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.unipd.it
Contents

Introduction v

I The Complexity of Reduction 1

1 Optimality and Complexity 3
  1.1 Harbingers ....................................................... 4
    1.1.1 The typed λ-calculus is not elementary recursive .......... 4
    1.1.2 Counting beta reductions in the pure typed lambda-calculus . . .... 6
    1.1.3 Optimality by Lévy ........................................ 7
  1.2 Looking for efficient implementations of functional languages .... .... 13
  1.3 Lamping’s Algorithm ........................................... 15
    1.3.1 Sharing Graphs ............................................. 16
    1.3.2 The Full Algorithm ....................................... 21
    1.3.3 Initial Encoding .......................................... 26
    1.3.4 Read-back .................................................. 28
  1.4 The complexity of Lamping’s algorithm .................................. 30
  1.5 Inherent complexity of implementing optimal reduction ............... 33
    1.5.1 The general case .......................................... 33
    1.5.2 The polynomial case ....................................... 34
  1.6 Conclusions .................................................... 36

2 Complexity of optimal sharing 37

2.1 Linear Logic .................................................... 37
  2.1.1 Elementary Affine Logic [Gir98, Asp98] ........................ 41
  2.1.2 Light Affine Logic [Gir98, Asp98] ............................. 43
  2.1.3 Soft Linear Logic [Laf01] .................................... 43
  2.2 Decorating terms ................................................. 44
  2.3 Coding type theory into EAL proofs .................................. 46
  2.4 Conclusions ..................................................... 70

3 EAL-typing 73

3.1 Type inference in EAL ............................................. 73
  3.1.1 NEAL ......................................................... 75
  3.1.2 Example of type inference .................................... 79
  3.1.3 Type Inference .............................................. 90
  3.1.4 The full algorithm .......................................... 91
  3.1.5 Properties of the Type Synthesis Algorithm .................... 98
CONTENTS

3.2 Principal type ................................................................. 105
  3.2.1 Abstract EAL-terms ................................................... 105
  3.2.2 Principal Typing for \( \vdash_{abs} \) ................................ 109
  3.2.3 Canonical Forms ....................................................... 111
  3.2.4 Canonical Forms Algorithm \( C \) .................................. 116
  3.2.5 Conclusions .............................................................. 121

II The Implementation of Functional Languages 123

4 Optimal Reducers 125
  4.1 A tool for reducers comparison ....................................... 125
  4.2 Implementation details ................................................ 138
    4.2.1 Sharing graphs .................................................... 139
    4.2.2 Graph nodes ........................................................ 142
    4.2.3 Graphical User Interface ....................................... 144

Conclusions 145

Bibliography 147
“... 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 \textit{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 \((\lambda x. M N) \rightarrow_{\beta} M\{^N/x\}\) 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
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, $F' = \lambda a. \cdots a \cdots$ and $A'$ and then contracting $(F' \ A')$ to $\cdots A' \cdots$. 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 order 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 $\lambda I$-calculus [Bar84] and the $\lambda I$-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
\[
(\lambda g. (g \lambda x.x)) \lambda h. (\lambda f. (f (\lambda z.z)) \lambda w. (h(w \lambda y.y)))
\]
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 \(\lambda\)-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 \(\beta\)-conversion
between simple typed lambda terms. We give Statman’s encoding in the next sections.

Type theory

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

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

In \(\Omega\) we can define equality in the following way:
\[
x^k =_k y^k \iff \forall z^{k-1} (z^{k-1} \in x^k \iff z^{k-1} \in y^k).
\]
Then, every element \( D \in \mathcal{D}_n \) can be identified with the set of its elements, i.e. if \( D = \{d_1, \ldots, d_m\} \), it can be expressed in \( \Omega \) as \( \{x^{n-1} \mid x^{n-1} = d_{m-1} \lor \cdots \lor x^{n-1} = d_1 \} \).

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

**Proof:** See [Mey74]. \( \square \)

**Statman’s Encoding**

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

In order to encode \( \Omega \)-formulas we need the following simple typed terms:

\[
\begin{align*}
  a_1 &= 2 : \mathbb{N}_0 \\
  a_{n+1} &= (a_n \{o \to o/o\} \ a_1) : \mathbb{N}_0
\end{align*}
\]

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

\[
\begin{align*}
  e_0 &= \lambda x.\lambda y.((\text{sum}(\text{mul}(\text{sg}(x))(\overline{s}g)(y)))(\text{mul}(\text{sg}y)(\overline{s}g)x))) : \mathbb{N}_0 \to (\mathbb{N}_0 \to \mathbb{N}_0) \\
  \forall_1 &= \lambda h.(\text{sum}(h)(h1)) : \mathbb{N}_1 \to \mathbb{N}_0 \\
  C &= \lambda g.(\text{sum}(g(\lambda x.1))(g(\lambda x.x))) : \mathbb{N}_2 \to \mathbb{N}_0 \\
  p_{n+1}(x, y) &= (C \ \lambda f.(\forall_n \ \lambda w.(\lambda y.((\text{mul}(f(e_nwy)(xy))))))) : \mathbb{N}_0 \\
  e_{n+1} &= \lambda x.\lambda y.((\forall_n \ \lambda z.(e_0(xz))(yz))) : \mathbb{N}_{n+1} \to (\mathbb{N}_{n+1} \to \mathbb{N}_0) \\
  \forall_{n+1} &= \lambda y.((a_{n+1}\{\mathbb{N}_{n+2}/o\} \ \lambda z.\lambda x.\lambda w.1)(x, z, y)) : \mathbb{N}_{n+2} \to \mathbb{N}_0
\end{align*}
\]

The encoding \( (\cdot)^* \) function from \( \Omega \)-formulas to simply-typed lambda terms is the following:

\[
\begin{align*}
  (0)^* &= 0 \\
  (1)^* &= 1 \\
  (x^n)^* &= x : \mathbb{N}_n \\
  (t_1 \in t_2)^* &= (\text{sg}((t_1)^*(t_2)^*)) \\
  (A \land B)^* &= (\text{sg}((A)^*(B)^*)) \\
  (\neg A)^* &= (\overline{s}g(A)^*) \\
  (\forall x A)^* &= (\text{sg}((\forall_n \ \lambda x^{\mathbb{N}_n}.(A)^*)))
\end{align*}
\]

Consider now the definition of an object of type \( n \) as

\[
\begin{align*}
  \text{def}^0(0) &= \{0\} \\
  \text{def}^0(1) &= \{1\} \\
  \text{def}^{n+1} (\alpha) &= \{\lambda y^{\mathbb{N}_n} (\text{mul} \ r_1 (\cdots (\text{mul} \ r_{s(n+1)} (\lambda w^{\mathbb{N}_n} 1 \ y)) \cdots) | \\
  & \quad r_i = 1 \text{ or } r_i = (e_n \ t \ y) \text{ for } t \in \text{def}^n (\beta) \}
\end{align*}
\]

for each \( \beta \in \alpha \) for some \( t \in \text{def}^n (\beta) \) there is some \( i \) s.t. \( r_i = (e_n \ t \ y) \}

We can state the following proposition:
Proposition 2 Suppose $\alpha, \beta \in D_n$, $\gamma \in D_{n+1}$, $t_1 \in \text{def}^n(\beta)$, $t_2 \in \text{def}^n(\alpha)$, and $t_3 \in \text{def}^{n+1}(\gamma)$, then

1. $\beta = \alpha$ iff $(e_n t_1 t_2) \to 0$

2. $\beta \in \gamma$ iff $(t_3 t_1 t_2) \to 0$.

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

Theorem 3 Let $A(x_1^{n_1}, \ldots, x_m^{n_m})$ be an $\Omega$-formula with free variables $\{x_1^{n_1}, \ldots, x_m^{n_m}\}$, and let $\alpha_i \in D_{n_i}$ and $t_i \in \text{def}^{n_i}(\alpha_i)$ for $1 \leq i \leq m$, then $A[\alpha_1/x_1^{n_1}, \ldots, \alpha_m/x_m^{n_m}]$ is true if and only if the simple typed lambda term $(\lambda x_1^{N_1} \cdots \lambda x_m^{N_m}. (A)^* t_1 \cdots t_m)$ reduces to 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 $\lambda$-symbol, i.e. $\text{length}(\lambda x.x) = \text{length}(\lambda x.\lambda y.y) = 1$. 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 $\ell_n$ 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

$$s(n) \leq \text{length of the normal form of } a_n \leq (\text{length of } a_n)^{2^n} = (3n)^{2^n} \leq 2^{2n+t_n} \quad (\text{since } 3n \leq 2^n)$$

□

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

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

The third level of Grzegorczyk hierarchy is the class of elementary functions, i.e. \( E = E_3 \), 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 \( \tau \) be defined inductively by \( l(o) = 0 \), \( l(\sigma \rightarrow \tau) = \max\{l(\sigma) + 1, l(\tau)\} \) 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 \( E^4 \) upper bounded.

**Theorem 5**

1. There is an \( E^4 \) 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 \( \leq f(\max\{\text{length}(M), l(M)\}) \).

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 \( \leq g_m(\text{length}(M)) \).

Finally Schwichtenberg shows that combining his result with those of Gandy in [Gan80], one can obtain a universal \( E^4 \) 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 \( \tau \), 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 \( E^4 \) 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 \( \max\{\text{length}(M), l(M)\} \). Since by Gandy’s construction of \( \overline{M} \), \( \text{length}(\overline{M}) \) depends only linearly on \( \text{length}(M) \) and \( l(\overline{M}) = l(m) \), we also have an \( E^4 \) upper bound on the number of standard beta reduction steps in terms of \( \max\{\text{length}(M), l(M)\} \). Next note that any standard beta reduction step at most squares the length of the original term. So we have an \( E^4 \) upper bound on the length (and hence on the numerical value) of the normal form of \( \overline{M} \), again in terms of \( \max\{\text{length}(M), l(M)\} \). 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 [Lévy78, Lévy80]. 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 [Lévy80]

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 of parallel reduction contracting singletons.

Definition 3 (Residuals) Let \( R \) be a redex in a lambda term \( M \) and let \( \rho : M \rightarrow N \) be a reduction (a sequence of redexes) from \( M \) to \( N \).

The set of residuals of \( R \) by \( \rho \) is defined by induction on the length of \( \rho \):

\[
R/0 = \{R\} \quad \text{where 0 is the empty reduction}
\]

\[
R/\rho \sigma = \{T \mid \forall S \in R/\rho(T \in S/\sigma)\}.
\]

If \( \rho \) consists of a single 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.
**Definition 4** A reduction $\rho = R_1 \ldots R_n \ldots$ is relative to $F$ iff

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

Moreover $\rho$ is a development of $F$ iff $\rho$ is relative to $F$ and $F/\rho = \emptyset$.

**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 $\rho$ and $\sigma$ 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 $M \xrightarrow{F_1} M_1 \xrightarrow{F_2} \cdots \xrightarrow{F_n} M_n$ 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 $F \uparrow C = F(C/F)$.

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

1. $F \uparrow C$ and $C \uparrow F$ end at the same expression,
2. $H/(F \uparrow C) = H/(C \uparrow F)$ 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. $F \uparrow C \equiv C \uparrow F$ 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:

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

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

$$0/\rho = 0$$

$$(\sigma F)/\rho = (\sigma/\rho)(F/(\rho/\sigma))$$

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 $\rho R$ as the redex occurrence $R$ with history $\rho$.

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

**Definition 8 (Family equivalence $\simeq$)** Two redexes $R$ and $S$ with histories $\rho$ and $\sigma$ 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 $\tau T$ 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 $\rho R$ with respect to $\simeq$ and let $\rho = F_1 F_2 \ldots F_n \ldots$. The set of family classes of the redexes contracted in $\rho$ is defined in the following way:

$$\text{FAM}(\rho) = \{[F_1 F_2 \ldots F_{i-1} R_i] \mid R_i \in F_i, i \geq 1\}$$

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

**Definition 9** $\rho$ is relative to $X$ if $\text{FAM}(\rho) \subseteq X$.

A reduction $\rho$ relative to $X$ is a development of $X$ if there is no redex $R$ such that $[\rho R] \in X$.

The finite developments theorem can be easy generalized to the definition above.
Definition 10 (Complete reduction) A parallel reduction $F_1F_2\ldots F_n\ldots$ is complete iff for every $n \geq 1$, $F_n \neq \emptyset$ is a maximum set of redexes such that, for all $R \in F_n$ and $S \in F_n$, $F_1F_2\ldots F_{n-1}R \simeq F_1F_2\ldots F_{n-1}S$.

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

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

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

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 $\rho$ is terminating iff its final expression is in normal forms.

Let $\rho = F_1F_2\ldots F_n\ldots$ and let $\mathcal{R}(\rho)$ be the set of redexes one of whose residuals is contracted in $\rho$:

$$\mathcal{R}(\rho) = \{ R \mid R/F_1F_2\ldots F_{i-1} \cap F_i \neq \emptyset, i \geq 1 \}.$$

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

- A parallel reduction $\rho = F_1F_2\ldots F_n\ldots$ 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 $\rho$ is a set of copies of a single redex iff there is one redex $S$ with history $\sigma$ such that $\sigma S \leq \rho R$ for every $R \in F$.

The reduction $\rho = F_1F_2\ldots F_n\ldots$ 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:

$$\text{cost}(\rho) = |\rho|$$

(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 $\lambda$-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 $\lambda$-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 $\infty$). We define also $\mu_\beta(M)$ as the minimum number of standard beta reduction steps required to reduce $M$ (as for $\mu_\pi$, if $M$ is not reducible then $\mu_\beta(M) = \infty$), and $l_{\text{normal}}(M)$ 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 $l_{\text{normal}}(M)$. Notice that this is true also if $M$ has no normal form since in that case $l_{\text{normal}}$ is simply $\infty$.

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 $\mu_\pi$ as a component of the input size parameter. Other candidates could be $\mu_\beta$ and $l_{\text{normal}}$, but $l_{\text{normal}}$ 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 $l_{\text{normal}}(A_k) = 2^{\Omega(\mu_\beta(A_k))}$.

**Proof:** Let

$$A_0 = \lambda x.x$$
$$A_k = (\lambda x.(x x) A_k)$$

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:
\[
\begin{align*}
S(0) &= 0 \\
S(k) &= 2S(k-1) + 2 \\
T(0) &= 0 \\
T(k) &= T(k-1) + 2
\end{align*}
\]
that have solutions
\[
\begin{align*}
S(k) &= 2^{k+1} - 2 \\
T(k) &= 2k
\end{align*}
\]
from which the proposition follows. \(\square\)

Thus measuring the complexity of an implementation using \(l_{\text{normal}}\) can give unrealistically optimistic assessments in the sense that the predicted cost is much less than the actual cost. \(\mu_\pi\) is better than \(\mu_\beta\) for a number of reasons. First, it is known that \(\mu_\pi\) is always less then \(\mu_\beta\). Second, we have seen in Section 1.1.3 that an optimal strategy is known for parallel reductions. Furthermore \(\mu_\pi\) and \(\mu_\beta\) 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 \(M_n = (\lambda x.(2 \cdot (2 \cdot (2 \cdot x) \cdots))) \cdot \lambda x.\lambda y.(x(x \circ y))).\) \(\square\)

**Definition 13 (The complexity of an implementation of the \(\lambda\)-calculus)**

- A procedure \(I\) is an implementation of the lambda calculus if \(I\) on input a lambda term \(M\) outputs \(N\), the normal form of \(M\) (if \(M\) has no normal form then \(I\) 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 \(I\) of the lambda calculus has (worst-case) complexity \(T(\nu)\) if \(T : \mathbb{N} \to \mathbb{N}\) satisfies \(T(\nu) = \max\{\text{run time of } I \text{ on input } M \mid M \in S_\nu\}\) when \(S_\nu \neq \emptyset\) and where \(S_\nu = \{M \in \Lambda \mid \nu(M) = \nu\}\).
- An implementation \(J\) of the lambda calculus of worst case complexity \(T_J(\nu)\), is an optimal implementation if for any implementation \(I\) of the lambda calculus of worst case complexity \(T_I(\nu)\), we have \(T_J(\nu) = O(T_I(\nu))\).

\(^1\)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 $S_\nu$ are always finite, and there is a number $\nu_0$ such that for $\nu > \nu_0$, $S_\nu$ is always nonempty.

The size of $M$ must appear in $\nu$ 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 families 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 reductions. 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 $\mu_\pi = \mu_\beta = l_{\text{normal}}$, 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. □

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.
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 $\lambda$ 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:

\[
(\Delta (\lambda h. (\Delta (h I)))) \rightarrow (\lambda h. (\Delta (h I)) \lambda h. (\Delta (h I)))
\]

\[
\rightarrow (\Delta (\lambda h. (\Delta (h I))) I)
\]

\[
\rightarrow (\Delta (\Delta (I I)))
\]

\[
\rightarrow (\Delta (\Delta I))
\]

\[
\rightarrow (\Delta (I I))
\]

\[
\rightarrow (\Delta I)
\]

\[
\rightarrow (I I)
\]

\[
\rightarrow I.
\]

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
best case, can not avoid to duplicate one redex ((Δ (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 (Δ λh.(Δ (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.

Figure 1.7: First beta steps of $\Delta \lambda h. (\Delta (h \ I))$. 
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 initial 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.
Let us go on with our example. After the application of the fan-\(\lambda\) rule—Figure 1.10(4)—a new \(\beta\)-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 \(\beta\)-redexes or fan-\(\lambda\)-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 lose 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.
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-\(\oplus\) rule it is possible to perform a new fan-\(\lambda\) 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 \(\ast\) with \(\ast\), 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 \(\lambda\) 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 \(\beta\)-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 \(\beta\)-redex we obtain the expected normal form.

The normalization of this term required only seven applications of the \(\beta\)-rule, against eight \(\beta\)-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
CHAPTER 1. OPTIMALITY AND COMPLEXITY

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

Figure 1.14: Fan-fan iterations.
Figure 1.15: Reduction of $\Delta \lambda h. (\Delta (h I))$.

Figure 1.16: 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 \( \lambda \)-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-\( \lambda \) 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 \( \Lambda \).

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.
1.3. LAMPING’S ALGORITHM

Figure 1.18: Labeling fans fails.

Figure 1.19: The croissant node.
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 \textit{beta-rule}. The other three are the \textit{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 \( i \) \textit{duplicates} any node at level greater then \( i \), a croissant \textit{decreases} and a bracket \textit{increases} the level of any node at higher level it faces.

In the following we will consider (and count as)

- \textit{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;

- \textit{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 \textit{garbage node}—is created and its principal,
1.3. LAMPING’S ALGORITHM

Figure 1.22: Interaction rules.

Figure 1.23: Duplication and level management.
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 $\langle \langle \cdots \langle C', a_{n-1}, \cdots, a_1, a_0 \rangle, \cdots \rangle, a_0 \rangle$ 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.:

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

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

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

When, during the read-back we pass through a fan-in of level \( n \), we have to modify the context \( C^n[⟨b, a⟩] \) into \( C^n[⟨b, o a⟩] \) if entering from the \( o \) auxiliary port, \( C^n[⟨b, * a⟩] \) otherwise. Conversely, passing through a fan out, we will exit from the \( o \) port if the context is \( C^n[⟨b, o a⟩] \), from the \( * \) port if the context is \( C^n[⟨b, * a⟩] \). 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 \( □ \) 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
$C^n[⟨⟨b, a⟩, c⟩]$ becomes $C^n[⟨b, ⟨a, c⟩⟩]$. Traversing in the opposite direction, we have to operate the inverse transformation.

```
<table>
<thead>
<tr>
<th>C^n[⟨⟨b, a⟩, c⟩]</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
</tr>
<tr>
<td>C^n[⟨b, ⟨a, c⟩⟩]</td>
</tr>
</tbody>
</table>
```

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 $⟨\cdot, \circ⟩$. 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 $⟨⟨\cdot, \circ⟩, \circ⟩$. 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 $⟨⟨\cdot, \circ⟩⟩$. 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 $\ast$ auxiliary port obtaining the context $⟨\cdot, \ast⟩$ and coming to an application node. Hence the $M_1$ sub-term is again an application ($M_1\ M_2$) and we can go on with the read-back of the functional part $M_1$ 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 $\ast\circ$ and then we exit from the $\ast$ port with context $⟨⟨\cdot, \circ⟩⟩$. At this stage we meet a lambda node, thus the sub-term $M_1'$ has the form $\lambda x.M_1''$. 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 $⟨⟨\cdot, \circ⟩⟩$. 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 $M_1''$ is a variable $x$.

The lambda term we have read-back so far is $((\lambda x.\ x\ M_1)\ M_2)$ where $M_1$ 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 $⟨⟨\cdot, \ast⟩⟩$ and exit from the argument port for reading-back $M_1$. 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 $⟨⟨\cdot, \circ⟩⟩$ 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.
Consider the term

\[ C_n = \lambda x. \left( 2 \left(2 \left( \cdots \left(2 \left(x\right) \right) \cdots \right) \right) \right) \]

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

**Theorem 20**  
Lamping’s algorithm takes \(\Omega(2^n)\) oracle interactions to effect \(n\) parallel \(\beta\)-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:

\[
two' = \lambda x.\lambda y. (\lambda z. (z \, (z \, y))) \, \lambda w. (x \, w))
\]

\[
d = \lambda x. (x \, x)
\]

\[
g = \lambda n. (((n \, d) \, two') \, \lambda x. (q))
\]

\[
h = \lambda n. (((n \, two') \, two') \, \lambda w. (w) \, q)
\]

For any Church integer \( n \) the number of fan interactions in the reduction of \((g \, n)\) and \((h \, n)\) 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:

\[
((\lambda x. M)^\ell \, N) \rightarrow M^\ell [N^\ell / x]
\]

For example

\[
((\lambda x. (x^1 \, z^2)^3) \, y^5) \rightarrow (\lambda y. ((\lambda z. (y^5 \, z^6)^7) \, z^8) \, \lambda w. (w)^9) \rightarrow
\]

\[
((\lambda y. ((\lambda z. (y^5 \, z^6)^7) \, z^8) \, \lambda w. (w)^9) \rightarrow ((\lambda z. ((\lambda y. ((\lambda z. (y^5 \, z^6)^7) \, z^8) \, \lambda w. (w)^9) \rightarrow
\]

\[
((\lambda z. ((\lambda y. ((\lambda z. (y^5 \, z^6)^7) \, z^8) \, \lambda w. (w)^9) \rightarrow ((\lambda z. ((\lambda y. ((\lambda z. (y^5 \, z^6)^7) \, z^8) \, \lambda w. (w)^9) \rightarrow
\]

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 \((\lambda z. \ldots)\) 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 $K_\ell(n)$ as $K_0(n) = n$ and $K_{\ell+1}(n) = 2^{K_\ell(n)}$. 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 $\Omega(K_\ell(n))$. 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 $\sigma$ be a simple type generated by the grammar

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

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

$$\eta_o(x) = x$$

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

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 $\text{or}(M)$ 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 $\text{or}(M)$ 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 $\text{or}(M)$ 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 $\sigma$ 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 $D_1 = \{true, false\}$ in the simply typed lambda calculus such that any formula $F$ is true if and only if its encoding $\lambda x.\lambda y.x$ reduces to $\lambda x.\lambda y.x$. Moreover if $F$ quantifies over universes $D_i$ for $i \leq k$, the size of its encoding is $|F| = O(|F|(2k)!)$.

Finally they state the following theorem:

**Theorem 25** Let $TM$ be a fixed Turing Machine that accepts or rejects an input $x$ in $K_\ell(|x|)$ 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 $D_i$ for $i \leq (\log^* |x|) + \ell + 6$ and has length $O(|x| \log^* |x|)$.

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 $\lambda$-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 $\Omega(K_\ell(|n|))$ 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 $\text{lab}(i) \in \{\text{true, false, and, or}\}$;

\footnote{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.}
1.5. INHERENT COMPLEXITY OF IMPLEMENTING OPTIMAL REDUCTION

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

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

- the gates with $\text{indegree}(i) = 0$ 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 $[\text{lab}(i)]$ of labels is the following:

- $[\text{true}] = \lambda x. \lambda y. x$
- $[\text{false}] = \lambda x. \lambda y. y$
- $[\text{and}] = \lambda b_1. \lambda b_2. ((b_1 \ [\text{false}]) b_2)$
- $[\text{or}] = \lambda b_1. \lambda b_2. ((b_1 \ [\text{true}]) b_2)$.

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

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

If the algorithm reaches a gate $j$ with $\text{outdegree}(j) > 1$—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 $\text{outdegree}(i) > 1$. When it reaches such a gate, it writes on the top of the buffer “$(\lambda x_i.$”, 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 $\hat{C}$ encoding a boolean circuit $C$, in the graph normalization of the sharing graph corresponding to $\hat{C}$ 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.
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\(^1\):

\[
\begin{align*}
\Gamma, x : A, y : B, \Delta \vdash_{LL} M : C \\
\Gamma, y : B, x : A, \Delta \vdash_{LL} M : C
\end{align*}
\]

\[\text{Ex}\]

\(^1\)\(M\{N/x\}\) denotes the usual substitution.
\[
\begin{align*}
\frac{x : A \vdash_{\text{LL}} x : A}{A x} & \quad \frac{\Gamma \vdash_{\text{LL}} M : A \quad \Delta, x : A \vdash_{\text{LL}} N : B}{\Gamma, \Delta \vdash_{\text{LL}} N_{\{M/x\}} : B} \quad \text{Cut} \\
\frac{\Gamma \vdash_{\text{LL}} M : A \quad x : B, \Delta \vdash_{\text{LL}} N_{\{y \ M/x\}} : C}{\Gamma, y : A \multimap B, \Delta \vdash_{\text{LL}} N_{\{y \ M/x\}} : C} \quad (\rightarrow, l) & \quad \frac{\Gamma, x : A \vdash_{\text{LL}} M : B}{\Gamma \vdash_{\text{LL}} \lambda x. M : A \multimap B} \quad (\rightarrow, r)
\end{align*}
\]

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:

\[
\begin{align*}
\frac{x : !A, y : !A, \Gamma \vdash_{\text{LL}} M : B}{z : !A, \Gamma \vdash_{\text{LL}} M \{z/x, z/y\} : B} \quad \text{Contr} \\
\frac{\Gamma \vdash_{\text{LL}} M : B}{\Gamma, x : !A \vdash_{\text{LL}} M : B} \quad \text{Weak}
\end{align*}
\]

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

\[
\begin{align*}
\frac{x : A, \Gamma \vdash_{\text{LL}} M : B}{x : !A, \Gamma \vdash_{\text{LL}} M : B} \quad \varepsilon & \quad \frac{\Gamma \vdash_{\text{LL}} M : B}{\Gamma, !M \vdash_{\text{LL}} M : B} \quad (!, r)
\end{align*}
\]

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:

\[
\begin{align*}
\frac{\Gamma \vdash_{\text{LL}} M : B}{\Gamma, !M \vdash_{\text{LL}} M : B} \quad \delta & \quad \frac{x : !A, \Gamma \vdash_{\text{LL}} M : B}{x : !A, \Gamma \vdash_{\text{LL}} M : B}
\end{align*}
\]

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 \( A_1, \ldots, A_n \vdash_{\text{LL}} B \) 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 \( A \to B \) and with all negative conclusions of \( N \) but \( A \);
• 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. $A \dashv B$ 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;
• 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 2.5: Contraction.](image)

• 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 2.6: Weakening.](image)

• 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 2.7: Left introduction of !.](image)

• 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 2.8: Delta rule.](image)
2.1. LINEAR LOGIC

- 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.](image)

- 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 $\text{Contr}$ behaves like a fan, $\epsilon$ like a croissant, $\delta$ like a bracket and $\text{Weak}$ 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 $D = !D \to D$.

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 $\epsilon$ and $\delta$ 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:

\[
\frac{A \vdash \text{EAL} A}{Ax} \quad \frac{\Gamma \vdash \text{EAL} A, A, \Delta \vdash \text{EAL} B}{\text{Cut}} \quad \frac{\Gamma, A, B, \Delta \vdash \text{EAL} C}{\text{Ex}}
\]

\[
\frac{\Gamma \vdash \text{EAL} A}{\Gamma, A \to B, \Delta \vdash \text{EAL} C} \quad \frac{\Gamma, A \vdash \text{EAL} B}{\Gamma, A \vdash \text{EAL} A \to B} \quad \frac{\Gamma \vdash \text{EAL} C}{\Gamma, A \vdash \text{EAL} C} \quad \frac{\Gamma, !A, !A \vdash \text{EAL} B}{\text{Contr}}
\]

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 $\delta$ and $\epsilon$. 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 \(O(K_d(n))\).

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 Lampiing’s abstract algorithm. In fact, as we have already observed, no operators for \(\delta\) and \(\epsilon\) 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.

\[
\begin{align*}
A \vdash_{LAL} A & \quad \text{Ax} & \frac{\Gamma \vdash_{LAL} A \quad A, \Delta \vdash_{LAL} B}{\Gamma, \Delta \vdash_{LAL} B} & \text{Cut} & \frac{\Gamma, A, B, \Delta \vdash_{LAL} C}{\Gamma, B, A, \Delta \vdash_{LAL} C} & \text{Ex} \\
\Gamma \vdash_{LAL} A & \quad B, \Delta \vdash_{LAL} C & \frac{\Gamma, A \vdash_{LAL} B}{\Gamma, A \vdash_{LAL} B} & (\sim, l) & \frac{\Gamma, A \vdash_{LAL} B}{\Gamma, A \vdash_{LAL} B} & (\sim, r) \\
\Gamma \vdash_{LAL} C & \quad \frac{\Gamma, A \vdash_{LAL} C}{\Gamma, A \vdash_{LAL} C} & \text{Weak} & \frac{\Gamma, !A, !A \vdash_{LAL} B}{\Gamma, !A \vdash_{LAL} B} & \text{Contr} & \frac{\Gamma \vdash_{LAL} B}{\Gamma \vdash_{LAL} !B} & !_0 \\
& \quad \frac{\Gamma \vdash_{LAL} B}{\Gamma \vdash_{LAL} !B} & \text{Contr} & \frac{\Gamma \vdash_{LAL} B}{\Gamma \vdash_{LAL} !B} & \text{Contr} & \frac{\Gamma \vdash_{LAL} B}{\Gamma \vdash_{LAL} !B} & !_1 \\
& \quad \frac{A_1, \ldots, A_n, A_{n+1}, \ldots, A_{n+k} \vdash_{LAL} B}{!A_1, \ldots, !A_n, \$A_{n+1}, \ldots, \$A_{n+k} \vdash_{LAL} \$B} & \text{Ex} & \frac{\Gamma \vdash_{LAL} B}{\Gamma \vdash_{LAL} !B} & !_0 \\
\end{align*}
\]

### 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.
substituting them with a sort of general contraction

\[
\frac{\Gamma, A^{(n)} \vdash_{\text{SLL}} B}{\Gamma, !A \vdash_{\text{SLL}} B} \quad \text{Contr}
\]

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

\[
\frac{A \vdash_{\text{SLL}} A}{\Gamma \vdash_{\text{SLL}} A} \quad \text{Ax} \quad \frac{\Gamma \vdash_{\text{SLL}} A, \Delta \vdash_{\text{SLL}} B}{\Gamma, \Delta \vdash_{\text{SLL}} B} \quad \text{Cut} \quad \frac{\Gamma, A, B, \Delta \vdash_{\text{SLL}} C}{\Gamma, B, A, \Delta \vdash_{\text{SLL}} C} \quad \text{Ex}
\]

\[
\frac{\Gamma \vdash_{\text{SLL}} A, B, \Delta \vdash_{\text{SLL}} C}{\Gamma, A \vdash_{\text{SLL}} B, \Delta \vdash_{\text{SLL}} C} \quad (-, l) \quad \frac{\Gamma, A \vdash_{\text{SLL}} B}{\Gamma \vdash_{\text{SLL}} A \vdash_{\text{SLL}} B} \quad (-, r)
\]

\[
\frac{\Gamma, A^{(n)} \vdash_{\text{SLL}} B}{\Gamma, !A \vdash_{\text{SLL}} B} \quad \text{Contr} \quad \frac{A_1, \ldots, A_n \vdash_{\text{SLL}} B}{!A_1, \ldots, !A_n \vdash_{\text{SLL}} !B} \quad !
\]

### 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 \( \lambda\)-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
\(\lambda\)-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

\[ N = (\lambda n.\lambda y.(n \ \lambda z.\ z) \ y) \ \lambda x.(x \ x \ w.w) ) \]

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 \(\Rightarrow\)) as in Figure 2.11 (a),
where \(I_\alpha = \alpha \Rightarrow \alpha\) for every type \(\alpha\).

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 \(M = \lambda x.(x \ (x \ w.w))\). 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

\[
\begin{array}{c}
o \vdash o \\
\frac{(Ax)}{I_o} \\
\frac{I_o \vdash I_o}{I_o \vdash I_o} \\
\frac{(Ax) \quad (\neg o, l)}{I_o \vdash I_o \\
\frac{I_o, I_o \vdash I_o}{I_o \vdash I_o} \\
\frac{I_o \vdash I_o}{I_o \vdash I_o} \\
\frac{\text{Contr?}}{I_o \vdash I_o} \\
\frac{I_o \vdash I_o}{I_o \vdash I_o} \\
\end{array}
\]

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

\[
\begin{array}{c}
o \vdash o \\
\frac{(Ax)}{I_o} \\
\frac{I_o \vdash I_o}{I_o \vdash I_o} \\
\frac{(Ax) \quad (\neg o, l)}{I_o \vdash I_o \\
\frac{I_o, I_o \vdash I_o}{I_o \vdash I_o} \\
\frac{I_o \vdash I_o}{I_o \vdash I_o} \\
\frac{!I_o, !I_o \vdash !I_o}{!I_o \vdash !I_o} \\
\frac{\text{Contr.}}{!I_o \vdash !I_o} \\
\frac{I_o \vdash I_o}{I_o \vdash I_o} \\
\end{array}
\]

The corresponding typing is represented in Figure 2.11 (b), where the type of \(x\) inside the
box is \(I_o\), whereas it is \(!I_o\) 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 \(!I_o\).

Observe now that the leftmost innermost application is wrong. We need to box the argument \(\lambda z.\ z\), which must have type \(!I_o\), as it is shown in Figure 2.12 (c).

Finally, in order to apply \((n \ \lambda z.\ z)\) of type \(!I_o\) 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 \(!I_o\) becomes \(o \Rightarrow o\), 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:

\[
\frac{L_o \vdash L_o}{(Ax)} \quad \frac{x_o \vdash (Ax)}{L_o, x_o \vdash (Ax)} \quad \frac{\vdash \neg \sigma \vdash \sigma}{L_o, \neg \sigma \vdash \sigma} \quad \frac{\vdash \sigma \vdash \neg \sigma, \neg \sigma}{L_o, \neg \sigma, \neg \sigma \vdash \neg \sigma} 
\]

\[\vdash L_o \vdash (Ax) \quad \frac{L_o \vdash (Ax)}{L_o, x_o \vdash L_o} \quad \frac{\vdash \neg \sigma \vdash \sigma}{L_o, \neg \sigma \vdash L_o} \quad \frac{\vdash \sigma \vdash \neg \sigma, \neg \sigma}{L_o, \neg \sigma, \neg \sigma \vdash L_o} \quad \frac{\vdash \neg \sigma \vdash \sigma}{\vdash L_o \vdash \neg \sigma, \neg \sigma} \quad \frac{\vdash \sigma \vdash \neg \sigma, \neg \sigma}{\vdash L_o \vdash \sigma, \sigma} \quad (Cut)
\]

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 \(^m!\) \( I_o \), by adding \( n \) !-rules at the end of the derivation. Or, to be more general, we may give \( N \) the type \(^m+n!\) \( I_o \), if we introduce \( m \) !'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 \(^{m+n}!\) \( I_o \).

Finally we remark that there exists simply typed terms without any EAL types. For example the simply typed \( \lambda \)-term

\[
(\lambda n. (n \lambda y. (n \lambda z. y)) \lambda x. (x \ y))
\]

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 \( \lambda \)-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 \( \lambda \)-terms. The (type-free) \( \lambda \)-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 \( x^1 < y^1 = \exists x^2. \text{true} \in x^2 \land \text{false} \notin x^2 \land y^1 \in x^2 \land x^1 \notin x^2 \) and \( x^1 = y^1 = \neg (x^1 < y^1 \lor y^1 < x^1) \).

Define the type of Booleans as \( B = !0 \rightarrow !0 \rightarrow !0 \); write \( L^* \) for the EAL type of the generic lists of elements of type \( \alpha \):

\[
L^* = !(\alpha \rightarrow \tau \rightarrow \tau) \rightarrow !0(\tau \rightarrow \tau).
\]

Following [Mai92], quantifiers can be encoded by using iteration over lists. Given \( n \geq 0 \) and some EAL type \( \sigma \), suppose to have coded with the \( \lambda \)-term \( Q : L^* \tau \) the set \( Q = \{e_1, \ldots, e_m\} \) of elements of type \( \sigma \); suppose moreover that \( \Psi : \sigma \rightarrow !^n B \) is a term encoding a generic formula \( \Psi \). Then \( (Q \lambda y. (\text{AND} \ (\Psi \ y)) \text{true}) \) 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
2.3. CODING TYPE THEORY INTO EAL PROOFS

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

\[
\begin{align*}
  x^1 = y^1 &= x^1 \leftrightarrow y^1 \\
  x^{k-1} \in_k y^k &= \exists z^{k-1} \in y^k \; z^{k-1} =_{k-1} x^{k-1} \\
  x^k = y^k &= \forall z^{k-1} \in \mathcal{D}_{k-1} \; (z^{k-1} \in_k x^k \leftrightarrow z^{k-1} \in_k y^k).
\end{align*}
\]

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 $\to$ into $\rightarrow$.

**Definition 22 (Size of terms)**

- For $\sigma$ simple type, $||\sigma||$ is defined inductively as follows:
  \[
  \begin{align*}
  ||\sigma|| &= 1 \\
  ||\alpha \rightarrow \beta|| &= 1 + ||\alpha|| + ||\beta||.
  \end{align*}
  \]

- For $M : \sigma$ simply typed term, size $|M|$ is defined inductively as follows:
  \[
  \begin{align*}
  |x| &= ||\sigma|| \text{ if } x \text{ has type } \sigma \\
  |\lambda x.M| &= 1 + |M| \\
  |(M \; N)| &= 1 + |M| + |N|.
  \end{align*}
  \]

- For $M : A$ EAL-typed term, size $|M|$ is defined inductively as follows:
  \[
  \begin{align*}
  |x| &= ||(A)^*|| \text{ if } x \text{ has type } A \\
  |\lambda x.M| &= 1 + |M| \\
  |(M \; N)| &= 1 + |M| + |N|.
  \end{align*}
  \]

**Definition 23 (Depth)**

- For a EAL type, depth $d(A)$ is defined inductively as follows:
  \[
  \begin{align*}
  d(o) &= 0 \\
  d(A \rightarrow B) &= \max\{d(A), d(B)\} \\
  d(!A) &= 1 + d(A) + d(A).
  \end{align*}
  \]

- 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 $\alpha, \gamma, \tau$ and are relevant only to the calculus of $D_k$. Depth of $eq_k$ is relevant only to the calculus of depth of $member_k$.

(**) $\Delta_1$ and $\Delta_k$ 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>
<thead>
<tr>
<th>Formula</th>
<th>Depth</th>
<th>Size</th>
<th>Time</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>((1 - \gamma)z + \eta + (\iota))</td>
<td>((a))</td>
<td>({0, \eta}) max + ((1 - \gamma)z + \eta)</td>
<td>(</td>
<td>x</td>
</tr>
<tr>
<td>((1 - \gamma)z + \eta + (\iota))</td>
<td>((a))</td>
<td>({0, \eta}) max + ((1 - \gamma)z + \eta)</td>
<td>(</td>
<td>x</td>
</tr>
<tr>
<td>((1 - \gamma)z + \eta + (\iota))</td>
<td>((a))</td>
<td>({0, \eta}) max + ((1 - \gamma)z + \eta)</td>
<td>(</td>
<td>x</td>
</tr>
<tr>
<td>((1 - \gamma)z + \eta + (\iota))</td>
<td>((a))</td>
<td>({0, \eta}) max + ((1 - \gamma)z + \eta)</td>
<td>(</td>
<td>x</td>
</tr>
<tr>
<td>((1 - \gamma)z + \eta + (\iota))</td>
<td>((a))</td>
<td>({0, \eta}) max + ((1 - \gamma)z + \eta)</td>
<td>(</td>
<td>x</td>
</tr>
<tr>
<td>((1 - \gamma)z + \eta + (\iota))</td>
<td>((a))</td>
<td>({0, \eta}) max + ((1 - \gamma)z + \eta)</td>
<td>(</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 2.1: Encoding.
Lemma 29 (Booleans)  We encode boolean values in EAL as follows:

1. true = λx. y x :!B for any n ≥ 0, |true| = 3, d(true :!B) = n and the deepest type that appears in the EAL-derivation of true :!B is !(B of depth d(!B) = n + 1;

2. false = λx. y y :!B for any n ≥ 0, |false| = 3, d(false :!B) = n, deepest type !(B of depth d(!B) = n + 1;

3. NOT = λb x. y (b x y) :!B → !(B for any n ≥ 0, |NOT| = 12, d(NOT :!B → ...) = n, deepest type !(B of depth d(!B) = n + 1;

4. AND = λb1 b2 x y (b1 (b2 x y)) :!B → !(B → !(B for any n ≥ 0, |AND| = 21, d(AND : !B → ...) = n, deepest type !(B of depth d(!B) = n + 1;

5. OR = λb1 b2 x y (b1 (b2 x y)) :!B → !(B → !(B for any n ≥ 0, |OR| = 21, d(OR :!B → ...) = n, deepest type !(B of depth d(!B) = n + 1;

6. IFF = λb1 b2 x y (b1 (b2 x y)) :!B → !(B → !(B for any n ≥ 1, |IFF| = 29, d(fff :!B → ...) = n, deepest type !(B of depth d(!B) = n + 1;

Proof:  Simple inspection of the EAL-derivations.

Following [AM98], for any k the domain D_k can be encoded by a λ-term D_k representing D_k as the list of its values. Define

\[ D_1 = \lambda c. \lambda n. (c \text{ true} (c \text{ false} n)) \]
\[ D_k = (\text{powerset } D_{k-1}), \]

where

\[ \text{powerset} = \lambda A^*. (A^* \text{ double } \lambda c. \lambda n. (c \lambda c'. \lambda n'. n') (n)) \]
\[ \text{double} = \lambda x. \lambda l. \lambda c. \lambda n. (l \lambda e. (c \lambda c'. \lambda n'. (c' x (c c') n')) (l c n)). \]

Lemma 30  For any EAL-type α, γ and τ and for n ≥ 0:

- double has type !^α+1 !L^γ+1 !L^γ+1 ; d(double :!^α+1 !L^γ+1 → ...) = 3 + i and the deepest type in the EAL-type derivation of double is !(L^γ+1 !L^γ+1 ) of depth 2 + max{d(γ), 1 + i + d(τ)}, 1 + i + d(α).

- powerset has type L_i+1 !L^γ+1 τ → !(L^γ+1 !L^γ+1 ). d(powerset : L_i+1 !L^γ+1 τ → ...) = 4 + i and the deepest type in the EAL-type derivation of powerset is !(L^γ+1 !L^γ+1 ) of depth 3 + max{d(γ), 1 + i + d(τ), 1 + i + d(α)}.

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 :!^α+1 !L^γ+1 !L^γ+1 is the final type.

\[ d(!^α+1 !L^γ+1 !L^γ+1 ) = \]
\[ = \max\{3 + i + d(\alpha), 1 + \max\{1 + d(\gamma), 1 + i + \max\{1 + d(\tau), 1 + d(\alpha)\}\}\} \]
\[ = d(!(L^γ+1 !L^γ+1 )). \]
Figure 2.15: EAL type of powerset
2.3. CODING TYPE THEORY INTO EAL PROOFS

Analogously, for \( \text{powerset} : L_{i+\alpha}^\gamma \rightarrow \omega(\omega^{\alpha}) \):

\[
d(L_{i+\alpha}^\gamma \rightarrow \omega(\omega^{\alpha})) =
\]

\[
= \max \left\{ \max \left\{ 1 + 1 + 1 + 3 + i + d(\alpha), 2 + \max\{1 + d(\gamma), 1 + i + \max\{1 + d(\tau), 1 + d(\alpha)\}\} \right\} \right\}
\]

\[
= d(\omega(\omega^{\alpha})).
\]

\(\square\)

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

Given the type schema

\[
\begin{align*}
\Delta_0 &= 1^{n_0}B \\
\Delta_k &= 1^{n_k}L_{\Delta_{k-1}}^\tau
\end{align*}
\]

we can prove the following:

**Lemma 31**

1. \( \forall \tau_1, \ldots, \tau_k \) types in EAL, \( \forall n_0, \ldots, n_k \geq 0, \forall n_k \geq 2(k - 1), D_k \) has type \( \Delta_k \).

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

3. The deepest type in the derivation of \( D_k \) is \( \Delta_k \) of depth

\[
d(\Delta_k) = \max_{0 \leq j \leq k} \left\{ \sum_{i=j}^{k} (1 + n_i) + d(\tau_j) \right\}.
\]

4. \( |D_k| \leq d(2k)! \) where \( d \) is a fixed constant.

**Proof:** For the proof of the size of \( D_k \) 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 \( \Delta_1 = 1^{n_1}(L_{\tau_0 \tau_1}^\gamma) \). The proof is shown in Figure 2.16.

2. Looking at Figure 2.16 is easy to see that the depth of \( D_1 \) is \( 1 + n_0 + n_1 \).

3. Again, looking at Figure 2.16, the deepest type is \( 1^{n_1}(L_{\tau_0 \tau_1}^\gamma) = \Delta_1 \) and

\[
d(1^{n_1}L_{\tau_0 \tau_1}^\gamma) = n_1 + 1 + \max\{d(\tau_1), n_0 + 1\}.
\]
CHAPTER 2. COMPLEXITY OF OPTIMAL SHARING

Figure 2.16: EAL type of $D_1$

Figure 2.17: EAL type of $D_2$

$k \geq 2$

- if $k = 2$ by Lemma 30, powerset has type $\text{powerset}^\gamma L_\gamma^\tau$ for any type $\alpha$, $\gamma$ and $\tau$ and for $i \geq 0$, and, in particular, when $\alpha = !^\beta$, $j \geq 0$, powerset has type:

$$!^\alpha L_i^\gamma L_\gamma^\tau \rightarrow_o !^\beta (L_i^\gamma L_\gamma^\tau)$$

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

$$!^\gamma (L_i^\tau)^{n_0}$$

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

$$!^{2+m} (L_i^\gamma L_\gamma^\tau)$$

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

$$\Delta_2 = !^{n_2} (L_i^\gamma L_\gamma^\tau)^{n_0}$$

- if $k > 2$, $D_k = (\text{powerset} \ D_{k-1})$ by definition.

By inductive hypothesis $\forall \tau_1, \ldots, \tau_{k-1}$ types in EAL, $\forall n_0, \ldots, n_{k-2} \geq$
2.3. CODING TYPE THEORY INTO EAL PROOFS

0, ∀nk−1 ≥ 2(k − 2), Dk−1 has type \( Δ_{k−1} = !^{nk−1}(L^τ_{k−1}(L^{τ_{k−2}}_{Δ_{k−3}})) \).

Similar to the previous case, for \( α = !^{i}(L^{τ_{k−2}}_{Δ_{k−3}}), \) \( nk−1 = 2(k − 2), nk−2 = 3 + i + j \) and \( τ_{k−1} = !^{i}(L^γ_{j}(L^{τ_{k−2}}_{Δ_{k−3}})) \) one has ∀m ≥ 0, Dk of type

\[
!^{2(k−1)+m}(L^γ_{j}(L^{τ_{k−2}}_{Δ_{k−3}}))
\]

(see Figure 2.18, where the inner bold box represents 2(k − 2) normal boxes and the outer one represents m normal boxes). Hence ∀τ1, . . . , τk, ∀n0, . . . , nk−1 ≥ 0, ∀nk ≥ 2(k − 1), Dk has type

\[
Δ_k = !^{nk}(L^τ_{nk−1}(L^{τ_{k−2}}_{Δ_{k−3}})).
\]

2. For \( k ≥ 2 \) one has:

\[
D_k = (powerset_{k−1})^{L^{\Delta_{k−3}}_{n_{k−2}}(n_{k−2}(L^\Delta_{k−3}))}
\]

\[
(powerset_{k−2})^{L^{\Delta_{k−4}}_{\underbrace{(3+n_{k−1}+n_{k−2})}}(n_{k−3}(L^\Delta_{k−4}))}
\]

\[
\ldots
\]

\[
(powerset_1)^{L^{\Delta_{n_{08}}}_{\underbrace{(3+\ldots(3+n_{k−1}+n_{k−2})\ldots+n_{1})}}(n_{08})
\]

\[
D_1^{L^{\Delta_{n_{08}}}_{\underbrace{(3+\ldots(3+n_{k−1}+n_{k−2})\ldots+n_{1})}}(n_{08})}
\]

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 powerset \( j \) in the construction above is inside 2(\( j−1 \)) boxes, hence depth of \( D_k \) is

\[
\max \left\{ \max_{1 ≤ j ≤ k−1} \{ d(powerset_j : L^{\Delta_{n_{08}}}_{\underbrace{(3+\ldots(3+n_{k−1}+n_{k−2})\ldots+n_{1})}}(n_{08}) \} + 2(\ j−1 \) \} + m
\]
but
\[
d(\text{powerset}_j : L_{3(k-j)+\sum_{i=j-1}^{k-1} n_i}^j) + 2(j - 1) > \]
\[
d(\text{powerset}_{j+1} : L_{3(k-(j-1)+\sum_{i=j}^{k-1} n_i}^j) + 2j
\]
indeed
\[
1 + 3(k - j) + \sum_{i=j-1}^{k-1} n_i + 2(j - 1) > 1 + 3(k - j - 1) + \sum_{i=j}^{k-1} n_i + 2j
\]
\[> -1 + n_{j-1} > -2\]
and by hypothesis every \( n_i \) is greater than zero. Hence depth of \( D_k \) is
\[
\max \left\{ \begin{array}{l}
d(\mathbf{D}_1 : L_{3(k-1)+\sum_{i=0}^{k-1} n_i}^k), \\
d(\text{powerset} : L_{3(k-1)+\sum_{i=0}^{k-1} n_i}^k)
\end{array} \right\} + m
\]
\[
= \max \left\{ \begin{array}{l}
1 + 3(k - 1) + \sum_{i=0}^{k-1} n_i, \\
1 + 3(k - 1) + \sum_{i=0}^{k-1} n_i
\end{array} \right\} + m
\]
\[
= k + \sum_{i=0}^{k} n_i.
\]
3. for \( D_k \) of \( \tau_k \) of \( \tau_{k-1} \) of \( \tau_{k-2} \) of \( \tau_{k-3} \), looking at Figure 2.18, we prove first that the deepest type is \( \sum_{i=0}^{k-1} n_i \), indeed
\[
d(\tau_k) =
\]
\[
= 2(k - 1) + 1 +
\]
\[
+ \max \left\{ \begin{array}{l}
d(\tau_k), \\
1 + 1 + \max \left\{ \begin{array}{l}
d(\tau_{k-2}), \\
d(\Delta_{k-1})
\end{array} \right\}
\end{array} \right\}
\]
\[
= 2(k - 2) + 1 + \max \left\{ \begin{array}{l}
d(\tau_k), \\
3 + n_{k-1} + n_{k-2} + 1 + \max \left\{ \begin{array}{l}
d(\tau_{k-2}), \\
d(\Delta_{k-3})
\end{array} \right\}
\end{array} \right\}
\]
\[
= d(\tau_{k-2}) L_{\Delta_{k-3}}^{\tau_{k-2}} L_{\tau_{k-2}}^{\tau_{k-1}} L_{\tau_{k-1}}^{\tau_k}
\]
Then we have depth of type of \( D_{k-1} \) equals to depth of type of \( D_{k-1} \) in the derivation of \( D_k \). Moreover depth of type of \( D_{k-1} \) is greater than depth of type of \( \text{powerset}_{k-1} \), hence the deepest type is the type of \( D_k \). Finally, assuming \( \tau_0 = o \), we prove:
\[
d(\Delta_k) = \max_{0 \leq j < k} \left\{ \sum_{i=j}^{k} (1 + n_i) + d(\tau_j) \right\}
\]
By induction on $k$:

$$d(\Delta_0) = d(!^0) = n_0 + 1;$$

$$d(\Delta_k) = 1 + n_k + \max\{d(\tau_k), d(\Delta_{k-1})\}$$

$$= \max \left\{ 1 + n_k + d(\tau_k), 1 + n_k + \max_{0 \leq j \leq k-1} \left\{ \sum_{i=j}^{k-1} (1 + n_i) + d(\tau_j) \right\} \right\}.$$

**Lemma 32** Let

$$M_1 = B$$

$$M_k = L_{1:M_{k-1}}^{2k-3}B.$$

Then

1. $member_k$ has EAL-type

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

for any $m \geq 0$.

2. $\exists c \geq 0. \exists d \geq 0. \ |member_k| \leq c \ast k^2 + d(2(k-1))!$.

3. $d(member_k : !^{m + \max\{2k-5,2\}} M_{k-1} - \circ \cdots) = m + 2(k - 1) + \max\{2k - 7, 0\}$.

4. The deepest type in $member_k$ is $!^{m + \max\{4k-9,2(k-1)\}} B$ 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

$$eq_1 = \text{IFF with } n = 1$$

$$member_k = \lambda x^{k-1}. \lambda y. (y^k \lambda y^{k-1}. (\text{OR} (eq_{k-1} \cdot x^{k-1} \cdot y^{k-1})) \text{ false})$$

$$eq_k = \lambda x^k. \lambda y^k. (\lambda op. (D_{k-1} \cdot \lambda z^{k-1}. (\text{AND} (\text{IFF} (op z^{k-1} \cdot x^k) \cdot (op z^{k-1} \cdot y^k))) \text{ true})) member_k)$$

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

Prime formulas are encoded by the following terms.

$$eq_1 = \text{IFF}$$

$$member_k = \lambda x^{k-1}. \lambda y^k. (y^k \lambda y^{k-1}. (\text{OR} (eq_{k-1} \cdot x^{k-1} \cdot y^{k-1})) \text{ false})$$

$$eq_k = \lambda x^k. \lambda y^k. (\lambda op. (D_{k-1} \cdot \lambda z^{k-1}. (\text{AND} (\text{IFF} (op z^{k-1} \cdot x^k) \cdot (op z^{k-1} \cdot y^k))) \text{ true}))$$

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

Moreover notice that:

$$
\text{member}_k = \text{member}_k^i[D_1/d_1, \ldots, D_{k-2}/d_{k-2}]
$$

$$
\text{eq}_k = \text{eq}_k^i[D_1/d_1, \ldots, D_{k-1}/d_{k-1}].
$$

**Definition 25** We define types $E^k = E_1^k \rightarrow E_2^k \rightarrow E_3^k$ of eq$_k^i$, for $k \geq 1$, and $M^k = M_1^k \rightarrow M_2^k \rightarrow M_3^k$ of member$_k^i$, for $k \geq 2$ as follows:

$$
E_1^k = \mathbb{B}
$$

$$
M_1^k = \mathbb{E}^k_1
$$

$$
E_2^k = \mathbb{M}^k_2
$$

$$
M_2^k = L_{E_1^k}^{k-1}
$$

$$
E_3^k = \mathbb{M}^k_3
$$

$$
M_3^k = E_3^{k-1}
$$

Hence

$$
E^1 = \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}
$$

$$
M^k = \mathbb{E}^{k-1}_{E_1^k} \rightarrow L_{E_1^k}^{k-1} \rightarrow \mathbb{E}^{k-1}_3
$$

$$
E^k = \mathbb{M}^k_2 \rightarrow \mathbb{M}^k_2 \rightarrow \mathbb{M}^k_3
$$

**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. \(\Delta_k = \text{m}(L_{E_1^k}^{k+1})\)

**Lemma 34** \(\forall k \geq 1 \ \exists n_0, \ldots, n_{k-1} \geq 0 \ \exists \tau_1, \ldots, \tau_k \) such that \(\forall n_k \geq 0 \ \Delta_k = \text{m}(L_{M_1^k}^{M_{k+1}})\).

**Lemma 35**

1. For any $k \geq 2$, member$_k^i$ has type $M^k$ in EAL, with $\text{FV}(\text{member}_k^i) = \{d_1, \ldots, d_{k-2}\}$, $d_i$ of type $\mathbb{I}^{2(k-2) + 1}(M_{M_1^k}^{M_{k+1}})$; moreover $d(\text{member}_k^i : M^k) = 2(k-1)$;

2. for any $k \geq 1$, eq$_k^i$ has type $E^k$ in EAL, with $\text{FV}(\text{eq}_k^i) = \{d_1, \ldots, d_{k-1}\}$, $d_i$ of type $\mathbb{I}^{2(k-1) + 1}(M_{M_1^k}^{M_{k+1}})$ for $k \geq 1$; moreover $d(\text{eq}_k^i : E^k) = 2(k-1) + 1$.

**Proof:** By mutual induction on $k$.

First of all, note that $\forall k \geq 2 \ \ M^k_3 \equiv \mathbb{I}^{2(k-1)} \mathbb{B}$ and $\forall k \geq 1 \ \ E^k_3 \equiv \mathbb{I}^{2(k-1) + 1} \mathbb{B}$.

The proof for member$_k^i$ is shown in Figure 2.19, where $\mathbb{D}$ has type

$$
\mathbb{I}^{2(k-2) + 1} \mathbb{B} \rightarrow \mathbb{I}^{2(k-2) + 1} \mathbb{B} \rightarrow \mathbb{I}^{2(k-2) + 1} \mathbb{B}
$$

for the observation above, and the bold box represents $2(k-2) + 1$ normal boxes. Note that eq$_{k-1}^i$ has free variables $d_1 : \mathbb{I}^{2(k-3)}(M_{M_1^k}^{M_{k-1}}), \ldots, d_{k-2} : M_{M_1^k}^{M_{k-1}}$ and it is inside a box; hence member$_k^i$ has the same free variables with the same types, but with one more $!$, that is $d_1 : \mathbb{I}^{2(k-3) + 1}(M_{M_1^k}^{M_{k-1}}), \ldots, d_{k-2} : !(M_{M_1^k}^{M_{k-1}})$. The proof for eq$_k^i$ is shown in Figure 2.20. Note that member$_k^i$ has free variables $d_1 : \mathbb{I}^{2(k-3) + 1}(M_{M_1^k}^{M_{k-1}}), \ldots, d_{k-2} : !(M_{M_1^k}^{M_{k-1}})$ and it is inside a box therefore, the free
variables of $eq_k'$ are $d_{k-1} : L^{M^k_{M^1}}$ and the variables of $member_k'$ with the same type with one more $!$, that is $d_1 : !^{2(k-2)}(L^{M^2_{M^1}}), \ldots, d_{k-1} : L^{M^k_{M^1}}$.

Looking at figures 2.20 and 2.19,

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

$$d(member'_k : M^k) = \max\{d(false : E^{k-1}_3), d(OR : E^{k-1}_3 \rightarrow \cdots \cdots), d(eq'_{k-1} : E^{k-1}_3)\} + 1$$

$$= \max\{2(k-2) + 1, d(eq'_{k-1} : E^{k-1}_3)\} + 1$$

$$\equiv \max\{2(k-2) + 1, 2(k-2) + 1\} + 1 = 2(k-1)$$

$$d(eq'_k : E^k) = \max\{2(k-1), d(member'_k : M^k)\} + 1$$

$$\equiv \max\{2(k-1), 2(k-1)\} + 1$$

$$= 2(k-1) + 1.$$


\[ \square \]

**Lemma 36** \( \forall k \geq 2 \forall n \geq \max\{2k-7, 0\} \) member\(_k\) has type \(!^n M^k_1 \rightarrow !^n M^k_2 \rightarrow !^n M^k_3\) in EAL.

**Proof:** By the previous lemma member\(_k\)' has type \(M^k\) in EAL, with free variables \(d_1 : !^{2(k-3)+1}(L^{M^2_{M^1}}), \ldots, d_{k-2} : !^{(L^{M^k_{M^1}}_{M^1})}\), 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, ∀n₀, . . . , nᵢ−₁ ≥ 0, ∀nᵢ ≥ 2(j − 1), Dᵢ has type Δᵢ, in particular, by Lemma 34, Dᵢ has type \( \text{max}(2(j−1),2(k−2−j)+1) \) \( (L^{M_{i+1}}_{M_i}) \). The proof is shown in Figure 2.21, where

1. \( i = \left[ \frac{k}{2} \right] \) is the lowest \( i \) s.t. \( 2(i − 1) > 2(k − 2 − i) + 1 \),
2. \( D_1 : \text{max}(2(k−3)+1)(L^{M_2}_{M_1}) \)
   \[ \vdots \]
3. the bold box inserted at step three represent \( m = \begin{cases} 1 & \text{k even boxes,} \\ 3 & \text{k odd boxes,} \end{cases} \)
4. \( p = \text{max}\{2k−7,0\} \).

The other bold boxes represent four normal boxes. \( \square \)

**Proof:** [of Lemma 32]

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

\[
\text{max}(k−3)+1 \iff \text{max}(k−3)+1 \iff \text{max}(k−3)+1 
\]

and vice versa \( \forall m \geq 0, \exists n \geq \text{max}\{2k−7,0\} \) such that the same equivalence holds.

(a) \( \text{max}(k−3)+1 \implies \text{max}(k−3)+1 \implies \text{max}(k−3)+1 \)

(b) by induction on \( k \geq 2 \) we prove \( M^k = M_k \):

\( k = 2 \): \( M^2 = L^B_2 = L^1_{M_1} = M_2 \);

\( k > 2 \): \( M^k = L^2_{M_{k−1}} \implies L^2_{M_{k−1}} = M_k \).

Hence trivially \( \text{max}(k−3)+1 \implies \text{max}(k−3)+1 \implies \text{max}(k−3)+1 \).

(c) \( \text{max}(k−3)+1 \implies \text{max}(k−3)+1 \implies \text{max}(k−3)+1 \).

2. First notice the following:

\[
(E^1_k)^* = (\Delta_0)^* = (B)^* \\
(E^1_k)^* = (L^B_{\Delta_{k−1}})^* = (L^B_{\Delta_{k−2}})^* = (\Delta_{k−2})^* \\
(M^2_k)^* = (\Delta_0)^* = (B)^* \\
(M^k_k)^* = (L^B_{\Delta_{k−2}})^* = (L^B_{\Delta_{k−3}})^* = (\Delta_{k−2})^* 
\]
Looking at Figure 2.21, by Lemma 36 we have depth 2.3. CODING TYPE THEORY INTO EAL PROOFS then, using the result above:

\[ n \]

First we need to show that for all \( n \)

\[ k \]

hence

\[ | \text{member}_k | = 8 + | \text{OR} | + | \text{eq}_k - 1 | + | \text{false} | + | y^k | + | x^{k-1} | + | y^{k-1} | \]

\[ = 32 + | \text{eq}_k - 1 | + ||(\Delta^B_{k-1})^*|| + 2||(\Delta^B_{k-2})^*|| \]

\[ = 32 + 24(k-1) + 5 + 48(k-2) + 10 + | \text{eq}_k - 1 | \]

\[ = 72k - 73 + | \text{eq}_k - 1 | \]

\[ | \text{eq}_k | = 14 + | \text{AND} | + | \text{IFF} | + | \text{true} | + | D_{k-1} | \]

\[ + | \text{member}_k | + | op | + 2 | z^{k-1} | + | x^k | + | y^k | \]

\[ = 14 + 21 + 29 + 3 + | D_{k-1} | + | \text{member}_k | \]

\[ + ||(\Delta^B_{k-2} \rightarrow \Delta^B_{k-1} \rightarrow B)^*|| + 2||(\Delta^B_{k-2})^*|| \]

\[ = 144k - 112 + | D_{k-1} | + | \text{member}_k | \]

\[ \leq 144k - 112 + d(2(k-1))! + | \text{member}_k | \]

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

\[
\max \left\{ \begin{array}{c}
d(\text{member}_k') : M^k) + \max\{2k - 7, 0\}, \\
\max_{1 \leq i < [k/2]} \{d(D_j : L_{M_j}^{(2i-1)}L_{M_j}^{(2i)}) + \max\{2k - 7, 0\}\} \\
\max_{[k/2] \leq j \leq k-2} \{d(D_j : L_{M_j}^{(2(j-1)}L_{M_j}^{(2j-1)}) + 4(k - 2 - j)\} \\
m \end{array} \right\}
\]

First we need to show that for all \( k \geq 1 \)

\[ E^k_1 = \Delta_{k-1} \]

with \( n_{k-1} = \cdots = n_0 = 1 \) (we use notation \( \Delta_{k-1}^{(1,\ldots)} \)) and for all \( k \geq 2 \)

\[ M^k_1 = \Delta_{k-2} \]

with \( n_{k-2} = 2 \) and \( n_{k-3} = \cdots = n_0 = 1 \) (we use \( \Delta_{k-2}^{(2,1,\ldots)} \)). By induction on \( k \):

\[ E^1_1 = \text{!B} = \Delta^{(1,\ldots)}_0 \]

\[ E^k_1 = \text{!}E_{k-1} \Rightarrow \Delta^{(1,\ldots)}_k = \Delta^{(1,\ldots)}_{k-1} \]

then, using the result above:

\[ M^1_1 = \text{!}E^1_1 = \text{!}2 \text{B} = \Delta^{(2)}_0 \]

\[ M^k_1 = \text{!}E^{k-1}_1 = \text{!}2 M^{k-1}_1 = \text{!}2 \text{L}_{E^{k-2}_1} = \text{!}2 \text{L}_{\Delta^{(1,\ldots)}_{k-2}} = \Delta^{(2,1,\ldots)}_{k-2} \]
therefore, in the calculus of $d(D_j)$ for $d(member_k)$, we have $\sum_{\ell=0}^{j-1} n_\ell = j+1$. Hence

$$d(member_k) = \max\{2k-5.2, n \}$$. 

$$d(member_k) = \max\{2k-7.0, \Delta_{k-1} \}$$. 

$$d(member_k) = \max\{2k-7.0, \}$$. 

$$d(member_k) = \max\{2k-7.0, \} + 4k-2-j \}$$

$$d(member_k) = \max\{2k-7.0, \} + 4k-2-j \}$$

$$d(member_k) = \max\{2k-7.0, \} + \max\{2k-7.0, \}$$

$$d(member_k) = \max\{2k-7.0, \} + \max\{2k-7.0, \}$$

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 D_i.

First notice that $M_k = M_k^2 = \Delta_{k-1}$ with $n_{k-1} = 0$ and $n_{k-2} = \ldots = n_0 = 1$ and $\tau_{k_i} = \max\{2k-3, B\}$, then

$$d(M_k) = d(M_k^2) = d(\Delta_{k-1})$$

$$d(M_k) = d(M_k^2) = d(\Delta_{k-1})$$

$$d(M_k) = d(M_k^2) = d(\Delta_{k-1})$$

$$d(M_k) = d(M_k^2) = d(\Delta_{k-1})$$

Hence depth of type of member_k is:

$$d(\max\{2k-5.2, n \}) = d(\max\{2k-7.0, \}) = \max\{4k-8, 2k-1 \} = d(\max\{4k-9, 2(k-1) \})$$

By Lemma 36, the D_i have type $\max\{2(i-1), 2(k-2-i) \} + 1 \}$ for $1 \leq i \leq k-2$.

$$d(\max\{2(i-1), 2(k-2-i) \} + 1 \}) = 2k-1 \).$$

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

\[\square\]

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

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

Let $\Psi(x_1^{k_1}, \ldots, x_n^{k_n})$ be the term encoding an arbitrary formula $\Phi$ with free variables $x_{i+1}^{k_{i+1}}, \ldots, x_n^{k_n}$, $0 \leq i \leq n$, and $m$ quantifiers over $D_1, \ldots, D_i$. Then

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

such that in EAL

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

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

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

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

2. the deepest type in the EAL-derivation of \(\Psi[x_1^{k_1}, \ldots, x_n^{k_n}]\) is the final one, of depth \(\leq f(k_j) + d + \sum_{\ell \in J} 2(k_{\ell - 1}) + 1\); 

3. the size of \(\Psi[x_1^{k_1}, \ldots, x_n^{k_n}]\) is \(O(|\Phi|2k_{\text{MAX}}!))\) where \(k_{\text{MAX}}\) is the greatest \(k\) such that \(\text{member}_k\) appears in \(\Psi[x_1^{k_1}, \ldots, x_n^{k_n}]\).

**Proof:** By induction on \(\Psi\).

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

We show the proof for the general case \(\Psi[x_k^{-1}, x_k] = (\text{member}_k x_k^{-1} x_k)\), leaving the cases \(\Psi[x^2] = (\text{member}_2 \text{true} x^2)\) and \(\Psi[x^2] = (\text{member}_2 \text{false} x^2)\) to the reader.

By Lemma 32 \(\Psi[x_k^{-1}, x_k] : !^mB\) for every \(m_3 \geq \max\{4k - 9, 2(k - 1)\}\). Now, \(f(k) \geq \max\{4k - 9, 2(k - 1)\}\) hence \(\Psi[x_k^{-1}, x_k] : !f(k)B\).

Again by Lemma 32 \(x_k : !^mM_k\) for any \(m_2 \geq \max\{2k - 7, 0\}\), and in particular \(x_k : !^{\max(2k - 7, 1) + f(k) - f(k_j)}M_{k_j}\) 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:

   \[
   d(\Psi[x_k^{-1}, x_k] : !f(k)B) = d\left(\text{member}_k : !^{\max(2k - 5, 3)}M_{k - 1}\right)
   \]

   which is of the form:

   \[
   f(k) = \begin{cases} 
   2(k - 1) + 1 & k \leq 3 \\
   4k - 9 & k \geq 4
   \end{cases}
   \]

   Than the thesis holds with \(k_j = k\). Moreover,

2. considering the EAL-derivations, the deepest type is the type of \(\text{member}_k\) of depth \(\max\{4k - 8, 2k - 1\} \leq f(k) + 1\) by Lemma 36.

3. Finally, by Lemma 36, size of \(\Psi\) is \(O(|\Phi|(2k)!))\).

**inductive step :**

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

1. about depth we have:

   \[
   d(\Psi[x_1^{k_1}, \ldots, x_n^{k_n}] : !f(k_j) + d + \sum_{\ell \in J} 2(k_{\ell - 1})B) = \\
   = \max\{d(\text{NOT} : !f(k_j) + d + \sum_{\ell \in J} 2(k_{\ell - 1})B), d(\Psi' : !f(k_j) + d + \sum_{\ell \in J} 2(k_{\ell - 1})B)\} \\
   \]

   which is of the form:

   \[
   f(k_j) + d + \sum_{\ell \in J} 2(k_{\ell - 1}) \\
   = f(k_j) + d + \sum_{\ell \in J} 2(k_{\ell - 1})
   \]
2. It is easy to see that the deepest type is the final one, of depth
\[ d(f(k_j) + d + \sum_{\ell \in J} 2(k_\ell - 1)B) = f(k_j) + d + \sum_{\ell \in J} 2(k_\ell - 1) + 1. \]

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

If \(\Psi[x_1^{k_1}, \ldots, x_n^{k_n}] = (\text{AND } \Psi_1[x_1^{k_1_1}, \ldots, x_i^{k_{i_1}}, \ldots, x_n^{k_1}]) \Psi_2[x_1^{k_1_2}, \ldots, x_n^{k_1}]\)

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 \(\Psi_1 : !f(k_{j_1}) + d_1 + \sum_{\ell \in J_1} 2(k_\ell - 1)B\) and \(x_h^{k_h} : \max\{2k_h - 1, 1\} + f(k_{j_1}) - f(k_h) + d_1 + \sum_{\ell \in J_1} 2(k_\ell - 1)M_{kh}\)

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 \(\Psi_2 : !f(k_{j_2}) + d_2 + \sum_{\ell \in J_2} 2(k_\ell - 1)B\) and \(x_h^{k_h} : \max\{2k_h - 1, 1\} + f(k_{j_2}) - f(k_h) + d_2 + \sum_{\ell \in J_2} 2(k_\ell - 1)M_{kh}\)

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 \(\Psi_2\) (\(t\) boxes in Figure 2.22). Then
\[
\Psi_2 : !\left( f(k_{j_2}) + d_2 + \sum_{\ell \in J_2} 2(k_\ell - 1) + f(k_{j_1}) + d_1 + \sum_{\ell \in J_1} 2(k_\ell - 1) - (f(k_{j_2}) + d_2 + \sum_{\ell \in J_2} 2(k_\ell - 1)) \right)B
\]

hence
\[
\Psi_2 : !f(k_{j_1}) + d_1 + \sum_{\ell \in J_1} 2(k_\ell - 1)B
\]

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 \( \Psi_2[x_{i_2}^{k_2}, \ldots, x_n^{k_n}] \) as in Figure 2.22. Then
\[
d(\Psi[x_1^{k_1} \ldots x_n^{k_n}]:!f(k_{j_1}) + d_1 + \sum_{\ell \in J_1} 2(k_{\ell} - 1) B) = \max \left\{ \begin{align*}
d(AND :!f(k_{j_1}) + d_1 + \sum_{\ell \in J_1} 2(k_{\ell} - 1) B), \\
d(\Psi_1[x_1^{k_1}, \ldots, x_{i_1}^{k_{i_1}}]:!f(k_{j_1}) + d_1 + \sum_{\ell \in J_1} 2(k_{\ell} - 1) B), \\
d(\Psi_2[x_{i_2}^{k_2}, \ldots, x_n^{k_n}]:!f(k_{j_2}) + d_2 + \sum_{\ell \in J_2} 2(k_{\ell} - 1) B) + \\
\quad + f(k_{j_1}) + d_1 + \sum_{\ell \in J_1} 2(k_{\ell} - 1) - \\
\quad - f(k_{j_2}) + d_2 + \sum_{\ell \in J_2} 2(k_{\ell} - 1) \end{align*} \right\}
= f(k_{j_1}) + d_1 + \sum_{\ell \in J_1} 2(k_{\ell} - 1) \]

2. It is easy to see that the deepest type is the final one, of depth
\[
d(!f(k_j) + d + \sum_{\ell \in J} 2(k_{\ell} - 1) B) = f(k_j) + d + \sum_{\ell \in J} 2(k_{\ell} - 1) + 1. \]

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

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

If \( \Psi[x_1^{k_1}, \ldots, x_n^{k_n}] = (D_{k_1} \lambda y_{k_1} (\text{AND} \ \lambda y_{k_1}. \Psi'[x_1^{k_1}, \ldots, x_n^{k_n}] \ y_{k_1}) \text{true}) \) then by inductive hypothesis \( \Psi' \) has \( m' \) quantifiers over \( D_{i_1}, \ldots, D_{i'} \) and there exist
\[ J' \subseteq \{1, \ldots, i'\} \land j' \in \{1, \ldots, n\} \land d' \leq m' \]
such that
\[
\Psi' :!f(k_{j'}) + d' + \sum_{\ell \in J'} 2(k_{\ell} - 1) B
\]
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 \( \Psi \) has \( m = m' + 1 \) quantifiers over \( D_{i_1}, \ldots, D_{i'}, D \), then we can type \( \Psi \) 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 \( d = d' + 1. \)
1. About depth we have the following:

\[
\begin{align*}
    d(\Psi[x_1^{k_1}, \ldots, x_n^{k_n}]) &= f(k_{j_1}) + (d'+1) + \sum_{\ell \in J'} 2(k_{\ell-1}) M_{k_{h}} \\
    &= \max \\
    &\begin{cases}
        d(\text{true}) : f(k_{j}) + d' + \sum_{\ell \in J'} 2(k_{\ell-1}) M_{k_{h}} + 2(k_{h} - 1) + 1, \\
        d(\text{AND}) : f(k_{j}) + d' + \sum_{\ell \in J'} 2(k_{\ell-1}) M_{k_{h}} + 2(k_{h} - 1) + 1, \\
        d(\Psi'[x_1^{k_1}, \ldots, x_n^{k_n}]) : f(k_{j}) + d' + \sum_{\ell \in J'} 2(k_{\ell-1}) M_{k_{h}} + 2(k_{h} - 1) + 1, \\
        d(\mathbf{D}_{k_{h}}) : f(2(k_{h-1}) M_{k_{h}}) + 2(k_{h} - 1) + 1 \quad \text{if } p\max\{2k_{h}-7,1\} + f(k_{j}) + d' + \sum_{\ell \in J'} 2(k_{\ell-1}) M_{k_{h}} = \Delta_{k_{h}-1} \end{cases}
    \\
    &= \max \quad 2(k_{h} - 1) + \max\{2k_{h}-7,1\} - f(k_{h}) = 0 \quad \text{then} \end{align*}
\]

Figure 2.23: EAL type of an arbitrary formula
2. Looking at Figure 2.23, the deepest type is again the final one, indeed
\[ d(2^{(k-1)}(L_{p1}^B)) = 2(k_i - 1) + 1 + \max \left\{ f(k_j) + d' + \sum_{\ell \in j'} 2(k_{\ell} - 1) + 1, \right. \]
\[ \left. \text{max}\{2k_i - 7, 1\} + f(k_j) - f(k_i) + d' + \sum_{\ell \in j'} 2(k_{\ell} - 1) + 2k_i - 1 \right\} \]
but \( f(k_i) = 2(k_i - 1) + \max\{2k_i - 7, 1\} \), then
\[ = 2(k_i - 1) + 1 + \max \left\{ f(k_j) + d' + \sum_{\ell \in j'} 2(k_{\ell} - 1) + 1 \right. \]
\[ \left. \text{max}\{2k_i - 7, 1\} + f(k_j) - f(k_i) + d' + \sum_{\ell \in j'} 2(k_{\ell} - 1) + 2k_i - 1 \right\} \]
\[ = f(k_j) + (d' + 1) + \sum_{\ell \in j'} 2(k_{\ell} - 1) + 1 \]
\[ = d(2^{(k-1)}(L_{p1}^B)) = 2^{(k-1)}(L_{p1}^B) \]

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

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

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 \( \Psi \) be a term encoding an arbitrary formula. Then \( \Psi \) has type in EAL \( !\text{B} \) with \( t = O(n \cdot k_{\text{MAX}}) \) where \( k_{\text{MAX}} \) is the greatest \( k \) such that \( \text{member}_k \) appears in \( \Psi \) and \( n \) is the number of quantifiers in \( \Psi \).

**Proof:** By Theorem 37 \( \Psi \) has type \( !\text{B} \) with \( t = f(k_j) + \sum_{\ell \in j'} 2(k_{\ell} - 1) \leq f(k_{\text{MAX}}) + n + 2n(k_{\text{MAX}} - 1). \) \( \Box \)

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

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

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

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

**Proof:** It is sufficient to investigate depth of types in the derivation of \( \Psi \). Indeed, depth of \( \text{or}(\Psi) \) cannot exceed depth of \( \Psi \) plus the maximal depth of a type in the derivation of \( \Psi \), because, in the worst case, we can \( \eta \)-expand the deepest variable of deepest type in \( \Psi \) and \( d(\eta(x)_\sigma) \) is trivially \( d(\sigma). \) \( \Box \)
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 $\lambda$-terms $E_n : B$ which normalize in at most $n$ shared $\beta$-reductions, where the number of non $\beta$-interactions that are required to normalize $E_n$ using Lamping’s abstract algorithm grows as $\Omega(K_\ell(n))$ 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: $\forall x^3(\exists x^2 \text{ true } \in x^2 \land \neg(\forall x^2 x^2 \in x^3))$
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 $\lambda$-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.
Since clearly these constraints admit solutions, we conclude the decoration procedure obtaining

If we change every \( \rightarrow \) in \( \leftarrow \), 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

becomes

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

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

\[
\begin{align*}
n_1, n_2, n_3, n_4, n_5, n_6 \in \mathbb{N} & \quad \land \quad n_5 = n_3 + n_5 \quad \land \quad n_1 + n_3 = n_2 \quad \land \quad n_4 = n_1 \quad \land \quad n_5 + n_6 \geq 1.
\end{align*}
\]

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

\[
\begin{align*}
n_1, n_2, n_6 \in \mathbb{N} & \quad \land \quad n_3 = 0 \quad \land \quad n_1 = n_2 = n_4 \quad \land \quad n_5 + n_6 \geq 1.
\end{align*}
\]

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

\[
\frac{x : \alpha \vdash \alpha}{\text{EAL-\lambda x y.}(x(y)) : \alpha}
\]
Thus \( \text{two} \) has EAL types \( \Gamma \vdash \text{two} \), 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 \( \text{two} \) (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 \( \Gamma \vdash_{\text{NEAL}} A \) then \( B, \Gamma \vdash_{\text{NEAL}} A \).

**Proof:** Trivially by induction for \( \text{contr}, (\rightarrow I) \) and \( (\rightarrow E) \) rules and by definition for \( \text{ax} \) and \( ! \) rules. \( \square \)

To annotate NEAL derivations, we use terms generated by the following grammar (elementary affine terms \( \Lambda^{EA} \)):
\[
M ::= x | \lambda x. M | (M M) | !(M) [M/x, \ldots, M/x] | \|M\|_{x,x}^M
\]

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 \( M \), \( \text{FV}(M) \), inductively as follows:
Figure 3.2: Natural Elementary Affine Logic in sequent style notation

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

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. \( \parallel N \parallel_{y, z}^P \{M/x\} = \parallel N[y'/y] \{z'/z\} \{M/x\} \parallel_{y', z'}^{P(M/x)} \) if \( x \notin \{y, z\} \), where \( y', z' \) are fresh variables;
9. \( \parallel N \parallel_{y, z}^P \{M/x\} = \parallel N \parallel_{y, z}^{P(M/x)} \) if \( x \in \{y, z\} \).

Elementary terms may be mapped to \( \lambda \)-terms, by forgetting the exponential structure:

- \( x^* = x \)
- \( (\lambda x. M)^* = \lambda x. M^* \)
3.1. TYPE INFERENCE IN EAL

\[
\begin{array}{c}
\Gamma, x : A \vdash_{\text{NEAL}} x : A \quad \text{ax} \\
\Gamma \vdash_{\text{NEAL}} M : B \quad \text{contr} \\
\Gamma, x : A \vdash_{\text{NEAL}} M : B \\
\Gamma \vdash_{\text{NEAL}} \lambda x. M : A \vdash_{\text{NEAL}} (M N) : B \\
\Gamma, \Delta_1, \ldots, \Delta_n \vdash_{\text{NEAL}} ! (N) [M_1/x_1, \ldots, M_n/x_n] : !B
\end{array}
\]

Figure 3.3: Term Assignment System for Natural Elementary Affine Logic

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 \(\text{FV}(M_1) \cap \text{FV}(M_2) = \emptyset\);
4. \(! (M) [M_1/x_1, \ldots, M_n/x_n]\) is legal iff \(M\) and \(M_i\) are legal for any \(i \leq n\) and \(\text{FV}(M) = \{x_1, \ldots, x_n\}\) and \(i \neq j \Rightarrow \text{FV}(M_i) \cap \text{FV}(M_j) = \emptyset\);
5. \(\parallel M \parallel_{x_1, x_2}^N\) is legal iff \(M\) and \(N\) are both legal and \(\text{FV}(M) \cap \text{FV}(N) = \emptyset\).

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

Proof: By trivial induction on the structure of \(M\) using definitions of legal terms and \(\text{FV}\).

\[\square\]

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. \(\text{dom}(\Gamma) = \{x_1, \ldots, x_n\}\); \(\Gamma(x_i) = A_i\); \(\Gamma \upharpoonright V = \{x : A | x \in V \land A = \Gamma(x)\}\).

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

Lemma 44

1. If \(\Gamma \vdash_{\text{NEAL}} M : A\) then \(\text{FV}(M) \subseteq \text{dom}(\Gamma)\);
2. If $\Gamma \vdash_{\text{NEAL}} M : A$ then $\Gamma \vdash \text{FV}(M) \vdash_{\text{NEAL}} M : A$.

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

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

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

Lemma 47 (Unique Derivation) For any legal term $M$ and formula $A$, if there is a valid derivation of the form $\Gamma \vdash_{\text{NEAL}} M : A$, 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 ($\beta$ and $\text{dup}$) 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.

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

$$(\|N\|_{x,y}^{M}[M_1/x_1,\ldots,M_n/x_n]) \rightarrow_{\text{dup}}^{M_1} \cdots \rightarrow_{M_n}^{M_n} N_{x_1,y_1}^{x_1}$$

$$(\|M\|_{x_1,x_2}^{M_1}) \rightarrow_{\alpha-c}^{x_1} \cdots \rightarrow_{\alpha-c}^{x_2} N_{x_1,x_2}^{x_1,x_2}$$

$$(\|M\|_{y,z}^{N}[y_1,y_2]) \rightarrow_{\iota-c}^{y_1} \cdots \rightarrow_{\iota-c}^{y_2} N_{y_1,y_2}^{y_1,y_2}$$

$$(\|M\|_{x_1,x_2}^{N}[y_1,y_2]) \rightarrow_{\alpha-c} \cdots \rightarrow_{\alpha-c} \lambda x. M\{N_{y_1,y_2}/y \}$$

where $M'$ in the $\rightarrow_{\text{dup}}$-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 $\rightarrow_{\alpha-c}$-rule, $y'$ and $z'$ in the $\rightarrow_{\iota-c}$-rule and $y_1',y_2'$ in the $\rightarrow_{\alpha-c}$-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 $\rightarrow_{\beta}$, $\rightarrow_{\text{dup}}$, $\rightarrow_{\iota-c}$, $\rightarrow_{\alpha-c}$, $\rightarrow_{c-c}$, $\rightarrow_{\lambda-c}$.

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

Proposition 49 Let $M \rightarrow_r N$ where $r$ is not $\rightarrow_{\beta}$, then $M^* = N^*$. 


3.1. TYPE INFERENCE IN EAL

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 \( \Theta \) is generated by the following grammar:

\[
\Theta ::= \top_{n_1 + \cdots + n_k} \Theta \keleton (\Theta \rightarrow \Theta)
\]

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

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

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

\[
n : (((o \rightarrow o) \rightarrow (o \rightarrow o)) \rightarrow (o \rightarrow o)).
\]

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 \( \mathcal{P} \):

\[
\mathcal{P}(\Theta) = \top_{n_1 + \cdots + n_k} \Theta \keleton (\Theta \rightarrow \Theta) \quad \mathcal{P}(\sigma) = \Theta \quad \mathcal{P}(\tau) = \Gamma \quad \mathcal{P}(\sigma \rightarrow \tau) = \top_{p_i(\Theta \rightarrow \Gamma)}
\]

The main function of the algorithm—the type synthesis function \( S \)—may now be applied. In the case of a variable \( x \) of simple type \( \sigma \) the rule is:

\[
\mathcal{P}(\sigma) = \Theta \\
S[x : \sigma] = (\Theta, \{x : \Theta\}, \emptyset, \emptyset)
\]

Observe that, given a term \( M \) of simple type \( \sigma \), \( S(M : \sigma) \) returns a quadruple:
CHAPTER 3. EAL-TYPING

(general EAL-type, base \(\{x_i : \Theta_i\}_i\) of pairs (variable:general EAL-type), set of linear constraints, critical points).

In our example we obtain:

\[ n : !p_1 (!p_2 (!p_3 (\varnothing !p_5 \varnothing) \varnothing !p_8 \varnothing) \varnothing !p_9 \varnothing) \]  \(3.4\)

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 \((n \rightarrow m)\) instead of \((!n \varnothing !m \varnothing)\), for a better reading.

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

\[ z : p_{12} (p_{13} \rightarrow p_{14}) \]  \(3.5\)

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 \(B\), 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 \(B\), 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 \(C\), which will be described later).

The rule is the following:

\[
C(\Theta_1, \ldots, \Theta_h) = A_3 \\
B \left( M, B_1, \Gamma_1, cpts \cup \left\{ \text{sl}_1(x) \right\}, A_1 \right) = \left\{ B \cup \left\{ x : \Theta_1 \right\}, \ldots, x : \Theta_h \right\}, \Gamma, A_2 \right) \\
S(M : \tau) = \langle \Gamma_1, B_1, A_1, cpts \cup \{\text{sl}_1(x), \ldots, \text{sl}_k(x)\} \rangle \\
S(\lambda x.M : \sigma \rightarrow \tau) = \langle \Theta_1 \rightarrow \sigma, \Gamma, B, \left\{ A_2, A_3 \right\}, cpts \rangle \]  \(3.6\)

In our example, there is only one occurrence of \(z\) and therefore the contraction function \(C\) is called with only one type and does not produce any constraint. Also the boxing function \(B\) 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:

\[ B(z, \{ z : p_{12} (p_{13} \rightarrow p_{14}) \}, p_{12} (p_{13} \rightarrow p_{14}) \}, \varnothing) = \{\{ z : p_{12} (p_{13} \rightarrow p_{14}) \}, p_{12} (p_{13} \rightarrow p_{14}), \varnothing\}. \]

\(^2\)A base here is a multiset, i.e. there could be several 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.
The rôle of *cpts* and *sl* will be discussed in the context of the critical points, below. Coming back to our example, for \( \lambda z.z : ((o \rightarrow o) \rightarrow (o \rightarrow o)) \) we infer the EAL-type

\[
\lambda z.z : p_{12}(p_{13} \rightarrow p_{14}) \rightarrow p_{12}(p_{13} \rightarrow p_{14})
\]  

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

\[
\mathbb{B}(\lambda z.z, \emptyset, p_{12}(p_{13} \rightarrow p_{14}) \rightarrow p_{12}(p_{13} \rightarrow p_{14}), \emptyset, \emptyset) = \\
= (\emptyset, b_1(p_{12}(p_{13} \rightarrow p_{14}) \rightarrow p_{12}(p_{13} \rightarrow p_{14})), \emptyset)
\]

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

\[
p_1 = 0
\]  

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

\[
\mathcal{U} \left( 
\begin{array}{c}
b_1(p_{12}(p_{13} \rightarrow p_{14}) \rightarrow p_{12}(p_{13} \rightarrow p_{14})), \\
p_2(p_{3}(p_{4} \rightarrow p_{5}) \rightarrow p_{6}(p_{7} \rightarrow p_{8}))
\end{array}
\right).
\]

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:

\[
\begin{align*}
b_1 &= p_2 \\
p_{12} &= p_3 \\
p_{13} &= p_4 \\
p_{14} &= p_5 \\
p_{12} &= p_6 \\
p_{13} &= p_7 \\
p_{14} &= p_8
\end{align*} \Leftrightarrow
\begin{align*}
b_1 &= p_2 \\
p_3 &= p_6 = p_{12} \\
p_4 &= p_7 = p_{13} \\
p_5 &= p_8 = p_{14}.
\end{align*}
\]  

The type synthesis rule\(^5\) for an application, provided that \( M \) and \( N \) are not applications

\(^5\)We will explain \( \mathcal{U} \) later
themselves, is:

\[ U(\Theta_1, \Theta_3) = A_4 \]

\[ \mathcal{B}(N, B_2, \Theta_2, \text{cpts}_2, A_2) = \langle B_3, \Theta_3, A_3 \rangle \]

\[ S(N : \sigma) = \langle \Theta_2, B_2, A_2, \text{cpts}_2 \rangle \]

\[ S(M : \sigma \rightarrow \tau) = \langle \sum n_i (\Theta_1 \rightarrow \Gamma), B_1, A_1, \text{cpts}_1 \rangle \]

(3.10)

\[ S((M \ N) : \tau) = \langle \Gamma, B_1 \cup B_3, \begin{cases} A_1 \\ A_3 \\ A_4 \end{cases}, \text{cpts}_1 \cup \text{cpts}_2, \sum n_i = 0 \rangle \] (3.11)

Figure 3.5 shows the decoration obtained so far for:

\[ n : b_1(p_3(p_4 \rightarrow p_5) \rightarrow p_3(p_4 \rightarrow p_5)) \rightarrow p_9(p_{10} \rightarrow p_{11}) \vdash (n \ \lambda z. z) : p_9(p_{10} \rightarrow p_{11}). \]

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 \( \mathcal{B} \) 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:

\[ p_9 = 0. \] (3.12)

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

\[ U(p_{10}, p_{15}) = \begin{cases} p_{10} = p_{15} \end{cases} \] (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 \( x : o \rightarrow o, y : o \). The first derivation, via a left \( \rightarrow \)-rule, obtains \( ((n \ \lambda z. z) y) : o \); then it bounds \( y \), giving \( \lambda y.((n \ \lambda z. z) y) : o \rightarrow (o \rightarrow o) \). The second derivation permutes the rules: it starts by binding \( y \), obtaining \( \lambda y.((x \ y) \ y) \) and only at this point substitutes \( (n \ \lambda z. z) \) for \( x \), via the left \( \rightarrow \)-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 \( \lambda y.((x \ y) \ y) \) 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 \( \rightarrow \)-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,
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 \(\sqcup\) 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:

\[(a) \; \text{FVO}(x) = [x]; \]
\[(b) \; \text{FVO}(\lambda x. M) = \text{FVO}(M) - x; \]
\[(c) \; \text{FVO}((M_1 \; M_2)) = \text{FVO}(M_1) :: \text{FVO}(M_2) \; (\text{the concatenation of lists}). \]

\(^6\)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\textsuperscript{7}:

\[ sl = \{(A^{j_1}, [y_{11}, \ldots, y_{1h}]), \ldots, (A^{j_k}, [y_{k1}, \ldots, y_{kh}])\} \]

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:

\[
\begin{align*}
\text{cpts} &= (\text{cpts}_1 \cup \{(\sum n_i = 0, \text{FVO}((M_1 M_2)))\}) \cup \text{cpts}_2 \\
\mathcal{W}(\Theta_1, \Theta_3) &= A_4 \\
\mathcal{B}(N, B_2, \Theta_2, \text{cpts}_2, A_2) &= (B_3, \Theta_3, A_3) \\
\mathcal{S}(N : \sigma) &= (\Theta_2, B_2, A_2, \text{cpts}_2) \\
\mathcal{S}((M_1 M_2) : \sigma \rightarrow \tau) &= (\sum n_i (\Theta_1 \rightarrow \Gamma), B_1, A_1, \text{cpts}_1) \\
\mathcal{S}((M_1 M_2) N) : \tau) &= \left\langle \Gamma, B_1 \cup B_3, \begin{cases} A_1 \\ A_3 \\ A_4 \end{cases}, \text{cpts} \right\rangle \\
&\quad \Sigma n_i = 0
\end{align*}
\]

(3.14)

In the example case we obtain:

\[
\begin{align*}
\left\{ \begin{array}{l}
\text{n} : b_1(p_3(p_4 \rightarrow p_5) \rightarrow p_3(p_4 \rightarrow p_5)) \\
\text{y} : p_{10}
\end{array} \right\} &\vdash ((n \lambda z.z) y) : p_{11} \\
&\quad \sum n_i = 0
\end{align*}
\]

(3.15)

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

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

\[
\mathcal{B}(x, B, \Gamma, \text{cpts}, A) = (B, \Gamma, A)
\]

(3.16)

\textsuperscript{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.

\[
\mathcal{B}(B, \Gamma, cpts, A) = \langle B_1, \Gamma_1, A_1 \rangle \\
\mathcal{B}(M, B, \Gamma, cpts, A) = \langle b B_1, b \Gamma_1, A_1 \rangle
\] (3.17)

\(\mathcal{B}\) takes care of the list of critical points, by adding boxes “inside” the term as in Figure 3.7; at the end, \(\mathcal{B}\) adds \(b\) boxes “around” the term.

\[
\mathcal{B}(B, \Gamma, \emptyset, A) = \langle B, \Gamma, A \rangle
\] (3.18)

\(\mathcal{B}\) with no critical points produces no changes.

\[
\mathcal{B}(B_1, \Gamma, cpts, A_2) = \langle B, \Delta, A_1 \rangle
\]

\[
B_1 = \left\{ x_i : \left\{ \begin{array}{ll}
! \Theta_i & x_i \notin sl \\
\Theta_i & x_i \in sl
\end{array} \right. \right\}
\]

\[
A_2 = \left\{ \begin{array}{ll}
A_i & A_i \notin sl \\
A_i - c & A_i \in sl
\end{array} \right. \right\}
\]

\[
\mathcal{B}(\{ x_i : \Theta_i \}, \Gamma, \{ sl \} \cup cpts, A) = \langle B, \Delta, A_1 \rangle
\] (3.19)

Therefore, rule (3.6) gives in our case:

\[
\mathcal{S}(\lambda y.((n \lambda z. z) y) : o \rightarrow o) = \\
b_2 + c_1 + p_{10} \rightarrow b_2 + c_1 + p_{11} \\
\{ n : b_2(b_1(p_3(p_4 \rightarrow p_5) \rightarrow p_3(p_4 \rightarrow p_5)) \rightarrow p_9(p_{10} \rightarrow p_{11})) \},
\]

\[
\left\{ \begin{array}{l}
p_9 - c_1 = 0 \\
\vdots \\
\{ (p_9 - c_1 = 0, [n]) \}
\end{array} \right. 
\] (3.20)

where \(p_9 - c_1 = 0\) is the unique constraint (Equation (3.12)) modified by \(\mathcal{B}\). 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 \(\mathcal{B}\) would be performed, see the rule (3.59) for function \(\mathcal{S}\).

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

\[
\mathcal{S}(\lambda n.\lambda y.((n \lambda z. z) y) : (((o \rightarrow o) \rightarrow (o \rightarrow o)) \rightarrow (o \rightarrow o)) \rightarrow (o \rightarrow o)) = \\
b_3 + b_2(b_1(p_3(p_4 \rightarrow p_5) \rightarrow p_3(p_4 \rightarrow p_5)) \rightarrow p_9(p_{10} \rightarrow p_{11})) \\
\rightarrow b_3 + c_2(b_2 + c_1 + p_{10} \rightarrow b_2 + c_1 + p_{11}) \\
\emptyset,
\]

\[
\left\{ \begin{array}{l}
p_9 - c_1 - c_2 = 0 \\
\vdots \\
\emptyset
\end{array} \right. 
\] (3.21)
where \( p_9 - c_1 - c_2 = 0 \) is the unique constraint 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—\( x^{(2)} : (o \rightarrow o) \rightarrow (o \rightarrow o) \)—we infer the EAL-type

\[
p_{16}(p_{17}(p_{18} \rightarrow p_{19}) \rightarrow p_{20}(p_{21} \rightarrow p_{22}));
\]  

(3.22)

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

\[
p_{23}(p_{24}(p_{25} \rightarrow p_{26}) \rightarrow p_{27}(p_{28} \rightarrow p_{29})).
\]  

(3.23)

The EAL-type of \( w : o \) is \( p_{30} \) and then \( \lambda w.w : o \rightarrow o \) is typeable in EAL with type \( p_{30} \rightarrow p_{30} \).

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

\[
p_{23} = 0,
\]  

(3.24)

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

\[
\mathcal{U}'(p_{24}(p_{25} \rightarrow p_{26}), b_4(p_{30} \rightarrow p_{30})) = \begin{cases} 
 b_4 = p_{24} \\
 p_{25} = p_{26} = p_{30}.
\end{cases}
\]  

(3.25)
When the algorithm processes \((x^{(2)} \ (x^{(1)} \ \lambda w. w))\), it adds \(b_5\) boxes around the argument, imposes
\[
p_{16} = 0
\] (3.26)
and unifies the types
\[
\mathcal{U} (p_{17}(p_{18} \to p_{19}), b_5 + p_{27}(p_{28} \to p_{29})) = \begin{cases} 
p_{17} = b_5 + p_{27} 
p_{18} = p_{28} 
p_{19} = p_{29}.
\end{cases}
\] (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:
\[
\begin{cases} 
x^{(1)} : b_5(b_4(p_{25} \to p_{25}) \to p_{27}(p_{18} \to p_{19})), 
x^{(2)} : p_{17}(p_{18} \to p_{19}) \to p_{20}(p_{21} \to p_{22})
\end{cases} \vdash (x^{(2)} \ (x^{(1)} \ \lambda w. w)) : p_{20}(p_{21} \to p_{22})
\] (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 \(C\), whose rules are the following.
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 \( \mathcal{B} \) function modifies the first constraint in Equation (3.27):

\[
p_{17} = b_5 + p_{27} - c_3
\]  

Then the algorithm contracts the types of \( x \):

\[
C\left( \begin{array}{c}
b_6 + b_5(b_4(p_{25} \rightarrow p_{25}) \rightarrow p_{27}(p_{18} \rightarrow p_{19})), \\
b_6 + c_3(p_{17}(p_{18} \rightarrow p_{19}) \rightarrow p_{20}(p_{21} \rightarrow p_{22}))
\end{array} \right) = \left\{ \begin{array}{l}
b_6 + b + 5 \geq 1 \\
b_5 = c_3 \\
b_4 = p_{17} \\
p_{18} = p_{19} = p_{21} = p_{22} = p_{25} \\
p_{20} = p_{27}
\end{array} \right.
\]  

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:

\[
\vdash \lambda x. (x \ (x \lambda w. w)) : b_6 + b_5(b_4(p_{18} \rightarrow p_{18}) \rightarrow p_{20}(p_{18} \rightarrow p_{18})) \rightarrow b_6 + b_5 + p_{20}(p_{18} \rightarrow p_{18}).
\]  

The algorithm processes now the whole term \( (\lambda n.\lambda y.((n \lambda z. z) \ y) \ \lambda x. (x \ (x \lambda w. w))) : o \rightarrow o \). It adds \( b_7 \) boxes around the argument of the application and unifies the EAL-types for the
3.1. TYPE INFERENCE IN EAL

Figure 3.11: correct application:

\[
\begin{align*}
\{ & b_3 + b_2(b_1(p_3(p_4 \rightarrow p_5) \rightarrow p_3(p_4 \rightarrow p_5)) \rightarrow p_9(p_{10} \rightarrow p_{11}), \\
& b_7(b_6 + b_5(b_4(p_{18} \rightarrow p_{18}) \rightarrow p_{20}(p_{18} \rightarrow p_{18}) \rightarrow b_6 + b_5 + p_{20}(p_{18} \rightarrow p_{18})) \rightarrow b_7 + b_3 + b_2 \\
& b_1 = b_6 + b_5 \\
& b_4 = p_3 = p_{20} \\
& p_4 = p_5 = p_{10} = p_{11} = p_{18} \\
& p_9 = b_6 + b_5 + p_{20} \}\end{align*}
\]

(3.34)

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

\[
(\lambda n.\lambda y.((n \lambda z.z) y) \lambda x.(x (x \lambda w.w))) : o \rightarrow o
\]

(3.35)

has EAL-type

\[
b_8 + b_3 + c_2 \rightarrow (b_2 + c_1 + p_4) \rightarrow (b_2 + c_1 + p_4)
\]

(3.36)

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

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

(3.37)

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

\footnote{We have boxed the constraints which were not modified by \(S\) until the end of the type inference process in the exposition above. They are now all collected in the set of constraints below.}
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 $\text{!}^{n_3 + n_5} (\text{!}^{m_1 + n_2 + n_4 + n_6} o - \text{!}^{n_1 + n_2 + n_4 + n_6} o)$
   and decoration shown in Figure 3.13, or

2. for any $m_1, \ldots, m_7 \in \mathbb{N}, m_1 \geq 1 \land m_2 + m_3 = m_1 + m_5$ with EAL-type
   $\text{!}^{m_3 + m_4 + m_6} (\text{!}^{m_2 + m_7} o - \text{!}^{m_2 + m_7} o)$ 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.
3.1. TYPE INference IN EAL

Definition 31 (Type Synthesis Algorithm) Given a simply typeable lambda term and its principal type scheme $M : \sigma$, the type synthesis algorithm $\mathcal{S}(M : \sigma)$ returns a triple $(\Theta, B, A)$, where $\Theta$ 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 $\mathcal{S}(M : \sigma)$ 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, $\mathcal{B}$ (see Section 3.1.4).

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

The algorithm is exponential in the size of the $\lambda$-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, $\uplus$, in Section 3.1.4) that are roughly bounded by the size of the term.

Correctness and completeness of $\mathcal{S}$ 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 $!^{n_1}(o^{n_2}\Theta)$ 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 $\Theta$ and for any base $B = \{x_1 : \Theta_1, \ldots, x_n : \Theta_n\}$, we denote with $X(\Theta)$ the instantiation of $\Theta$ 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: $\mathcal{U}$

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.

\[
\mathcal{U}(\Sigma n_{i1} \Theta_1, \ldots, !\Sigma n_{ih} \Theta_h) = \begin{cases} 
\sum n_{i1} - \sum n_{i2} = 0 \\
\vdots \\
\sum n_{ih-1} - \sum n_{ih} = 0 
\end{cases}
\]

(3.38)

\[
\mathcal{U}(\Theta_1, \ldots, \Theta_h) = A_1 \\
\mathcal{U}(\Theta_2, \ldots, \Theta_h) = A_2
\]

(3.39)

**Contraction ($C$) and Type Processing ($P$)**

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

\[
\overline{C}(\Theta) = \emptyset
\]

(3.40)

\[
\mathcal{U}(\Sigma n_{i1} \Theta_1, \ldots, !\Sigma n_{ih} \Theta_h) = A \\
\overline{C}(\Sigma n_{i1} \Theta_1, \ldots, !\Sigma n_{ih} \Theta_h) = \begin{cases} 
n_1 + \cdots + n_h \geq 1 \\
A_1 & A_2
\end{cases}
\]

(3.41)

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

\[
\overline{P}(o) = !p_o
\]

(3.42)

\[
\overline{P}(\sigma) = \Theta \\
\overline{P}(\tau) = \Gamma \\
\overline{P}(\sigma \rightarrow \tau) = !p(\Theta \rightarrow \sigma \Gamma)
\]

(3.43)

**Boxing: $B$ and $B$**

\[
\overline{B}(B, \Gamma, \emptyset, A) = \langle B, \Gamma, A \rangle
\]

(3.44)

\[
\overline{B}(B_1, !^c \Gamma, cpts, A_2) = \langle B, \Delta, A_1 \rangle \\
B_1 = \begin{cases} 
x_i : \{ !^c \Theta_i : x_i \notin sl \} \\
\Theta_i : x_i \in sl 
\end{cases} \\
A_2 = \begin{cases} 
A^j : A^j \notin sl \\
A^j - c : A^j \in sl 
\end{cases}
\]

\[
\overline{B}(\{x_i : \Theta_i\}_i, \Gamma, \{sl\} \cup cpts, A) = \langle B, \Delta, A_1 \rangle
\]

(3.45)
3.1. TYPE INFERENCE IN EAL

\[ \mathcal{B}(x, B, \Gamma, \text{cpts}, A) = \langle B, \Gamma, A \rangle \] (3.46)

\[ \mathcal{A}(B, \Gamma, \text{cpts}, A) = \langle B_1, \Gamma_1, A_1 \rangle \]
\[ \mathcal{B}(M, B, \Gamma, \text{cpts}, A) = \langle ^{\eta}B_1, ^{\eta}\Gamma_1, A_1 \rangle \] (3.47)

**Proposition 53** Let \( b, c_1, \ldots, c_k \) be the fresh variables introduced by \( \mathcal{B}(M, B, \Gamma, \text{cpts}, A) = \langle ^{\eta}B_1, ^{\eta}\Gamma_1, A_1 \rangle \) 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:** \( \cup \)

\[ \emptyset \cup \text{cpts} = \text{cpts} \] (3.48)

\[ \text{cpts} \cup \emptyset = \text{cpts} \] (3.49)

\[ \begin{cases} \{s_{l_1} \} \cup \{s_{l_2} \} \\
\vdots \\
\{s_{l_n} \} \end{cases} = \text{cpts} \]

\[ \begin{cases} \{s_{l_1} \} \cup \{s_{l_2} \} \\
\vdots \\
\{s_{l_n} \} \end{cases} = \{s_{l_1}, s_{l_1} \cup s_{l_2}, \ldots, s_{l_1} \cup s_{l_2} \} \cup \text{cpts} \] (3.50)

**Type synthesis:** \( S \)

Let \( \neg \text{app}(M) \leftrightarrow \exists M_1, M_2 \text{ s.t. } M = (M_1, M_2) \).

\[ \mathcal{P}(\sigma) = \Theta \]

\[ \mathcal{S}(x : \sigma) = \langle \Theta, \{x : \Theta\}, \emptyset, \emptyset \rangle \] (3.51)

\[ h \geq 1 \]

\[ \mathcal{C}(\Theta_1, \ldots, \Theta_h) = A_3 \]

\[ \mathcal{B} \left( M, B_1, \Gamma_1, \text{cpts} \cup \{s_{l_1}(x) \} \cup \{s_{l_2}(x) \} \cup \ldots \cup \{s_{l_k}(x) \} \right) = \left( B \cup \{x : \Theta_1 \} \cup \{x : \Theta_2 \} \cup \ldots \cup \{x : \Theta_h \} \right) \cup \Gamma, A_1 \]

\[ \mathcal{S}(M : \tau) = \langle \Gamma_1, B_1, A_1, \text{cpts} \cup \{s_{l_1}(x), \ldots, s_{l_k}(x) \} \rangle \]

\[ \mathcal{S}(\lambda x : M : \sigma \rightarrow \tau) = \langle \Theta_1 \rightarrow \Gamma, B, \{A_2, A_3 \}, \text{cpts} \rangle \] (3.52)
\[ x \notin \text{FV}((M_1, M_2)) \]
\[ \text{cpts} = \text{cpts}_1 \cup \{(\sum n_i - n = 0, \text{FV0}(M_1, M_2))\} \]
\[ \mathcal{P}(\sigma) = \Theta \]
\[ \mathbb{B}((M_1, M_2), B_1, \Gamma_1, \text{cpts}_1, A_1) = \langle B, !^n\Gamma, A \rangle \]
\[ S((M_1, M_2) : \tau) = \langle \Gamma_1, B_1, A_1, \text{cpts}_1 \rangle \]
\[ S(\lambda x. (M_1, M_2) : \sigma \rightarrow \tau) = \left( \Theta \rightarrow !^n\Gamma, B, \left\{ \begin{array}{c} A \\ \sum n_i - n = 0 \\ \text{cpts} \end{array} \right\} \right) \] (3.53)
\[ -\text{app}(M) \]
\[ x \notin \text{FV}(M) \]
\[ \mathcal{P}(\sigma) = \Theta \]
\[ \mathbb{B}(M, B_1, \Gamma_1, \text{cpts}, A_1) = \langle B, \Gamma, A \rangle \]
\[ S(M : \tau) = \langle \Gamma_1, B_1, A_1, \text{cpts}_1 \rangle \]
\[ S(\lambda x. M : \sigma \rightarrow \tau) = \left( \Theta \rightarrow \Gamma, B, A, \text{cpts} \right) \] (3.54)
\[ \left\{ \begin{array}{c} \neg\text{app}(M) \wedge \neg\text{app}(N) \\ \mathcal{P}(\Theta_1, \Theta_3) = A_4 \\ \mathbb{B}(N, B_2, \Theta_2, \text{cpts}_2, A_2) = \langle B_3, \Theta_3, A_3 \rangle \\ S(N : \sigma) = \langle \Theta_2, B_2, A_2, \text{cpts}_2 \rangle \\ S(M : \sigma \rightarrow \tau) = \langle !^n\sum_i (\Theta_1 \rightarrow \Gamma), B_1, A_1, \text{cpts}_1 \rangle \end{array} \right\} \]
\[ S((M, N) : \tau) = \left( \Gamma, B_1 \cup B_3, \left\{ \begin{array}{c} A_1 \\ A_3 \\ A_4 \\ \sum n_i = 0 \\ \text{cpts}_1 \cup \text{cpts}_2 \end{array} \right\} \right) \] (3.55)
\[ \left\{ \begin{array}{c} \neg\text{app}(M) \\ \text{cpts} = \text{cpts}_1 \cup \{(A_4, \text{FV0}((N_1, N_2)))\}\cup \text{cpts}_2 \\ \mathcal{P}(\Theta_3, \Theta_1) = A_4 \\ \mathbb{B}((N_1, N_2), B_2, \Theta_2, \text{cpts}_2, A_2) = \langle B_3, \Theta_3, A_3 \rangle \\ S((N_1, N_2) : \sigma) = \langle \Theta_2, B_2, A_2, \text{cpts}_2 \rangle \\ S(M : \sigma \rightarrow \tau) = \langle !^n\sum_i (\Theta_1 \rightarrow \Gamma), B_1, A_1, \text{cpts}_1 \rangle \end{array} \right\} \]
\[ S((M, (N_1, N_2)) : \tau) = \left( \Gamma, B_1 \cup B_3, \left\{ \begin{array}{c} A_1 \\ A_3 \\ A_4 \\ \sum n_i = 0 \\ \text{cpts} \end{array} \right\} \right) \] (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 \(M\).
3.1. Type Inference in EAL

cpts_4 = cpts_2 \cup \{ (A_1, \text{FVO}(N_1, N_2)) \}

cpts_3 = cpts_1 \cup \{ (\sum n_i = 0, \text{FVO}(M_1, M_2)) \}

\mathcal{V}(\Theta_3, \Theta_1) = A_4

\mathcal{E}((N_1, N_2), B_2, \Theta_2, cpts_2, A_2) = \langle B_3, \Theta_3, A_3 \rangle

S((N_1, N_2) : \sigma) = (\Theta_2, B_2, A_2, cpts_2)

S((M_1, M_2) : \sigma \rightarrow \tau) = \langle \sum n_i (\Theta_1 \rightarrow \Gamma), B_1, A_1, cpts_1 \rangle

\mathcal{S}((M_1, M_2) : N_1, N_2) : \tau) = \left\langle \Gamma, B_1 \cup B_3, \left\{ \begin{array}{l} A_1 \\ A_3 \\ A_4 \\ \sum n_i = 0 \end{array} \right. , cpts_3 \cup cpts_4 \right\rangle

Type synthesis algorithm: \mathcal{S}

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

C(\Theta_{k_1}, \ldots , \Theta_{k_h}) = A_1 
\vdots 
C(\Theta_{k_1}, \ldots , \Theta_{k_h}) = A_h

\mathcal{E}(M, B_1, \Theta_1, cpts, A') = \left\langle \left\{ \begin{array}{l} x_1 : \Theta_{k_1}, \ldots , x_1 : \Theta_{k_1} \\ \vdots \\ x_h : \Theta_{k_1}, \ldots , x_h : \Theta_{k_h} \end{array} \right. , \Theta, A \right\rangle

S(M : \sigma) = (\Theta_1, B_1, A', cpts)

\mathcal{S}(M : \sigma) = \left\langle \Theta, \left\{ x_1 : \Theta_{k_1}, x_2 : \Theta_{k_2}, \ldots , x_h : \Theta_{k_h} \right\} , \left\{ \begin{array}{l} A \\ A_1 \\ \vdots \\ A_h \end{array} \right. \right\rangle

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 \rightarrow o) \rightarrow o) \rightarrow o)

\mathcal{S}(n : (o \rightarrow o) \rightarrow o)

\mathcal{S}(n : (o \rightarrow o) \rightarrow o)

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

\mathcal{S}(\lambda y.(n \lambda z.y) : o)
\[\mathcal{R}(o \rightarrow o \rightarrow o) = p_0(p_7(p_8 \rightarrow p_9) \rightarrow p_{10})\]
\[= (p_0(p_7(p_8 \rightarrow p_9) \rightarrow p_{10}), \{n : p_0(p_7(p_8 \rightarrow p_9) \rightarrow p_{10})\}, \emptyset, \emptyset)\]

\[
\mathcal{S}(\lambda z : o \rightarrow o)
\begin{align*}
\mathcal{S}(y : o) &= \mathcal{R}(o) = p_{11} \\
&= \langle p_{11}, \{y : p_{11}\}, \emptyset, \emptyset \rangle
\end{align*}
\]
\[
\mathcal{B}(y, \{y : p_{11}\}, p_{12}, \emptyset, \emptyset) = \langle \{y : p_{11}\}, p_{12}, \emptyset, \emptyset \rangle
\]
\[
\mathcal{B}(\lambda z : o \rightarrow o) \\
\mathcal{B}(\lambda z : o \rightarrow o) \\
\mathcal{B}(\lambda z : o \rightarrow o)
\]
\[
\mathcal{B}(l y. (n \lambda z : o \rightarrow o), \{n : b_2(b_1(p_8 \rightarrow p_9) \rightarrow p_{10})\}, b_2 + b_1 + p_9 \rightarrow b_2 + p_{10}, \emptyset, \emptyset)
\]
\[
= \langle \{n : b_3 + b_2(b_1(p_8 \rightarrow p_9) \rightarrow p_{10})\}, b_3(b_2 + b_1 + p_9 \rightarrow b_2 + p_{10}), \emptyset \rangle
\]
\[
\mathcal{B}(p_2(p_3, p_4), b_3(b_2 + b_1 + p_9 \rightarrow b_2 + p_{10})) = \langle p_2 = b_3, p_3 = b_2 + b_1 + p_9, p_4 = b_2 + p_{10} \rangle
\]
\[
= \langle p_5, \{n : p_1(p_2(p_3, p_4) \rightarrow p_5), n : b_3 + b_2(b_1(p_8 \rightarrow p_9) \rightarrow p_{10}) \}, p_2 = b_3, p_3 = b_2 + b_1 + p_9, p_4 = b_2 + p_{10}, p_5, \emptyset, \emptyset \rangle
\]
\[
= \langle p_5, \{n : b_5(b_2 + b_1 + p_9 \rightarrow b_2 + p_{10}) \rightarrow p_5, n : b_3 + b_2(b_1(p_8 \rightarrow p_9) \rightarrow p_{10}) \}, \emptyset, \emptyset \rangle
\]
\[
\mathcal{B}(l y. (n \lambda z : o \rightarrow o), \{n : b_4(b_3(b_2 + b_1 + p_9 \rightarrow b_2 + p_{10}) \rightarrow p_5), n : b_3 + b_2(b_1(p_8 \rightarrow p_9) \rightarrow p_{10}) \}, b_4 + b_3 + b_2(b_1(p_8 \rightarrow p_9) \rightarrow p_{10})
\]
\[
= \langle \{(b_4 \geq 1, b_4 = b_4 + b_3 + b_2, b_4 + b_1 + p_9 = p_8, b_2 + p_{10} = p_9, p_5 = p_{10}\} \rangle
\]
\[
= \langle b_4(p_5 \rightarrow p_5) \rightarrow p_{10}, \emptyset, \{b_4 \geq 1\}, \emptyset \rangle
\]
\[ S(\lambda x. (x \; y)) : (o \rightarrow o) \rightarrow o \]

\[
S'((x \; y)) : o
\]
\[
S((x \; y)) : o
\]
\[
S(x : o \rightarrow o) = \langle p_1(p_2 \rightarrow p_3), \{ x : p_1(p_2 \rightarrow p_3) \}, \emptyset, \emptyset \rangle
\]
\[
S(y : o) = \langle p_8, \{ y : p_8 \}, \emptyset, \emptyset \rangle
\]
\[
\mathbb{V}(p_6, p_8) = \{ p_6 = p_8 \}
\]
\[
\mathbb{V}(p_7, \{ x : p_6 \rightarrow p_7, y : p_6 \}, \emptyset, \emptyset)
\]
\[
\mathbb{V}(b_1 + p_7, p_2) = \{ b_1 + p_7 - p_2 - 0 = 0 \}
\]
\[
cpts = \{ x : b_1 + p_7 - p_2 = 0, \{ x : b_1(p_6 \rightarrow p_7), \} \}
\]
\[
S(x : o \rightarrow o) = \langle p_4(p_6 \rightarrow p_7), \{ x : p_4(p_6 \rightarrow p_7) \}, \emptyset, \emptyset \rangle
\]
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 Synthesis Algorithm

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

**Lemma 54** Let \( S(M : \sigma) = (\Theta, B, A, \text{cpts}) \). For any slice \( sl \) in \( \text{cpts} \), \( sl = \{\text{cpt}_1, \ldots, \text{cpt}_k\} \), for every path from the root of the syntax tree of \( M \) to any leaf, there exists at least one \( \text{cpt}_i \) 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 \( S \) and for the product union. \( \square \)

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 \( S(M : \sigma) = (\Theta, B, A, \text{cpts}) \).

1. \( \forall \{\text{cpt}_1, \ldots, \text{cpt}_k\} = sl \in \text{cpts} \) there exist \( P, (N_1_1, N_2_1), \ldots, (N_1_k, N_2_k) \) such that \( P \) is not a variable, \( x_1, \ldots, x_k \in \text{FV}(P) \) 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, \( x_1, \ldots, x_k \in \text{FV}(P) \) and \( M = P\{(N_1_1, N_2_1)/x_1, \ldots, (N_1_k, N_2_k)/x_k\} \), there exists \( \{\text{cpt}_1, \ldots, \text{cpt}_k\} = sl \in \text{cpts} \) such that \( \text{cpt}_i \) 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 \( \text{cpts} = \emptyset \). 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 \( \text{cpt}_i \) 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 \( \text{cpt}_i \). Then we take \( P = (y P_2) \). The other cases are analogous.

2. If \( M \) is a variable then \( \exists P \) 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 \( \text{cpts} \) in the rule for \( S(\lambda x.(M_1 M_2) : \sigma) \) where \( x \notin \text{FV}(M_1 M_2) \). 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. \( \square \)
Consider the length $L(M)$ of an EAL-term $M$ defined inductively:

$$
L(x) = 0
$$

$$
L(\lambda x. M) = 1 + L(M)
$$

$$
L((M \ N)) = 1 + L(M) + L(N)
$$

$$
L(\{M_1/x_1, \ldots, M_n/x_n\}) = L(M) + \sum_{i=1}^{n} L(M_i)
$$

$$
L(\| M \|_{x,y}^N) = L(M) + L(N).
$$

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

1. $M$ has no subterm of the form $\| M_1 \|_{x,y}^{M_2}$ 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 $\{1!-!, @ - c, ! - c, c - c, \lambda - c, dup\}$-normal form;
2. $P$ is simple;
3. if $\| R \|_{x,y}^{Q} \] is a subterm of $P$, then $x, y \in \text{FV}(R)$;
4. if $\| R \|_{x,y}^{Q_1/x_1, \ldots, Q_k/x_k} \]$ is a subterm of $P$, then $R$ is not a variable.

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

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

**Proof:** The request on the $\{1!-!, @ - c, ! - c, c - c, \lambda - c, dup\}$-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 $\| R \|_{x,y}^{Q_1, Q_2}$ or $\| R \|_{x_1, \ldots, x_n}^{P_1, Q_1, Q_2, \ldots, P_n}$ with $\| S \|_{y,z}^{x}$ 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 $\beta$-redex (inside $(Q_1 Q_2)$). However, the corresponding $\lambda$-term $P^*$, cannot share any redex—there is no sufficient syntax for this in the $\lambda$-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 $\Gamma \vdash_{\text{NEAL}} \| x \|_{M/x} : \Psi$ if and only if $\Gamma \vdash_{\text{NEAL}} M : \Psi$. Our aim is to identify $\lambda$-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) $\lambda$-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 $\Gamma, x : \Psi \vdash_{\text{NEAL}} x : \Psi$ then $S(x : \Psi) = \langle \mathcal{P}(\Psi), \{x : \mathcal{P}(\Psi)\}, \emptyset \rangle$ and the thesis holds by Lemma 57 being any $X$ solution of the empty set of constraints.

• If the type derivation ends with
\[
\frac{\Gamma \vdash_{\text{NEAL}} x : !\Phi \quad \Delta, y : !\Phi, z : !\Phi \vdash_{\text{NEAL}} N : \Psi}{\Gamma, \Delta \vdash_{\text{NEAL}} \|N\|^y_{y,z} : \Psi}
\]
then the thesis holds by inductive hypothesis on $\Delta, y : !\Phi, z : !\Phi \vdash_{\text{NEAL}} N : \Psi$.

• If $P$ is an abstraction then the type derivation is
\[
\frac{\Gamma, x : \Psi \vdash_{\text{NEAL}} M : \Phi}{\Gamma \vdash_{\text{NEAL}} \lambda x.M : \Psi \rightarrow \Phi}
\]
The thesis holds by inductive hypothesis. Notice that the solution $X$ instantiates all variables introduced by the $B$ call of the rule for $S$ to 0. It is easy to see looking at the rules for $B$ that if in the solution $X$ there is one variable introduced by $B$ that is not set to zero, then the type is exponential and $\Psi \rightarrow \Phi$ is not.

• If $P$ is an application
\[
\frac{\Gamma \vdash_{\text{NEAL}} M : \Phi \rightarrow \Psi \quad \Delta \vdash_{\text{NEAL}} N : \Phi}{\Gamma, \Delta \vdash_{\text{NEAL}} (M \: N) : \Psi}
\]
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 $B$ call to 0. Thus the constraint $\sum n_j = 0$ of the rule for $S$ 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
\[
\frac{\Delta_1 \vdash_{\text{NEAL}} M_1 : !\Phi_1 \cdots !\Phi_n \quad \vdash_{\text{NEAL}} M_n : !\Phi_n \quad x_1 : \Phi_1, \ldots, x_n : \Phi_n \vdash_{\text{NEAL}} N : \Psi}{\Gamma, \Delta_1, \ldots, \Delta_n \vdash_{\text{NEAL}} \|N\| [M_1/x_1, \ldots, M_n/x_n] : !\Psi}
\]
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 $B$ 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 $B$ for the critical point corresponding to $M_i$ increased by one.

\[\square\]

Notation

We use
\[
\frac{\Gamma \vdash M : !^n A \quad x : A \vdash N : B}{\Gamma \vdash !^n(N)[M/x] : !^n B}
\]
as a shorthand for

\[
x_2 \vdash ! ! A \vdash x_2 \vdash ! ! A \quad x_1 : ! ! A \vdash x_1 : ! ! A \quad x : A \vdash N : B
\]

\[
\vdash M : ! ! A \quad \vdash x_{n-1} : ! ! A \quad \vdash ([\vdots (N[x_1/x] \cdots )]^x_{n-1}/x_{n-2}] : ! \cdots ! B
\]

\[
\vdash [\vdots ! (N[x_1/x] \cdots )]^M/x_{n-1}] : ! \cdots ! B
\]

**Lemma 59 (Superimposing of derivations)** Let \( \mathcal{I} (M : \sigma) = (\Theta, B, A) \) 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 \( !^n_2(\alpha \rightarrow !^m_1(\beta \rightarrow \gamma)) \) at level 0 (nearest to \( x_0 \)) and has label \( (\alpha \rightarrow !^m_1(\beta \rightarrow \gamma)) \) at level \( n_2 \). This is the graphical counterpart of the !-rule

\[
\vdash [\vdots ! (N[x_1/x] \cdots )]^M/x_{n-1}] : ! \cdots ! B
\]

The complete decoration of Figure 3.15 can be produced in NEAL in two ways:

by the instantiation of

\[
!^n_2 ((((x_0 x_1) y)((x_4 x_5) w))) [(x_2 x_3)/y, (x_6 x_7)/w]
\]

and\(^9\)

\[
!^m_1 (((z(x_2 x_3))((x_4 x_5) w))) [(x_0 x_1)/z, (x_6 x_7)/w],
\]

\(^9\)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 \( !^n_2 ((((x_0 x_1) y)((x_4 x_5) w))) [x_0/x_0, x_1/x_1, (x_2 x_3)/y, x_4/x_4, x_5/x_5, (x_6 x_7)/w] \)).
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:

\[ !^1 (![z - y] (x_4 x_3 w)) [x_0 x_1/z] [x_2 x_3/y] (x_6 x_7)/w \]

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 \( S(M : \sigma) = \langle \Theta, B, A \rangle \) 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 \( \ell \) and \( b \) of level \( \ell + k \), then the type labeling the wire is \( !^k \Psi \) near \( a \) and \( \Psi \) near \( b \), i.e. the sum of level and number of exponentials for types labeling the syntax tree is an invariant.
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 $\ell$ boxes and set $i$ to $\ell$.

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:

$$
\begin{align*}
\Gamma_k(M\{^{th-k}(N)^Q/z/x\})^{P/y} & \text{ if } k \leq h \\
\Gamma_k(M)^{P/y, ^{th}(N)^Q/z/x} & \text{ if } k > h \\
\Gamma_h(\Gamma_k^{h-k}(M)^N/x)^Q/z, P/y & \text{ if } k > h
\end{align*}
$$

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.
CHAPTER 3. EAL-TYPING

Theorem 60 (Soundness) Let $\mathcal{J}(M: \sigma) = \langle \Theta, B, A \rangle$. For every $X$ integer solution of $A$, there exists $P$ elementary affine term such that $P^* = M$ and $X(B) \vdash_{\text{NEAL}} P : X(\Theta)$.

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, $\Theta$ 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 $\rightarrow$ 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 $B$.

rules for $B$. In Figure 3.19 it is shown how build a correctly decorated syntax...
3.2 Principal type

We prove that it is possible to identify a principal type for any term in $\Lambda^{EA}$. 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 $M \in \Lambda$. $\Gamma \vdash_{\text{NEAL}} M : A$ if and only if $\Gamma \vdash_{\text{NEAL}} R : A$ and $(R)^{\ast} = M$, 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 $\text{Abs}^{EA}$ of abstract EAL-terms is generated by the following grammar:

$$M ::= x \mid \lambda x.M \mid (M \ M) \mid [M]_{N_1 \rightarrow (\tilde{x}_1), \ldots, N_k \rightarrow (\tilde{x}_k)} \mid \nabla (M)[N_1/x_1, \ldots, N_k/x_n]$$

with the condition that every variable occurs just once in a term ($\tilde{x}_i$ is a placeholder for $x_1^i, \ldots, x_{n_i}^i$).

Definition 36 We define the following operations on abstract EAL-terms:

1. the set of free variables $\text{FV}$:

$$\text{FV}(x) = \{x\} \quad (3.60)$$
$$\text{FV}(\lambda x.M) = \text{FV}(M) \setminus \{x\} \quad (3.61)$$
$$\text{FV}((M_1 \ M_2)) = \text{FV}(M_1) \cup \text{FV}(M_2) \quad (3.62)$$
$$\text{FV}([M]_{C\text{list}}) = (\text{FV}(M) \setminus \text{CV}(C\text{list})) \cup \text{SSV}(C\text{list}) \cup \text{SFV}(C\text{list}) \quad (3.63)$$
$$\text{FV}(\nabla (M)[N_1/x_1, \ldots, N_k/x_k]) = \text{FV}(M) \setminus \{x_1, \ldots, x_k\} \cup \bigcup_{i=1}^{k} \text{FV}(N_i) \quad (3.64)$$
2. the set of contracted variables $CV$:

$$CV(M \rightarrow (x_1, \ldots, x_n)) = \{x_1, \ldots, x_n\}$$  \hspace{1cm} (3.65)

$$CV(M \rightarrow (x_1, \ldots, x_n), Slist) = \{x_1, \ldots, x_n\} \cup CV(Slist)$$  \hspace{1cm} (3.66)

3. the set of shared free variables $SFV$:

$$SFV(M) = \emptyset$$  \hspace{1cm} (3.67)

$$SFV([M]Slist) = SFV(M) \cup SFV(Slist)$$  \hspace{1cm} (3.68)

$$SFV(x \rightarrow (x_1, \ldots, x_n)) = \emptyset$$  \hspace{1cm} (3.69)

$$SFV(x \rightarrow (x_1, \ldots, x_n), Slist) = SFV(Slist)$$  \hspace{1cm} (3.70)

$$SFV(M \rightarrow (x_1, \ldots, x_n)) = FV(M)$$  \hspace{1cm} (3.71)

$$SFV(M \rightarrow (x_1, \ldots, x_n), Slist) = SFV(Slist)$$  \hspace{1cm} (3.72)

where $M$ in Equation (3.67) is not a contraction (i.e. it is not of the form $[M]Slist$) and $M$ in equations (3.71) and (3.72) is not a variable;

4. the set of single shared variables $SSV$:

$$SSV(M) = \emptyset$$  \hspace{1cm} (3.73)

$$SSV([M]Slist) = SSV(M) \cup SSV(Slist)$$  \hspace{1cm} (3.74)

$$SSV(x \rightarrow (x_1, \ldots, x_n)) = \{x\}$$  \hspace{1cm} (3.75)

$$SSV(x \rightarrow (x_1, \ldots, x_n), Slist) = \{x\} \cup SSV(Slist)$$  \hspace{1cm} (3.76)

$$SSV(M \rightarrow (x_1, \ldots, x_n)) = \emptyset$$  \hspace{1cm} (3.77)

$$SSV(M \rightarrow (x_1, \ldots, x_n), Slist) = SSV(Slist)$$  \hspace{1cm} (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 $ST$:

$$ST(M) = \emptyset$$  \hspace{1cm} (3.79)

$$ST([M]Slist) = ST(Slist)$$  \hspace{1cm} (3.80)

$$ST(M \rightarrow (x_1, \ldots, x_n)) = \{M\}$$  \hspace{1cm} (3.81)

$$ST(M \rightarrow (x_1, \ldots, x_n), Slist) = \{M\} \cup ST(Slist)$$  \hspace{1cm} (3.82)

where $M$ in Equation (3.79) is not a contraction;

6. the set of banged variables $BV$:

$$BV(M/x) = \{x\}$$  \hspace{1cm} (3.83)

$$BV(M/x, Blist) = \{x\} \cup BV(Blist)$$  \hspace{1cm} (3.84)

7. the set of single banged variables $SBV$:

$$SBV(y/x) = \{x\}$$  \hspace{1cm} (3.85)

$$SBV(M/x) = \emptyset$$  \hspace{1cm} (3.86)

$$SBV(y/x, Blist) = \{x\} \cup SBV(Blist)$$  \hspace{1cm} (3.87)

$$SBV(M/x, Blist) = SBV(Blist)$$  \hspace{1cm} (3.88)

where $M$ in equations (3.86) and (3.88) is not a variable.
On $\text{Abs}^{EA}$ a reduction relation can be defined as follows.

**Definition 37** The reduction relation $\rightarrow_{\text{an}}$ on $\text{Abs}^{EA}$ is the transitive and contextual closure of the following set of reduction relations \{ $\rightarrow_{\text{collaps}}$, $\rightarrow_{\text{var}}$, $\rightarrow_{c}$, $\rightarrow_{\text{not-c}}$, $\rightarrow_{\text{not-c'}}$, $\rightarrow_{\text{lambda}}$, $\rightarrow_{\text{not-c-1}}$, $\rightarrow_{\text{not-c-2}}$, $\rightarrow_{\text{not-c-3}}$, $\rightarrow_{\text{not-c-4}}$ \}, defined as follows:

$$\nabla (\nabla (M) [y_1/x_1, \ldots , y_n/x_n]) [M_1/y_1, \ldots , M_n/y_n] \rightarrow_{\text{collaps}} \nabla (M) [M_1/x_1, \ldots , M_n/x_n] \quad (3.89)$$

$$[[M], \ldots , x_i \rightarrow (y_1, \ldots , y_{n_i}), \ldots , \ldots , N_j \rightarrow (z_j^1, \ldots , z_{m_j}^j), \ldots ] \rightarrow_{\text{var}} \nabla \nabla (M) [N_{x_i}] [M_1/x_1, \ldots , P_1/y_1, \ldots , P_n/y_n, \ldots , M_n/x_n] \quad (3.90)$$

**Note:** if the sharing list of the inner contraction becomes empty, then the square brackets are removed.

$$[[M]_{N_i \rightarrow (x^1)}, \ldots , N_i \rightarrow (x_{i_1}^1, \ldots , x_{i_{m_i}}^i), \ldots ] \nabla \rightarrow_{\text{collaps}} \nabla (M) [M_1/x_1, \ldots , M_n/x_n] \quad (3.91)$$

**Note:** if the sharing list of the inner contraction becomes empty, then the square brackets are removed.

$$\nabla (M) [M_1/x_1, \ldots , \nabla (N) [P_1/y_1, \ldots , P_n/y_n]/x_i, \ldots , M_n/x_n] \rightarrow_{\text{not-c}}$$

$$\nabla (M) [\nabla (N) [x_i]] [M_1/x_1, \ldots , P_1/y_1, \ldots , P_n/y_n, \ldots , M_n/x_n] \quad (3.92)$$

$$\nabla (M) [\nabla (N) [x_i]] [M_1/x_1, \ldots , P_1/y_1, \ldots , P_n/y_n, \ldots , M_n/x_n] \quad (3.93)$$

$$\nabla (M) [\nabla (N) [x_i]] [M_1/x_1, \ldots , P_1/y_1, \ldots , P_n/y_n, \ldots , M_n/x_n] \quad (3.94)$$

$$[[M], \ldots , N_i \rightarrow (x^1), \ldots , N_k \rightarrow (y^k)]/x_i, \ldots \rightarrow_{\text{not-c'}}$$

$$\nabla (M) [\nabla (N) [x_i]] [M_1/x_1, \ldots , P_1/y_1, \ldots , P_n/y_n, \ldots , M_n/x_n] \quad (3.95)$$

$$[[M], \ldots , N_i \rightarrow (x^1), \ldots , N_k \rightarrow (y^k)]/x_i, \ldots \rightarrow_{\text{not-c'}}$$

$$\nabla (M) [\nabla (N) [x_i]] [M_1/x_1, \ldots , P_1/y_1, \ldots , P_n/y_n, \ldots , M_n/x_n] \quad (3.96)$$

$$[[M], \ldots , N_i \rightarrow (x^1), \ldots , N_k \rightarrow (y^k)]/x_i, \ldots \rightarrow_{\text{not-c'}}$$

$$\nabla (M) [\nabla (N) [x_i]] [M_1/x_1, \ldots , P_1/y_1, \ldots , P_n/y_n, \ldots , M_n/x_n] \quad (3.97)$$
\( \lambda x. [M]_{\text{Slist}} \rightarrow_{\lambda -c} [\lambda x. M]_{\text{Slist}} \) where \( x \notin \text{SFV}(\text{Slist}) \cup \text{SSV}(\text{Slist}) \) \hspace{1cm} (3.98)

\[
\nabla(x)[M/x] \rightarrow_{\nabla -c -1} M \\
\nabla(M)[\cdot \cdot N/x, \cdot \cdot] \rightarrow_{\nabla -c -2} \nabla(M)[\cdot \cdot] \) where \( x \notin \text{FV}(M) \) \hspace{1cm} (3.99)

\[
[x]_{\cdot \cdot N \rightarrow (\cdot \cdot x_{\cdot \cdot}) \cdot \cdot} \rightarrow_{c -c -1} N \\
[M]_{\cdot \cdot N \rightarrow (\cdot \cdot x_{\cdot \cdot}) \cdot \cdot} \rightarrow_{c -c -2} [M(N/x)]_{\cdot \cdot} \hspace{1cm} (3.100)
\]

All new variables introduced in the equations above are intended fresh.

**Lemma 61** \( \rightarrow_{\text{\&an}} \) is Church-Rosser.

**Definition 38** The type assignment system assigning formulas of EAL to terms of \( \text{Abs}^{\text{EA}} \) is shown in Table 3.1.

\[
\begin{array}{c}
\Gamma, x : A \vdash_{\text{abs}} x : A \\
\Gamma, x : A \vdash_{\text{abs}} M : B \\
\Gamma \vdash_{\text{abs}} \lambda x. M : A \rightarrow B \\
\Gamma \vdash_{\text{abs}} N : A \\
\Delta \vdash_{\text{abs}} M N : B \\
\Gamma, \Delta \vdash_{\text{abs}} M N : B \\
\Delta_1 \vdash_{\text{abs}} N_1 : !A_1 \\
\vdots \\
\Delta_k \vdash_{\text{abs}} N_k : !A_k \\
\Gamma, \Delta_1, \ldots, \Delta_k \vdash_{\text{abs}} \nabla(M)[N_1/x_1, \ldots, N_k/x_k] : !B \\
\end{array}
\]

Table 3.1: Type assignment system for \( \text{Abs}^{\text{EA}} \)-terms.

**Lemma 62 (Subject reduction)** Let \( M \in \text{Abs}^{\text{EA}} \) and \( M \xrightarrow{\rightarrow_{\text{\&an}}} N \), then

\( \Gamma \vdash_{\text{abs}} M : A \Rightarrow \Gamma \vdash_{\text{abs}} N : A. \)

**Proof:** By induction on the length of the reduction. By cases on the last step of the reduction.

\( \rightarrow_{c -c -1} \) then

\( [x]_{\cdot \cdot N \rightarrow (\cdot \cdot x_{\cdot \cdot}) \cdot \cdot} \rightarrow_{c -c -1} N \)
the type assignment is

\[ \Gamma \vdash_{abs} N : ! A' \]
\[ \vdots \]
\[ \ldots, x : ! A', \ldots \vdash_{abs} x : ! A' \]
\[ \ldots, \Gamma \vdash [x]_{\ldots, N \rightarrow (\ldots, x, \ldots), \ldots} : ! A' \]
\[ \text{contr} \]

hence, trivially, \( \ldots, \Gamma \vdash_{abs} N : ! A' \).

\[ \rightarrow \nabla_{-\epsilon-1} \]
then

\[ \nabla(x)[N/x] \rightarrow \nabla_{-\epsilon-1} N \]

the type assignment is

\[ \Gamma \vdash_{abs} N : ! \ldots ! A \]
\[ \vdots \]
\[ \Delta, \Gamma \vdash_{abs} \nabla(x)[N/x] : ! \ldots ! A \]
\[ m \]

\[ ! m \]

hence, trivially, \( \Delta, \Gamma \vdash_{abs} N : ! \ldots ! A \).

Other cases are equally easy. \[ \square \]

### 3.2.2 Principal Typing for \( \vdash_{abs} \)

In this section we will prove that \( \vdash_{abs} \) enjoys 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:

\[ \sigma ::= \alpha \mid \sigma \rightarrow \sigma \mid ! \sum_{i=1}^{n_1}(\sigma) \]

where \( \alpha \) belongs to a countable set of scheme variables, ranged over by \( \alpha, \beta, \gamma \); type schemata are ranged over by \( \sigma, \tau, \rho \); let \( T \) denote the set of type schemata;

ii) A scheme substitution is a function from type schemata to types, replacing scheme variables by types and \( ! / x^{k_i=1}_{p} \) by \( ! / x^{k_i=1}_{p} \), 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 \( ! \sum_{i=1}^{k} / x^{k_i=1}_{p} \) by \( ! / x^{k_i=1}_{p} \), 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 \( T \times T \) 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 $! \Sigma^{m_i}$:

$$
F(! \Sigma^{n_i}(x)) = ! \Sigma^{n_i}(x) \quad F(x) = x
$$

$$
F(\tau) = ! \Sigma^{m_j}(\tau) \quad F(! \Sigma^{n_i}(\sigma)), F(\sigma + \Sigma^{m_j}(\tau)) =
$$

$$
F(\sigma \rightarrow \tau) = \rho \quad F(\sigma) = \sigma_1 \quad F(\tau) = \tau_1
$$

In the following we assume that types unified by $U$ are already transformed by $F$, i.e. it is not the case that $! m(\sigma)$ is a subtype of them.

$$
U(\alpha, \alpha) = < \emptyset, [\alpha] > \\
\alpha \text{ is a scheme variable not occurring in } \tau
$$

$$
U(\alpha, \tau) = < \emptyset, [\alpha \mapsto \tau] > \\
\alpha \text{ is a scheme variable not occurring in } \sigma
$$

$$
U(\sigma, \alpha) = < \emptyset, [\alpha \mapsto \sigma] > \\
U(\rho, \mu) = < C, s >
$$

$$
U(! \Sigma^{n_i} \rho, ! \Sigma^{m_j} \mu) = < C \cup \{ n_i - m_j = 0 \}, s > \\
U(\sigma_1, \tau_1) = < C_1, s_1 > U(s_1(\sigma_2), s_1(\tau_2)) = < C_2, s_2 > \\
U(\sigma_1 \rightarrow \sigma_2, \tau_1 \rightarrow \tau_2) = < C_1 \cup C_2, s_1 \circ s_2 >
$$

**Lemma 63**

i) (correctness) $U(\sigma, \tau) = < C, s >$ implies $\forall S \text{ scheme substitution associated to } < C, s >, S(\sigma) \equiv S(\tau)$

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 $\sigma, \tau$, decomposing $S$. □

Let

$$
U(\sigma_1, \sigma_2, \ldots, \sigma_n) = \text{let } U(\sigma_1, \sigma_2) = < C_1, s_1 > \text{ and } \text{let } U(s_1 \circ \cdots \circ s_i(\sigma_{i+1}), s_1 \circ \cdots \circ s_i(\sigma_{i+2})) = < C_{i+1}, s_{i+1} > \text{ in } < C_1 \cup \cdots \cup C_n, s_1 \circ \cdots \circ s_n >.
$$

The following function $PT$ takes as input a term of $\Lambda^{EA}$ 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 $\alpha$ is fresh;
- \( PT(\lambda x. M) = \text{let } PT(M) = < B, \sigma, C > \text{ in } \begin{array}{l}
\text{if } B = B' \cup \{ x : \tau \} \text{ then } < B', \tau \rightarrow \sigma, C > \\
\text{else } < B, \alpha \rightarrow \sigma, C >
\end{array} \) where $\alpha$ is fresh;
3.2. PRINCIPAL TYPE

- \( PT(M \ N) = \text{let } PT(M) = \langle B_1, \sigma_1, C_1 \rangle \text{ and } PT(N) = \langle B_2, \sigma_2, C_2 \rangle \text{ and let they be disjoint in } \langle U(\sigma_1, \sigma_2 \backslash \alpha) = \langle C, s \rangle \rangle \in \langle s(B_1 \cup B_2), s(\alpha), C \cup C_1 \cup C_2 \rangle \rangle; \)

- \( PT([M]_{N_1 \rightarrow (\widehat{x}_1)}, ..., N_k \rightarrow (\widehat{x}_k)) = \text{let } PT(M) = \langle B, \sigma, C \rangle \text{ and } PT(N_i) = \langle B_i, \sigma_i, C_i \rangle \rangle \text{ (all disjoint and disjoint from } PT(M) \rangle \text{ and let } U(B(x^1_1), ..., B(x^m_1), \sigma_1, !^{m_1} \alpha_1) = \langle C'_1, s_1 \rangle \rangle \text{ and } U(s_1 \circ \cdot \cdot \cdot \circ s_i(B(x^1_{i+1})), ..., s_1 \circ \cdot \cdot \cdot \circ s_i(B(x^m_{i+1})), s_1 \circ \cdot \cdot \cdot \circ s_i(\sigma_{i+1}), !^{m_1+1} \alpha_{i+1}) = \langle C'_{i+1}, s_{i+1} \rangle \rangle \text{ and } s = s_1 \circ \cdot \cdot \cdot \circ s_k \rangle \in \langle s(B \cup \bigcup_{i \leq j \leq k} B_j), s(\sigma), C \cup \bigcup_{i \leq j \leq k} C_j \cup \bigcup_{i \leq j \leq k} C'_j \rangle \rangle. \) (\( \alpha_i \) and \( m_i \) 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 \( PT(M) = \langle \Theta, \sigma, C \rangle \) and \( \Gamma \supseteq S(\Theta), A = S(\sigma), \) for some scheme substitution \( S \) satisfying \( C \).

**Proof:** By structural induction on \( M \). \( \square \)

### 3.2.3 Canonical Forms

A term of \( Abs^{EA} \) is a term in a meta-language, representing an infinite set of terms of \( \Lambda^{EA} \). In particular, to every term in \( \Lambda^{EA} \) a canonical form can be assigned, which is a term of \( Abs^{EA} \) in normal form.

First consider the embedding function from \( \Lambda^{EA} \) to \( Abs^{EA} \):

\[
\begin{align*}
emb(x) &= x \\
emb(\lambda x.M) &= \lambda x.emb(M) \\
emb((M_1 \ M_2) &= (emb(M_1) \ emb(M_2)) \\
emb(! (M) [N_1/x_1, ..., N_k/x_k]) &= \nabla(emb(M))[^{emb(N_1)}/x_1, \cdots, ^{emb(N_k)}/x_k] \\
emb(||M||_{N,x,y}) &= [emb(M)]_{emb(N) \rightarrow (x,y)}
\end{align*}
\]
The canonical form of $M \in \Lambda^{EA}$ can be obtained by reducing to normal form (with respect to $\rightarrow_{can}$) the term $emb(M) \in Abs^{EA}$. Thanks to the Lemma 61 the canonical form of a term is unique: let us call it $\mathcal{C}an(M)$.

**Lemma 65** $\forall M \in \Lambda^{EA}$ $\Gamma \vdash_{NEAL} M : A \Rightarrow \Gamma \vdash_{abs} emb(M) : A$.

**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, \ldots, N_k/x_k]} $ the thesis holds by the observation that $!$-rule in the assignment system for $\Lambda^{EA}$ is simply a particular case of $!^{m}$-rule in the assignment system for $Abs^{EA}$. \hfill $\Box$

**Lemma 66** $\forall M \in \Lambda^{EA}$. $\Gamma \vdash_{NEAL} M : A$ then $\Gamma \vdash_{abs} \mathcal{C}an(M) : A$.

**Proof:** By Lemma 65 and Lemma 62. \hfill $\Box$

Consider the following embedding function from $Abs^{EA}$ to $\mathcal{P}(\Lambda^{EA})$:

\[
\text{bme}(x) = \{x\} \\
\text{bme}(\lambda x.M) = \{\lambda x.M' | M' \in bme(M)\} \\
\text{bme}((M_1 M_2) = \{(M_1' M_2') | M_1' \in bme(M_1) \land M_2' \in bme(M_2)\} \\
\text{bme}(\nabla(M)[N_1/x_1, \ldots, N_k/x_k]) = \\
\{[\cdots ![M']_1/x_1, \ldots, z_k/x_k)] \cdots \\
\cdots [N_1/z_1, \ldots, N_k/z_k] | \\
M' \in bme(M) \land N'_i \in bme(N_i) \land n > 0\} \\
\text{bme}([M]_{N=(x_1,x_2,\ldots,x_1^{n_1})}) = \text{bme}([M]_{y=(x_1,x_2,\ldots,x_1^{n_1})}) \\
y \text{ fresh variable} \\
\text{bme}([M]_{N=(x_1,x_2),N_2=(x_2)}) = \text{bme}([M]_{N_1=(x_1,x_2)}) \\
\text{bme}([M]_{N=(x_1,x_2)}) = \{M' | M' \in bme(M) \land N' \in bme(N)\}
\]

**Lemma 67** $\forall M \in Abs^{EA}$ $\Gamma \vdash_{abs} M : A \Rightarrow \exists M' \in bme(M) \Gamma \vdash_{NEAL} M' : A$.

**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. \hfill $\Box$

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):

```
``
### 3.2. Principal Type

<table>
<thead>
<tr>
<th>Productions</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C ::= S \mid NS$</td>
<td>a canonical form is either a sharing term or a non-sharing term.</td>
</tr>
<tr>
<td>$S ::= [LB]_{S\text{list}}$ where $\text{CV}(S\text{list}) \subseteq \text{FV}(LB)$</td>
<td>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.</td>
</tr>
<tr>
<td>$[S]_{S\text{list}}$ where $\text{CV}(S\text{list}) \cap \text{SSV}(S) = \emptyset$ and $\text{CV}(S\text{list}) \subseteq \text{FV}(S)$ and $\forall S_i \in \text{ST}(S)$ $\text{CV}(S\text{list}) \cap \text{FV}(S_i) \neq \emptyset$</td>
<td>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).</td>
</tr>
</tbody>
</table>
### Productions

<table>
<thead>
<tr>
<th>Production</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Slist</strong>: $$NS \rightarrow (\hat{x}) \mid NS \rightarrow (\hat{x}), Slist$$&lt;br&gt;where $$</td>
<td>\hat{x}</td>
</tr>
<tr>
<td><strong>NS</strong>: $$LB \mid x$$</td>
<td>a non-sharing term is either a linear or banged one or is a variable.</td>
</tr>
<tr>
<td><strong>LB</strong>: $$L \mid B$$&lt;br&gt;<strong>L</strong>: $$(NS NS) \mid \lambda x.NS$$</td>
<td>a linear term can be an application or an abstraction of non-sharing terms.</td>
</tr>
<tr>
<td>$$\lambda x.S$$ where $$S = [C]_{NS-(\hat{y})}$$ and $$x \in FV(NS)$$</td>
<td>a linear term can also be an abstraction of a sharing term provided that it shares a single subterm and that the variable binded by the abstraction is free in the shared subterm.</td>
</tr>
<tr>
<td><strong>DSL</strong>: $$D \mid S \mid L$$&lt;br&gt;&lt;br&gt;<strong>B</strong>: $$D \mid \nabla(DSL)[\hat{x}/\hat{y}]$$ where $$SBV(\hat{x}/\hat{y}) \cap SSV(DSL) = \emptyset$$ and $$BV(\hat{x}/\hat{y}) = FV(DSL)$$</td>
<td>this kind of box is all around the term and only (and all the) variables exit from the $$\nabla$$. Actually, inside the box the term can be linear, or a sharing term with no single variables contracted, or a banged term “of type $$D$$”, i.e. with a box closed before a subterm.</td>
</tr>
<tr>
<td><strong>D</strong>: $$\nabla(DSL)[Blist]$$ where $$SBV(Blist) \cap SSV(DSL) = \emptyset$$ and $$BV(Blist) = FV(DSL)$$</td>
<td>this kind of banged terms put the box before a subterm. As in the previous case, if the term inside the box is a sharing term, then it must have no single variable contracted. The distinction between two kinds of banged terms means that we can put a sequence of boxes closing them before one or more subterms (type $$D$$) and finally eventually box all the term (type $$B$$).</td>
</tr>
</tbody>
</table>
### Lemma 68 (Soundness of $\mathcal{C}an$)

For all $M \in \Lambda^{EA}$, $\mathcal{C}an(M) \in C^{EA}$.

#### Proof:

By absurd let $M$ be the smallest normal form such that $M \notin \mathcal{L}_C$ (the language of $C^{EA}$). By structural induction on $M$ we will show that either $M \in \mathcal{L}_C$ or $M$ is not in $\rightarrow_{\mathcal{C}an}$-normal form.

$M = x$ then $M \in \mathcal{L}_C$ thanks to the derivation $C \rightarrow NS \rightarrow x$;

$M = \lambda x.M'$ and by hypothesis $M'$ is in $\mathcal{N}F$ and $M' \in \mathcal{L}_C$ because it is smaller than $M$.

But we have the following derivations:

\[
C \rightarrow NS \rightarrow LB \rightarrow L \rightarrow \lambda x.NS
\]

\[
\lambda x.S \in [C]_{NS \rightarrow \mathcal{N}F} \text{ and } x \in \mathcal{FV}(NS)
\]

$M' \in \mathcal{L}_C$ then, by the first production rule of the grammar, either $M' \in \mathcal{L}_S$ or $M' \in \mathcal{L}_S$. If $M' \in \mathcal{L}_S$ we have by derivation above $M \in \mathcal{L}_C$. Then it must be $M' \in \mathcal{L}_S$. If $M'$ contracts more than one subterm, i.e. it has the form $[M'']_{N_1 \rightarrow (\bar{x}), N_2 \rightarrow (\bar{x})}$, then since in $M$ all variables are linear, $x$ cannot be at the same time in the free variables of both $N_1$ and $N_2$, hence there is a $\rightarrow_{\lambda}$-redex. The same if $M'$ contracts only one subterm but $x$ is not in the free variables of such subterm. Then it must be $M' = [M'']_{N \rightarrow \mathcal{N}F}$ and $x \in \mathcal{FV}(N)$ thus $M \in \mathcal{L}_C$.

$M = (M_1, M_2)$ and by hypothesis $M_1, M_2$ are normal forms in $\mathcal{L}_C$. Consider now the derivation:

\[
C \rightarrow NS \rightarrow LB \rightarrow L \rightarrow (NS NS)
\]

If both $M_1$ and $M_2$ are in $\mathcal{L}_S$ then $M \in \mathcal{L}_C$, then it must be either $M_1 \notin \mathcal{L}_S$ or $M_2 \notin \mathcal{L}_S$. But $M_1 \in \mathcal{L}_C$ then either $M_1 \in \mathcal{L}_S$ or $M_1 \in \mathcal{L}_S$. But if $M_1 \in \mathcal{L}_S$ then there is a $\rightarrow_{\alpha_{\mathcal{C}an} - 1}$-redex. Analogously for $M_2$.

$M = [M_1]_{\mathcal{S}list}$ and by hypothesis $M_1$ and every $N$ of $\mathcal{S}list$ are normal forms in $\mathcal{L}_C$. Consider the derivation:

\[
C \rightarrow S \rightarrow [LB]_{\mathcal{S}list} \quad \mathcal{C}V(\mathcal{S}list) \subseteq \mathcal{F}V(LB)
\]

\[
[S]_{\mathcal{S}list} \quad \mathcal{C}V(\mathcal{S}list) \cap \mathcal{S}S\mathcal{V}(S) = \emptyset
\]

\[
\forall S_i \in \mathcal{S}T(S) \quad \mathcal{C}V(\mathcal{S}list) \cap \mathcal{F}V(S_i) \neq \emptyset
\]

If $\mathcal{C}V(\mathcal{S}list) \subseteq \mathcal{F}V(M_1)$ then there is a $\rightarrow_{\mathcal{C}an} - 2$-redex, thus $\mathcal{C}V(\mathcal{S}list) \subseteq \mathcal{F}V(M_1)$. Now $M_1 \in \mathcal{L}_C$ then $M_1 \in \mathcal{L}_S$ or $M_1 \in \mathcal{L}_S = \mathcal{L}_S \cup \mathcal{L}_x$ i.e. there are three cases:

<table>
<thead>
<tr>
<th>Productions</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Blist ::= L/x</td>
<td>L/x, Blist</td>
</tr>
</tbody>
</table>
1. $M_1 \in \mathcal{L}_x$ then there is a $\rightarrow_{c_1-1}$-redex;
2. $M_1 \in \mathcal{L}_{LB}$ then $M \in \mathcal{L}_C$ by derivation above;
3. $M_1 \in \mathcal{L}_S$ then there are two cases:
   (a) $\mathcal{CV}(\text{Clist}) \cap \mathcal{SSV}(M) \neq \emptyset$ then there is a $\rightarrow_{\text{var-collaps}}$-redex;
   (b) $\exists S_i \in \mathcal{ST}(S) \mathcal{CV}(\text{Stlist}) \cap \mathcal{FV}(S_i) = \emptyset$ then there is a $\rightarrow_{c_1}$-collaps-redex.

$M = \nabla(M_1)[\text{Mlist}]$ and $M_1$ and each $N_i$ in Mlist are normal forms $\in \mathcal{L}_C$. Consider the derivation:

$$C \rightarrow NS \rightarrow LB \rightarrow B \rightarrow \nabla(DSL)[\hat{x}/\hat{y}] \quad \mathcal{BV}(\hat{x}/\hat{y}) = \mathcal{FV}(DSL)$$
$$\quad \backslash \quad \mathcal{SBV}(\hat{x}/\hat{y}) \cap \mathcal{SSV}(DSL) = \emptyset$$
$$\nabla(DSL)[\text{Blist}] \quad \mathcal{BV}(\text{Blist}) = \mathcal{FV}(DSL)$$
$$\quad \backslash \quad \mathcal{SBV}(\text{Blist}) \cap \mathcal{SSV}(DSL) = \emptyset$$

If $\mathcal{BV}(\text{Mlist}) \neq \mathcal{FV}(M_1)$ then there is a $\rightarrow_{\nabla-c_2}$-redex, thus $\mathcal{BV}(\text{Mlist}) = \mathcal{FV}(M_1)$.

Now $M_1 \in \mathcal{L}_C = \mathcal{L}_S \cup \mathcal{L}_{NS}$.

1. If $M_1 \in \mathcal{L}_S \subseteq \mathcal{L}_{DSL}$ then if $\mathcal{SBV}(\text{Mlist}) \cap \mathcal{SSV}(M_1) \neq \emptyset$ then there is a $\rightarrow_{c_2}$-redex else there must exist a $N_i \notin \mathcal{L}_L \cup \mathcal{L}_x$, with $N_i \in \mathcal{L}_C$ by hypothesis.
   (a) $N_i \in \mathcal{L}_S$ then there is a $\rightarrow_{\nabla-c}$-redex;
   (b) $N_i \in \mathcal{L}_{NS}$ then it must be $N_i \in \mathcal{L}_B$ then there is a $\rightarrow_{\nabla_{\nabla}}$-redex; otherwise $M \in \mathcal{L}_C$ by the derivation above.
2. If $M_1 \in \mathcal{L}_{NS}$
   (a) if $M_1 \in \mathcal{L}_L$ then there must exist a $N_1 \notin \mathcal{L}_L \cup \mathcal{L}_x$, with $N_i \in \mathcal{L}_C$ by hypothesis. As above by cases on $N_i$;
   (b) if $M_1 \in \mathcal{L}_x$ then there is a $\rightarrow_{\nabla_1}$-redex;
   (c) if $M_1 \in \mathcal{L}_B$ it must be $M_1 \notin \mathcal{L}_D$ then there is a $\rightarrow_{\nabla_{\nabla}}$-redex; otherwise $M \in \mathcal{L}_C$ by the derivation above.

\[\Box\]

**Lemma 69** $\forall M \in C^{EA}$, $M$ is in $\rightarrow_{\nabla_{\text{can}}}$ normal form.

**Proof:** By induction on the length of derivation of $M$, there is no redex in it.

\[\Box\]

### 3.2.4 Canonical Forms Algorithm $\mathcal{C}$

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 $C(M) = \{ N \mid \exists R \in \Lambda^{EA} \text{ such that } (R)^* = M, \text{ R is simple and } N = \text{can}(R) \}$.

**Lemma 70** For every $M \in \Lambda$, $C(M)$ is finite.
We will show an algorithm $\mathcal{C}$ such that, for every $M \in \Lambda$, $\mathcal{C}(M)$ gives either $C(M)$ or a negative answer. $\mathcal{C}$ is correct and complete.

Let $L(M)$ be the *linearization* of $M$ with respect to all its free variables and let $L_x(M)$ be the set of fresh variables generated by $L$ during the linearization of $x$ in $M$. I.e. let $M = (x \ (x \ y \ y))$ then $L(M) = (x_1 \ (x_2 \ y_1 \ y_2))$ and $L_x(M) = \{x_1, x_2\}$, $L_y(M) = \{y_1, y_2\}$ and $L_z(M) = \emptyset$ for any other variable $z$.

The algorithm is defined by the following equations:

$$\mathcal{C}(M) = \begin{cases} \text{if } \exists x_1, \ldots, x_k \in \iota \ FV(M) \text{ then } \\ \left[ T(L(M)) \cup F(L(M)) \right]_{x_1 \to (L_{x_1}(M)), \ldots, x_k \to (L_{x_k}(M))} \\ \text{else} \\ T(M) \cup F(M) \end{cases}$$

$$T(x) = \{x\}$$

$$T(\lambda x. M) = \begin{cases} \text{if } x \in \iota \ FV(M) \text{ then } \\ \lambda x. \left[ T(L(M)) \cup F(L(M)) \right]_{x \to (L_x(M))} \\ \text{else} \\ \lambda x. (T(M) \cup F(M)) \end{cases}$$

$$T((M \ N)) = T(M) \circ T(N) \cup F(N)$$

$$F(x) = \emptyset$$

$$F(M \neq x) = F'(M) \cup T((F'(M) \{z/y \ FV(M)\}) \cup T(M \{z/y \ FV(M)\}))[FV(M)/z]$$

$$F'(x) = \emptyset$$

$$F'(M \neq x) = \forall A_1, \ldots, A_n. P, n > 0 \text{ s.t.}$$

$$M = A_1 \ P \{A_1/y_1, \ldots, A_n/y_n\}$$

$$\{y_1, \ldots, y_{n+k}\} = FV(P) \ P \neq x$$

$$\forall 1 \leq i \leq n \ A_i = (A_i_1 \ A_i_2)$$

$$\nabla \left( T(P) \{\hat{z}_1/y_{n+1}^{n+k}\} \cup (F'(P) \{\hat{z}_1/y_{n+1}^{n+k}\}) \left[ T(A_1) / y_1, \right. \right.$$

$$\left. \ldots, T(A_n) / y_n, y_{n+1}^{n+k} / z_1 \right]\right.$$  

Where $\hat{z}$ and $\hat{z}_1$ are fresh variables. $\hat{z}_1/y_{n+1}^{n+k}$ stands for $z_1/y_{n+1}, \ldots, z_k/y_{n+k}$, $\hat{z}/FV(M)$ stands for the complete renaming of free variables of $M$ width fresh ones, and $FV(M)/\hat{z}$ stands for the inverse substitution.

**Fact 71** $\forall C \in \mathcal{C}(M) \ (C)^* = M$

In order to prove soundness and completeness of $\mathcal{C}$, 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):

---

**Note:** The text contains mathematical expressions and formal logic statements that are typical for a technical or academic paper. The context suggests a discussion on formal systems and their properties, possibly within the domain of logic or computer science.
### Productions

<table>
<thead>
<tr>
<th>Production</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CC</strong> ::=</td>
<td>([K]_{Clist}) where (CV(Clist) \subseteq FV(K))</td>
</tr>
<tr>
<td><strong>Clist</strong> ::=</td>
<td>(y \rightarrow (\hat{x}))</td>
</tr>
<tr>
<td><strong>K</strong> ::=</td>
<td>(\nabla (B)[\hat{x}/\hat{y}]) where (BV(\hat{x}/\hat{y}) = FV(B))</td>
</tr>
<tr>
<td><strong>B</strong> ::=</td>
<td>(\nabla (B)[L]) where (BV(L) = FV(B))</td>
</tr>
<tr>
<td><strong>L</strong> ::=</td>
<td>(A/x)</td>
</tr>
<tr>
<td><strong>R</strong> ::=</td>
<td>(\lambda x.[K]_{x \rightarrow {x_1, \ldots, x_n}}) where ({x_1, \ldots, x_n} \subseteq FV(K))</td>
</tr>
<tr>
<td><strong>A</strong> ::=</td>
<td>((R K))</td>
</tr>
</tbody>
</table>

where all variables are linear, \(\hat{x}\) stands for \(x_1, \ldots, x_n\) and \(n > 0\). 

Notice that side condition \(|\hat{x}| \geq 2\) in the production of **Clist** implies \(|x|_{Clist}\) is not a possible term in \(CC^{EA}\) by side condition \(CV(Clist) \subseteq FV(K)\) in production of **CC** and by \(\{x_1, \ldots, x_n\} \subseteq FV(K)\) 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 \( M \).

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 \( \square \).

**Proof:** If \( M \) is simple then \( SFV(N) = \emptyset \) for any subterm \( N \) of \( M \). Hence looking at the grammar, the only possible contraction is \( [LB]_{Slist} \).

Lemma 75 *If \( M \in C^{EA} \) is simple then \( \nexists N \) subterm of \( M \) such that \( N \) has the form \( \nabla \).

**Proof:** If \( M \) is simple then \( SSV([N']_{Slist}) \subseteq FV([N']_{Slist}) = BV(Blist) \). Now suppose \( SSV([N']_{Slist}) \neq \emptyset \), then there exists \( x \in SSV([N']_{Slist}) \). \( x \notin SBV(Blist) \) by the grammar, then \( x \in BV(Blist) \setminus SBV(Blist) \) hence \( M \) contracts a subterm that is not a variable and this contradicts the hypothesis.

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 \( \Gamma \vdash_{abs} M : A \land M \notin CC^{EA} \). 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.

Lemma 77 Let \( L_X \) 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. \( T(M) \subseteq L_R \cup \{ x \} \)
   (a) \( T(M \neq x) \subseteq L_R \)
   (b) \( T((M_1 \ M_2)) \subseteq L_A \)
2. \( F(M) \subseteq L_B \)
3. \( \eta(M) \subseteq L_K \)

Lemma 78 Let \( L_{CC} \) the language of canonical forms generated by the grammar of Definition 42, then

\[ \forall M \in \Lambda \ \ \eta(M) \subseteq L_{CC} \]

Lemma 79

1. \( M \in L_R \cup \{ x \} \Rightarrow M \in T((M)^*) \)
2. \( M \in L_A \Rightarrow (M)^* = (M_1 \ M_2) \land M \in T((M_1 \ M_2)) \)
3. \( M \in L_B \Rightarrow M \in F((M)^*) \cup T((M)^*) \)
4. \( M \in L_K \Rightarrow M \in F((M)^*) \cup T((M)^*) \)
Lemma 80

\[ M \in \mathcal{L}_{CC} \Rightarrow M \in \mathcal{C}((M)^*) \]

Theorem 81

\[ \bigcup_{M \in \Lambda} \mathcal{C}(M) = \mathcal{L}_{CC} \]

Proof: By Lemma 80 and Lemma 78.

Theorem 82 (Soundness and Completeness of \(\mathcal{C}\)) \(\forall M \in \Lambda\)

1. \(\mathcal{C}(M) \subseteq C(M)\);
2. \(N \in C(M)\) and \(\exists \Gamma, A\) s.t. \(\Gamma \vdash_{abs} N : A \Rightarrow N \in \mathcal{C}(M)\).

Proof: We recall the definition of \(C(M)\):

\[ C(M) = \{ N \mid \exists R \in \Lambda^{EA} \text{ s.t. } (R)^* = M, R \text{ is simple and } N = \text{Can}(R) \} \]

1. By Lemma 78 for any \(C \in \mathcal{C}(M)\) we have \(C \in C^{CEA}\) and hence, by Lemma 72, \(C \in C^{EA}\). Moreover \((C)^* = M\) by Fact 71. Then \(C\) is in \(C(M)\) because it exists \(R \in bme(C)\) s.t. \((R)^* = M\), \(R\) is simple and it is sufficient to notice that \(\text{emb}(bme(C))\) is either equal to \(C\) (that is in \(\neg\text{Can}\) normal form) or there are a set of \(\neg\text{var-collaps}, \neg\text{c-collaps}\) and \(\neg\text{v-collaps}\) redexes after firing them we get \(C\) again.
2. By Lemma 68 \(\text{Can}(R) = N \in C^{EA}\). \(R\) is simple by hypothesis and then \(N\) is simple too. Moreover \(\Gamma \vdash_{abs} N : A\), hence, by Lemma 76, \(N \in C^{CEA}\) and then \(N \in \mathcal{C}(M)\) by Lemma 80.

Theorem 83 (Principal typing for \(\Lambda\) in \(EAL\)) \(\forall M \in \Lambda \Gamma \vdash_{\text{NEAL}} M : A\) 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 \(N \in \mathcal{C}(M)\).

Proof:

(If) Let be \(PT(N) = < \Theta, \sigma, C >\), then by Theorem 64 \(\Gamma \vdash_{abs} N : A\) and by Lemma 67 \(\exists N' \in bme(N) \Gamma \vdash_{\text{NEAL}} N' : A\), hence the thesis.

(Only if) \(\Gamma \vdash_{\text{NEAL}} M : A\) then by definition \(\exists R \in \Lambda^{EA} \Gamma \vdash_{\text{NEAL}} R : A\) and \(R\) is simple. Then \(\Gamma \vdash_{abs} \text{Can}(R) : A\) by Lemma 66. Moreover \(\text{Can}(R) \in C^{EA}\) by Lemma 68 and then, being \(\text{Can}(R)\) typeable, \(\text{Can}(R) \in C^{CEA}\) by Lemma 76. This is sufficient to prove that \(\text{Can}(R) \in \mathcal{C}(M)\) (by Lemma 81). The thesis holds by principal typing for \(\text{Abs}^{EA}\).
3.2.5 Conclusions

We presented two different approaches for obtaining complete algorithms to derive EAL-types for $\lambda$-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 $\lambda$-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 $C$ 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.
The Implementation of Functional Languages
4

OptimalReducers

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 $\lambda$-terms. The grammar accepted by the parser is showed in Table 4.1. The parser accepts a list of definitions of the form $\text{Id} = \text{Term}$ 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 $\lambda x. y. M$ is the same of $\lambda x. y. M$;
- 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 $\lambda$-terms:

\[
\frac{x : \text{newvar} \vdash x : \text{newvar}}{\text{ax}} \quad \frac{B \vdash M : t}{B \cdot \{x\} \vdash \lambda x. M : B(x) \rightarrow t} \rightarrow
\]

\[
\frac{B_1 \vdash M_1 : t_1 \quad B_2 \vdash M_2 : t_2}{B_1, B_2 \vdash (M_1 M_2) : s'(\text{newvar})}
\]

\[
\text{unify}(\text{commonvars}(B_1, B_2)) = s \quad \text{unify}(s(t_2) \rightarrow \text{newvar}, s(t_1)) = s'
\]

where $s, s'$ are substitutions of the form $\{\text{typevar} \leftarrow \text{type}, \ldots, \text{typevar} \leftarrow \text{type}\}$. 
CHAPTER 4. OPTIMAL REDUCERS

Figure 4.1: Architecture of the Optimal Reduction tool.

Figure 4.2: The GUI.

Table 4.1: Grammar accepted by the optimal reducer.

\[
\begin{align*}
\text{Start} & ::= \text{ListOfDefs}((\text{Term})?)? <\text{EOF}> \\
& | \text{Term} <\text{EOF}> \\
\text{Def} & ::= \text{Id} = \text{Term} \\
\text{ListOfDefs} & ::= \text{Def}(:\text{Def})^* \\
\text{Term} & ::= \langle \text{Id}\rangle^+.\text{Term} \\
& | (\text{Term} (\text{Term})^+) \\
& | \text{Id} \\
\text{Id} & ::= ([a-z][A-Z][0-9])^+ 
\end{align*}
\]
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 $(\lambda z.z)$, 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 4.3: Void total effect.](image1)

![Figure 4.4: Critical pair.](image2)
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:
4.1. A TOOL FOR REDUCERS COMPARISON

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:

     \[
     F(x) = \lambda x.M F(N) F(M) = F(M) F(N)
     \]

     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
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 \( \lambda \)-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 \( \lambda \)-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
4.1. A TOOL FOR REDUCERS COMPARISON

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 \(((3 \cdot 2) \cdot (\lambda x. x) \cdot (\lambda x. x))^2\) 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.

\footnote{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 absolampinggraph = 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.
4.2. IMPLEMENTATION DETAILS

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`:

```java
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:

```java
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 {
            n =}

}
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 {
                if (headnormalform)
                    ...
            }
        }
    }
}

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
CHAPTER 4. OPTIMAL REDUCERS

\}
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];
  }
A `LampingGraphNode` is a graph node with a level:

```java
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
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:

```java
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.

```java
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:

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Sta79]</td>
<td>Typed λ-calculus is not elementary recursive.</td>
</tr>
<tr>
<td>[Sch82]</td>
<td>Number of β in the reduction of simply typed lambda terms is $\Omega \left( \frac{2 \cdot \ldots \cdot 2^n}{\varepsilon} \right)$. Existence of a strategy s.t. $#\beta \leq f \in \mathcal{E}^4$.</td>
</tr>
<tr>
<td>[FS91]</td>
<td>$\exists M$ s.t. $\mu_\beta(M) &gt; 5 \mu_\pi(M)$. Propose a new measure $\nu$. Turner and Hughes implementations are $2^{\Omega(\nu)}$.</td>
</tr>
<tr>
<td>[LM96, LM97]</td>
<td>Abstract Lamping’s algorithm is $\Omega \left( 2^\nu \right)$. Propose two new measures based on Lévy’s labels.</td>
</tr>
<tr>
<td>[AM98]</td>
<td>Optimal beta is not elementary recursive.</td>
</tr>
</tbody>
</table>

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++).


