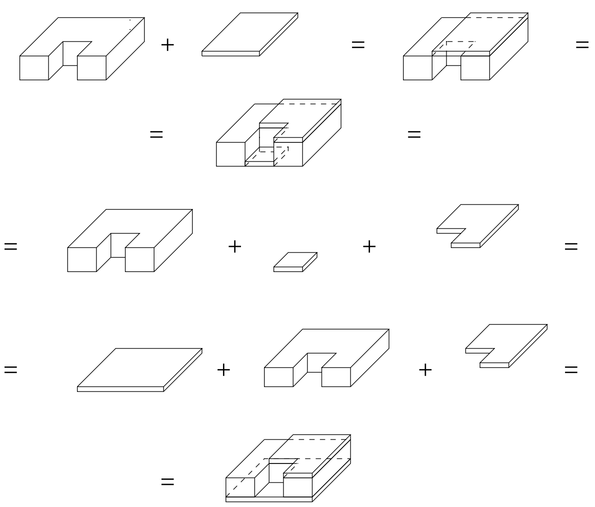


Figure 3.16: Equivalences of boxes.

which are boxes belonging to two different derivations. Graphically such an instantiation can be represented as in the first row of Figure 3.16, where incompatibility is evident by the fact that the boxes are not well stacked, in particular the rectangular one covers a hole. To have a correct EAL-derivation it is necessary to find the equivalent, well stacked configuration (that corresponds to the subsequent application of boxes from the topmost to the bottommost).

The procedure by which we find the well stacked box configuration is visualized in Figure 3.16. The reader may imagine the boxes subject to gravity (the passage from the first to the second row of Figure 3.16) and able to fuse each other when they are at the same level (the little square in the third row fuse with the solid at its left in the passage from the third to the fourth row).

The "gravity operator" corresponds to finding the minimal common subterm of all the superimposed derivations and it is useful for finding the correct order of application of the ! rule. The "fusion operator" corresponds to the elimination of a cut between two exponential formulas. Moreover, the final configuration of Figure 3.16 corresponds to a particular solution of the set of constraints produced by the type synthesis algorithm, that instantiates the following boxes:

$$!^{n_1}\left(!^{n_2-n_1}\left(!^{n_1}\left(((z\ y)((x_4\ x_5)w))\right)[(x_0\ x_1)/z]\right)[(x_2\ x_3)/y]\right)(x_6\ x_7)/w]$$

Finally, notice that during the procedure all types labelling the boundary edges of the lambda-term never changes, i.e. the instantiations of the term type (the label of the topmost edge) and the base types (the labels of the edges at the bottom) remain unchanged.

Now let $\mathcal{S}(M : \sigma) = \langle \Theta, B, A \rangle$ and let X be the solution that instantiates k overlapping—thus incompatible—boxes. Consider the boxed syntax tree of M and associate to any node its level, i.e. the number of boxes containing the node. Notice that if there is a wire connecting tho nodes a of level ℓ and b of level $\ell+k$, then the type labeling the wire is $!^k\Psi$ near a and Ψ near b, i.e. the sum of level and number of exponentials for types labeling the syntax tree is an invariant.

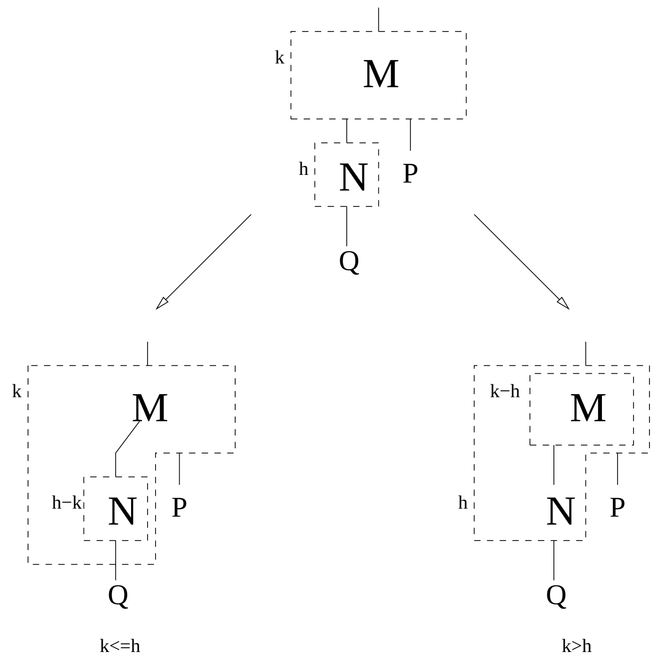


Figure 3.17: Fusion of boxes.

We break the boxes using the following procedure: starting from the root of the syntax tree of M, we are at level $i = 0$; we proceed with a breath first visit and whenever encounter a node of level $\ell \neq i$ we close i boxes, open ℓ boxes and set i to ℓ.

At the end of the procedure described above there are no more overlapping boxes, but it could be happen that there is a variable x not in the same boxes of its binding lambda node. Such configuration of boxes is not correct. However the level of the variable and lambda node is the same because the procedure of breaking boxes does not change level of nodes. Moreover all nodes belonging to the path from the lambda node to the variable have level higher or equal to the level of the variable since they all were initially in the same box and some of them were eventually also in some overlapping boxes that increase the level. Hence we can fuse boxes until variable and corresponding binder are in the same box. The fusion operation is shown in Figure 3.17 and described by the following equation:

$$
\begin{array}{ccl}
 & & !^k(M\{^{!^{h-k}(N)[^Q/z]}/x\})[^P/y] \text{ if } k \leq h \\
 & \nearrow & \\
!^k(M)[^P/y, {}^{!^h(N)[^Q/z]}/x] & & \\
 & \searrow & \\
 & & !^h(!^{k-h}(M)[^N/x])[^Q/z, {}^P/y] \text{ if } k > h
\end{array}
$$

After all fusions are performed, all variables are in the same boxes of their lambda binders and there are no more overlapping boxes, thus the decoration obtained corresponds to an EAL-derivation. By completeness exists X_2 solution corresponding to such decoration. Moreover types labeling the syntax tree are unchanged by the transformations applied, hence the thesis. □

Theorem 60 (Soundness) *Let $\mathscr{S}(M:\sigma) = \langle \Theta, B, A\rangle$. For every X integer solution of A, there exists P elementary affine term such that $P^* = M$ and $X(B) \vdash_{\mathsf{NEAL}} P : X(\Theta)$.*

Proof: By induction on the structure of M, using the superimposing lemma. We first need a definition:

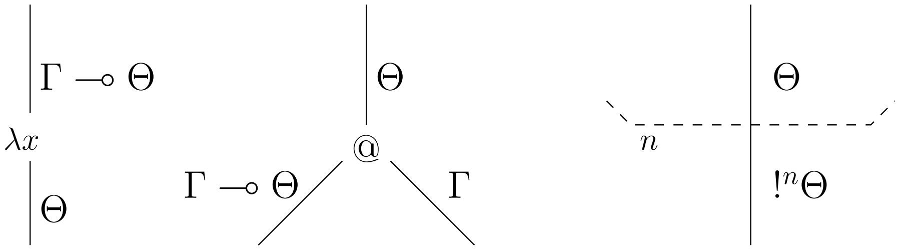


Figure 3.18: Type labels for decorated syntax trees.

Definition 34 *A syntax tree T is* correctly decorated *if the edges of the graph are labeled according to Figure 3.18 (in the rightmost picture, Θ is inside n boxes). Moreover all edges connecting a variable x occurring multiple, are labeled with the same type $!^n\Gamma$. In the case the variable is abstracted, the type label of variable is syntactically identical to the argument part of the type label of the edge at the root of the abstraction.*

Given a correctly decorated syntax tree, and an instantiation X for the general EAL-types labeling its edges such that the number of exponentials for types of multiple variables is greater than 1, it is easy to build the corresponding NEAL derivation, using the Curry-Howard isomorphism and eventually applying a contraction before the $\multimap$ introduction for binded variables and at the end of the derivation for free variables.

Thus, in order to prove soundness of our algorithm, it is sufficient to prove by structural induction on M that we can build a correctly decorated syntax tree. If the solution taken into account instantiates two overlapping boxes we use Lemma 59. Hence without loss of generality we can consider X such that all boxes are compatible. The only interesting part of the proof is the checking of

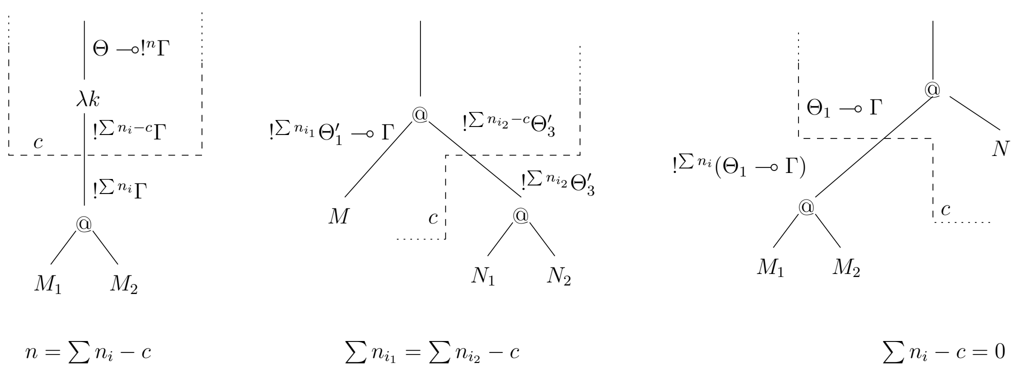


Figure 3.19: Decorations given by $\mathbb{B}$.

rules for $\mathbb{B}$. In Figure 3.19 it is shown how build a correctly decorated syntax tree when the solution X instantiates a box passing through a critical point (all three cases of critical points are depicted). $\square$

3.2 Principal type

We prove that it is possible to identify a principal type for any term in Λ^{EA}. The principal type algorithm produces an EAL-type with type and natural variables, and a set of linear constraints. All EAL-types for the elementary term can be obtained via substitutions of type and natural variables fulfilling the set of constraints. We can associate to any elementary term a *canonical form* that, intuitively, postpone contractions whenever is possible and collapses consecutive ! introduction rules. Finally we give a procedure to calculate the set of canonical forms of EAL corresponding to a given lambda term. Such a procedure, in combination with the principal type algorithm, allows to identify the *set of principal types* of a lambda term in EAL.

Definition 35 *Let $M \in \Lambda$. $\Gamma \vdash_{\mathsf{NEAL}} M : A$ if and only if $\Gamma \vdash_{\mathsf{NEAL}} R : A$ and $(R)^* = M$, for some $R \in \Lambda^{EA}$, R simple.*

As we have already mentioned in the previous section, in order to synthesize EAL-types for lambda terms, we do not consider EAL-derivations contracting subterms. Our goal is to identify lambda terms that are reducible using the abstract Lamping's algorithm. If we allow contraction of subterm, we should respect such contractions during the translation into sharing graphs. We are intentioned to investigate this possibility in the future.

Finally notice that we still allow weakening of subterms in the above definition.

3.2.1 Abstract EAL-terms

Before introducing the canonical forms, we need an intermediate language. The set Abs^{EA} of *abstract EAL-terms* is generated by the following grammar:

$$M ::= x \mid \lambda x.M \mid (M\ M) \mid [M]_{N_1 \to (\widehat{x_1}), \ldots, N_k \to (\widehat{x_k})} \mid \nabla(M)[^{N_1}/x_1, \cdots, {}^{N_k}/x_n]$$

with the condition that every variable occurs just once in a term ($\widehat{x_i}$ is a placeholder for $x_i^1, \ldots, x_i^{n_i}$).

Definition 36 *We define the following operations on abstract EAL-terms:*

1. *the set of* free variables $\mathtt{FV}$*:*

$$\mathtt{FV}(x) = \{x\} \tag{3.60}$$

$$\mathtt{FV}(\lambda x.M) = \mathtt{FV}(M) \setminus \{x\} \tag{3.61}$$

$$\mathtt{FV}((M_1\ M_2)) = \mathtt{FV}(M_1) \cup \mathtt{FV}(M_2) \tag{3.62}$$

$$\mathtt{FV}([M]_{Clist}) = (\mathtt{FV}(M) \setminus \mathtt{CV}(Clist)) \cup \mathtt{SSV}(Clist) \cup \mathtt{SFV}(Clist) \tag{3.63}$$

$$\mathtt{FV}(\nabla(M)[^{N_1}/x_1, \cdots, {}^{N_k}/x_k]) = \mathtt{FV}(M) \setminus \{x_1, \ldots, x_k\} \cup \bigcup_{i=1}^{k} \mathtt{FV}(N_i) \tag{3.64}$$

2. *the set of* contracted variables CV*:*

$$\texttt{CV}(M \to (x_1, \ldots, x_n)) = \{x_1, \ldots, x_n\} \tag{3.65}$$
$$\texttt{CV}(M \to (x_1, \ldots, x_n), \mathit{Slist}) = \{x_1, \ldots, x_n\} \cup \texttt{CV}(\mathit{Slist}) \tag{3.66}$$

3. *the set of* shared free variables SFV*:*

$$\texttt{SFV}(M) = \emptyset \tag{3.67}$$
$$\texttt{SFV}([M]_{\mathit{Slist}}) = \texttt{SFV}(M) \cup \texttt{SFV}(\mathit{Slist}) \tag{3.68}$$
$$\texttt{SFV}(x \to (x_1, \ldots, x_n)) = \emptyset \tag{3.69}$$
$$\texttt{SFV}(x \to (x_1, \ldots, x_n), \mathit{Slist}) = \texttt{SFV}(\mathit{Slist}) \tag{3.70}$$
$$\texttt{SFV}(M \to (x_1, \ldots, x_n)) = \texttt{FV}(M) \tag{3.71}$$
$$\texttt{SFV}(M \to (x_1, \ldots, x_n), \mathit{Slist}) = \texttt{FV}(M) \cup \texttt{SFV}(\mathit{Slist}) \tag{3.72}$$

where M in Equation (3.67) *is not a contraction (i.e. it is not of the form $[M']_{\mathit{Slist}}$) and M in equations* (3.71) *and* (3.72) *is not a variable;*

4. *the set of* single shared variables SSV*:*

$$\texttt{SSV}(M) = \emptyset \tag{3.73}$$
$$\texttt{SSV}([M]_{\mathit{Slist}}) = \texttt{SSV}(M) \cup \texttt{SSV}(\mathit{Slist}) \tag{3.74}$$
$$\texttt{SSV}(x \to (x_1, \ldots, x_n)) = \{x\} \tag{3.75}$$
$$\texttt{SSV}(x \to (x_1, \ldots, x_n), \mathit{Slist}) = \{x\} \cup \texttt{SSV}(\mathit{Slist}) \tag{3.76}$$
$$\texttt{SSV}(M \to (x_1, \ldots, x_n)) = \emptyset \tag{3.77}$$
$$\texttt{SSV}(M \to (x_1, \ldots, x_n), \mathit{Slist}) = \texttt{SSV}(\mathit{Slist}) \tag{3.78}$$

where M in Equation (3.73) *is not a contraction and M in equations* (3.77) *and* (3.78) *is not a variable;*

5. *the set of* shared terms ST*:*

$$\texttt{ST}(M) = \emptyset \tag{3.79}$$
$$\texttt{ST}([M]_{\mathit{Slist}}) = \texttt{ST}(\mathit{Slist}) \tag{3.80}$$
$$\texttt{ST}(M \to (x_1, \ldots, x_n)) = \{M\} \tag{3.81}$$
$$\texttt{ST}(M \to (x_1, \ldots, x_n), \mathit{Slist}) = \{M\} \cup \texttt{ST}(\mathit{Slist}) \tag{3.82}$$

where M in Equation (3.79) *is not a contraction;*

6. *the set of* banged variables BV*:*

$$\texttt{BV}(M/x) = \{x\} \tag{3.83}$$
$$\texttt{BV}(M/x, \mathit{Blist}) = \{x\} \cup \texttt{BV}(\mathit{Blist}) \tag{3.84}$$

7. *the set of* single banged variables SBV*:*

$$\texttt{SBV}(y/x) = \{x\} \tag{3.85}$$
$$\texttt{SBV}(M/x) = \emptyset \tag{3.86}$$
$$\texttt{SBV}(y/x, \mathit{Blist}) = \{x\} \cup \texttt{SBV}(\mathit{Blist}) \tag{3.87}$$
$$\texttt{SBV}(M/x, \mathit{Blist}) = \texttt{SBV}(\mathit{Blist}) \tag{3.88}$$

where M in equations (3.86) *and* (3.88) *is not a variable.*

On Abs^{EA} a reduction relation can be defined as follows.

Definition 37 *The reduction relation $\rightarrow_{\mathscr{C}an}$ on Abs^{EA} is the transitive and contextual closure of the following set of reduction relations $\{\rightarrow_{\nabla-collaps}, \rightarrow_{var-collaps}, \rightarrow_{c-collaps}, \rightarrow_{\nabla-\nabla}, \rightarrow_{\nabla-c}, \rightarrow_{c-\nabla}, \rightarrow_{c-c}, \rightarrow_{@-c-1}, \rightarrow_{@-c-2}, \rightarrow_{\lambda-c}, \rightarrow_{\nabla-\epsilon-1}, \rightarrow_{\nabla-\epsilon-2}, \rightarrow_{c-\epsilon-1}, \rightarrow_{c-\epsilon-2}, \rightarrow_{c-\epsilon-3}\}$, defined as follows:*

$$\nabla\big(\nabla(M)[y_1/x_1, \cdots, y_n/x_n]\big)[^{M_1}/y_1, \cdots, ^{M_n}/y_n] \rightarrow_{\nabla-collaps} \nabla(M)\left[^{M_1}/x_1, \ldots, ^{M_n}/x_n\right] \tag{3.89}$$

$$\big[[M]_{\ldots,x_i\rightarrow(y^i_1,\ldots,y^i_{n_i}),\ldots}\big]_{\ldots,N_j\rightarrow(z^j_1,\ldots,z^j_{k-1},x_i,z^j_{k+1},\ldots,z^j_{m_j}),\ldots} \rightarrow_{var-collaps} \big[[M]_{\ldots}\big]_{\ldots,N_j\rightarrow(z^j_1,\ldots,z^j_{k-1},y^i_1,\ldots,y^i_{n_i},z^j_{k+1},\ldots,z^j_{m_j}),\ldots} \tag{3.90}$$

Note: if the sharing list of the inner contraction becomes empty, then the square brackets are removed.

$$\big[[M]_{N_1\rightarrow(\widehat{x^1}),\ldots,N_i\rightarrow(x^i_1,\ldots,x^i_{n_i}),\ldots}\big]_{Slist} \rightarrow_{c-collaps} \big[[M]_{N_1\rightarrow(\widehat{x^1}),\ldots,N_{i-1}\rightarrow(\widehat{x^{i-1}}),N_{i+1}\rightarrow(\widehat{x^{i+1}}),\ldots}\big]_{Slist,N_i\rightarrow(x^i_1,\ldots,x^i_{n_i})}$$
$$\textit{if } \mathtt{FV}(N_i)\cap\mathtt{CV}(Slist)=\emptyset. \tag{3.91}$$

Note: if the sharing list of the inner contraction becomes empty, then the square brackets are removed.

$$\nabla(M)[^{M_1}/x_1, \cdots, ^{\nabla(N)\left[^{P_1}/y_1,\ldots,^{P_m}/y_m\right]}/x_i, \cdots, ^{M_n}/x_n] \rightarrow_{\nabla-\nabla} \nabla(M\{N/x_i\})[^{M_1}/x_1, \cdots, ^{P_1}/y_1, \cdots, ^{P_m}/y_m, \cdots ^{M_n}/x_n] \tag{3.92}$$

$$\nabla(M)[\cdots, ^{[M_i]_{N_1\rightarrow(\widehat{y^1}),\ldots,N_k\rightarrow(\widehat{y^k})}}/x_i, \cdots] \rightarrow_{\nabla-c} \left[\nabla(M)[\cdots, ^{M_i\{\widehat{z^1}/\widehat{y^1},\ldots,\widehat{z^k}/\widehat{y^k}\}}/x_i, \cdots]\right]_{N_1\rightarrow(\widehat{z^1}),\ldots,N_k\rightarrow(\widehat{z^k})} \tag{3.93}$$

$$\nabla([M]_{\ldots,x\rightarrow(\widehat{x}),\ldots})[\cdots, y/x, \cdots] \rightarrow_{c-\nabla} \big[\nabla([M]_{\ldots})[\cdots, \widehat{y}/\widehat{x}, \cdots]\big]_{y\rightarrow(\widehat{y})} \tag{3.94}$$

$$[M]_{\ldots,[N]_{P_1\rightarrow(\widehat{y^1}),\ldots,P_k\rightarrow(\widehat{y^k})}\rightarrow(\widehat{x^i}),\ldots} \rightarrow_{c-c} \left[[M]_{\ldots,N\{\widehat{z^1}/\widehat{y^1},\cdots,\widehat{z^k}/\widehat{y^k}\}\rightarrow(\widehat{x^i}),\ldots}\right]_{P_1\rightarrow(\widehat{z^1}),\ldots,P_k\rightarrow(\widehat{z^k})} \tag{3.95}$$

$$([M]_{M_1\rightarrow(\widehat{x^1}),\ldots,M_k\rightarrow(\widehat{x^k})}\ N) \rightarrow_{@-c-1} \left[(M\{\widehat{y^1}/\widehat{x^1}, \ldots, \widehat{y^k}/\widehat{x^k}\}\ N)\right]_{M_1\rightarrow(\widehat{y^1}),\ldots,M_k\rightarrow(\widehat{y^k})} \tag{3.96}$$

$$(M\ [N]_{N_1\rightarrow(\widehat{x^1}),\ldots,N_k\rightarrow(\widehat{x^k})}) \rightarrow_{@-c-2} \left[(M\ N\{\widehat{y^1}/\widehat{x^1}, \ldots, \widehat{y^k}/\widehat{x^k}\})\right]_{N_1\rightarrow(\widehat{y^1}),\ldots,N_k\rightarrow(\widehat{y^k})} \tag{3.97}$$

$$\lambda x.[M]_{Slist} \rightarrow_{\lambda-c} [\lambda x.M]_{Slist} \text{ where } x \notin \mathtt{SFV}(Slist) \cup \mathtt{SSV}(Slist) \tag{3.98}$$

$$\nabla(x)[^M/x] \rightarrow_{\nabla-\epsilon-1} M \tag{3.99}$$
$$\nabla(M)[\cdots, {}^N/x, \cdots] \rightarrow_{\nabla-\epsilon-2} \nabla(M)[\cdots] \text{ where } x \notin \mathtt{FV}(M) \tag{3.100}$$
$$[x]_{\ldots,N\rightarrow(\ldots,x,\ldots),\ldots} \rightarrow_{c-\epsilon-1} N \tag{3.101}$$
$$[M]_{\ldots,N\rightarrow(\ldots,x,\ldots),\ldots} \rightarrow_{c-\epsilon-2} [M]_{\ldots,N\rightarrow(\ldots),\ldots} \text{ where } x \notin \mathtt{FV}(M) \tag{3.102}$$
$$[M]_{\ldots,N\rightarrow(x),\ldots} \rightarrow_{c-\epsilon-3} [M\{N/x\}]_{\ldots} \tag{3.103}$$

All new variables introduced in the equations above are intended fresh.

Lemma 61 *$\rightarrow_{\mathscr{C}an}$ is Church-Rosser.*

Definition 38 *The type assignmennt system assigning formulas of EAL to terms of Abs^{EA} is shown in Table 3.1.*

$$\frac{}{\Gamma, x : A \vdash_{abs} x : A}\ ax \qquad \frac{\Gamma, x : A \vdash_{abs} M : B}{\Gamma \vdash_{abs} \lambda x.M : A \multimap B}\ (\multimap I)$$

$$\frac{\Gamma \vdash_{abs} M : A \multimap B \quad \Delta \vdash_{abs} N : A}{\Gamma, \Delta \vdash_{abs} M\ N : B}\ (\multimap E)$$

$$\frac{\begin{array}{l}\Gamma_1 \vdash_{abs} N_1 :!A_1 \\ \quad\vdots \\ \Gamma_k \vdash_{abs} N_k :!A_k \quad \Delta, \widehat{x_1 :!A_1}, \ldots, \widehat{x_k :!A_k} \vdash_{abs} M : B\end{array}}{\Gamma_1, \ldots, \Gamma_k, \Delta \vdash_{abs} [M]_{N_1\rightarrow(\widehat{x_1}),\ldots,N_k\rightarrow(\widehat{x_k})} : B}\ contr$$

$$\frac{\begin{array}{l}\Delta_1 \vdash_{abs} N_1 : \overbrace{!\ldots!}^{m} A_1 \\ \quad\vdots \\ \Delta_k \vdash_{abs} N_k : \overbrace{!\ldots!}^{m} A_k \quad x_1 : A_1, \ldots, x_k : A_k \vdash_{abs} M : B \quad m > 0\end{array}}{\Gamma, \Delta_1, \ldots, \Delta_k \vdash_{abs} \nabla(M)[^{N_1}/x_1, \cdots, {}^{N_k}/x_n] : \underbrace{!\ldots!}_{m} B}\ !^m$$

Table 3.1: Type assignment system for Abs^{EA}-terms.

Lemma 62 (Subject reduction) *Let $M \in Abs^{EA}$ and $M \rightarrow^*_{\mathscr{C}an} N$, then*

$$\Gamma \vdash_{abs} M : A \Rightarrow \Gamma \vdash_{abs} N : A.$$

Proof: By induction on the length of the reduction. By cases on the last step of the reduction.

$\rightarrow_{c-\epsilon-1}$ then

$$[x]_{\ldots,N\rightarrow(\ldots,x,\ldots),\ldots} \rightarrow_{c-\epsilon-1} N$$

the type assignment is

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash_{abs} N :!A'\\ \vdots\end{array} \qquad \ldots, x :!A', \ldots \vdash_{abs} x :!A'}{\ldots, \Gamma \vdash_{abs} [x]_{\ldots, N \to (\ldots, x, \ldots), \ldots} :!A'}\ contr$$

hence, trivially, $\ldots, \Gamma \vdash_{abs} N :!A'$.

$\to_{\nabla-\epsilon-1}$ then

$$\nabla(x)[^N/x] \to_{\nabla-\epsilon-1} N$$

the type assignment is

$$\dfrac{\Gamma \vdash_{abs} N : \overbrace{!\ldots!}^{m} A \quad x : A \vdash_{abs} x : A \quad m > 0}{\Delta, \Gamma \vdash_{abs} \nabla(x)[^N/x] : \underbrace{!\ldots!}_{m} A}\ !^m$$

hence, trivially, $\Delta, \Gamma \vdash_{abs} N : \underbrace{!\ldots!}_{m} A$.

Other cases are equally easy. $\square$

3.2.2 Principal Typing for $\vdash_{abs}$

In this section we will prove that $\vdash_{abs}$ enjoies the principal typing property, i.e., every typing can be derived from a particular one by means of a suitable substitution. First of all, let us introduce the notion of *type scheme*.

Definition 39 *i)* Type schemata *are defined by the following grammar:*

$$\sigma ::= \alpha \mid \sigma \multimap \sigma \mid !^{\sum n_i}(\sigma)$$

where α belongs to a countable set of scheme variables, ranged over by α, β, γ; type schemata are ranged over by σ, τ, ρ; let $\mathscr{T}$ denote the set of type schemata;

ii) A scheme substitution *is a function from type schemata to types, replacing scheme variables by types and $!^{\sum_{i=1}^{k} n_i}$ by $\underbrace{! \cdots !}_{p}$, for some $p > k$.*

We can decompose a scheme substitution S into a pair $(\bar{S}, X)$, where $\bar{S}$ substitutes scheme variables with type schemata, X replaces $!^{\sum_{i=1}^{k} n_i}$ by $\overbrace{! \cdots !}^{p}$, for some $p > k$, and $S = \bar{S} \circ X$.

$\equiv$ denotes the syntactical identity between schemata.

In order to define the principal typing of a canonical form, we need a unification algorithm for type schemes. The unification algorithm, which we will present in SOS style, is a function U from $\mathscr{T} \times \mathscr{T}$ to pairs of the shape $< C, s >$, where C (the *modality* set) is a set of natural linear constraints, and s is a substitution, replacing scheme variables by type schemes.

A scheme substitution S is associated to $< C, s >$ if the decomposition of S is (s, X) where X is a solution of C.

Before U we introduce a function F collapsing all $!^{\sum n_i}$:

$$\frac{}{F(!^{\sum n_i}(x)) = !^{\sum n_i}(x)} \qquad \frac{}{F(x) = x}$$

$$\frac{F(\sigma) = !^{\sum m_j}(\tau)}{F(!^{\sum n_i}(\sigma)) = !^{\sum n_i + \sum m_j}(\tau)}$$

$$\frac{F(\sigma \multimap \tau) = \rho}{F(!^{\sum n_i}(\sigma \multimap \tau)) = !^{\sum n_i}(\rho)} \qquad \frac{F(\sigma) = \sigma_1 \quad F(\tau) = \tau_1}{F(\sigma \multimap \tau) = \sigma_1 \multimap \tau_1}$$

In the following we assume that types unified by U are already transformed by F, i.e. it is not the case that $!^n(!^m(\sigma))$ is a subtype of them.

$$\frac{}{U(\alpha, \alpha) =< \emptyset, [] >}$$

$$\frac{\alpha \text{ is a scheme variable not occurring in } \tau}{U(\alpha, \tau) =< \emptyset, [\alpha \mapsto \tau] >}$$

$$\frac{\alpha \text{ is a scheme variable not occurring in } \sigma}{U(\sigma, \alpha) =< \emptyset, [\alpha \mapsto \sigma] >}$$

$$\frac{U(\rho, \mu) =< C, s >}{U(!^{\sum n_i}\rho, !^{\sum m_j}\mu) =< C \cup \{\sum n_i - \sum m_j = 0\}, s >}$$

$$\frac{U(\sigma_1, \tau_1) =< C_1, s_1 > U(s_1(\sigma_2), s_1(\tau_2)) =< C_2, s_2 >}{U(\sigma_1 \to \sigma_2, \tau_1 \multimap \tau_2) =< C_1 \cup C_2, s_1 \circ s_2 >}$$

Lemma 63 *i) (correctness) $U(\sigma, \tau) =< C, s >$ implies $\forall S$ scheme substitution associated to $< C, s >$, $S(\sigma) \equiv S(\tau)$*

ii) (completeness) $S(\sigma) \equiv S(\tau)$ and the decomposition of S is (s, X) implies $U(\sigma, \tau) =< C, s' >$ and X is a solution of C and $s = s' \circ s''$, for some s''.

Proof: By structural induction on σ, τ, decomposing S. □

Let

$$\begin{aligned} U(\sigma_1, \sigma_2, \dots, \sigma_n) = \; & \texttt{let } U(\sigma_1, \sigma_2) =< C_1, s_1 > \texttt{ and} \\ & \texttt{let } U(s_1 \circ \dots \circ s_i(\sigma_{i+1}), s_1 \circ \dots \circ s_i(\sigma_{i+2})) =< C_{i+1}, s_{i+1} > \\ & \texttt{in } < C_1 \cup \dots \cup C_n, s_1 \circ \dots \circ s_n > . \end{aligned}$$

The following function PT takes as input a term of Λ^{EA} and gives as output a triple of a scheme context, i.e. a finite set of scheme assignments, a type scheme and a set of linear natural constraints. PT is defined modulo names of type variables.

- $PT(x) =< \{x : \alpha\}, \alpha, \emptyset >$, where α is fresh;
- $PT(\lambda x.M) = \texttt{let } PT(M) =< B, \sigma, C > \texttt{ in}$
  $\texttt{if } B = B' \cup \{x : \tau\} \texttt{ then } < B', \tau \multimap \sigma, C >$
  $\texttt{else } < B, \alpha \multimap \sigma, C >$
  where α is fresh;

- $PT(M\ N) = \texttt{let}\ PT(M) =< B_1, \sigma_1, C_1 >\ \texttt{and}$
  $PT(N) =< B_2, \sigma_2, C_2 >\ \texttt{and}$
  $\texttt{let}$ they be disjoint $\texttt{in}$
  $\texttt{let}\ U(\sigma_1, \sigma_2 \multimap \alpha) =< C, s >$
  (α fresh) $\texttt{in}$
  $< s(B_1 \cup B_2), s(\alpha), C \cup C_1 \cup C_2 >$;

- $PT([M]_{N_1 \to (\widehat{x_1}), \ldots, N_k \to (\widehat{x_k})}) =$
  $\texttt{let}\ PT(M) =< B, \sigma, C >\ \texttt{and}$
  $PT(N_i) =< B_i, \sigma_i, C_i >$
  (all disjoint and disjoint from $PT(M)$) $\texttt{and}$
  $\texttt{let}\ U(B(x_1^1), \ldots, B(x_1^{n_1}), \sigma_1, !^{m_1}\alpha_1) =< C'_1, s_1 >$
  $\texttt{and}\ U(s_1 \circ \cdots \circ s_i(B(x_{i+1}^1)), \ldots, s_1 \circ \cdots \circ s_i(B(x_{i+1}^{n_i})),$
  $s_1 \circ \cdots \circ s_i(\sigma_{i+1}), !^{m_{i+1}}\alpha_{i+1}) =< C'_{i+1}, s_{i+1} >$
  $\texttt{and}\ s = s_1 \circ \cdots \circ s_k$
  $\texttt{in} < s(B \cup \bigcup_{1 \le j \le k} B_j), s(\sigma), C \cup \bigcup_{1 \le j \le k} C_j \cup \bigcup_{1 \le j \le k} C'_j >$.
  (α_i and m_i fresh)

- $PT(\nabla(M)[N_1/x_1, \ldots, N_m/x_m]) =$
  $\texttt{let}\ PT(M) =< B, \sigma, C >$
  $PT(N_i) =< B_i, \sigma_i, C_i >$
  (all disjoint and disjoint from $PT(M)$) $\texttt{and}$
  $\texttt{in}\ U(!^n B(x_1), \sigma_1) =< C'_1, s_1 >$
  $\texttt{and}\ U(!^n(s_1 \circ \cdots \circ s_i(B(x_{i+1}))), s_1 \circ \cdots \circ s_i(\sigma_{i+1})) =< C'_{i+1}, s_{i+1} >$
  $\texttt{and}\ s = s_1 \circ \cdots \circ s_m$
  $\texttt{in} < s(\bigcup_{1 \le j \le m} B_j), !^n(s(\sigma)), C \cup \bigcup_{1 \le j \le m} C_j \cup \bigcup_{1 \le j \le m} C'_j >$
  (n fresh).

  where x_j^i denotes the i-th component of $\widehat{x_j}$.

Theorem 64 (Principal typing for Abs^{EA}) *$\Gamma \vdash_{abs} M : A$ if and only if $PT(M) =< \Theta, \sigma, C >$ and $\Gamma \supseteq S(\Theta)$, $A = S(\sigma)$, for some scheme substitution S satisfying C.*

Proof: By structural induction on M. □

3.2.3 Canonical Forms

A term of Abs^{EA} is a term in a meta-language, representing an infinite set of terms of Λ^{EA}. In particular, to every term in Λ^{EA} a *canonical form* can be assigned, which is a term of Abs^{EA} in normal form.

First consider the embedding function from Λ^{EA} to Abs^{EA}:

$$\begin{aligned}
emb(x) &= x \\
emb(\lambda x.M) &= \lambda x.emb(M) \\
emb((M_1\ M_2) &= (emb(M_1)\ emb(M_2)) \\
emb(!\,(M)\,[N_1/x_1, \ldots, N_k/x_k]) &= \nabla(emb(M))[{}^{emb(N_1)}/x_1, \cdots \\
&\qquad \cdots, {}^{emb(N_k)}/x_k] \\
emb(\|M\|^N_{x,y}) &= [emb(M)]_{emb(N) \to (x,y)}
\end{aligned}$$

The canonical form of $M \in \Lambda^{EA}$ can be obtained by reducing to normal form (with respect to $\rightarrow_{\mathscr{C}an}$) the term $emb(M) \in Abs^{EA}$. Thanks to the Lemma 61 the canonical form of a term is unique: let us call it $\mathscr{C}an(M)$.

Lemma 65 $\forall M \in \Lambda^{EA} \quad \Gamma \vdash_{\mathsf{NEAL}} M : A \Rightarrow \Gamma \vdash_{abs} emb(M) : A$

> **Proof:** By structural induction on M, the thesis holds trivially for $M = x$ and by hypothesis for $M = \lambda x.M'$, $M = (M_1\ M_2)$ and $M = [M_1]_{N=x,y}$. For $M = !(M_1)[^{N_1}/x_1, \cdots, {}^{N_k}/x_k]$ the thesis holds by the observation that !-rule in the assignment system for Λ^{EA} is simply a particular case of $!^m$-rule in the assignment system for Abs^{EA}. □

Lemma 66 $\forall M \in \Lambda^{EA}$. $\Gamma \vdash_{\mathsf{NEAL}} M : A$ *then* $\Gamma \vdash_{abs} \mathscr{C}an(M) : A$.

> **Proof:** By Lemma 65 and Lemma 62. □

Consider the following embedding function from Abs^{EA} to $\mathscr{P}(\Lambda^{EA})$:

$$\begin{aligned}
bme(x) &= \{x\}\\
bme(\lambda x.M) &= \{\lambda x.M' | M' \in bme(M)\}\\
bme((M_1\ M_2) &= \{(M_1'\ M_2') | M_1' \in bme(M_1) \wedge M_2' \in bme(M_2)\}\\
bme\left(\nabla(M)[^{N_1}/x_1, \cdots {}^{N_k}/x_k]\right) &=\\
&\quad \{\overbrace{!(\cdots!(}^{n} M')[z_1^1/x_1, \ldots, z_k^1/x_k]) \cdots\\
&\qquad \cdots)[^{N_1'}/z_1^n, \ldots, {}^{N_k'}/z_k^n] \mid\\
&\qquad M' \in bme(M) \wedge N_i' \in bme(N_i) \wedge n > 0\}\\
bme([M]_{N \rightarrow (x_1^1, x_1^2, \ldots, x_1^{n_1}), \ldots}) &= bme\left(\left[[M]_{y \rightarrow (x_1^1, x_1^2)}\right]_{N \rightarrow (y, x_1^3, \ldots, x_1^{n_1}), \cdots}\right)
\end{aligned}$$

y fresh variable

$$\begin{aligned}
bme([M]_{N_1 \rightarrow (x_1^1, x_1^2), N_2 \rightarrow (\widehat{x_2}), \cdots}) &= bme\left(\left[[M]_{N_1 \rightarrow (x_1^1, x_1^2)}\right]_{N_2 \rightarrow (\widehat{x_2}), \cdots}\right)\\
bme([M]_{N \rightarrow (x_1, x_2)}) &= \{\left\|M'\right\|_{x_1, x_2}^{N'} \mid M' \in bme(M) \wedge N' \in bme(N)\}
\end{aligned}$$

Lemma 67 $\forall M \in Abs^{EA}$

$$\Gamma \vdash_{abs} M : A \Rightarrow \exists M' \in bme(M)\ \Gamma \vdash_{\mathsf{NEAL}} M' : A.$$

> **Proof:** It is sufficient to choose the necessary expansion of ! according to the derivation $\Gamma \vdash_{abs} M : A$ and to notice that *bme* simply impose an order of binary contraction that does not prejudice the possibility of deriving a type. □

We can now formally define the set of EAL-canonical forms.

Definition 40 *The set* C^{EA} *of* canonical forms *is generated by the following grammar (C is the starting symbol):*

Productions			Comment
C	$::=$	$S \mid NS$	*a canonical form is either a sharing term or a non-sharing term.*
S	$::=$	$[LB]_{Slist}$ *where* $\mathtt{CV}(Slist) \subseteq \mathtt{FV}(LB) \mid$	*a sharing term can be a linear or banged one contracting a list of terms. In this case the set of variables contracted must be a subset of the free variables of* LB*. In other words, it is not allowed to contract a weakened variable.*
		$[S]_{Slist}$ *where* $\mathtt{CV}(Slist) \cap \mathtt{SSV}(S) = \emptyset$ *and* $\mathtt{CV}(Slist) \subseteq \mathtt{FV}(S)$ *and* $\forall S_i \in \mathtt{ST}(S)$ $\mathtt{CV}(Slist) \cap \mathtt{FV}(S_i) \neq \emptyset$	*a sharing term can also be built using another sharing term, but in this case it is not possible to contract a previously contracted variable (there is no need of contracting a set of variables in two steps). As in the previous case it is not allowed to contract a weakened variable. Moreover every subterm previously contracted must have at least a free variable that is contracted now (if this is not the case, there is no need of two different steps of contraction and it is possible to collapse them).*

Productions			Comment
$Slist$	$::=$	$NS \to (\widehat{x}) \mid NS \to (\widehat{x}), Slist$ *where* $\lvert\widehat{x}\rvert \geq 2$	*a list of shared terms is made of non-sharing terms. Moreover there must be at least two contracted variables.*
NS	$::=$	$LB \mid x$	*a non-sharing term is either a linear or banged one or is a variable.*
LB	$::=$	$L \mid B$	
L	$::=$	$(NS\ NS) \mid \lambda x.NS \mid$	*a linear term can be an application or an abstraction of non-sharing terms.*
		$\lambda x.S$ *where* $S = [C]_{NS \to (\widehat{y})}$ *and* $x \in \mathtt{FV}(NS)$	*a linear term can also be an abstraction of a sharing term provided that it shares a single subterm and that the variable binded by the abstraction is free in the shared subterm.*
DSL	$::=$	$D \mid S \mid L$	
B	$::=$	$D \mid \nabla(DSL)[\widehat{x}/\widehat{y}]$ *where* $\mathtt{SBV}(\widehat{x}/\widehat{y}) \cap \mathtt{SSV}(DSL) = \emptyset$ *and* $\mathtt{BV}(\widehat{x}/\widehat{y}) = \mathtt{FV}(DSL)$	*this kind of box is all around the term and only (and all the) variables exit from the* ∇*. Actually, inside the box the term can be linear, or a sharing term with no single variables contracted, or a banged term "of type* D*", i.e. with a box closed before a subterm.*
D	$::=$	$\nabla(DSL)[Blist]$ *where* $\mathtt{SBV}(Blist) \cap \mathtt{SSV}(DSL) = \emptyset$ *and* $\mathtt{BV}(Blist) = \mathtt{FV}(DSL)$	*this kind of banged terms put the box before a subterm. As in the previous case, if the term inside the box is a sharing term, then it must have no single variable contracted. The distinction between two kinds of banged terms means that we can put a sequence of boxes closing them before one or more subterms (type* D*) and finally eventually box all the term (type* B*).*

PRODUCTIONS	COMMENT
$Blist \quad ::= \quad L/x \mid L/x, Blist \mid x/y, Blist$	*the list of subterms that are "out of the box" contains at least a term that is not a variable. Such a term is a linear one.*

all variables are linear.

Lemma 68 (Soundness of $\mathscr{C}an$) $\forall M \in \Lambda^{EA}\ \mathscr{C}an(M) \in C^{EA}$

Proof: By absurd let M be the smallest normal form s.t. $M \notin \mathscr{L}_C$ (the language of C^{EA}). By structural induction on M we will show that either $M \in \mathscr{L}_C$ or M is not in $\rightarrow_{\mathscr{C}an}$-normal form.

$M = x$ then $M \in \mathscr{L}_C$ thanks to the derivation $C \rightarrow NS \rightarrow x$;

$M = \lambda x.M'$ and by hypothesis M' is in n.f. and $M' \in \mathscr{L}_C$ because it is smaller than M. But we have the following derivations:

$$\begin{array}{lll} C \rightarrow NS \rightarrow LB \rightarrow L & \rightarrow & \lambda x.NS \\ & \searrow & \\ & & \lambda x.S \quad S = [C]_{NS\rightarrow(\widehat{y})} \text{ and } x \in \mathtt{FV}(NS) \end{array}$$

$M' \in \mathscr{L}_C$ then, by the first production rule of the grammar, either $M' \in \mathscr{L}_{NS}$ or $M' \in \mathscr{L}_S$. If $M' \in \mathscr{L}_{NS}$ we have by derivation above $M \in \mathscr{L}_C$. Then it must be $M' \in \mathscr{L}_S$. If M' contracts more than one subterm, i.e. it has the form $[M'']_{N_1\rightarrow(\widehat{x_1}),N_2\rightarrow(\widehat{x_2}),\ldots}$ then since in M all variables are linear x can not be at the same time in the free variables of both N_1 and N_2, hence there is a $\rightarrow_{\lambda-c}$-redex. The same if M' contracts only one subterm but x is not in the free variables of such subterm. Then it must be $M' = [M'']_{N\rightarrow(\widehat{y})}$ and $x \in \mathtt{FV}(N)$ thus $M \in \mathscr{L}_C$.

$M = (M_1\ M_2)$ and by hypothesis M_1, M_2 are normal forms $\in \mathscr{L}_C$. Consider now the derivation:

$$C \rightarrow NS \rightarrow LB \rightarrow L \rightarrow (NS\ NS)$$

If both M_1 and M_2 are $\in \mathscr{L}_{NS}$ then $M \in \mathscr{L}_C$, then it must be either $M_1 \notin \mathscr{L}_{NS}$ or $M_2 \notin \mathscr{L}_{NS}$. But $M_1 \in \mathscr{L}_C$ then either $M_1 \in \mathscr{L}_S$ or $M_1 \in \mathscr{L}_{NS}$. But if $M_1 \in \mathscr{L}_S$ then there is a $\rightarrow_{@-c-1}$-redex. Analogously for M_2.

$M = [M_1]_{Clist}$ and by hypothesis M_1 and every N of $Clist$ are normal forms $\in \mathscr{L}_C$. Consider the derivation:

$$\begin{array}{llll} C \rightarrow S & \rightarrow & [LB]_{Slist} & \mathtt{CV}(Slist) \subseteq \mathtt{FV}(LB) \\ & \searrow & & \\ & & [S]_{Slist} & \mathtt{CV}(Slist) \cap \mathtt{SSV}(S) = \emptyset \\ & & & \mathtt{CV}(Slist) \subseteq \mathtt{FV}(S) \\ & & & \forall S_i \in \mathtt{ST}(S)\ \mathtt{CV}(Slist) \cap \mathtt{FV}(S_i) \neq \emptyset \end{array}$$

If $\mathtt{CV}(Clist) \not\subseteq \mathtt{FV}(M_1)$ then there is a $\rightarrow_{c-\epsilon-2}$-redex, thus $\mathtt{CV}(Clist) \subseteq \mathtt{FV}(M_1)$. Now $M_1 \in \mathscr{L}_C$ then $M_1 \in \mathscr{L}_S$ or $M_1 \in \mathscr{L}_{NS} = \mathscr{L}_{LB} \cup \mathscr{L}_x$ i.e. there are three cases:

1. $M_1 \in \mathscr{L}_x$ then there is a $\rightarrow_{c-\epsilon-1}$-redex;
2. $M_1 \in \mathscr{L}_{LB}$ then $M \in \mathscr{L}_C$ by derivation above;
3. $M_1 \in \mathscr{L}_S$ then there are two cases:
   (a) $\mathtt{CV}(Clist) \cap \mathtt{SSV}(M) \neq \emptyset$ then there is a $\rightarrow_{var-collaps}$-redex;
   (b) $\exists S_i \in \mathtt{ST}(S)$ $\mathtt{CV}(Slist) \cap \mathtt{FV}(S_i) = \emptyset$ then there is a $\rightarrow_{c-collaps}$-redex.

$M = \nabla(M_1)[Mlist]$ and M_1 and each N_i in $Mlist$ are normal forms $\in \mathscr{L}_C$. Consider the derivation:

$$\begin{array}{lll} C \rightarrow NS \rightarrow LB \rightarrow B & \rightarrow & \nabla(DSL)[\widehat{x}/\widehat{y}] \quad \mathtt{BV}(\widehat{x}/\widehat{y}) = \mathtt{FV}(DSL) \\ & \searrow & \qquad \text{and } \mathtt{SBV}(\widehat{x}/\widehat{y}) \cap \mathtt{SSV}(DSL) = \emptyset \\ & & \nabla(DSL)[Blist] \quad \mathtt{BV}(Blist) = \mathtt{FV}(DSL) \\ & & \qquad \text{and } \mathtt{SBV}(Blist) \cap \mathtt{SSV}(DSL) = \emptyset \end{array}$$

If $\mathtt{BV}(Mlist) \neq \mathtt{FV}(M_1)$ then there is a $\rightarrow_{\nabla-\epsilon-2}$-redex, thus $\mathtt{BV}(Mlist) = \mathtt{FV}(M_1)$.

Now $M_1 \in \mathscr{L}_C = \mathscr{L}_S \cup \mathscr{L}_{NS}$.

1. If $M_1 \in \mathscr{L}_S \subseteq \mathscr{L}_{DSL}$ then if $\mathtt{SBV}(Mlist) \cap \mathtt{SSV}(M_1) \neq \emptyset$ then there is a $\rightarrow_{c-\nabla}$-redex else there must exists a $N_i \notin \mathscr{L}_L \cup \mathscr{L}_x$, with $N_i \in \mathscr{L}_C$ by hypothesis.
   (a) $N_i \in \mathscr{L}_S$ then there is a $\rightarrow_{\nabla-c}$-redex;
   (b) $N_i \in \mathscr{L}_{NS}$ then it must be $N_i \in \mathscr{L}_B$ then there is a $\rightarrow_{\nabla-\nabla}$-redex;

   otherwise $M \in \mathscr{L}_C$ by the derivation above.
2. If $M_1 \in \mathscr{L}_{NS}$
   (a) if $M_1 \in \mathscr{L}_L$ then there must exists a $N_1 \notin \mathscr{L}_L \cup \mathscr{L}_x$, with $N_i \in \mathscr{L}_C$ by hypothesis. As above by cases on N_i;
   (b) if $M_1 \in \mathscr{L}_x$ then there is a $\rightarrow_{\nabla\ \epsilon-1}$-redex;
   (c) if $M_1 \in \mathscr{L}_B$ it must be $M_1 \notin \mathscr{L}_D$ then there is a $\rightarrow_{\nabla-\nabla}$-redex;

   otherwise $M \in \mathscr{L}_C$ by the derivation above.

□

Lemma 69 *$\forall M \in C^{EA}$, M is in $\rightarrow_{\mathscr{C}an}$ normal form.*

Proof: By induction on the length of derivation of M, there is no redex in it.
□

3.2.4 Canonical Forms Algorithm $\mathscr{C}$

We have seen in Section 3.1 that lambda terms typeable in EAL are skeletons that can be decorated. We can obtain decidability of type inference of lambda terms in EAL in a different way using the principal type theorem and the canonical forms.

Definition 41 *Let $M \in \Lambda$. The set of canonical forms corresponding to M is $C(M) = \{N \mid \exists R \in \Lambda^{EA}$ such that $(R)^* = M,\ R$ is simple and $N = \mathscr{C}an(R)\}$.*

Lemma 70 *For every $M \in \Lambda$, $C(M)$ is finite.*

We will show an algorithm $\mathscr{C}$ such that, for every $M \in \Lambda$, $\mathscr{C}(M)$ gives either $C(M)$ or a negative answer. $\mathscr{C}$ is correct and complete.

Let $\mathbb{L}(M)$ be the *linearization* of M with respect to all its free variables and let $\mathbb{L}_x(M)$ be the set of fresh variables generated by $\mathbb{L}$ during the linearization of x in M. I.e. let $M = (x\ (x\ y\ y))$ then $\mathbb{L}(M) = (x_1\ (x_2\ y_1\ y_2))$ and $\mathbb{L}_x(M) = \{x_1, x_2\}$, $\mathbb{L}_y(M) = \{y_1, y_2\}$ and $\mathbb{L}_z(M) = \emptyset$ for any other variable z.

The algorithm is defined by the following equations:

$$
\begin{aligned}
\mathscr{C}(M) = {} & \texttt{if}\ \exists x_1, \dots, x_k \in_{i>1} \texttt{FV}(M)\ \texttt{then} \\
& \quad \big[\mathbb{T}(\mathbb{L}(M)) \cup \mathbb{F}(\mathbb{L}(M))\big]_{x_1 \to (\mathbb{L}_{x_1}(M)), \dots, x_k \to (\mathbb{L}_{x_k}(M))} \\
& \texttt{else} \\
& \quad \mathbb{T}(M) \cup \mathbb{F}(M)
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{T}(x) &= \{x\} \\
\mathbb{T}(\lambda x.M) &= \texttt{if}\ x \in_{i>1} \texttt{FV}(M)\ \texttt{then} \\
& \qquad \lambda x.\big[\mathbb{T}(\mathbb{L}(M)) \cup \mathbb{F}(\mathbb{L}(M))\big]_{x \to (\mathbb{L}_x(M))} \\
& \quad \texttt{else} \\
& \qquad \lambda x.(\mathbb{T}(M) \cup \mathbb{F}(M)) \\
\mathbb{T}((M\ N)) &= \mathbb{T}(M)@\Big(\mathbb{T}(N) \cup \mathbb{F}(N)\Big)
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{F}(x) &= \emptyset \\
\mathbb{F}(M \neq x) &= \mathbb{F}'(M) \cup \nabla\big(\mathbb{F}'(M\{\widehat{z}/\texttt{FV}(M)\}) \cup \mathbb{T}(M\{\widehat{z}/\texttt{FV}(M)\})\big)[\texttt{FV}(M)/\widehat{z}]
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{F}'(x) &= \emptyset \\
\mathbb{F}'(M \neq x) &= \forall A_1, \dots, A_n, P, n > 0 \text{ s.t.} \\
& \qquad M =_\alpha P\{A_1/y_1, \cdots, A_n/y_n\} \\
& \qquad \{y_1, \dots, y_{n+k}\} = \texttt{FV}(P)\ P \neq x \\
& \qquad \forall 1 \leq i \leq n\ A_i = (A_{i_1}\ A_{i_2}) \\
& \quad \nabla\big(\mathbb{T}(P\{\widehat{z_1^k}/\widehat{y_{n+1}^{n+k}}\}) \cup (\mathbb{F}'(P\{\widehat{z_1^k}/\widehat{y_{n+1}^{n+k}}\})\big)\big[\mathbb{T}(A_1)/y_1, \\
& \qquad \cdots, \mathbb{T}(A_n)/y_n, \widehat{y_{n+1}^{n+k}}/\widehat{z_1^k}\big]
\end{aligned}
$$

Where $\widehat{z}$ and $\widehat{z_1^k}$ are fresh variables. $\widehat{z_1^k}/\widehat{y_{n+1}^{n+k}}$ stands for $z_1/y_{n+1}, \cdots, z_k/y_{n+k}$, $\widehat{z}/\texttt{FV}(M)$ stands for the complete renaming of free variables of M width fresh ones, and $\texttt{FV}(M)/\widehat{z}$ stands for the inverse substitution.

Fact 71 $\forall \mathbb{C} \in \mathscr{C}(M) \qquad (\mathbb{C})^* = M$

In order to prove soundness and completeness of $\mathscr{C}$, we define the subset of EAL-canonical forms we are interested in, i.e. the set of canonical forms contracting at most variables.

Definition 42 (CC^{EA}) *The set of* simple canonical EAL-terms CC^{EA} *is generated by the following grammar (CC is the starting symbol):*

Productions	Comment
$CC \quad ::= \quad [K]_{Clist}$ *where* $\mathtt{CV}(Clist) \subseteq \mathtt{FV}(K) \mid K$	*a simple canonical form can eventually contract some variables. No other subterm of it can be a contraction except before a lambda abstraction binding the contracted variable.*
$Clist \quad ::= \quad y \to (\widehat{x}) \mid y \to (\widehat{x}), Clist$ *where* $\lvert\widehat{x}\rvert \geq 2$	*each component of a contraction list contracts at least two variables. Only variables are shared. No more complex subterm.*
$K \quad ::= \quad \nabla(B)[\widehat{x}/\widehat{y}]$ *where* $\mathtt{BV}(\widehat{x}/\widehat{y}) = \mathtt{FV}(B) \mid B \mid x$	*this kind of box corresponds to "B" production in canonical forms grammar. There is only one side condition because* $\mathtt{SSV}(B) = \emptyset$*.*
$B \quad ::= \quad \nabla\,(B)\,[L]$ *where* $\mathtt{BV}(L) = \mathtt{FV}(B) \mid R$	*this kind of box corresponds to "D" in canonical forms grammar.*
$L \quad ::= \quad A/x \mid y/x, L \mid A/x, L$	*differently from canonical forms, the list of subterms out of the box is made of applications.*
$R \quad ::= \quad \lambda x.[K]_{x \to (x_1,\ldots,x_n)}$ *where* $\{x_1, \ldots, x_n\} \subseteq \mathtt{FV}(K)$ $\mid \lambda x.K \mid A$	*it is possible to introduce a contraction just before the shared variable has been binded by a lambda abstraction.*
$A \quad ::= \quad (R\ K) \mid (x\ K)$	*the functional part of an application can be either an abstraction or an application itself or a variable. No contraction or box is allowed.*

where all variables are linear, $\widehat{x}$ *stands for* $x_1, \ldots, x_n$ *and* $n > 0$*.*

Notice that side condition $\lvert\widehat{x}\rvert \geq 2$ in the production of $Clist$ implies $[x]_{Clist}$ is not a possible term in CC^{EA} by side condition $\mathtt{CV}(Clist) \subseteq \mathtt{FV}(K)$ in production of CC and by $\{x_1, \ldots, x_n\} \subseteq \mathtt{FV}(K)$ in production of R.

Lemma 72

$$CC^{EA} \subseteq C^{EA}$$

Proof: By absurd, let M be the smallest term in CC^{EA} but not in C^{EA}. The proof is easy by induction on the structure of M. $\square$

Fact 73 *Each term in CC^{EA} contracts at most variables.*

Lemma 74 *If $M \in C^{EA}$ is simple then $\nexists N$ subterm of M such that N has the form $\big[[N']_{Slist_1}\big]_{Slist_2}$.*

Proof: If M is simple then $\mathtt{SFV}(N) = \emptyset$ for any subterm N of M. Hence looking at the grammar, the only possible contraction is $[LB]_{Slist}$. $\square$

Lemma 75 *If $M \in C^{EA}$ is simple then $\nexists N$ subterm of M such that N has the form $\nabla\big([N']_{Slist}\big)[Blist]$.*

Proof: If M is simple then $\mathtt{SSV}([N']_{Slist}) \subseteq \mathtt{FV}([N']_{Slist}) = \mathtt{BV}(Blist)$. Now suppose $\mathtt{SSV}([N']_{Slist}) \neq \emptyset$, then there exists $x \in \mathtt{SSV}([N']_{Slist})$. $x \notin \mathtt{SBV}(Blist)$ by the grammar, then $x \in \mathtt{BV}(Blist) \setminus \mathtt{SBV}(Blist)$ hence M contracts a subterm that is not a variable and this contradicts the hypothesis. $\square$

Lemma 76 *$\forall M \in C^{EA}$, M simple,*

$$\Gamma \vdash_{abs} M : A \Rightarrow M \in CC^{EA}.$$

Proof: By absurd let M be the smallest canonical form in C^{EA} such that $\Gamma \vdash_{abs} M : A \wedge M \notin CC^{EA}$. By structural induction on M is easy to prove that either there is a derivation in the grammar of CC^{EA} or M is not typeable. $\square$

Lemma 77 *Let $\mathscr{L}_X$ be the language generated by the grammar of Definition 42 with starting element X, for $X \in \{R, A, B, K\}$, then the following hold:*

1. $\mathbb{T}(M) \subseteq \mathscr{L}_R \cup \{x\}$
   - (a) $\mathbb{T}(M \neq x) \subseteq \mathscr{L}_R$
   - (b) $\mathbb{T}((M_1\ M_2)) \subseteq \mathscr{L}_A$
2. $\mathbb{F}'(M) \subseteq \mathscr{L}_B$
3. $\mathbb{F}(M) \subseteq \mathscr{L}_k$

Lemma 78 *Let $\mathscr{L}_{CC}$ the language of canonical forms generated by the grammar of Definition 42, then*

$$\forall M \in \Lambda \quad \mathscr{C}(M) \subseteq \mathscr{L}_{CC}$$

Lemma 79 1. $M \in \mathscr{L}_R \cup \{x\} \Rightarrow M \in \mathbb{T}((M)^*)$

2. $M \in \mathscr{L}_A \Rightarrow (M)^* = (M_1\ M_2) \ \wedge\ M \in \mathbb{T}((M_1\ M_2))$
3. $M \in \mathscr{L}_B \Rightarrow M \in \mathbb{F}'((M)^*) \cup \mathbb{T}((M)^*)$
4. $M \in \mathscr{L}_K \Rightarrow M \in \mathbb{F}((M)^*) \cup \mathbb{T}((M)^*)$

Lemma 80

$$M \in \mathscr{L}_{CC} \Rightarrow M \in \mathscr{C}((M)^*)$$

Theorem 81

$$\bigcup_{M \in \Lambda} \mathscr{C}(M) = \mathscr{L}_{CC}$$

Proof: By Lemma 80 and Lemma 78. □

Theorem 82 (Soundness and Completeness of $\mathscr{C}$) $\forall M \in \Lambda$

1. $\mathscr{C}(M) \subseteq C(M)$;

2. $N \in C(M)$ *and* $\exists \Gamma, A$ *s.t.* $\Gamma \vdash_{abs} N : A \Rightarrow N \in \mathscr{C}(M)$.

Proof: We recall the definition of $C(M)$:

$$C(M) = \{N \mid \exists R \in \Lambda^{EA} \text{ s.t. } (R)^* = M,\ R \text{ is simple and } N = \mathscr{C}an(R)\}$$

1. By Lemma 78 for any $\mathbb{C} \in \mathscr{C}(M)$ we have $\mathbb{C} \in CC^{EA}$ and hence, by Lemma 72, $\mathbb{C} \in C^{EA}$. Moreover $(\mathbb{C})^* = M$ by Fact 71. Then $\mathbb{C}$ is in $C(M)$ because it exists $R \in bme(\mathbb{C})$ s.t. $(R)^* = M$, R is simple and it is sufficient to notice that $emb(bme(\mathbb{C}))$ is either equal to $\mathbb{C}$ (that is in $\rightarrow_{\mathscr{C}an}$ normal form) or there are a set of $\rightarrow_{var-collaps}$, $\rightarrow_{c-collaps}$ and $\rightarrow_{\nabla-collaps}$ redexes after firing them we get $\mathbb{C}$ again.
2. By Lemma 68 $\mathscr{C}an(R) = N \in C^{EA}$. R is simple by hypothesis and then N is simple too. Moreover $\Gamma \vdash_{abs} N : A$, hence, by Lemma 76, $N \in CC^{EA}$ and then $N \in \mathscr{C}(M)$ by Lemma 80.

□

Theorem 83 (Principal typing for Λ in EAL) $\forall M \in \Lambda$ $\Gamma \vdash_{\mathsf{NEAL}} M : A$ *if and only if* $PT(N) =< \Theta, \sigma, C >$ *and* $\Gamma \supseteq S(\Theta)$, $A = S(\sigma)$, *for some scheme substitution* S *satisfying* C *and for some* $N \in \mathscr{C}(M)$.

Proof:

(If) Let be $PT(N) =< \Theta, \sigma, C >$, then by Theorem 64 $\Gamma \vdash_{abs} N : A$ and by Lemma 67 $\exists N' \in bme(N)$ $\Gamma \vdash_{\mathsf{NEAL}} N' : A$, hence the thesis.

(Only if) $\Gamma \vdash_{\mathsf{NEAL}} M : A$ then by definition $\exists R \in \Lambda^{EA}$ $\Gamma \vdash_{\mathsf{NEAL}} R : A$ and R is simple. Then $\Gamma \vdash_{abs} \mathscr{C}an(R) : A$ by Lemma 66. Moreover $\mathscr{C}an(R) \in C^{EA}$ by Lemma 68 and then, being $\mathscr{C}an(R)$ typeable, $\mathscr{C}an(R) \in CC^{EA}$ by Lemma 76. This is sufficient to prove that $\mathscr{C}an(R) \in \mathscr{C}(M)$ (by Lemma 81). The thesis holds by principal typing for Abs^{EA}.

□

3.2.5 Conclusions

We presented two different approaches for obtaining complete algorithms to derive EAL-types for λ-terms. One of our main goals is the characterization of those lambda terms that can be optimally reduced without the oracle, for which EAL-typeability is a sufficient condition. One should not see (N)EAL as a programming language; instead, it is a kind of intermediate language: if a λ-term is typeable in EAL, then we can compile it in a special manner with excellent performances during reduction, otherwise we compile it in the usual way, using the oracle.

Considering the type synthesis algorithm of Section 3.1.4, a puzzling open problem is whether there exist terms yielding constraints with only non integer solutions. Of course they have to be non EAL-typeable terms, in view of our completeness theorem. Our extensive experiments never produced such a scenario, yet we could not prove that the constraints have always integral solutions. Would there be any logical meaning for a term with a non integral number of boxes?

Following the other approach, the one of canonical forms, it could be interesting to investigate the possibility to extend the algorithm $\mathscr{C}$ to the full set of canonical forms. In this case the extended algorithm should search for common subterm to contract. The existence of such a (EAL-typeable) canonical form for a given lambda term will suggest the possibility of reducing it inside the Lamping's abstract algorithm provided that it is translated taken into account the common sub-terms identified by the canonical forms.

Finally, as we have already mentioned at the beginning of the Chapter, it is worth to investigate the possible extensions to other fragments of Linear Logic and in particular Soft Linear Logic, in the affine version, seems to be a good candidate.

II

The Implementation of Functional Languages

4

Optimal Reducers

4.1 A tool for reducers comparison

We developed a tool for the comparison of various implementation of optimal reduction. The tool consists in some modules common to all implementations plus the specific modules for every different reducer (see Figure 4.1). It has been developed in Java with the help of Java Compiler Compiler as parser generator and consists of about 13.500 lines of source code.

All different reducers implemented are based on graph rewriting. However it is possible to extend the tool adding other reducers based, for example on GOI as in [PQ00, Pin01] or based on interaction nets as in [Mac00].

Common modules are:

1. the **parser**. This module simply collects definitions and produces the syntax tree of λ-terms. The grammar accepted by the parser is showed in Table 4.1. The parser accepts a list of definitions of the form `Id` = `Term` separated by a semicolon, eventually ended by a term that represents the functional program to be evaluated. A term in a definition must be a closed term, i.e. all variables not bound must be identifiers of previous definitions. We allow some "syntactic sugar":

   - the syntax tree of $\backslash x\ y.M$ is the same of $\backslash x.\backslash y.M$;
   - the syntax tree of $(M_1\ M_2\ M_3)$ is the same of $((M_1\ M_2)\ M_3)$.

2. The **simple type inference algorithm**. The module implements the usual simple type synthesis for λ-terms:

$$\frac{}{x : newvar \vdash x : newvar}\ ax \qquad \frac{B \vdash M : t}{B \setminus \{x\} \vdash \lambda x.M : B(x) \to t}\ \to$$

$$\frac{B_1 \vdash M_1 : t_1 \qquad B_2 \vdash M_2 : t_2 \qquad \begin{array}{l}\mathsf{unify}(\mathsf{commonvars}(B_1, B_2)) = s \\ \mathsf{unify}(s(t_2) \to newvar_2, s(t_1)) = s'\end{array}}{B_1, B_2 \vdash (M_1\ M_2) : s'(newvar)}\ @$$

where s, s' are substitutions of the form $\{typevar \leftarrow type, \ldots, typevar \leftarrow type\}$.

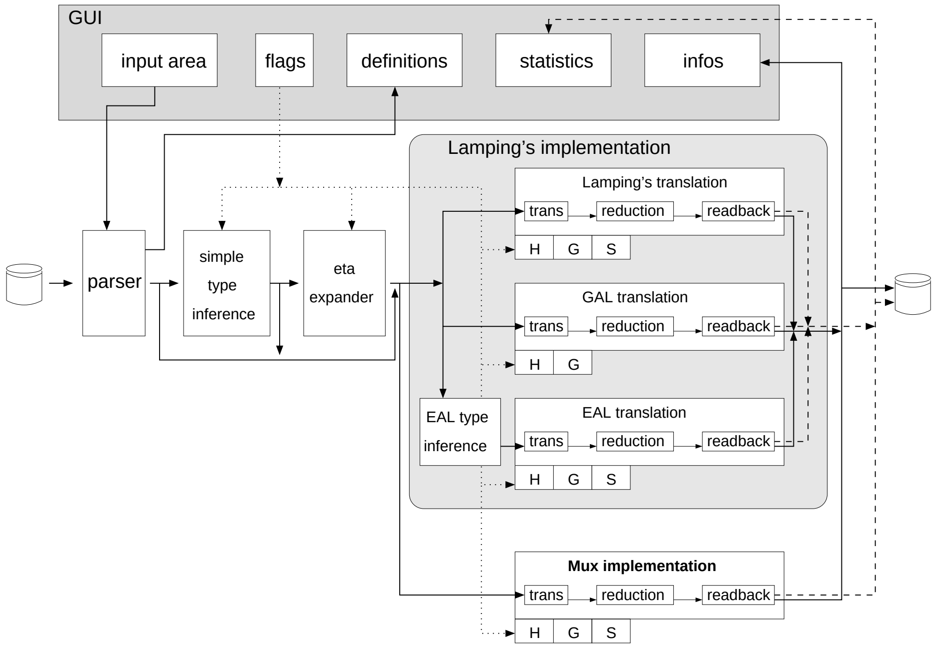


Figure 4.1: Architecture of the Optimal Reduction tool.

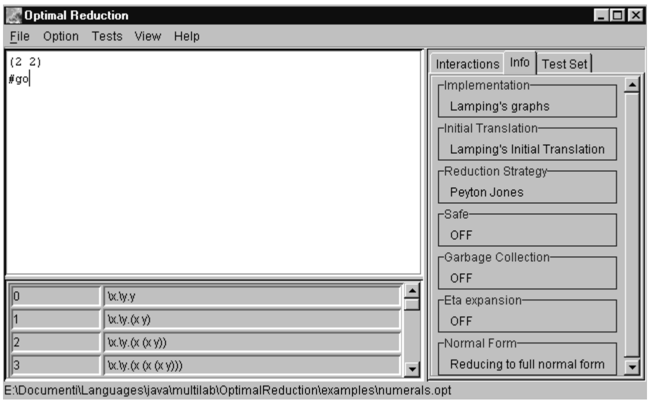


Figure 4.2: The GUI.

Start	::=	$\texttt{ListOfDefs}(;(\texttt{Term})?)? < \texttt{EOF} >$
	$\mid$	$\texttt{Term} < \texttt{EOF} >$
Def	::=	$\texttt{Id} = \texttt{Term}$
ListOfDefs	::=	$\texttt{Def}(;\texttt{Def})^*$
Term	::=	$\backslash(\texttt{Id})^+.\texttt{Term}$
	$\mid$	$(\texttt{Term}\ (\texttt{Term})^+)$
	$\mid$	$\texttt{Id}$
Id	::=	$([\texttt{a}-\texttt{z}][\texttt{A}-\texttt{Z}][0-9])^+$

Table 4.1: Grammar accepted by the optimal reducer.

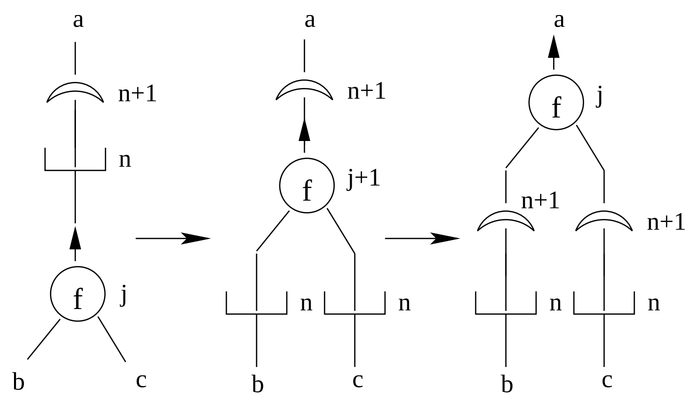


Figure 4.3: Void total effect.

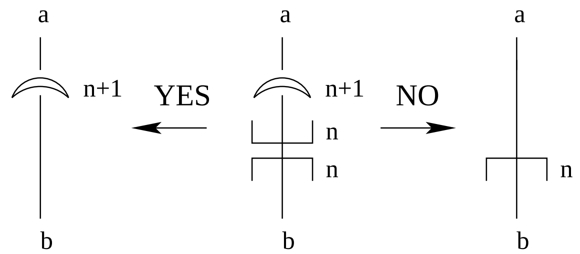


Figure 4.4: Critical pair.

3. The **eta expansion**. The module produces, given a simple typed lambda term M, the *optimal root* $or(M)$ as defined in Section 1.5. If the eta expansion option is checked, then the lambda term will be first typed, then eta expanded and finally its optimal root will be translated in sharing graphs.

4. The **safe heuristic**. The use of *safe* optimization was proposed by Asperti in [Asp95] and developed in [AC97]. As already pointed out by Lamping in his original paper, the initial translation introduces a certain number of useless control operators. Inspecting the graph reduction of, for example $(2\ \lambda z.z)$, we observe that in the final graph in normal form, representing $\lambda z.z$, there are redundant control nodes, accumulated during the reduction, in particular there are two useless brackets and two useless croissants. Moreover, if we consider the reduction of $\lambda x.((\lambda z.\lambda w.w\ x)\ x)$, we will see in the final representation of the normal form $\lambda x.x$ a redundant fan with a garbage node connected to its auxiliary door, a redundant bracket and a redundant croissant.

   The problem of accumulation of control nodes is not a minor problem of optimal reduction. Actually, the reduction of $\lambda n.((n\ 2)\ \lambda z.z)$ is exponential in n essentially for the accumulation of redundant control nodes. Now consider a specific configuration: a croissant of index $i + 1$ whose principal port is connected to the auxiliary port of a bracket of index i. Such configuration can be thought as a single "super node" with a void effect on level of nodes (the bracket first face and increase by one and then immediately the croissant decrease by the same amount, Figure 4.3). Thus one could be tempted to declare useless such control nodes and delete them. Unfortunately such optimization is not correct in the case the bracket node has to be annihilated by one of its copies, as in Figure 4.4. Consider also a configuration of a fan node with a garbage node connected to one of its auxiliary doors. The total effect of this configuration

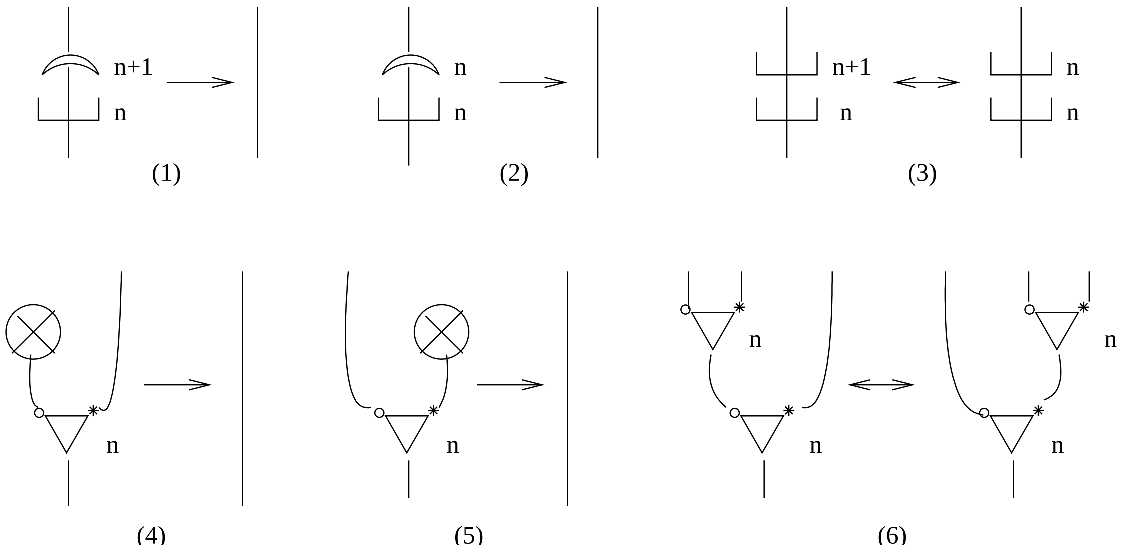


Figure 4.5: Safe rules.

is again void, and both nodes could be erased except in the case the fan has to be annihilated by one of its copies.

Definition 43 (safe operators) *A sharing operator s (fan, croissant or bracket) in a sharing graph G is* safe *if it can only match with itself.*

Proposition 84 *The rules in Figure 4.5 are correct provided the lower operator is safe. Moreover, given the configurations in the lhs of rules 1,2,3,6, when the lower operator is safe, the upper one is safe too.*

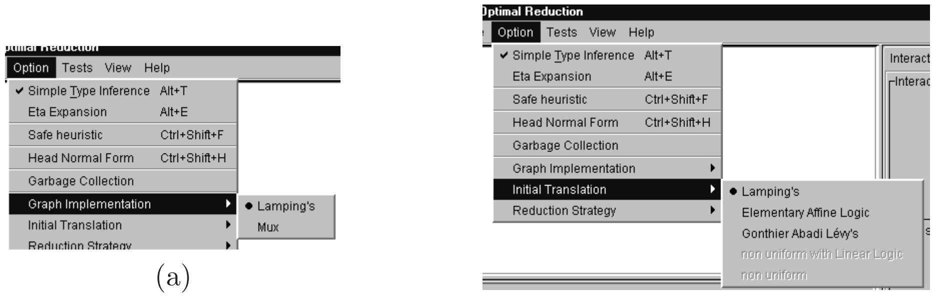
The problem of recognizing safe operator was partially solved by Asperti with the heuristic implemented in this module. The problem of giving a precise, operational characterization of safe nodes is still open. A sufficient condition for safeness of a control node is given by the safeness tag whose algorithm is described below:

- all sharing operators are initially tagged safe.
- Both residuals of an operator interacting with a lambda node are tagged unsafe.
- All other interaction rules preserve the tag of the (ancestor of) the interacting operators.
- Given the configurations in the lhs of the rules in Figure 4.5, if the lower operator is tagged safe, the upper operator can be tagged safe as well.

Theorem 85 *If a sharing operator is tagged safe, then it is safe.*

When the safe option is checked the optimal reducer tags the sharing operators as described above and implements the safe rules 1,2,4,5 of Figure 4.5.

Specific modules implementing optimal reducers are:



(a) (b)

Figure 4.6: Options.

1. **Lamping's graph implementation**, as it is described in [AG98]. For such an implementation it is possible to choose various initial translations:

   - *Lamping's initial translation.* In this case the lambda term is translated into Lamping's graph using the translation function in Figure 1.25. For this translation is available the *safe heuristic* (flag S in Figure 4.1).
   - *Gonthier, Abadi and Lévy initial translation.* In this case the translation function used is the one showed in Figure 4.7;

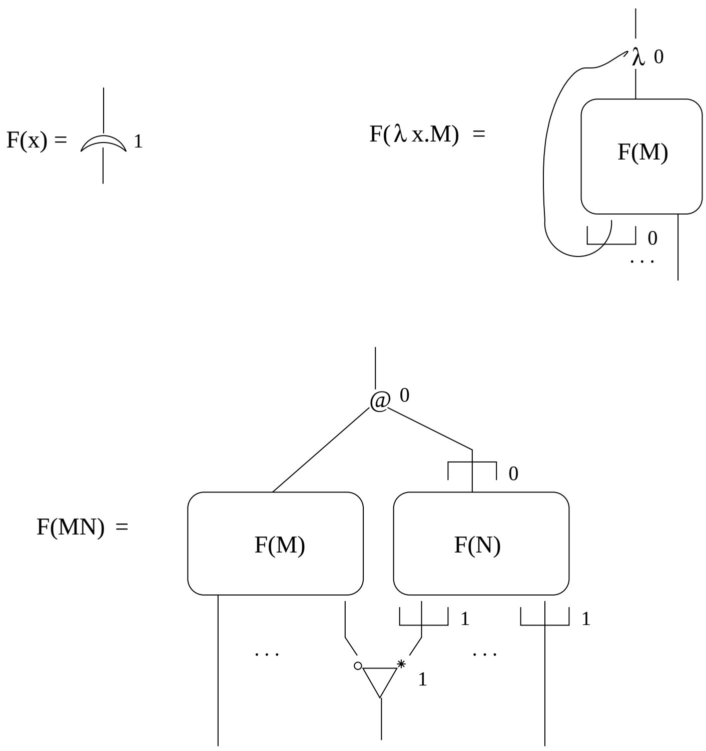
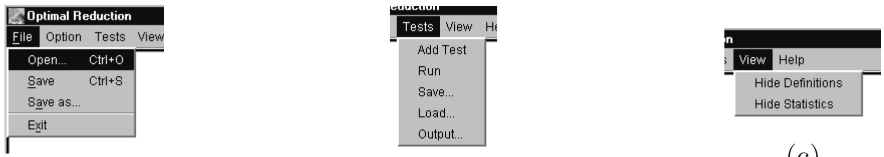


Figure 4.7: Gonthier, Abadi and Lévy initial translation.

   - *Elementary Affine Logic initial translation.* This module implements the type synthesis algorithm described in Section 3.1. If the term to evaluate is typeable



(a) (b) (c)

Figure 4.8: Menus

in Elementary Affine Logic, then it is reducible—and then reduced—inside the abstract Lamping's algorithm. For this translation is available the *safe heuristic*.

2. **Mux implementation**, described in [Gue96]. The initial translation used for this implementation is the Lamping's one. All sharing operators are substituted by a unique node, the multiplexer or mux. The idea behind mux is simple: consider two consecutive fans, one connected through its principal port to the auxiliary port of the other. Such configuration can be thought as—and behaves like—a single ternary fan. In general a tree of fans with n leaves can be substituted by a single mux with n auxiliary doors. Consider now a fan with a croissant at one of its auxiliary doors. The global effect on a node of this configuration is to duplicate and then lower the level of the copy where the croissant is connected. The same effect can be obtained with a single node that duplicates and increase the level of the copy connected to the auxiliary door where there was the croissant. Generalizing this two observations, the mux are generalized fans with $n \geq 1$ auxiliary ports having an integer weight associated. For this module is available the *safe heuristic*. Moreover if the safe option is checked, the reducer tries to apply the absorption rule as described in [GMM96]: if there are two muxes in series, they are collapsed provided that the lower is safe.

All modules use the reduction strategy introduced in [Pey87]: starting from the root of the graph, look for a redex ever exiting from the principal door of the nodes we encounter. There are some options available for the reducers that allow to

- reduce either in head normal form or in full normal form (flag H in Figure 4.1). In the first case the seeking of redexes ends when the algorithm finds a node connected to the root through its principal door. If the full normal form is selected, after the head normal form has been found, the algorithm insert a new temporary root behind the first node and re-start the reduction until either a new head normal form is reached or it reaches a lambda node from its binding port. The entire process continue until the whole graph has been visited;

- either activate or not the garbage collection rules (flag G in Figure 4.1).

The tool has a graphical interface, visible in figures 4.2—4.9, allowing the user to insert, view and delete definitions, evaluate λ-terms and check the chosen options. It is possible to save and load sets of definitions (Figure 4.8 (a)) and moreover it is possible to define, save, load and run a set of tests for a given λ-term (Figure 4.8 (b)). The last option is thought in order to give the opportunity of compare the performances of different reducer with the same input.

Finally the graphical interface shows, after the evaluation of a term, a set of statistics (Figure 4.9) that allow to compare the optimal reducers. The items taken into account for

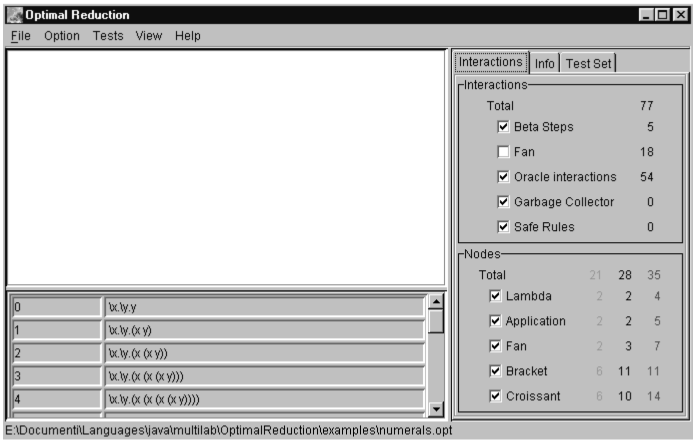


Figure 4.9: The statistics panel.

the statistics are reducer-dependent, i.e. if the selected reducer is the Lamping's implementation with Lamping's initial translation, the statistic panel will show, among the others, the number of optimal beta steps and the number of control interactions. If the selected initial translation is the Elementary Affine Logic one, the statistic panel will not show the number of control interaction because for that reducer there are no control interaction to count (remember that if a term is typeable in Elementary Affine Logic, then is reducible inside the abstract algorithm).

Our aim is to give a tool for the comparison of different graph reducers in terms of unit cost operations. Figures 4.10, 4.12 and 4.14 show a graph of the unit cost operations performed by, respectively, the Lamping's optimal reducer, the Lamping's optimal reducer with additional safe rules and the EAL-optimal reducer—essentially the abstract Lamping's provided that the lambda term has EAL type—during the reduction of the lambda term $((((3\ 2)\ 2)\ \lambda x.x)\ \lambda x.x)$[1] to the complete normal form. The pictures well show how the brackets and croissants interactions are predominant in the first case, the great majority in the second case and how the performances are dramatically improved both with safe rules by Asperti and in particular with EAL-typeable terms. Moreover in Figures 4.11, 4.13 and 4.15 it is shown the number of different nodes during the reduction. Also in this case it is evident the dramatic improvement due to the safe rules (compare Figure 4.11 with Figure 4.13).

Similar improvements are obtained for all Church numerals, booleans, lists and relative operations and in general all the terms used in Chapter 2. In particular it is possible to achieve these performance enhancements for the encoding of the elementary Turing machines.

[1] This particular term is one of the benchmarks used in [Mac00].

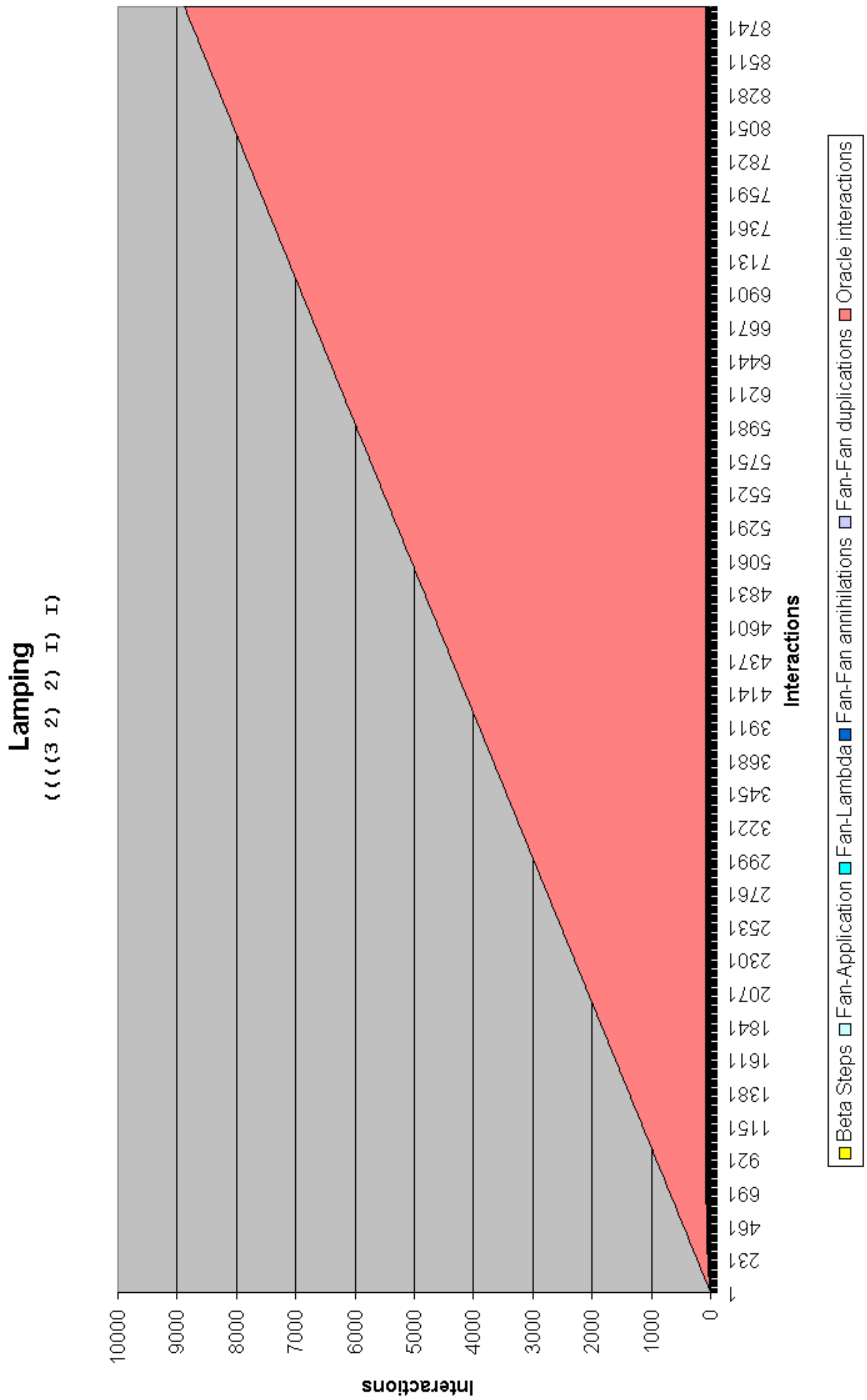


Figure 4.10: Unit cost operations for Lamping's optimal reducer.

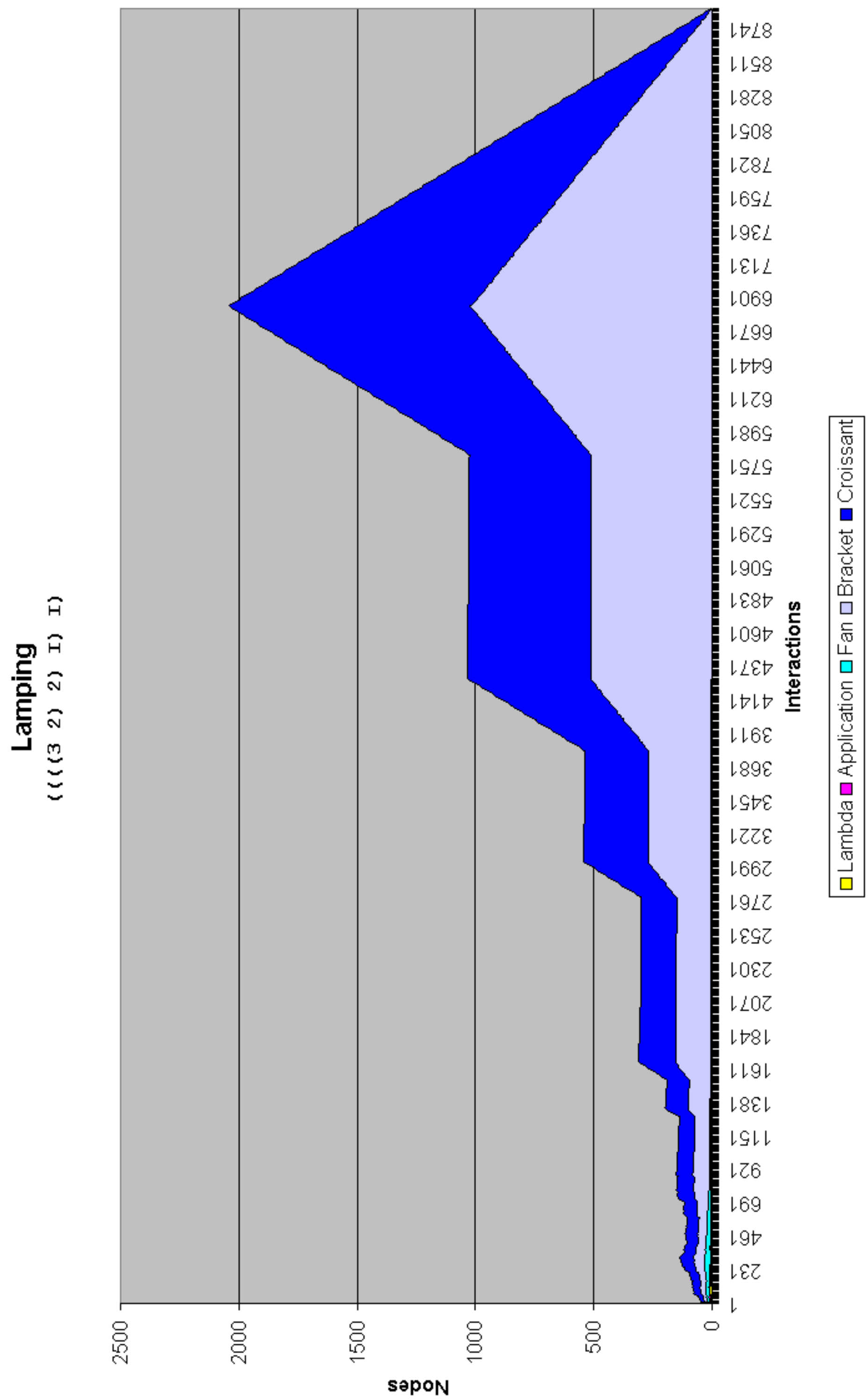


Figure 4.11: Number of graph nodes during the reduction for the Lamping's optimal reducer.

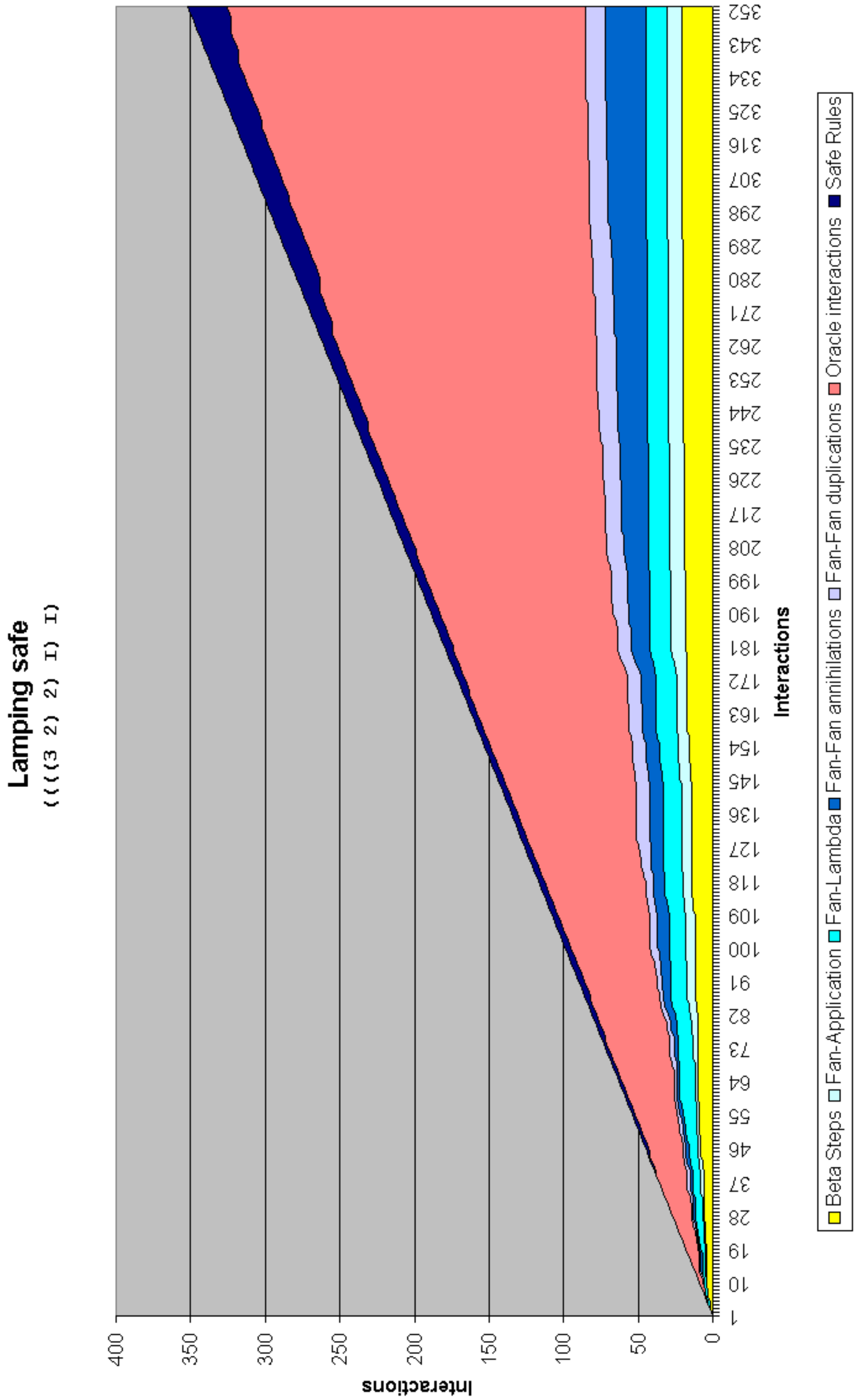


Figure 4.12: Unit cost operations for Lamping's optimal reducer with safe rules.

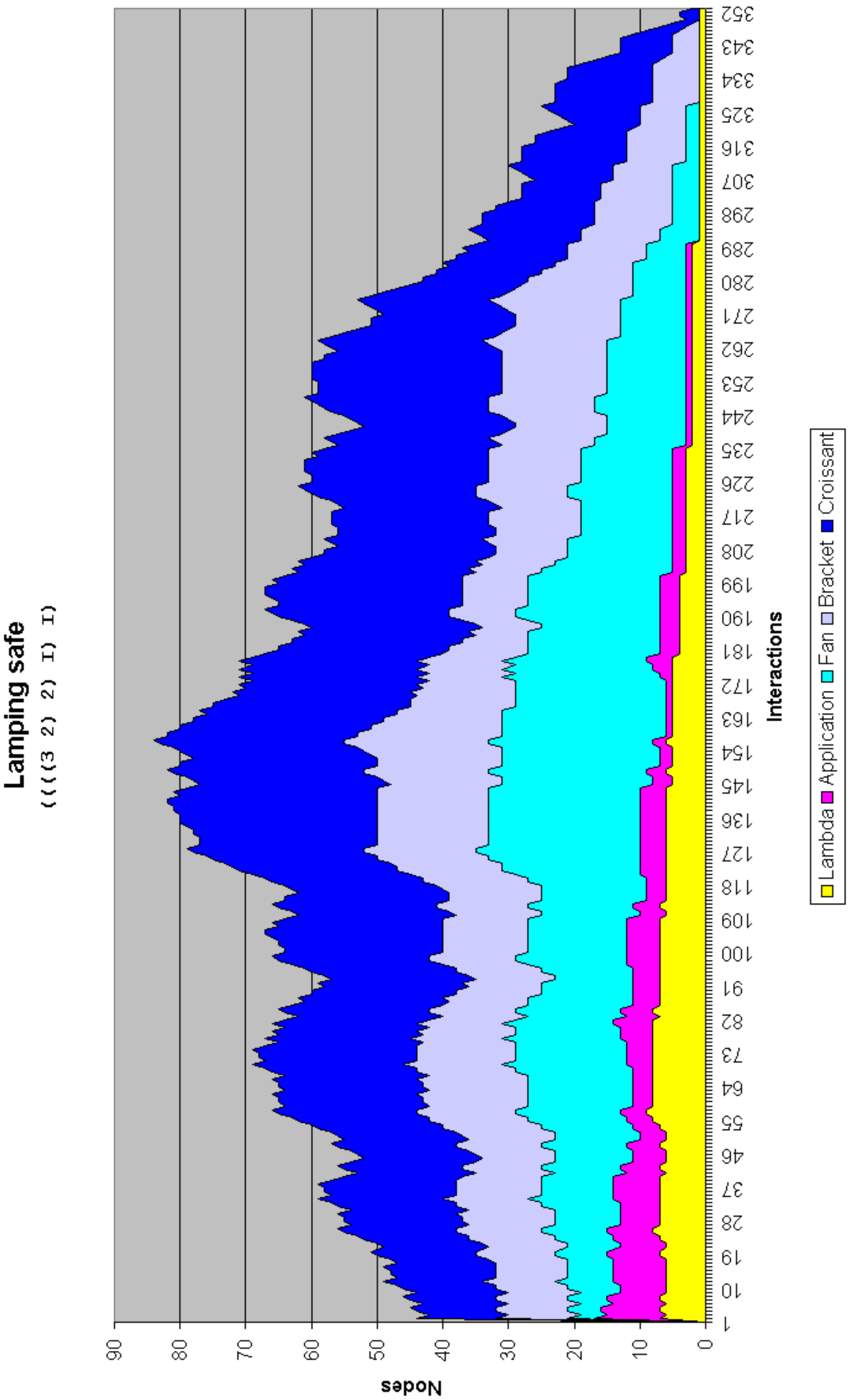


Figure 4.13: Number of graph nodes during the reduction for the Lamping's optimal reducer with safe rules.

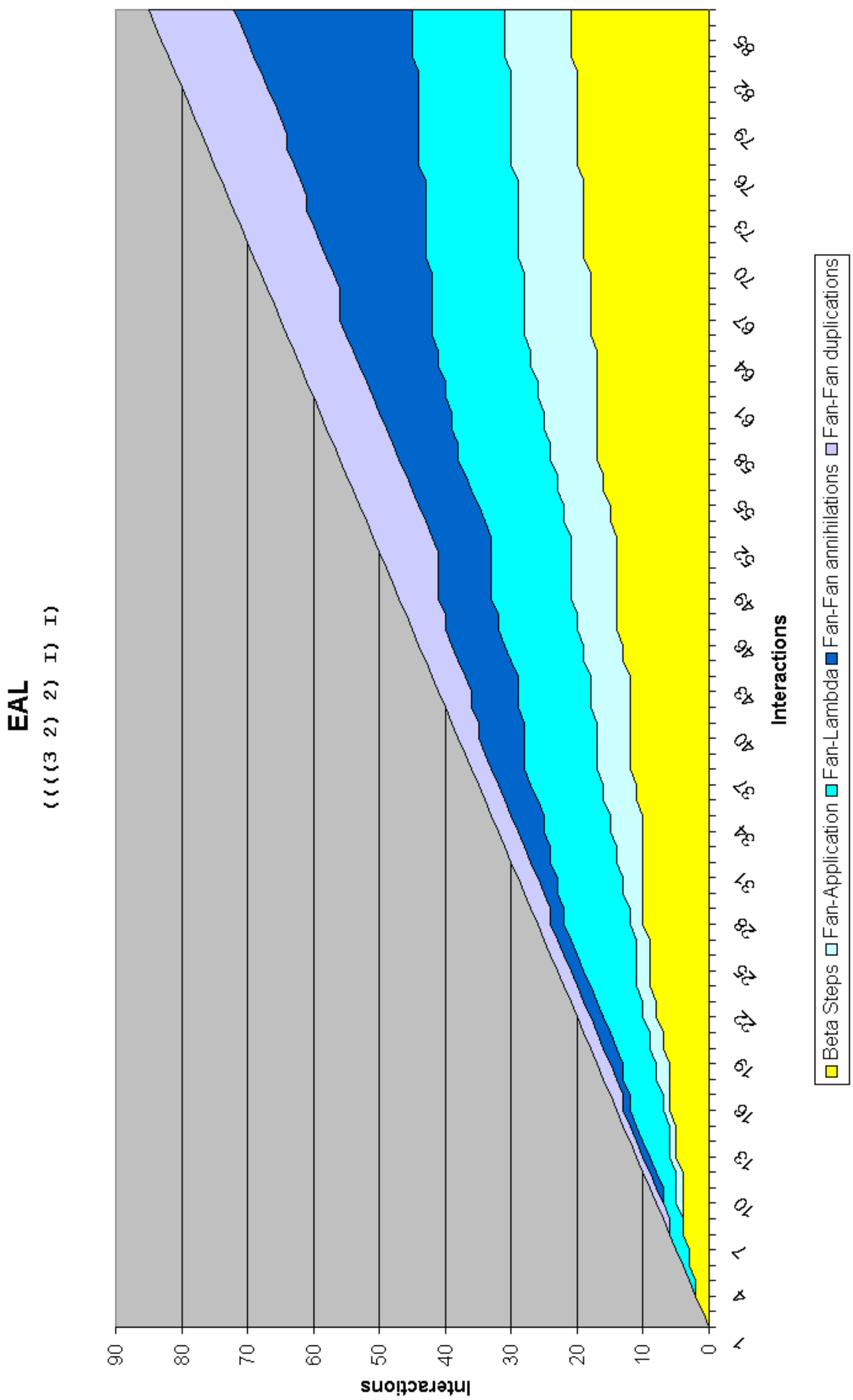


Figure 4.14: Unit cost operations for EAL-reducer.

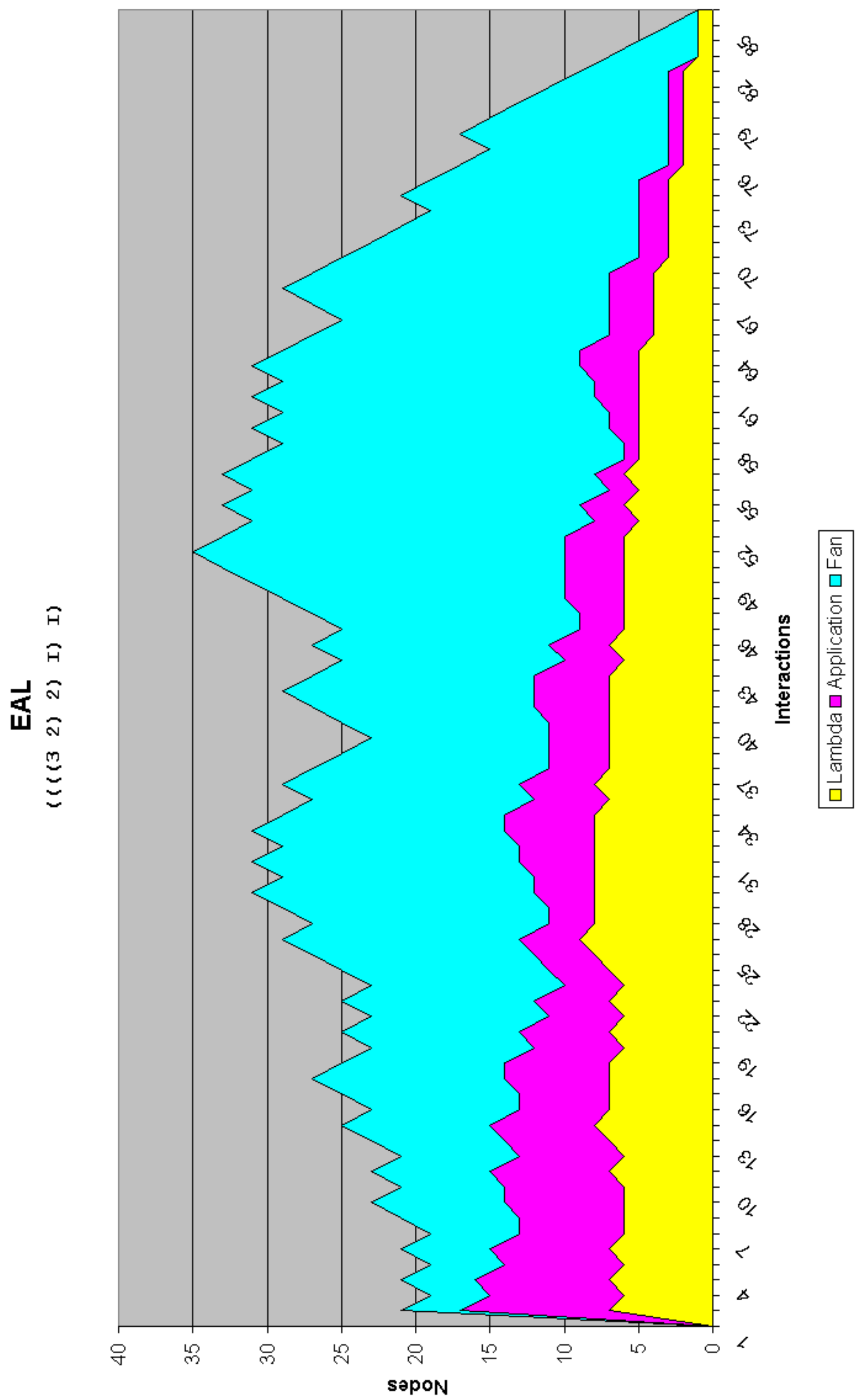


Figure 4.15: Number of graph nodes during the reduction for the EAL-reducer.

4.2 Implementation details

Note

Code in this section is a simplified version of the original one.

All the java classes of the tool are collected in the following packages structure:

- the root package is `OptimalReduction`. At this level of the hierarchy we found only the class `Main` and other classes providing help during the development. For example the class `Debug` provides methods for selective debugging.

```
public class Debug {
  public static boolean ON = false;
  private static final boolean FILE_OUTPUT = true;
  public static final int SyntaxTree_substitution = 0;
...
  public static final int abslampinggraph = 8;

  private static final boolean FILE_OUTPUT = true;
  private static boolean modules[] = {
                                    false,
                                    false,
                                    true,
                                    false,
                                    false,
                                    true,
                                    false,
                                    false,
                                    false
                                   };
```

  Wherever in the code there is the necessity of a debugging message, it is possible to insert a static method call

```
  if (Debug.ON) {Debug.println(<debug message>,Debug.<module>);}
```

  and the message <*debug message*> is printed only if the flag corresponding to `Debug.`<*module*> is `true`. Moreover notice that the flag `Debug.ON` is static and then all debug messages are eliminated at compile time if it is set to `false`.

- The package `OptimalReduction.parser` collects all classes used during precisely the parsing. These files are mostly produced automatically by the JavaCC parser generator. The main class, `OptimalReduction.parser.parser`, returns the syntax tree of the parsed lambda terms. This is a cause of inefficiency. As we have to parse a lambda term in order to translate it into sharing graphs, we could modify the parser in such a way it returns a graph. But doing this way we should build a new parser for every different sharing graph. Moreover we should modify the implementation of each sharing graph in the case we slightly modify the syntax of lambda terms. For these reasons we had preferred to insert an additional level of abstraction.

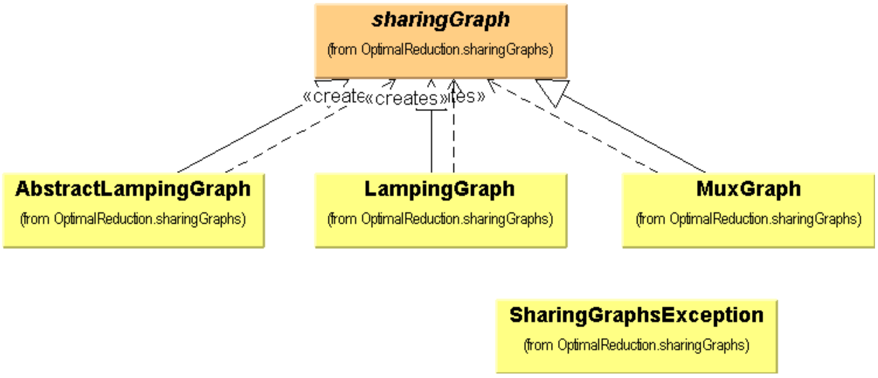


Figure 4.16: Java classes hierarchy for sharing graphs.

- The package `OptimalReduction.syntax` contains the classes implementing the abstraction level of syntax trees of lambda terms.
- `OptimalReduction.types` concern simple type inference and eta expansion.

In the next sections we will see more in depth the packages specific to the implementation of optimal reducers and to the comparison of them.

4.2.1 Sharing graphs

The package `OptimalReduction.sharingGraphs` contains three different implementation of optimal reducers (see Figure 4.16):

1. `LampingGraph`;
2. `MuxGraph`;
3. `AbstractLampingGraph`.

They are all subclasses of the abstract class `sharingGraph`:

```
public abstract class sharingGraph {
  public abstract sharingGraph translate(SyntaxTree term);
  public abstract sharingGraph reduce();
  public abstract String readBack();
...
```

Methods `translate`, `reduce` and `readBack` are implementation-dependent. All subclasses of `sharingGraph` fulfill them in different ways.

All sharing graphs can eventually implement different initial translations and different reduction strategies. For this purpose `sharingGraph` provides the following methods:

```
  public void setInitialTranslation(int translation)
                        throws SharingGraphsException {
    if (implementsInitialTranslation(translation)) {
      initialTranslation = translation;
    } else {
      throw(new SharingGraphsException());
```

```
    }
  }

  public void setReductionStrategy(char strategy)
                        throws SharingGraphsException {
    if (implementsReductionStrategy(strategy)) {
      reductionStrategy = strategy;
    } else {
      throw(new SharingGraphsException());
    }
  }

  public abstract boolean implementsInitialTranslation(int translation);
  public abstract boolean implementsReductionStrategy(char strategy);
```

Moreover any optimal reducer can either activate or not the garbage collector and can reduce either to full or head normal form. The methods for setting these options are common to all implementations and then are fulfilled by `sharingGraph`:

```
  public void setGarbageCollection(boolean value) {
    this.garbagecollection = value;
  }

  public void setHNF(boolean value) {
    this.hnf = value;
  }
```

All three subclasses of `sharingGraph` implement only one reduction strategy (the leftmost-outermost one). The main cycle in the method `reduce` simply visits the graph every exiting from the principal port of the nodes

```
      n = root.principal();
```

and checks every time if it is possible for the current node to interact:

```
  public sharingGraph reduce() {
    if (tryInteraction(root))
      headnormalform = true;
    else {
      n = root.principal();
      this.stack.push(root);
    }
    while (!headnormalform) {
      if (tryInteraction(n)) {
        try {
          n = this.stack.pop();
        } catch (EmptyStackException e) {
          //n was the root
          headnormalform = true;
        }
      } else {
```

```
        this.stack.push(n);
        n = n.principal();
        if (n instanceof freeVarNode)
          try {
            while (true) {
              n = this.stack.pop();
            }
          } catch (EmptyStackException e) {
            headnormalform = true;
          }
      }
    }
    if (!this.hnf) {
      reduceTofullNF();
    }
    return this;
  }
```

The use of the stack is suggested for efficiency reasons. The full normal form is obtained placing a new root after the lambda nodes reached at the end of the head-normal form reduction procedure

```
    RootNode newroot = passOn(currentRoot);
```

and going on reducing to the head normal form the body.

```
  private void reduceTofullNF() throws SharingGraphsException {
    RootNode newroot = passOn(currentRoot);
    if (newroot == null) {//a free var reached passing on nodes
                          //from their principal port
      if (!root.equals(currentRoot))//disconnect provisional root
      ...
      return;//full normal form reached
    } else {
      boolean headnormalform = false;
      if (tryInteraction(newroot))
        headnormalform = true;
      else {
        n = newroot.principal();
        this.stack.push(newroot);
      }
      while (!headnormalform &&
             !(n instanceof freeVarNode) && !n.visited) {
        if (tryInteraction(n)) {
          try {
            n = this.stack.pop();
          } catch (EmptyStackException e) {
            //n was the root
            headnormalform = true;
          }
```

```
        } else {
          this.stack.push(n);
          n = n.principal();
          if (n instanceof freeVarNode || n.visited)
          //empty the stack
            try {
              while (true) {
                this.stack.pop();
              }
            } catch (EmptyStackException e) {}
        }
      }
      if (headnormalform)
        reduceTofullNF();
      else {//found either a visited node or a free variable
```

if there is an application in the current normal form, a new root is placed on top of the argument and the procedure goes on recursively.

```
        RootNode newroot2 = lookForApplication();
        if (newroot2 == null) {//no application node found
          ...
          //eventually disconnect provisional root
          ...
          return;//full normal form reached
        } else {
          reduceTofullNF();
        }
      }
    }
  }
```

Finally `tryInteraction(n)` inserts an additional level of abstraction: code implementing local graph rewriting rules and also safe rules is written here. This allow the reuse of code when varying reduction strategy or reduction rules.

4.2.2 Graph nodes

Package `OptimalReduction.sharingGraphs.nodes` collect all classes implementing the agents, in the interaction nets terminology, of the sharing graphs. Figure 4.17 shows the hierarchy.

A `GraphNode` is simply a composed by a principal port, a set of auxiliary ports, the relative methods for the access and modification of the ports and a method returning a copy of the node (this is useful during the reduction, since graph nodes may be duplicated):

```
public class GraphNode {
  // Array of ports. ports[0] is the principal port.
  protected Port[] ports;

  public Port getPrincipalPort() {
    return ports[0];
```

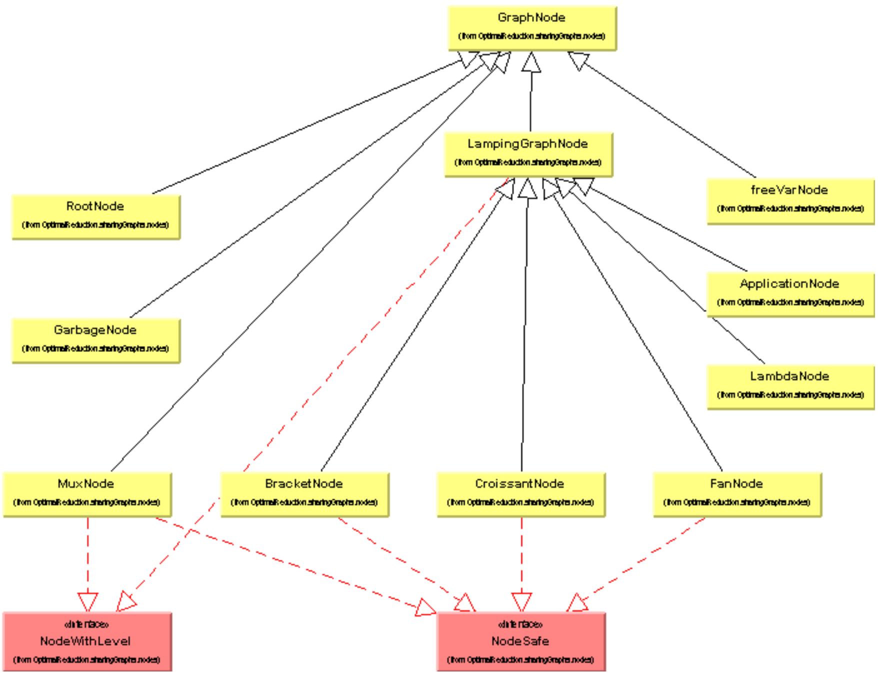


Figure 4.17: Java classes hierarchy for nodes.

```
  }
  protected void setPrincipalPort(Port p) {
  ...
  }
  protected void setAuxiliaryPort(Port p, int i) {
  ...
  }
  protected Port getAuxiliaryPort(int i) {
  ...
  }
  public Object copy() {
  ...
  }
}
```

A `LampingGraphNode` is a graph node with a level:

```
public class LampingGraphNode extends GraphNode
                              implements NodeWithLevel {
  public int level;
...
```

`RootNode`, `freeVarNode` and `GarbageNode` are all graph nodes with no auxiliary ports.

All the other classes extend `LampingGraphNodes` but `MuxNode`. Moreover the package provides an interface for the safe rules `NodeSafe`. All classes implementing such interface can interact when, during the reduction, a safe rule is performed.

Notice that the code implementing the interaction rules is not inside the classes of this package. A graph node, in general, can not "know" how to interact with another node. The

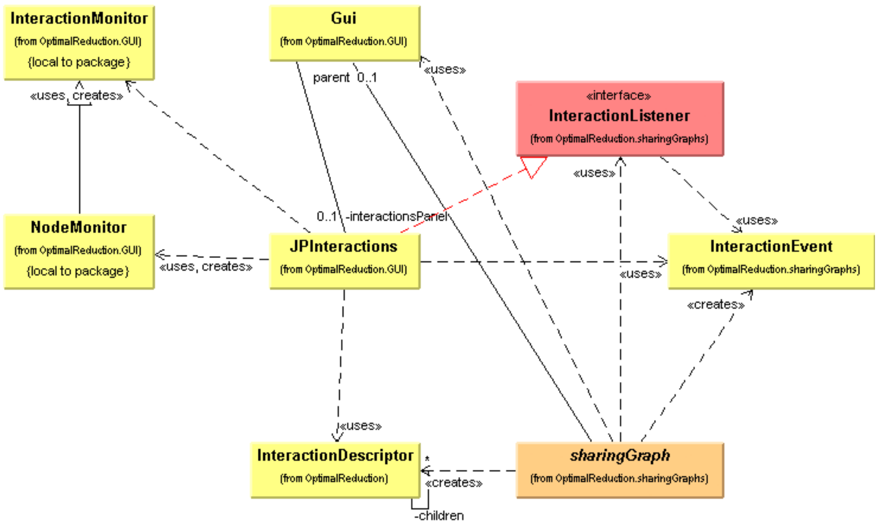


Figure 4.18: Java classes hierarchy for the GUI and statistics.

needed knowledge reside at level of the optimal reducer (the `tryInteraction` method).

4.2.3 Graphical User Interface

The class `Gui` of package `OptimalReduction.GUI` implements the user interface. All methods for the input/output are here. However, the most important feature implemented in this package is the management of statistics (see Figure 4.18).

Since the reduction of a lambda term can be a long process, all sharing graphs are implemented as threads running concurrently with the main one. Every time an interaction occurs, the sharing graph creates a new interaction event:

```
public class InteractionEvent extends EventObject {
...
```

In this way the graphical interface gives an immediate feedback during the reduction. Moreover it is possible to extend the tool in order to execute multiple reducers at the same time.

We have extended the `EventObject` of the java hierarchy in order to be conform to the java events management. Actually the GUI contains one or more components managing the interaction statistics (`JPInteractions`) that implements the `InteractionListener` interface. When a sharing graph creates a new event, it dispatches the event to the list of its registered listeners. This mechanism mimic the one used with Swing/JFC of Java2.

`JPInteractions` shows the evolution of interactions and nodes using two additional classes: `InteractionMonitor` and `NodeMonitor`. Moreover `JPInteractions` records the history of the computation and eventually outputs it in a file. Graphs of figures 4.10–4.15 are obtained using this feature.

```
public class JPInteractions extends JPanel
                            implements InteractionListener {
  private ArrayList history = new ArrayList();
  private PrintWriter historyFile;
  ...
  public void printHistory(PrintWriter out) {
  ...
```

Conclusions

We believe that an efficient implementation of the lambda calculus could exist. Lamping's implementation of Lévy's optimal reduction is a good starting point. We think that the computational overhead due to the accumulation of control nodes can be resolved using the safe rules of Asperti and refining the initial translation of lambda terms in sharing graphs. The decoration procedure explained in Chapter 3 can be easily extended to full Linear Logic and it is possible to calculate, for a given simply typed lambda terms, its decoration with minimal use of δ and ϵ rules, hence it is possible to find the initial translation with the minimal number of brackets and croissants. It remains to investigate the extensions to other logics of first and second order.

For what concerns the study of the complexity of lambda reduction we can sum up the work done so far with the following table:

[Sta79]	Typed λ-calculus is not elementary recursive.
[Sch82]	Number of β in the reduction of simply typed lambda terms is $\Omega\left(2^{\cdot^{\cdot^{\cdot^{2^n}}}}\right)$ $\exists$ a strategy s.t. $\#\beta \leq f \in \mathcal{E}^4$.
[FS91]	$\exists M$ s.t. $\mu_\beta(M) > 5\mu_\pi(M)$. Propose a new measure ν. Turner and Hughes implementations are $2^{\Omega(\nu)}$.
[LM96, LM97]	Abstract Lamping's algorithm is $\Omega\left(2^\nu\right)$. Propose two new measures based on Lévy's labels.
[AM98]	Optimal beta is not elementary recursive.

We proved in Chapter 2 that also optimal duplication is not elementary recursive.

What is a good measure of the reduction of functional programs remains an open question. We think that the proposal of Lawall and Mairson of considering the sum of new labels generated during the reduction as complexity measure is the most promising one. From one side, it introduces an element relative to the length of the reduction chain; from the other, it is implementation independent.

Further works will include the study and the characterization of the class of lambda terms reducible with the abstract Lamping's algorithm. The canonical forms of EAL can help in the understanding of possible regularity of lambda terms in this class.

It will be interesting to extend our type inference algorithm to a polynomial logic. At the moment the two most promising candidates are Light Affine Logic and Soft Linear Logic. In this way we will have a procedure to prove the polynomial complexity of the reduction of a given lambda term.

In Section 3.2 we defined the set of EAL-types of a lambda term referring to the set of EAL-terms contracting at most variable. Removing this constraint could be a possible extension of our work. In this case the existence of an EAL-type for a lambda terms will imply that exist a sharing graph whose read back is the term itself and it is reducible with the Lamping's abstract algorithm, but we should abandon the usual translation and adopt a new one taking care of the contraction imposed by the EAL-type derivation.

Finally, from the practical point of view, it will be worth to enrich the language of the optimal reduction tool of Chapter 4 adding naturals, boolean, lists and relative operations. Extending the type inference in EAL and the benefits of reduction with oracle $O(1)$ to this new language, will allow us to compare a prototype of optimal reducer with other implementation of functional languages using the benchmarks available in the literature. Clearly we should translate and refine the code in a more efficiently executable language than Java (for example C++).

Bibliography

[AC97] Andrea Asperti and Juliusz Chroboczek. Safe operators: brackets closed forever. *Applicable Algebra in Engineering, Communication and Computing*, 8(6), 1997.

[ACM00] Andrea Asperti, Paolo Coppola, and Simone Martini. (Optimal) duplication is not elementary recursive. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 96–107, N.Y., January 19–21 2000. ACM Press.

[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.

[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.

[Asp94] Andrea Asperti. Linear logic, comonads and optimal reductions. *Fundamentae Informaticae*, 22(1):3–22, 1994. Special Issue devoted to Categories in Computer Science.

[Asp95] Andrea Asperti. $\delta \circ ! \epsilon = 1$: Optimizing optimal λ-calculus implementations. In Jieh Hsiang, editor, *Proceedings of the 6th International Conference on Rewriting Techniques and Applications (RTA-95)*, volume 914 of *Lecture Notes in Computer Science*, pages 102–116, Berlin, April 5–7 1995. Springer-Verlag.

[Asp96] Andrea Asperti. On the complexity of beta-reduction. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 110–118, New York, NY, USA, 1996. ACM Press.

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

[Bac78] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, august 1978. ACM Turing Award Lecture.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus - Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam - New York - Oxford, 2 edition, 1984.

[BBdPH93] Nick Benton, Gavin Bierman, Valeria de Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. In M. Benzen and J.F. Groote, editors, *Typed Lambda Calculus and Applications. Int. Conference on Typed Lambda Calculus and Applications, TLCA'93*, volume 664 of *Lecture Notes in Computer Science*, pages 75–90, March 1993.

[Bru72] Nicolaas G. De Bruijn. Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[CM01] Paolo Coppola and Simone Martini. Typing Lambda Terms in Elementary Logic with Linear Constraints. In Samson Abramsky, editor, *Proc. of Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, volume 2044 of *Lecture Notes in Computer Science*, pages 76–90. Springer, may 2001.

[Cop97] Paolo Coppola. Complessità e Riduzione Ottimale nel Lambda Calcolo. Master's thesis, Università degli Studi di Udine, 1997.

[CR91] William Clinger and Jonathan A. Rees. The Revised^4 Report on the Algorithmic Language Scheme. *ACM LISP Pointers*, 4(3):1–55, 1991.

[DJS95] Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. On the linear decoration of intuitionistic derivations. In *Archive for Mathematical Logic*, volume 33, pages 387–412, 1995.

[FS91] Gudmund S. Frandsen and Carl Sturtivant. What is an efficient implementation of the λ-calculus? In John Hughes, editor, *Proceedings of Functional Programming Languages an Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 289–312, Berlin, Germany, August 1991. Springer.

[GAL92] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. Linear logic without boxes. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 223–234, Santa Cruz, California, 22–25 June 1992. IEEE Computer Society Press.

[Gan80] Robin O. Gandy. Proofs of strong normalization. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 457–477. Academic Press, London, 1980.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[Gir98] Jean-Yves Girard. Light linear logic. *Information and Computation*, 204(2):143–175, 1998.

[GMM96] Stefano Guerrini, Simone Martini, and Andrea Masini. Coherence for sharing proof nets. In Harald Ganzinger, editor, *Proceedings of the 7th International Conference on Rewriting Techniques and Applications (RTA-96)*, volume 1103 of *Lecture Notes in Computer Science*, pages 215–229, New Brunswick, NJ, USA, July 27–30 1996. Springer-Verlag.

[Grz53] Andrzej Grzegorczyk. Some classes of recursive functions. *Rozprawy Matematyczne*, 1953.

[Gue96] Stefano Guerrini. *Theoretical and Pratical Issues of Optimal Implementation of Functional Languages*. Phd thesis, Dipartimento di Informatica, Universitá di Pisa, Pisa, 1996. TD-3/96.

[Hen63] Leon Henkin. A theory of propositional types. *Fundamenta Mathematicae*, 52:323–344, 1963.

[Hug82] R. John M. Hughes. Super combinators: A new implementation method for applicative languages. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10. ACM, ACM, August 1982.

[Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. *ACM SIGPLAN Notices*, 19(6):58–69, June 1984.

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

[Laf01] Yves Lafont. Soft Linear Logic and Polynomial Time. ftp://iml.univ-mrs.fr/pub/lafont/soft.ps.gz, 2001.

[Lag01] Ugo Dal Lago. Semantica delle fasi per logiche lineari elementari. Master's thesis, Università degli Studi di Udine, 2001.

[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.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

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

[Lév80] Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, London, 1980.

[LM96] Julia L. Lawall and Harry G. Mairson. Optimality and inefficiency: What isn't a cost model of the lambda calculus? In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 92–101, Philadelphia, Pennsylvania, 24–26 May 1996.

[LM97] Julia L. Lawall and Harry G. Mairson. On global dynamics of optimal graph reduction. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 188–195, Amsterdam, The Netherlands, 9–11 June 1997.

[Mac00] Ian Mackie. Interaction nets for linear logic. *Theoretical Computer Science*, 247(1-2):83–140, september 2000.

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

[McC62] John McCarthy. *Lisp 1.5 Programmer's Manual.* Cambridge, Massachussets, 1962.

[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.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.

[Pap94] Christos H. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Computer Science. Prentice-Hall, 1987.

[Pin01] Jorge Sousa Pinto. Parallel Implementation Models for the λ-Calculus Using the Geometry of Interaction. In Samson Abramsky, editor, *Proc. of Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, volume 2044 of *Lecture Notes in Computer Science*, pages 385–399. Springer, may 2001.

[PQ00] Marco Pedicini and Francesco Quaglia. A parallel implementation for optimal lambda-calculus reduction. In *Proceedings of the 2nd Imternational ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)*, pages 3–14, N.Y., September 20–23 2000. ACM Press.

[PW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In ACM, editor, *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1993*, pages 71–84, New York, NY, USA, 1993.

[Rov98] Luca Roversi. A Polymorphic Language which is Typable and Poly-step. In *Proceedings of the Asian Computing Science Conference (ASIAN'98)*, volume 1538 of *Lecture Notes in Computer Science*, pages 43 – 60, Manila (The Philippines), December 1998. Springer Verlag.

[Sch82] Helmut Schwichtenberg. Complexity of normalization in the pure typed lambda-calculus. In A. S. Troelstra and D. van Dalen, editors, *Proceedings L. E. J. Brouwer Centenary Symp., Noordwijkerhout, The Netherlands, 8–13 June 1981*, volume 110 of *Studies in Logic and the Foundations of Mathematics*, pages 453–457. North-Holland, Amsterdam, 1982.

[Sch94] Harold Schellinx. *The Noble Art of Linear Decorating.* PhD thesis, Institute for Logic, Language and Computation, University of Amsterdam, 1994.

[Sch01] Aleksy Schubert. The Complexity of β-Reduction in Low Orders. In Samson Abramsky, editor, *Proc. of Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, volume 2044 of *Lecture Notes in Computer Science*, pages 400–414. Springer, May 2001.

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

[Ste84] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, 1984.

[Tur76] David A. Turner. The SASL language manual. Technical report, University of Kent, Canterbury, U.K., 1976.

[Tur79] David A. Turner. A new implementation technique for applicative languages. *Software – Practice and Experience*, 9:31–49, 1979.

[Tur85] David A. Turner. Miranda: a non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, September 1985.

[vEP93] Marko van Eekelen and Rinus Plasmeijer. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[Wad71] Christophe P. Wadsworth. *Semantics and pragmatics of the lambda calculus*. Ph.D. thesis, Programming Research Group, Oxford University, September 1971.