Typing and Optimal reduction for $\lambda$-calculus in variants of Linear logic for Implicit computational complexity

Présentée par Vincent Atassi pour être soutenue le 2 Décembre 2008

Rapporteurs :

Ian Mackie  
Simone Martini

École Polytechnique  
Università di Bologna

Rapporteur  
Rapporteur

Jury :

Patrick Baillot  
Jacqueline Vauzeilles  
Ian Mackie  
Simone Martini  
Christophe Fouqueré  
Virgile Mogbil  
Laurent Regnier  
Simona Ronchi della Rocca

CNRS/ENS Lyon  
Université Paris 13  
CNRS/École Polytechnique  
Università di Bologna  
Université Paris 13  
Université Paris 13  
Université Aix-Marseille 2  
Università di Torino

Directeur  
Directrice  
Rapporteur  
Rapporteur  
Examineur  
Examineur  
Examinatrice

Titre français : Typage et Réduction Optimale pour le lambda-calcul dans les variantes de la Logique Linéaire pour la complexité implicite

Laboratoire d’Informatique de Paris-Nord, UMR 7030 — CNRS, Univ. Paris 13
Acknowledgements
Contents

Acknowledgements iii

Introduction 1

1 Preliminaries 5
  1.1 Lambda calculus ........................................ 5
    1.1.1 Untyped lambda calculus .......................... 5
    1.1.2 Typed lambda calculus .............................. 7
    1.1.3 Curry-Howard isomorphism ......................... 9
  1.2 Linear Logic and Complexity bounded logics ............... 11
    1.2.1 Linear Logic ...................................... 11
    1.2.2 Complexity bounded logics ....................... 14
  1.3 Optimal Reduction and Geometry of Interaction .......... 18
    1.3.1 Lamping’s algorithm on Sharing Graphs .......... 19
    1.3.2 Semantics of execution: Geometry of Interaction 19
    1.3.3 Complexity of Optimal Reduction ................. 19

2 Type Inference in Dual light affine logic 21
  2.1 Introduction ............................................ 21
  2.2 From system F to DIAL .................................. 22
  2.3 Localisation of DIAL type inference ..................... 25
    2.3.1 Pseudo-terms ..................................... 25
    2.3.2 Pseudo-terms and proof-nets ...................... 26
    2.3.3 Local typing condition ........................... 27
    2.3.4 Boxing conditions ................................ 28
    2.3.5 Correctness of the conditions .................... 30
  2.4 Parameterisation and constraints generation .......... 34
    2.4.1 Parameterised terms and instantiations .......... 34
    2.4.2 Local typing constraints .......................... 36
    2.4.3 Boxing constraints ................................ 37
  2.5 Solving the constraints ................................ 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.1</td>
<td>Solving boolean constraints</td>
<td>39</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Solving integer constraints</td>
<td>40</td>
</tr>
<tr>
<td>2.6</td>
<td>Data-types and typing with domain specification</td>
<td>41</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Data-types</td>
<td>41</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Typing with domain specification</td>
<td>42</td>
</tr>
<tr>
<td>2.7</td>
<td>Implementation and examples</td>
<td>43</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Overview</td>
<td>43</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Two examples: reversing of list and predecessor</td>
<td>43</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Experiments with larger examples: polynomials</td>
<td>45</td>
</tr>
<tr>
<td>2.8</td>
<td>Discussion and further work for the case of propositional DLAL</td>
<td>48</td>
</tr>
<tr>
<td>2.9</td>
<td>Conclusion</td>
<td>49</td>
</tr>
<tr>
<td>3</td>
<td>Type inference with Coercions</td>
<td>51</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>51</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Some Background</td>
<td>52</td>
</tr>
<tr>
<td>3.2</td>
<td>Subtyping Coercions</td>
<td>53</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Coercions</td>
<td>53</td>
</tr>
<tr>
<td>3.2.2</td>
<td>On Base Types</td>
<td>53</td>
</tr>
<tr>
<td>3.2.3</td>
<td>General Types</td>
<td>55</td>
</tr>
<tr>
<td>3.3</td>
<td>Towards type inference</td>
<td>57</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Algorithmic system</td>
<td>57</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Equivalences</td>
<td>58</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Pseudo-terms, typability</td>
<td>61</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Correctness and completeness of criteria</td>
<td>62</td>
</tr>
<tr>
<td>3.4</td>
<td>Type Inference</td>
<td>66</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Restricted derivations</td>
<td>66</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Parametrised types and pseudo-terms</td>
<td>68</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Constraints generation</td>
<td>72</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Constraints resolution</td>
<td>74</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Correctness and completeness of the constraints</td>
<td>75</td>
</tr>
<tr>
<td>3.5</td>
<td>Conclusion</td>
<td>78</td>
</tr>
<tr>
<td>4</td>
<td>Optimal reduction for Soft Linear logic</td>
<td>79</td>
</tr>
<tr>
<td>4.1</td>
<td>Soft Sharing Graphs</td>
<td>79</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Introduction</td>
<td>79</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Soft sharing graphs</td>
<td>80</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Main properties</td>
<td>81</td>
</tr>
<tr>
<td>4.2</td>
<td>GoI semantics</td>
<td>84</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Traces</td>
<td>84</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Algebra of traces</td>
<td>84</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Word rewriting</td>
<td>85</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Multiplicative traces and exponential stacks</td>
<td>87</td>
</tr>
<tr>
<td>4.3</td>
<td>ReadBack</td>
<td>91</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Preliminary definitions</td>
<td>91</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Characteristic traces</td>
<td>92</td>
</tr>
<tr>
<td>4.3.3</td>
<td>ReadBack algorithm</td>
<td>94</td>
</tr>
<tr>
<td>4.4</td>
<td>Examples</td>
<td>96</td>
</tr>
<tr>
<td>4.4.1</td>
<td>A high complexity function in ssc</td>
<td>96</td>
</tr>
<tr>
<td>4.4.2</td>
<td>An involved ReadBack run</td>
<td>99</td>
</tr>
<tr>
<td>4.5</td>
<td>Proof of the main result</td>
<td>101</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Soft Proof-nets</td>
<td>101</td>
</tr>
</tbody>
</table>
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.2 <em>Preservation of ReadBack by cut-elimination</em></td>
<td>102</td>
</tr>
<tr>
<td>4.5.3 <em>GoI associated to a cut-free soft derivation</em></td>
<td>106</td>
</tr>
<tr>
<td>4.6 <em>Conclusion</em></td>
<td>114</td>
</tr>
<tr>
<td>Conclusion</td>
<td>115</td>
</tr>
<tr>
<td>Bibliography</td>
<td>117</td>
</tr>
<tr>
<td><strong>A Source code extracts</strong></td>
<td>121</td>
</tr>
<tr>
<td>A.1 <em>lambda_expr.ml</em></td>
<td>121</td>
</tr>
<tr>
<td>A.2 <em>params.ml</em></td>
<td>132</td>
</tr>
<tr>
<td>A.3 <em>equs.ml</em></td>
<td>139</td>
</tr>
</tbody>
</table>
Introduction

Lambda-calculus was introduced in the 1930’s by Alonzo Church, in order to study the mathematical functions from a computational point of view, thanks to a step by step reduction procedure, the beta-reduction. Within this framework, a function is not defined extensionally by a set of pairs argument/result, but intensionally, by a computational process, an algorithm. One soon realized that lambda-calculus could constitute a model of computable functions analogous to the so-called Turing machines. Later on, Turing machines have inspired the design of the first actual computers, and to this day machines still rely on this model. The most widely used programming languages, which are called imperative languages, such as C or Java are also based on the Turing model. But there also exist some less widespread languages such as Lisp and more recently, HASKELL or CAML that take lambda-calculus as basis, and are called functional programming languages. Lisp is based on pure lambda-calculus, as opposed to HASKELL or CAML which are based on typed lambda-calculus. Typing was first introduced to delineate a set of terminating lambda-terms. Several different type systems were then introduced. For the modern functional programming languages, type systems are used to statically guarantee program properties — for instance that an integer won’t be added with a real number — and for modularity.

Imperative languages are closely related to the computer architecture, therefore, the programs written in this style are easy to compute, while the evaluation of functional programs is more involved, since it requires their translation into another computational model. As a result, functional programming at first sight seems to be less efficient than imperative programming, and the cost of running a program is often more difficult to evaluate when it is a functional one than when it is an imperative one. To reduce a lambda-term, one has to contract redexes: a step of beta reduction corresponds to the contraction of a redex. However, the number of steps required does not offer a realistic measure of lambda-calculus complexity: beta reduction is not an elementary operation. Furthermore, there are several ways to implement it, and none of them turns out to be the most efficient one in all cases. This yields two main questions over lambda-calculus: what would an efficient reduction method consist in? What is the general cost of reducing a lambda-term?

Those questions have been formulated quite a long time ago, but they still haven’t been definitely answered. No current implementation of lambda-calculus reduction can claim to be provably the most efficient, and even more, we still don’t know what such an implementation would consist in.
INTRODUCTION

The present work fits into what now appears as a long tradition of research over those two questions. In order to tackle them, we will use the tools of Proof theory, of Computational complexity and of Optimal reduction.

**Proof theory.** In a seminal work, Gentzen defined a formulation for a deduction system, called *sequent calculus*, in which appeared a rule called *cut*, a kind of modus ponens. Together with the system, he defined a *cut-elimination* procedure, which consists in a step by step rewriting of proofs, each step removing an occurrence of a cut rule in the proof, until it reaches a proof in which there is no occurrence of the cut rule at all. This final proof proves the same formula as the initial one. So, this defines a *normal form* for proofs, and an algorithm to reach this normal form, called the cut-elimination procedure. Nowadays, *proof theory* is the field of logic which deals mainly with formal proofs and cut-elimination.

The *Curry-Howard correspondence* has been formulated in the 1970’s, and is often said to set up a bridge between proof theory and theoretical computer sciences: it states a correspondence between formal proofs and lambda-calculus terms; a correspondence between types and formulas and a correspondence between lambda-terms reduction and cut-elimination of proofs. Many works take advantage of the formulation of one side of the correspondence to express results on the other side. For instance, the system F type assignment system, introduced by Girard, was designed to show the consistency of second-order Peano arithmetic. *Linear logic* — also introduced by Girard, is in some way an outcome of system F, and is a logical system which emphasises the role of the duplication and erasing operations in logic. It is widely used in computer sciences as well as in proof theory. In particular, *Proof-nets*, a graphical syntax for proofs, will be a useful link between certain lambda-calculus reduction methods and cut-elimination of proofs; and the increased control on duplication will provide a better control on the computational complexity of the lambda-terms beta reduction.

**Complexity.** Computational complexity theory is the field of computer sciences that studies the complexity of problems and of functions, that is to say, the amount of space and time that their computation requires to terminate, and more precisely, the measure of the growth of the space and time that it requires to terminate when the size of the input grows. Let us mention briefly the *P-Time* complexity class, which gathers the functions that can be computed in time polynomial with respect to the size of their input. This class is often said to correspond to the *feasible functions*, since the growth remains acceptable and allows to compute in reasonable time. On the contrary, the class of Elementary functions is much less feasible: it contains the functions that can be computed in time bounded by a fixed-height tower of exponential: \( 2^{2^\ldots^2} \).

*Implicit computational complexity* (**icc**) aims at designing specific programming languages — or criteria on usual programming languages — with intrinsic bounded complexity properties. In the 1990’s, Daniel Leivant and Jean-Yves Marion did the first works that applied **icc** ideas to lambda-calculus.

During the 1990’s several variants of Linear logic enjoying intrinsic complexity properties for Proof-nets cut-elimination have been designed, first by Girard, and then by Asperti and Lafont: Bounded linear logic, Light linear logic and Soft linear logic for the *P-Time* class, and Elementary linear logic for the elementary class. Then the so-called affine variants, more convenient for typing, have been developed: Light affine logic and Elementary affine logic. And, using Curry-Howard correspondence, some type assignment systems were designed, which guarantee complexity bounds on typable programs. All those complexity-guided variants of Linear logic are called *Light logics*, even if Light linear logic is in fact the name of one particular system.

We can refresh the two above-mentioned questions thanks to cross-fertilisation of those different domains and concepts. Computational complexity theory has long been formulated over Turing machines, but with the Curry-Howard correspondence and Light logics, we are now able to formulate it independently of Turing machines. This has a direct application in
the typing of functional languages, to guarantee complexity bounds together with the usual properties of historical type systems. This is precisely what the two first contributions of the present work are about, since we present the development of a type inference algorithm for a P-Time logic, and the extension of an Elementary affine logic, together with an extension of its type inference procedure. We believe that further works on those subjects might lead us to a better understanding of complexity in the lambda-calculus, and possibly, of computational complexity in general.

**Optimal reduction.** A remarkable progress on the two previous questions was made by Lévy, when he developed the theory of Optimal reduction for the lambda-calculus. The idea of the theory is to share the redexes: with usual beta reduction, redexes may be duplicated along the reduction of a lambda-term. Lévy introduced the notion of *redex families*: two redexes belong to the same family if they are copies of a same initial redex. This defines an equivalence class, and Lévy has shown that one could decide whether two redexes belong to the same family. Optimal reduction is achieved if one manages to reduce all the redex of a family in one step. Furthermore, following a certain strategy, one can minimise the number of family reductions. Therefore, Lévy conjectured that the number of family reductions was a good measure of lambda-calculus reduction complexity.

Ten years later, Lamping designed an algorithm which implemented this idea. It was quite involved, and surprisingly enough at first sight, it turned out that Linear logic concepts could help to explain and simplify it, as did Gonthier, Abadi and Lévy. And some time later, Asperti noticed that the lambda-terms typable in Elementary affine logic could be reduced with a simplified version of Lamping’s algorithm (called Lamping’s abstract algorithm).

Nowadays, the most thrilling questions about Optimal reduction are still the ones which address its complexity, in order to try and give a definite answer to the aforementioned problematics. So far, only negative partial results have been formulated — for instance that the reduction of a family cannot be an elementary operation, but there is no definite conclusion on whether Optimal reduction can lead to the provably most efficient way of reducing lambda-terms.

Our last contribution is about the use of a variant of Optimal reduction for lambda-terms that are typable in a P-Time logic. We believe that, in spite of the partial negative results known about Optimal reduction, it is still a promising way to achieve a provably most efficient method for the lambda-calculus reduction. More precisely, we also think that the techniques and tools that have been developed in Light logics, notably the type inference procedures, could lead to a finer, and thus more efficient, Optimal reduction algorithm.

**Outline.** This is a brief outline of the present work:

**Chapter 1** is devoted to the basic definitions that we will need all along this work, mainly: pure and typed lambda-calculus, complexity, Light logics and Optimal reduction.

**Chapter 2** contains the development of a decoration algorithm which, starting from a system F term decides whether it is typable in a variant of Light linear logic — Dual light affine logic — and if so, gives a typing. It is the first Light types type inference algorithm which takes into account polymorphic types, and the first P-Time type inference algorithm for a polynomial type assignment system. We also provide a prototype implementation written in functional CAML. This work was done in collaboration with Baillot and Terui and published in [ABT06, ABT07].

**Chapter 3** is the account of an extension of Elementary affine logic (EAL), for which we give a correct and complete type inference algorithm. We define an extension of the usual type assignment system with subtyping, we prove that the type assignment system still guarantees complexity bounds, and we adapt the type inference algorithm to the presence of subtyping. This is a fully-fledged first-order type inference algorithm: we start from a pure lambda-term, and not a system F typed term, to produce its EAL typing if it exists.
Chapter 4 is dedicated to an application of Optimal reduction to the normalisation of lambda-terms typed in Soft linear logic. We introduce a slight variant of Lamping’s Optimal reduction algorithm. Then we show that it is correct and complete for the Optimal reduction of lambda-terms typable in Soft linear logic, and that the complexity bound is preserved.
Chapter 1

Preliminaries

1.1 Lambda calculus

The lambda calculus theory is at the heart of the present work. We will give some of its basic definitions and properties. See [CH06] for a historical survey on the subject.

1.1.1 Untyped lambda calculus

Given a countable set of variables \( \mathcal{V} \), whose elements will be denoted by \( x, y, z, \ldots \), the terms of lambda calculus, further on called lambda terms, are defined as follows (we use Krivine's syntax):

\[
t ::= \lambda x.t \mid (t) t \mid x, x \in \mathcal{V}
\]

They will be denoted by \( t, u, M, N, \ldots \) We may omit the application parenthesis, in which case \( t_1 \cdots t_n \) stands for \((\cdots((t_1)\cdots t_{n-1})t_n)\); if there are several binders we may write only the first one: \( \lambda x_1 \cdots x_n.t \) stands for \( \lambda x_1 \cdots \lambda x_n.t \).

In a term of the form \( \lambda x.t \), we say that \( x \) is bound in \( t \): occurrences of \( x \) are bound to the \( \lambda x \), this is why lambda’s are often called binders. On the contrary, a variable \( y \) occurring in a lambda term \( t \) and which is not bound is said to be free. We denote by \( FV(t) \) the set of free variables of \( t \). A lambda term \( t \) s.t. \( FV(t) = \emptyset \) is said to be closed.

We consider lambda terms up to \( \alpha \)-renaming. That is to say, we will identify terms which differ only by the name of their bound variables. A lambda term is said to follow Barendregt’s convention if all its bound variables have different names, which is sometimes a convenient way to treat \( \alpha \)-renaming.

A context is a kind of lambda term in which one can let, possibly, one or several holes. Intuitively a hole represents an undefined subterm. Contexts are defined as follows:

\[
C, D ::= [] \mid \lambda x.C \mid (C) D \mid x, x \in \mathcal{V}
\]
CHAPTER 1. PRELIMINARIES

$C[t]$ denotes the lambda term formed by the context $C$ in which the hole has been replaced by $t$. If there are several holes in $C$, $C[t_1, \ldots, t_n]$ denotes the lambda term formed by the context $C$ in which the holes have been replaced by $t_1, \ldots, t_n$.

A relation $R$ defined on lambda terms is **closed by context** if the following equivalence holds:

$t \mathrel{R} t' \iff C[t] R C[t']$, for any $C$

We need an operation on lambda terms called **substitution**: $t\{t'/x\}$ denotes the substitution of $x$ by $t'$ in $t$:

\[
x\{t'/x\} = t'
\]

\[
y\{t'/x\} = y, \quad y \neq x
\]

\[
[\lambda x.t_1]\{t'/x\} = \lambda x.t_1 \quad [(t_1) t_2]\{t'/x\} = (t_1\{t'/x\}) t_2\{t'/x\}
\]

We can denote several substitutions like this: $t\{u/x, v/y\}$. We may note $t\{u_i/x_i\}$ for $t\{u_1/x_1, \ldots, u_n/x_n\}$. Note that a free variable can be captured by a replacement, but not by a substitution.

Lambda calculus is equipped with a reduction relation denoted by $\Rightarrow$ named $\beta$-reduction, which is the smallest contextual closure of this relation:

\[
(\lambda x.t_1) t_2 \Rightarrow t_1\{t_2/x\}
\]

$\Rightarrow^*$ is the reflexive and transitive closure of $\Rightarrow$, and $\Rightarrow_\beta$ is the reflexive, transitive and symmetric closure of $\Rightarrow$.

An occurrence of a configuration $(\lambda x.t_1) t_2$ in a term is called a **redex**. A lambda term is said to be in **normal form** when it contains no redex. Normal lambda terms have the following syntax:

\[
B = \lambda x_1 \cdots x_n.z B \cdots B
\]

and for each subterm, $z$ is its **head variable**.

We will use lambda calculus as a programming language, in which integers are represented by **Church integers**:

\[
n = \lambda sz.(s)\cdots(s)z
\]

They can be seen as unary integers, or as iterators, since a church integer $n$ allows the iteration of a function $n$ times on a base argument. In the sequel, we will denote the Church integer $n$ by $\underline{n}$.

Now, the main properties of lambda calculus are the following:

**Theorem 1.1 (Confluence, [CR36]).** For all $t, t_1, t_2$ such that $t \Rightarrow^* t_1$ and $t \Rightarrow^* t_2$, there exist $t_3$ such that $t_1 \Rightarrow^* t_3$ and $t_2 \Rightarrow^* t_3$.

**Corollary 1.2.** If $t$ admits a normal form, then this normal form is unique.

**Theorem 1.3 ([Tur37]).** Lambda calculus is Turing complete.

**Graphical representation**

We will often manipulate lambda terms as graphical objects rather than as a language. We design lambda terms as trees in which the link between a variable and its binder, if any, is made explicit. Furthermore when there are several occurrences of a same variable, we can also make it explicit with a triangle. In those informal representation, we can use $n$-ary triangles to denote $n$ occurrences of a variable. Figure 1.1 gives the graphical representation of Church integer $2$. Lambda nodes are represented by a $\lambda$, and application nodes by a $\@$. 


1.1 LAMBDA CALCULUS

1.1.2 Typed lambda calculus

Pure or untyped lambda calculus is confluent, but not terminating. Types for lambda calculus have been first introduced by Church, following ideas of the Type theory of Whitehead and Russell.

Simple types

Given a countable set of variables $\mathcal{B}$, whose elements will be denoted by $\alpha, \beta, \ldots$, simple types are defined by the following grammar:

$$T ::= \alpha \mid T \Rightarrow T, \alpha \in \mathcal{B}$$

Keeping the pure lambda calculus syntax is the à la Curry syntax of simply typed lambda calculus. We can define an enriched syntax to give rise to à la Church simply typed lambda calculus:

$$t ::= x^T \mid (\lambda x^T \cdot t^{T_1})^{(T_1 \Rightarrow T_2)} \mid ((t_1)^{T_1 \Rightarrow T_2} t_2^{T_1})^{T_3}$$

Figure 1.2 gives the typing rules. The $\vdash$ symbol is called “turnstile”. At the left-hand side of the turnstile, we have the context, often denoted by $\Gamma, \Delta, \ldots$, which is a set of pairs $(x : A)$

$$\Gamma \vdash x : A$$

$$\Gamma, x : A \vdash t : B$$

$$\Gamma \vdash \lambda x.t : A \Rightarrow B$$

$$\Gamma \vdash (t_1) t_2 : B$$

Figure 1.3: Church integer 2 in simple types type assignment system
where \( x \) is a variable and \( A \) a type. The notation \(|\Gamma|\) denotes the number of elements in \( \Gamma \), and we write \( \Gamma \neq \Delta \) to state that \( \Gamma \) and \( \Delta \) have disjoint domains. Intuitively, the context collects the types of the free variables of a lambda term. On the right-hand side, we have an assertion of the form \( t : A \), where \( t \) is a lambda term and \( A \) a type. It is to be read " \( t \) is of type \( A \) ".

The \( \text{Ax} \) rules states that if \( x : A \) is in the context then, \( x \) has type \( A \); the \( \Rightarrow \text{-I} \) rule states that if one can type \( \Gamma, x : A \vdash t : B \), then \( \Gamma \vdash \lambda x. t : A \Rightarrow B \) is typable. Building a \textit{type derivation} is done by decomposing a lambda-term in a bottom-up way, using at each step the adequate typing rule until one reaches a variable.

The simple type of Church integers is \((\alpha \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha))\). Figure 1.1.2 gives a graphical representation of typing, on the graphical syntax we introduced. At the binder level, we represent the type of the bound variables (here for instance, the topmost lambda binds a variable of type \((\alpha \Rightarrow \alpha)). On each edge, the type associated to the subterm rooted at this point. Each variable edge carries the type of the associated variable; above the application nodes, we have the type of the result of the application; and above the abstraction nodes the type result of the abstraction.

The fundamental results on simples types are the following:

**Theorem 1.4** (termination of the reduction, [Taal67]). Given a lambda term \( t \) admitting a simple type derivation, any reduction sequence of \( t \) terminates.

**Theorem 1.5** (expressive power, [Sta77]). The problem of reducing simply typed lambda terms is non-elementary.

The notion of type assignment system for lambda calculus gives rise to the problems of \textit{typability}, and of \textit{type inference}:

**Problem 1.6** (typability for simple types). Given a pure lambda term \( t \) with free variables \( x_1, \ldots, x_n \), does there exists a type derivation of conclusion \( x_1 : A_1, \ldots x_n : A_n \vdash t : A \)?

**Problem 1.7** (type inference for simple types). Given a pure lambda term \( t \) with free variables \( x_1, \ldots, x_n \), give all its possible type derivations.

**Theorem 1.8.** Given a pure lambda term \( t \) typability and type inference are decidable.

### System F

The system F type system has been introduced by Girard in the early 1970′s ([Gir72]). Its typing rules are given in figure 1.4. It introduces \textit{polymorphism}, that is to say, universal quantification on type variables. It results in adding the two rules \( \forall \text{-I} \) and \( \forall \text{-E} \), and produces a type system dramatically more expressive than simple types. It has deep logical properties, but from our point of view, we mainly focus on it because it is one of the main basis of modern typed functional programming languages.
1.1 LAMBDA CALCULUS

Types are on the grammar:

\[ T ::= \alpha \mid T \Rightarrow T \mid \forall \alpha. T, \alpha \in B \]

In this system, we have to consider type substitution and \( \alpha \)-conversion on type variables. In the rule \( \forall \)-E, \( A[B/\alpha] \) denotes the type \( A \) in which free occurrences of \( \alpha \) are substituted by \( B \). As for lambda calculus variables, we now consider type variables up to \( \alpha \)-conversion.

We can provide an enriched \( \text{à la Church} \) syntax for system F terms:

\[
\begin{align*}
t &::= x^T \mid (\lambda x^{T_1}. t^{T_2})^{T_1\Rightarrow T_2} \mid ((t_1)^{T_1\Rightarrow T_2} \! t_2^{T_1})^{T_2} \mid (\forall \alpha. t^{T_2})^{\forall \alpha. T} \mid ((t_1)^{\forall \alpha. T_1} \! t_2^{T_1})^{T_2[T_2/\alpha]}
\end{align*}
\]

In this type system, church integers can be given the type \( \forall \alpha.(\alpha \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha) \). Figure 1.5 gives the typing of 2.

**Theorem 1.9 (Gir72).** If \( \Gamma \vdash t : A \) is derivable in F type assignment system, then any sequence of reductions of \( t \) terminates. The typable functions of system F correspond to second order Peano arithmetic functions.

There is no sensible termination bound on system F terms, and we have a negative answer to the problems of typability and type inference:

**Theorem 1.10 (Wel99).** The problems of typability and of type inference for system F are equivalent and undecidable.

### 1.1.3 Curry-Howard isomorphism

Very early, Haskell Curry noticed a correspondence between logical systems and combinators types. Nicolas DeBruijn then used lambda terms to represent proofs, and it was later, in 1969, properly formalised by Howard in a work which was published only in 1980: [How80]. For a comprehensive introduction to Curry-Howard isomorphism, see [GLT89].

The correspondence was first established between simple types type system and the so-called minimal logic, and this correspondence has then scaled to various type systems and to various logics, including system F. The logic corresponding to system F is second order Peano arithmetic.

The correspondence appears at different level: formulas of logic correspond to types in lambda calculus, then lambda terms correspond to proofs. Most interestingly the lambda calculus beta reduction corresponds to proofs cut-elimination, which we will present in the next section.
Sequent calculus and Cut-elimination

All logical systems that we will see admit two formulations, one in the *natural deduction style* and another in the *sequent calculus style*. Natural deduction is based on a symmetry between introduction and elimination of logical connectives, whereas sequent calculus is rather based on a symmetry between hypothesis and conclusions manipulations. In Figure 1.6, we give a sequent calculus presentation for minimal logic.

The first two rules define the *identity group*: axiom and cut. Axiom was present in type assignment (or natural deduction) systems. Note however that in sequent calculus, the context of the axiom is formed of only one hypothesis. The cut rule (also called modus ponens) is proper to sequent calculus presentations, and has a central role.

The next two rules shed some light on structural operations, and are thus called *structural rules*, in order to stress that they are indeed not *logical rules*. They are the rules of *contraction* and *weakening*. They manipulate hypothesis, but do not have a clear logical meaning.

The last two rules are the *logical rules*. Note that they both introduce the logical connective $\Rightarrow$, and that the symmetry is now based on the manipulated side: introduction on the right (conclusion) or on the left (hypothesis).

We can say that natural deduction or sequent calculus is mainly a matter of presentation because they are equivalent from the point of view of provability or typability.

The deepest importance of sequent calculus lies in the cut rule, and more precisely, in the ability to realize that it is redundant in this system: everything that we can prove with the cut rule can be proven without the cut rule. What interests us most in this result is the way the proof is done. Indeed, Gentzen defines a cut-elimination procedure, which is a kind of rewriting of proofs. Each step eliminates a cut, and the procedure is proven to be terminating — and it ends with a cut-free proof.

**Theorem 1.11** (*Gen35*). *If* $\Gamma \vdash t : A$ *is derivable then* $\Gamma \vdash t' : A$ *is derivable without cut.*

Following Curry-Howard correspondence, a sequence of of cut-elimination steps corresponds to a sequence of beta reduction steps in lambda calculus. So globally, cut-elimination corresponds to programs (lambda calculus terms) normalisation. This remark motivates the use of a logical system to control programs complexity: if we manage to design a logical system in which complexity of cut-elimination is time-bounded, then we will have a type assignment system bounding the complexity of programs execution.

**Implicit Computational Complexity**

Computational complexity is the field of computer sciences, whose concern is to determine bounds on the time or the space required by a *function* or by a *program* for its execution.
Mind that there is a big difference between those two concerns: determining the complexity of function amounts to determining the complexity of a problem in an abstract formulation. An example of this is the Theorem 1.5, which ensures that any program solving the problem of the reduction of simply typed lambda calculus will have a non elementary complexity.

In this work, we will rather take the less abstract problematic: determining an upper bound on the complexity of algorithm. Given a program, what is its running time w.r.t the size of its entries? — of course this question is undecidable in the general case. This question is more down-to-earth, but also more crucial in the field of computer sciences: what really matters is to assert that a program will indeed run in reasonable time and space, and not that the problem it solves has a certain theoretical complexity.

The field of Implicit computational complexity (icc) aims at designing programming languages — or of criteria on programming languages — ensuring complexity bounds, moreover those criteria should be implicit: the complexity bound must not directly appear in the program’s construction, so as to impede programming as little as possible. ICC has been launched by a study on primitive recursive functions in [BC92]. The first attempt at using lambda calculus as a source language has been made in [LM93]. The first logical system designed to enjoy complexity bounds on cut-elimination was Bounded linear logic, introduced in [GSS92]. In the sequel, we will focus on the different logical systems designed for their ICC properties, and their adaptation to lambda calculus typing using Curry-Howard correspondence.

1.2 Linear Logic and Complexity bounded logics

Linear logic (LL) has been introduced around 1986 by Girard, see [Gir87]. Its source is the semantics of the system F type system. In the present work, we use ideas taken from Linear logic all along, for our purposes: typing, Optimal reduction and Implicit complexity. More precisely, we will use ideas taken from variants of Linear Logic designed to have good complexity properties. Indeed, following the Curry-Howard correspondence, Linear logic can be adapted to provide lambda calculus with a type system.

1.2.1 Linear Logic

Linear logic relies on a decomposition of usual logical connectives. In particular in our case, the formula A ⇒ B becomes !A ⊸ B. The connective ⊸ is the linear implication, and ! (pronounced “bang” or “of course”) a modality.

In this work, we use only a slight part of the wealth of Linear logic: we always stick to small fragments of it that serve well our purposes. We will present the various system in a sequent calculus style. Figure 1.8 gives the sequent calculus rules for second order (polymorphic) types. This feature can be accommodated in all the systems we will present. For brevity, we will omit them in our presentation.

The fragment MLL−◦

Figure 1.7 presents a restricted fragment of Linear logic in sequent calculus style: the Intuitionistic implicational multiplicative linear logic (MLL−◦). It is intuitionistic because the sequent contains only one conclusion. It is multiplicative because it does not mention all the additive rules of linear logic (that we will never use in this work). It is implicative because it is based on the arrow: “⇒”.

MLL−◦ is the core of our different variants of Linear logic. Formulas are on the grammar

$$F ::= α | F −◦ F, α ∈ B$$
It is the same as for simple types, except that ⇒ is turned into the linear implication: □. To understand the idea, two things are to be noticed: in the axiom rule, the context must be reduced to the hypothesis we use; in the cut and the □-l rules, the context of the conclusion is split in two in the premises. From a logical point of view, the effect of those features is to ensure a linear use of hypothesis: each hypothesis is used once and exactly once. It cannot be duplicated, nor erased. In Figure [6.9] we give Intuitionistic Proof-nets for MLL□. As one can see they are very close to the graphical representation of lambda terms we introduced. The only cut-elimination rule that we have is so a kind of graphical representation of the beta reduction. Mind that this works only for the restricted case of MLL.

From a typing point of view, this systems enforces a linear use of variables, variables appear once and only once. This is a set of terms often referred to as linear lambda terms. In the graphical representation, they correspond to lambda terms which have no contraction node (the triangle). The simple cut-elimination rule that we give corresponds to beta reduction only for linear lambda terms.

The fragment MELL−□

In MELL−□, we introduce the modality “!” . The language of formulas, or types, is now:

\[ F ::= \alpha | F \rightarrow F | !F, \alpha \in B \]

Intuitively, we want in MELL−□ to retrieve the contraction and weakening of minimal logic, but in a more controlled way. This is the reason for the presence of a modality. The weakening and contraction rules will be allowed only on formulas marked with the “!” (called banged formulas), and “!” modalities will be introduced following certain rules.

Figure [6.10] gives the structural rules and the exponential rules of MELL−□. As announced, structural rules are restricted to banged formulas. We have three rules to manipulate modalities: box, dereliction and digging. The box rule corresponds to enclosing a previous derivation in a box (hence its name) and marking all its conclusion with a “!”, and it expresses the fact that the whole proof unit is now duplicable or erasable. Dereliction corresponds to the principle !A → A; digging to the principle !A → !!A. The number of boxes in which a node of a Proof-net (or a rule application of a derivation) is enclosed will be called its depth, and the depth of a Proof-net will be the maximal depth of its nodes. Figure [6.11] give the new Intuitionistic Proof-nets constructions one needs to represent MELL−□ and some of the additional cut-elimination rules. The cut-elimination between two boxes results in their merging;
1.2 LINEAR LOGIC AND COMPLEXITY BOUNDED LOGICS

Figure 1.9: MLL\(_{-}\) Proof-nets
the cut-elimination between a box and a weakening node results in the erasure of the box and the propagation of the weakening node. We didn’t show the last cut-elimination steps, Box-Digging and Box-Dereliction, because they will not be needed in the sequel. When reduced, the effect of a Box-Digging step is to remove a box, and the effect of a Box-Dereliction step is to add another box.

This presentation of *MELL* is not the historical one, nor the most common, but it is the one that corresponds best to our use of Linear logic. It is a slight variant of the presentation given in chapter 4 of [AG99].

In the scope of implicit complexity, Linear logic will be a tool of choice because of this increased control on structural rules. Indeed, we can control complexity by controlling duplication, or contraction, and Linear logic will allow us to control contraction by controlling modalities placement.

### 1.2.2 Complexity bounded logics

Now, we move to real complexity bounded logics. This field has been launched by a seminal work of Jean-Yves Girard in the mid-90’s, [Gir98]. Since then, several different complexity bounded logics have been designed that we will call in general *light logics*, even though *Light linear logic* is one particular system.

**Elementary Linear Logic**

Elementary linear logic (ELL) has been introduced in [Gir98] and further on studied in [Ped96, DJ03]. The exponential sequent calculus rules are given in figure 1.12. We can see it as a restriction of our presentation of full Linear logic: we simply remove dereliction and digging rules. This makes the exponential discipline more restrictive than in the general case. The main effect of this restriction is the so-called stratification: derelictions and diggings allow the depth of a box to vary along the reduction; their removal enforces that this depth will remain constant along the cut-elimination procedure. ELL Intuitionistic Proof-nets are obtained as a subset of *MELL* , Proof-nets construction rules and cut-elimination. One have to exclude dereliction and digging constructions given in the figure 1.11 to obtain the ELL Intuitionistic Proof-nets and their cut-elimination rules.

Elementary linear logic has the desired properties. One can exhibit an elementary bound on the reduction of a Proof-nets family, provided their depth remain constants. So, we say that ELL is *correct for elementary time*. Secondly, one can encode elementary functions in ELL Proof-nets, so we say that ELL is *complete for elementary time*. Mind that here we talk of functions in the sense of set theory, and not of *algorithms* or *programs* which are the different implementations of a function.

In [Asp98, AR02], authors have developed a simplification of ELL, called Elementary affine logic (EAL). The simplification consists in allowing *unrestricted weakening*: we don’t impose
Figure 1.11: MELL_{\omega} exponential Proof-nets construction rules
anymore that erased formulas carry a bang type. So the weakening rule becomes:

\[
\Gamma, x : B \vdash t : A
\]

This modification preserves the termination bounds (once the cut-elimination procedure is correctly adapted) and greatly simplifies the logic. In particular, subsequent efforts to develop light logics as programming languages or type systems all use affine logics rather than light logics. Note that the original intuition for this simplification came from Optimal reduction.

**Theorem 1.12 (correctness, [DJ03])**. Given a Proof-net of ELL \(P\) of depth \(d\), its reduction by the cut-elimination procedure takes at most \(2 \cdot 2^d\) steps, where \(d\) is the height of the exponential tower.

**Theorem 1.13 (completeness, [DJ03])**. Any elementary function from integers to integers can be encoded in a fixed-depth ELL Proof-net.

The type inference problem for ELL has been first tackled by Coppola and Martini in [CM01, CM06]. Then Baillot and Terui developed a more efficient algorithm in [BT05]. The principle that is used is to decorate a simple types type derivation into an ELL derivation by generating and solving linear integer constraints.

**Theorem 1.14 ([BT05])**. Given a simple type derivation, ELL type decoration is decidable in time polynomial w.r.t the size of the derivation.

**Soft Linear Logic**

Soft linear logic (SLL) is derived from Linear logic and designed to capture the polynomial time complexity class. It has been introduced by Lafont in [Laf04]. The exponential sequent calculus rules are given in Figure 1.13. The difference with ELL is that the contraction rule is replaced with a less general multiplexor rule. The multiplexor removes a modality on the type of the hypothesis it duplicates, and is n-ary: so it acts as weakening, when \(n = 0\), as dereliction when \(n = 1\), and as a less general duplication when \(n > 1\).

Figure 1.14 gives the construction of Multiplexor to obtain Intuitionistic Soft Proof-nets and the associated cut-elimination rule. Connected with a box, the multiplexor “opens” it before duplicating it.

While Elementary linear logic is based on stratification to enjoy complexity properties, Soft linear logic is rather based on a constant decrease of depth: each time a box is used (duplicated) then it is opened, so it cannot go through more duplications than its initial depth.
A restricted version of the affine variant of sll, called Soft type assignment (sta) has been developed in [GRDR07, GMRDR08].

**Theorem 1.15** ([GRDR07]). Let \( t \) be a lambda term typable in sta with a type derivation of depth \( d \), \( t \) reduces to its normal form in at most \( |t|^{d+1} \) beta reduction steps.

**Theorem 1.16** ([GRDR07]). Any polynomial time function can be represented in sta.

**Light Linear Logic**

Light linear logic has been introduced in [Gir98], and later refined by [Asp98, Rov99, AR02]. As sll, it is a logic of polynomial time. The language of its formulas is:

\[
F ::= \alpha \mid F \multimap F \mid !F \mid \#F, \alpha \in B
\]

There is one more modality than in the other systems, the “\( \# \)”.

Exponential sequent calculus rules are given in Figure 1.15. Contraction and weakening are as in ell, however, the boxes handling is original. As we have two modalities, we naturally have two different boxes. For the \( ! \)-box, the number of hypothesis must now be equal to zero or one: without getting into the technical details, let us just mention that it is this feature...
that ensures a polynomial termination. In figure 1.11 diagram (8) shows a Box-Cntr cut-elimination step in MELL. One can see that, if the number of hypothesis of the box is less or equal to one, the contraction node itself is not duplicated in the resulting Proof-net.

On the contrary, the $\Box$-box is rather there to recover a bit of expressiveness: without this rule, LLL would not be expressive enough. The interest of this rule regarding expressiveness is to allow a restricted dereliction, since an arbitrary part of the context can be “unbanged” when it is applied.

**Theorem 1.17** ([Rov99]). A Proof-net of LLL $P$ of depth $d$ can be reduced to its normal form in $|P|^{2^d}$ steps of cut-elimination.

**Theorem 1.18** ([Rov99]). Any polynomial function can be represented in LLL.

As for LLL, we have an affine variant of LLL, Light affine logic, in which weakening is not restricted to banged formulas. Type inference for LAL has been studied in [Bai02, Bai04]:

**Theorem 1.19** ([Bai04]). Type inference in LAL is decidable.

However, the type inference algorithm given in [Bai04] is of complexity at least exponential and is restricted to the propositional fragment of LAL, that is to say, without second-order types.

**Dual Light Affine Logic**

Dual light affine logic (DLAL) has been developed by Baillot and Terui in [BT04]. The base motivation for this work is that LAL, used as a type system, is not quite satisfactory: the complexity bound we obtain works for Proof-nets cut-elimination, but not for lambda calculus beta reduction. Furthermore, we don’t know of a polynomial algorithm for type inference, or for LAL type decoration, if one starts from a simply typed, or system F typed lambda term.

Chapter 2 is devoted to type inference for DLAL, so we won’t give many details here. But the main property of DLAL is:

**Theorem 1.20** ([BT04, BT08]). Let $t$ be a lambda term which has a derivation of depth $d$ in DLAL, then $t$ reduces to its normal form in $|t|^{2^d}$ beta reduction steps.

### 1.3 Optimal Reduction and Geometry of Interaction

The main reference for this part is [AG99]. In this section, after typing, we get more interested in lambda calculus reduction.

The beta reduction we defined in the first section yields a question: if a term contains several redexes, how do we choose the one to reduce first? Answering this question is called defining a reduction strategy. Quite naturally, we are led to consider the different possible strategies to determine which one is the best, and adopt it. Unfortunately, there is no such strategy. For instance, let us take the lambda term $E = (\lambda x. x \lambda v. v x \cdots x) \lambda y. (\lambda x. x \cdots x)(y) z$. In the course of reducing it, any reduction strategy will duplicate work, that is to say, will duplicate a redex. The goal of Optimal reduction is to define an algorithm which does not duplicate any redex.

Jean-Jacques Lévy developed a lambda calculus theory, labelled lambda calculus, in which reduction is done without any useless duplication, see [Lév80]. It is based on redex families: two redexes belong to the same family if they are copies of the same initial redex. He then conjectured that it was possible to implement this shared beta reduction in an elementary operation.
1.3.1 Lamping’s algorithm on Sharing Graphs

Ten years later, John Lamping and Vinod Kathail independently discovered two different algorithms in [Lam90] and [Kat90] which did implement Optimal reduction. Lamping’s solution was the most widely studied one. It presented an involved algorithm on sharing graphs, later rephrased and simplified in [GAL92] using Linear logic intuitions and more precisely Geometry of Interaction, previously developed in [Gir89].

It is a graph rewriting algorithm. The graphs on which it acts are called sharing graphs. They are composed of the nodes we defined for Intuitionistic Proof-nets, without boxes, but each node carries an integer index. We will not give the rewriting rules here, one can think of them as a local reformulation of the cut-elimination rules of the $MELL$ Proof-nets. In particular, the interaction between application and lambda nodes correspond exactly to the cut-elimination rule $\Rightarrow l \Rightarrow r$ for Intuitionistic Proof-nets given in Figure 1.9, where $\Rightarrow l$ is the application node, and $\Rightarrow r$ the lambda node.

However, in Proof-nets, the duplication operation is performed box-wise: a single step duplicates a whole box. In Optimal reduction, there isn’t anymore box, and the duplication is done node-wise. The difficulty is to determine where to stop the duplication, since we do not have the box delimitation anymore. To do so, one encloses the part of the Sharing graph to be duplicated between the contraction nodes, called fans in Optimal reduction. The lambda-application rewriting rule and the duplication rules are called the abstract part of Lamping’s algorithm.

When two fans face each other, it can mean: (1) that their duplication job is over, or (2) that one of the two must be duplicated by the other one. The oracle is the part of the algorithm that maintains the informations needed to decide whether we are in the situation (1) or in the situation (2) when two fans meet. This is done with the indexes that all the nodes carry. There are two nodes in the oracle: the croissant and the bracket. Their effect is to modify the indexes of the nodes, and thus ensure the correctness of the algorithm. The bracket increases the depth of a node, while the croissant decreases it.

In Linear logic Proof-nets those actions correspond to the dereliction and the digging principles. So the oracle can be seen as Linear logic dereliction and digging nodes that would act node-wise instead of box-wise along the reduction, as does the fan node.

In particular, as first noticed by Asperti in [Asp98] and recently more formally proved in [BCDL07], this is the reason why lambda terms typable in ELL or EAL can be correctly reduced with Lamping’s abstract algorithm. Indeed, those type systems do not enjoy the dereliction and digging principles.

1.3.2 Semantics of execution: Geometry of Interaction

Geometry of Interaction (GoI) has been developed by Jean-Yves Girard in a series of articles [Gir89, Gir90]. It has been then applied to Optimal reduction in [GAL92, DL06].

In the present work, we are interested in GoI regarding Optimal reduction, and thus in the flavours of GoI developed in [Reg92, GAL92, DR93, DR95].

1.3.3 Complexity of Optimal Reduction

The normalisation of lambda terms by Optimal reduction is done in three different parts: first the initial translation phase; then the Optimal reduction itself, and finally the read-back. The initial translation is the process of turning a pure lambda term into a Sharing graph. On the contrary the read-back is the process of turning a normal form sharing graph into a normal form lambda term.

A question that naturally arises is the complexity of this whole process. When he formulated his theory in the 1970’s, Lévy’s conjecture was that the number of redex families reduced during
the normalisation of a lambda term was a good measure of the cost of reducing a lambda term. When, some years later, Lamping published his algorithm, he conjectured that the cost of the oracle part was polynomial w.r.t to the cost of the abstract algorithm.

However, later works have proven both conjectures to be false. Indeed, Mairson and Lawall showed in [LM96] that any known initial translations procedures could lead to an exponential number of oracle steps w.r.t abstract algorithm steps, and Asperti showed in [Asp96] that the number of parallel beta steps (i.e. of family reductions) was not a good measure of the cost of the abstract algorithm.
Chapter 2

Type Inference in Dual light affine logic

2.1 Introduction

Background Several works have studied programming languages with intrinsic computational complexity properties. This line of research, Implicit computational complexity (icc), is motivated both by the perspective of automated complexity analysis, and by foundational goals, in particular to give natural characterisations of complexity classes, like P-time.

To use such ICC systems for programming purpose it is natural to wish to automatize the verification of the criterion. This way the user could stick to a simple programming language and the compiler would check whether the program satisfies the criterion, in which case a complexity property would be guaranteed.

In the present work we consider the approach of Light linear logic (LLL) (Gir98). The original formulation of LLL by Girard was quite complicated, but a first simplification was given by Asperti with Light affine logic (LAL) (AR02). There is a forgetful map to system F terms (polymorphic types) obtained by erasing some information (modalities) in types; if an LAL typed term \( t \) is mapped to an F-typed term \( M \) we also say that \( t \) is a decoration of \( M \).

So an LAL program can be understood as a system F program, together with a typing guarantee that it can be evaluated in polynomial time once that program is written and evaluated in the right syntax (see below). As system F is a reference system for the study of polymorphically typed functional languages and has been extensively studied, this seems to offer a solid basis to LAL.

However LAL itself is still difficult to handle and following the previous idea for the application of ICC methods, we would prefer to use plain lambda-calculus as a front-end language, without having to worry about the handling of modalities, and instead to delegate the LAL typing part to a type inference engine.

Motivations Type inference in LAL has two main drawbacks:
1. LAL typed terms need to be evaluated with a specific graph syntax, Proof-nets, in order to satisfy the polynomial bound, and plain beta reduction can lead to exponential blow-up.

2. Type inference seems too complex to be done in polynomial time.

To address those issues, in a previous work ([BT04], extended version [BT08]), Baillot and Terui defined a subsystem of LAL, called Dual light affine logic (DLAL). It is defined with both linear and non-linear function types. It is complete for P-Time just as LAL and its main advantage is that it is also P-Time sound w.r.t. beta reduction: a DLAL term admits a bound on the length of all its beta reduction sequences. Hence DLAL stands as a reasonable substitute for plain LAL for typing issues.

The contribution of the present work is to define an efficient algorithm to decide if a system F term can be decorated in a DLAL typed term. This was actually one of the original motivations for defining DLAL. We show here that decoration can be performed in polynomial time. This is obtained by taking advantage of intuitions coming from Proof-nets, but it is presented in a standard form with a first phase consisting in generating constraints expressing typability and a second phase for constraints solving.

The complete algorithm is already implemented in ML, in a way that follows closely the specification given in the article. It is modular and usable with any linear constraints solver. The code is commented, and available for public download (Section 2.7). With this program one might thus write terms in system F and verify if they are P-Time and obtain a time upper bound. It should in particular be useful to study further properties of DLAL and to experiment with reasonable size programs.

Outline In a first section, we recall from [BT04] the main properties of DLAL; to proceed towards type inference, we introduce an intermediary syntax, more informative, and a correct and complete typability criterion on this syntax; then we show how to parametrise system F terms and generate constraints for the type inference; we give the correctness and completeness result, and a constraints resolution procedure. The final sections are devoted to the problem of type inference for specific data-types, and to the test of the type inference program with large lambda terms.

The present chapter is a slightly modified version of the article [ABT07], which is a long version of [ABT06].

2.2 From system F to dlal

The language $L_F$ of system F types is given by:

$$T, U ::= \alpha \mid T \rightarrow U \mid \forall \alpha . T .$$

We assume that a countable set of term variables $x^T, y^T, z^T, \ldots$ is given for each type $T$. The terms of system F are built as follows (here we write $M^T$ to indicate that the term $M$ has type $T$):

$$x^T, (\lambda x^T . M^U)^{T \rightarrow U}, ((M^{T \rightarrow U})^N)^{T \rightarrow U}, (\Lambda \alpha . M^U)^{\forall \alpha . U}, ((M^{\forall \alpha . U})^{T \rightarrow U})^{T \rightarrow U / [T / \alpha]},$$

with the proviso that when building a term $\Lambda \alpha . M$, $\alpha$ does not occur free in the types of free term variables of $M$ (the eigenvariable condition). The set of free variables of $M$ is denoted $FV(M)$.

It is well known that there is no sensible resource bound (i.e. time/space) on the execution of system F terms in general. On the other hand, we are practically interested in those terms which can be executed in polynomial time. However the class $P$ of such terms is neither recursively enumerable nor co-recursively enumerable. This can be verified for instance in the
Figure 2.1: Typing system F terms in DLAL

following way, by reduction of the problem of solvability of Diophantine equations. For each
Diophantine equation \( P(x) = 0 \), build a lambda term \( M_P \) such that, when a binary word \( w \) is
given, \( M_P(w) \) returns \( \epsilon \) if \( P(x) = 0 \) has an integer solution \( n \) with \( -|w| \leq n \leq |w| \), and returns
a word of length \( 2^{|w|} \) otherwise. Then \( M_P \in P \) iff \( P(x) = 0 \) has an integer solution. There is
also a complementary reduction, establishing our claim. Actually a stronger result is shown in
[BMM05]: the class \( \mathcal{P} \) is \( \Sigma_\infty \)-complete.

So we are naturally led to the study of sufficiently large subclasses of \( \mathcal{P} \). The system DLAL
gives such a class in a purely type-theoretic way.

The language \( \mathcal{L}_{DLAL} \) of DLAL types is given by:

\[
A, B ::= \alpha \mid A \to B \mid A \mid \forall \alpha.A.
\]

We note \( \forall^0 \alpha.A = A \) and \( \forall^k+1 \alpha.A = \forall^k (\forall \alpha.A) \). The erasure map \((\cdot)^-\) from \( \mathcal{L}_{DLAL} \) to \( \mathcal{L}_F \) is defined by:

\[
(\forall \alpha.A)^- = A^- , \quad (A \to B)^- = (A \to B)^- = A^- \to B^- ,
\]

and \((\cdot)^-\) commutes with the other connectives. We say \( A \in \mathcal{L}_{DLAL} \) is a decoration of \( T \in \mathcal{L}_F \)
if \( A^- = T \).

A declaration is a pair of the form \( x^T : B \) with \( B^- = T \). It is often written as \( x : B \) for
simplicity. A judgement is of the form \( \Gamma \vdash M : A \), where \( M \) is a system F term, \( A \in \mathcal{L}_{DLAL} \)
and \( \Gamma \) and \( \Delta \) are disjoint sets of declarations. The intuition is that the (free) variables in \( \Gamma \)
are duplicable (non-linear), while the ones in \( \Delta \) are not (they are linear). When \( \Delta \) consists of
\( x_1 : A_1, \ldots, x_n : A_n \), \( \forall \Delta \) denotes \( x_1 : \forall x_1.A_1, \ldots, x_n : \forall x_n.A_n \). The type assignment rules are given on
Figure 2.2.1 Here, we assume that the substitution \( M/N/x \) used in \((\cdot)^e\) is capture-free. Namely,
no free type variable \( \alpha \) occurring in \( N \) is bound in \( M[N/x] \). We write \( \Gamma ; \Delta \vdash_{DLAL} M : A \) if
the judgement \( \Gamma ; \Delta \vdash M : A \) is derivable.

Examples of concrete programs typable in DLAL are given in Section 2.7.

Recall that binary words, in \( \{0,1\}^* \), can be given in system F the type:

\[
W_F = \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \alpha) .
\]
A corresponding type in DLAL, containing the same terms, is given by:

\[ W_{DLAL} = \forall \alpha. (\alpha \rightarrow \alpha) \Rightarrow (\alpha \rightarrow \alpha) \Rightarrow \Lw (\alpha \rightarrow \alpha) \, . \]

The depth \( d(A) \) of a DLAL type \( A \) is defined by:

\[
\begin{align*}
  d(\alpha) &= 0, \\
  d(A \rightarrow B) &= \max(d(A), d(B)), \\
  d(A \Rightarrow B) &= \max(d(A) + 1, d(B)).
\end{align*}
\]

A type \( A \) is said to be \( \Pi_1 \) if it does not contain a negative occurrence of \( \forall \); like for instance \( W_{DLAL} \).

The fundamental properties of DLAL are the following [BT04]:

**Theorem 2.1.**

1. For every function \( f : \{0,1\}^* \rightarrow \{0,1\}^* \) in DTIME[\( n^k \)], there exists a closed term \( M \) of type \( W_{DLAL} \rightarrow \Lw W_{DLAL} \) with \( d = O(\log k) \) representing \( f \).

2. Let \( M \) be a closed term of system F that has a \( \Pi_1 \) type \( A \) in DLAL. Then \( M \) can be normalised in \( O(|M|^2) \) steps by \( \beta \)-reduction, where \( d = d(A) \) and \( |M| \) is the structural size of \( M \). Moreover, the size of any intermediary term occurring in normalisation is also bounded by \( O(|M|^2) \).

Although DLAL does not capture all P-Time algorithms \( P \), the result 1 guarantees that DLAL is at least expressive enough to represent all P-Time functions. In fact, DLAL is as expressive as LAL even at the level of algorithms, because there exists a generic translation from LAL to DLAL given by:

\[
(!A)\circ = \forall \alpha.((A\circ \Rightarrow \alpha) \rightarrow \alpha), \quad (\cdot)\circ \text{ commutes with other connectives than !.}
\]

See [BT08] for details.

The result 2 on the other hand implies that if we ignore the embedded types occurring in \( M \), the normal form of \( M \) can be computed in polynomial time (by ordinary \( \beta \)-reduction; that is the difference from LAL).

Now, let \( M^{WF} \rightarrow WF \) be a system F typed term and suppose that we know that it has a DLAL type \( W_{DLAL} \rightarrow \Lw W_{DLAL} \) for some \( d \geq 0 \). Then, by the consequence of the above theorem, we know that the term \( M \) is P-Time. In fact, given a binary word \( w \in \{0,1\}^* \), consider its Church coding \( w \) of type \( W_{DLAL} \). Then we have that \( (M)w \) has type \( \Lw W_{DLAL} \), and can thus be evaluated in \( O(|w|^{2d+1}) \) steps. Thus by assigning a DLAL type to a given system F term, one can statically verify a polynomial time bound for its execution.

In order to use DLAL for resource verification of system F terms, we address the following problem:

**Problem 2.2 (DLAL typing).** Given a closed term \( M^T \) of system F, determine if there is a decoration \( A \) of \( T \) such that \( \vdash_{DLAL} M : A \).

(Here the closeness assumption is only for readability.)

In the sequel, we show that there is a polynomial time algorithm for solving the DLAL typing problem.
2.3 Localisation of dlal type inference

To solve the dlal typing problem, the main obstacle is that the typing rules of dlal are not syntax-directed. In particular, the rule (§ i) does not correspond to any constructs of system F terms, and the rule (§ e) involves term substitution. These features make local reasoning on types impossible.

To overcome the difficulty, we introduce (following [AR02]) an intermediary syntax which is more informative than system F terms, but not more informative than dlal derivations themselves (in 2.3.1). In particular, it has explicit constructs for (§ i). In addition, we replace the global typing rules of dlal (which involve substitution) with some local typing rules and a set of conditions (in 2.3.3 and 2.3.4). We then show that our Local typing rules and conditions exactly characterise system F terms typable in dlal (in 2.3.5).

2.3.1 Pseudo-terms

We begin with introducing an intermediary syntax, which consists of dlal* types and pseudo-terms.

First we decompose $A \Rightarrow B$ into $!A \rightarrow B$. The language $\mathcal{L}_{DLAL*}$ of dlal* types is given by:

$$
A ::= \alpha | D \rightarrow A | \forall \alpha. A | \#A, \\
D ::= A | !A.
$$

There is a natural map $(\cdot)^*$ from $\mathcal{L}_{DLAL*}$ to $\mathcal{L}_{DLAL*}$, such that $(A \Rightarrow B)^* = !A^* \rightarrow B^*$ and commutes with the other operations. The erasure map $(\cdot)^-$ from $\mathcal{L}_{DLAL*}$ to $\mathcal{L}_F$ can be defined as before. A dlal* type is called a bang type if it is of the form $!A$, and otherwise called a linear type. In the sequel, $A, B, C$ stand for linear types, and $D$ for either bang or linear types.

We assume there is a countable set of term variables $x^D, y^D, z^D, \ldots$ for each $D \in \mathcal{L}_{DLAL*}$. The pseudo-terms are defined by the following grammar:

$$
t, u ::= x^D | \lambda x^D.t | (t)u | \Lambda A.t | (t)A | \#t | \#\#t,
$$

where $A$ is a linear type and $D$ is an arbitrary one. The idea is that $\#$ corresponds to the main door of a §-box (or a !-box) in proof-nets ([GH97] [AR02]) while $\#\#$ corresponds to auxiliary doors. But note that there is no information in the pseudo-terms to link occurrences of $\#$ and $\#\#$ corresponding to the same box, nor distinction between $\#$-boxes and $\!$-boxes.

There is a natural erasure map from pseudo-terms to system F terms, which we will also denote by $(\cdot)^-$, consisting in removing all occurrences of $\#$, $\#\#$, replacing $x^D$ with $x^{D^*}$ and $(t)A$ with $(t)A^*$. When $t^- = M$, $t$ is called a decoration of $M$.

Let $t$ be a pseudo-term and $u$ be a subterm of $t$. We say that $u$ is a door-extreme subterm of $t$ if the following holds: if $u$ is of the form $u = \#u'$ or $u = \#\#u'$ then $\#u$ and $\#\#u$ are not subterms of $t$.

As an example consider $t = (x \#\#y)$. Its door-extreme subterms are $\{t, x, \#\#y, y\}$, and $\#\#y$ is a subterm of $t$ but not a door-extreme subterm.

For our purpose, it is sufficient to consider the class of regular pseudo-terms, given by:

$$
u ::= x^D | \lambda x^D.t | (t)u | \Lambda A.t | (t)A, \\
t ::= \#^m u,
$$

where $m$ is an arbitrary value in $\mathbb{Z}$ and $\#^m t$ denotes $\# \cdots \# t$ ($m$ times) if $m \geq 0$, and $\# \cdots \#\# t$ ($-m$ times) if $m < 0$.

In other words, a pseudo-term is regular if and only if it does not contain any subterm of the form $\#\#u$ or $\#\#\#u$. 

2.3.2 Pseudo-terms and proof-nets

In this section we illustrate the links between pseudo-terms and proof-nets. It is independent of the sequel and can be skipped without problem.

The translation \( (\cdot)^* \) from DLAL to LAL gives a mapping on derivations; therefore a DLAL type derivation corresponds to an LAL proof and thus to a proof-net \([\text{AR02}]\). To facilitate the reading we will use here a ‘syntax-tree like’ representation for intuitionistic LAL proof-nets.

As an example consider the following term:

\[
M = (\lambda f.((f)\; x))((\lambda h.h)\; g)
\]

It can be given the typing \( x : \; \beta \vdash M : \; \beta \), with the derivation of Fig. 2.2. The corresponding (intuitionistic) proof-net is given on Fig. 2.4. For readers more familiar with the classical representation of proof-nets (in the style of \( [\text{AR02}] \)), the corresponding representation is given on Fig. 2.3.

\[
\begin{array}{c}
f_2 : \alpha \rightarrow \alpha \vdash f_2 : \alpha \rightarrow \alpha \\
; f_2 : \alpha \rightarrow \alpha, x : \alpha \vdash (f_2)\; x : \alpha \\
; x : \alpha \vdash x : \alpha
\end{array}
\]

\[
\begin{array}{c}
f_3 : \beta \rightarrow \beta, \; e : \; \beta \vdash (f_3)\; ((f_3)\; x) : \; \beta \\
; f \vdash \beta : \beta, \; x : \; \beta \vdash (f(x))\; x : \; \beta \\
; x : \; \beta \vdash (\lambda f.((f)\; x)) \; f \vdash \beta
\end{array}
\]

\[
\begin{array}{c}
; \beta : \beta \vdash \beta \\
; g : \; \beta \vdash g : \; \beta
\end{array}
\]

where \( \beta = \alpha \rightarrow \alpha \).

Figure 2.2: Example: DLAL derivation for \( M \).

The pseudo-term corresponding to the previous derivation is:

\[
t = (\lambda f.(\lambda x.\; f(\; x))((\lambda h.h)\; g))
\]

It is represented graphically on Fig. 2.5 to \( \lambda \) and \( \tilde{\lambda} \) correspond respectively opening and closing doors.

In a proof-net, a box can be thought of as an opening door connected to a certain number (possibly none) of closing doors. If in the proof-net of Fig. 2.4 we disconnect opening doors from closing doors we get the graph of Fig. 2.5 corresponding to the pseudo-term.

Our method for type inference relies on a procedure for deciding if a pseudo-term comes from a DLAL derivation. This essentially corresponds to deciding if a pseudo-term corresponds to a proof-net, that is to say in particular deciding whether opening and closing doors can be matched in such a way to yield a correct distribution of boxes.
2.3 LOCALISATION OF DLAL TYPE INFERENE

2.3.3 Local typing condition

We now describe a way to assign types to pseudo-terms in a locally compatible way. A delicate point in DLAL is that it is sometimes natural to associate two types to one variable \( x \). For instance, we have \( x : A \vdash_{DLAL} x : \§ A \) in DLAL, and this can be read as \( x : ! A \vdash x : \§ A \) in terms of DLAL\( ^\star \) types. We thus distinguish between the input types, which are inherent to variables, and the output types, which are inductively assigned to all pseudo-terms. The condition (i) below is concerned with the output types. In the sequel, \( D^\circ \) denotes \( \§ A \) if \( D \) is of the form \( ! A \), and otherwise denotes \( D \) itself.

A pseudo-term \( t \) satisfies the Local typing condition if the following holds:

(i) one can inductively assign a linear type to each subterm of \( t \) in the following way (here the notation \( t : A \) indicates that \( t \) has the output type \( A \)):

\[
\begin{align*}
\lambda x : D, t : D &\vdash t : B \\
\lambda x : D, t : D &\vdash t : ! B \\
\lambda x, t : A &\vdash t : ! A \\
\lambda x.t : \forall \alpha. A &\vdash (t)B : A[B/\alpha]
\end{align*}
\]

(ii) when a variable \( x \) occurs more than once in \( t \), it is typed as \( x^! A \),
(iii) \( t \) satisfies the eigenvariable condition. Namely, for any subterm of the form \( \Lambda \alpha. u \) and any free term variable \( x^D \) in \( u \), \( \alpha \) does not occur free in \( D \).

We also say that \( t \) is locally typed.

The Local typing rules are syntax-directed, and assign a unique type to each pseudo-term whenever possible. Notice that there is a type mismatch between \( D \) and \( A \) in the application rule when \( D \) is a bang type. This mismatch will be settled by the \emph{Bang condition} below.

### 2.3.4 Boxing conditions

It is clear that local typability is not a sufficient condition for typability in DLAL, as it does not ensure that doors \( \$, \$ \) are well placed so that boxes can be built around them. Moreover, it does not distinguish \( \$ \)- and \(!\)-boxes. We therefore impose additional conditions on locally typed pseudo-terms.

We consider words over the language \( L = \{\$, \$\}^* \) and \( \leq \) the prefix ordering. If \( t \) is a pseudo-term and \( u \) is an occurrence of subterm in \( t \), let doors\((t, u)\) be the word inductively defined as follows. If \( t = u \), let doors\((t, u) = \epsilon \). Otherwise:

\[
\begin{align*}
\text{doors}(\$t, u) &= \$ :: (\text{doors}(t, u)), \\
\text{doors}(\$t, u) &= \$ :: (\text{doors}(t, u)), \\
\text{doors}(\lambda y^D.t_1, u) &= \text{doors}(\Lambda \alpha.t_1, u) = \text{doors}(t_1 A, u) = \text{doors}(t_1, u), \\
\text{doors}(t_1 t_2, u) &= \text{doors}(t_1, u), \text{ where } t_i \text{ is the subterm containing } u.
\end{align*}
\]
That is to say, $\text{doors}(t, u)$ collects the modal symbols $\$\!, \$\!$ occurring on the path from the root to the node $u$ in the term tree of $t$. We define a map $s : L \to Z$ by:

$$
s(\varepsilon) = 0,
\quad s(\$ : l) = 1 + s(l),
\quad s(\$ :: l) = -1 + s(l).
$$

A word $l \in L$ is weakly well-bracketed if $\forall l' \leq l$, $s(l') \geq 0$, and is well-bracketed if this condition holds and moreover $s(l) = 0$: think of $\$\!$ and $\$\!$ resp. as opening and closing brackets.

**Bracketing condition.** Let $t$ be a pseudo-term. We say that $t$ satisfies the Bracketing condition if:

(i) for any occurrence of free variable $x$ in $t$, $\text{doors}(t, x)$ is well-bracketed;

(ii) for any occurrence of an abstraction subterm $\lambda x. v$ of $t$:

(ii.a) $\text{doors}(t, \lambda x. v)$ is weakly well-bracketed, and

(ii.b) for any occurrence of $x$ in $v$, $\text{doors}(v, x)$ is well-bracketed.

This condition is sufficient to rule out the canonical morphisms for dereliction and digging, which are not valid in DLAL (nor in EAL):

$$
\lambda x^{\$ A}. \$ \!: \$ A \to A, \quad \lambda x^{\$ A}. \$ x : \$ A \to \$\! \$ A.
$$

Since $\text{doors}(\$ x, x) = \$\!$ and $\text{doors}(\$ x, x) = \$, they do not satisfy the Bracketing condition (ii.b).

**Remark 2.3.** On the graph representation of pseudo-terms, conditions (i), (ii.a) and (ii.b) can be visualised as conditions of bracketing holding on certain paths of the graph: for instance condition (ii.b) means that any (top-down) path from a $\lambda x$ binder to an edge corresponding to an occurrence of $x$ is well-bracketed (considering the opening and closing doors). For instance the pseudo-term graph of Fig. 2.4 satisfies these conditions; we show on the Figure two paths $\gamma_1, \gamma_2$ that have to be well-bracketed according to (ii.b).

**Bang condition.** Let $t$ be a locally typed pseudo-term. A subterm $u$ is called a bang subterm of $t$ if it occurs as $(t')u$ in $t$ for some $t' : ! A \to B$. We say that $t$ satisfies the Bang condition if for any bang subterm $u$ of $t$,

(i) $u$ contains at most one occurrence of free variable $x^{C}$, and it has a bang type $! C$.

(ii) for any subterm $v$ of $u$ such that $v \neq u$ and $v \neq x$, $s(\text{doors}(u, v)) \geq 1$.

Observe that this condition is sufficient to rule out the canonical morphisms for monoidalness $! A \otimes ! B \to ! (A \otimes B)$ and $\$ A \to ! A$ which are not valid in LAL (the following terms and types are slightly more complicated since $L_{\text{DLAL}}\!$ does not explicitly contain a type of the form $A \to ! B$):

$$
\lambda x^{!(A \to ! B)}. \lambda y^{! B \to C}. \lambda z^{! A}. (y)(x)(z), \quad \lambda x^{\$ A}. \lambda y^{! A \to B}. (z)(x).
$$

In the first pseudo-term, the bang subterm $\$ (\$ x)\$ z)$ contains more than one free variable. In the second pseudo-term, the bang subterm $\$ (\$ x)$ has a free variable $x$ with a linear type. Hence they both violate the Bang condition (i).

**Remark 2.4.** The intuition behind the Bang condition might be easier to understand on the graph representation of pseudo-terms. The idea is that in a proof-net, the argument of a non-linear application should be enclosed in a box, with at most one free variable, as in the Example of Fig. 2.4. This is enforced on the pseudo-term by Bang conditions (i) and (ii). Condition (ii) indeed forces the root of the argument of the application to start with an opening door, and this opening door can only be matched by a closing door on the edge corresponding to the free variable $x$. 
**A-Scope condition.** The previous conditions, Bracketing and Bang, would be enough to deal with boxes in the propositional fragment of DLAL. For handling second-order quantification though, we need a further condition to take into account the sequentiality enforced by the quantifiers. For instance consider the following two formulas (the second one is known as Barcan’s formula):

\[(1) \forall \alpha. A \rightarrow \forall \alpha. \exists A, \quad (2) \forall \alpha. \exists A \rightarrow \exists \forall \alpha. A .\]

Assuming \(\alpha\) occurs free in \(A\), formula (1) is provable while (2) is not. Observe that we can build the following pseudo-terms which are locally typed and have respectively type (1) and (2):

\[t_1 = \lambda x. \exists \forall \alpha. A. \Lambda \alpha. \exists ((\overline{\exists} x)\alpha), \quad t_2 = \lambda x. \exists \forall \alpha. A. \exists \Lambda \alpha. \exists ((x)\alpha) .\]

Both pseudo-terms satisfy the previous conditions, but \(t_2\) does not correspond to a DLAL derivation.

Let \(u\) be a locally typed pseudo-term. We say that \(u\) depends on \(\alpha\) if the type of \(u\) contains a free variable \(\alpha\). We say that a locally typed pseudo-term \(t\) satisfies the A-scope condition if: for any subterm \(\Lambda \alpha. u\) of \(t\) and for any subterm \(v\) of \(u\) that depends on \(\alpha\), doors \((u, v)\) is weakly well-bracketed.

Coming back to our example: \(t_1\) satisfies the A-scope condition, but \(t_2\) does not, because \((x)\alpha\) depends on \(\alpha\) and nevertheless doors \((\exists ((x)\alpha), (x)\alpha) = \frac{1}{2}\) is not weakly well-bracketed.

We now give a reformulation of the Bang condition (ii), which will be useful later:

**Lemma 2.5.** Assume that \(t\) is a locally typed regular pseudo-term that satisfies the Bracketing condition and that \(u\) is a bang subterm of \(t\) that satisfies the Bang condition (i). If \(u\) has a free variable call it \(x\). Then the Bang condition (ii) holds for \(u\) iff:

- for any door-extreme subterm \(v\) of \(u\) such that \(v \neq u, v \neq x, s(\text{doors}(u, v)) \geq 1;\) and \(s(\text{doors}(u, x)) = 0, \) if \(u\) has a free variable \(x\).

**Proof.** As for the ‘only-if’ direction, it suffices to show that \(s(\text{doors}(u, x)) = 0\) whenever \(u\) has a free variable \(x\). By the Bracketing condition, there is a subterm \(w\) of \(t\) such that doors \((w, x)\) is well-bracketed (\(w\) is of the form \(\lambda x. v,\) or \(w = t\) if \(x\) is free in \(t\)). Therefore \(s(\text{doors}(w, u)) \geq 0\) and \(s(\text{doors}(w, x)) = 0\), so \(s(\text{doors}(u, x)) \leq 0\). Let \(u'\) be the smallest subterm of \(u\) strictly containing \(x\). We have \(s(\text{doors}(u, u')) \geq 1\) and \(-1 \leq s(\text{doors}(u', x)) \leq 1\), so \(s(\text{doors}(u, x)) = 0\).

To show the ‘if’ direction, let \(v\) be a subterm of \(u\) such that \(v \neq u\). If \(u\) has a free variable \(x\) we also assume that \(v \neq x\). If \(v\) is a door-extreme subterm then \(s(\text{doors}(u, v)) \geq 1\). Otherwise there are two door-extreme subterms \(v_1, v_2\) of \(u\) such that:

- \(v_1 \subseteq v \subseteq v_2\), where \(\subseteq\) denotes the subterm relation,
- \(v_1\) is an immediate distinct door-extreme subterm of \(v_2\).

Because of regularity, we have:

- either \(s(\text{doors}(u, v_2)) \geq s(\text{doors}(u, v)) \geq s(\text{doors}(u, v_1))\),
- or \(s(\text{doors}(u, v_2)) < s(\text{doors}(u, v)) < s(\text{doors}(u, v_1))\).

Moreover we know that \(s(\text{doors}(u, v_2)) \geq 1\) and \(s(\text{doors}(u, v_1)) \geq 0\) (because if \(v_1 = x\) then \(s(\text{doors}(u, v_1)) = 0\), and otherwise \(s(\text{doors}(u, v_1)) \geq 1\)). Therefore we have \(s(\text{doors}(u, v)) \geq 1\). \(\square\)

### 2.3.5 Correctness of the conditions

So far we have introduced four conditions on pseudo-terms: Local typing, Bracketing, Bang and A-scope. Let us call a regular pseudo-term satisfying these conditions well-structured. It turns out that the well-structured pseudo-terms exactly correspond to the DLAL typing derivations.
Lemma 2.6. Let $M_0$ be a system F term. If

$$x_1: A_1, \ldots, x_m: A_m; y_1: B_1, \ldots, y_n: B_n \vdash_{DLAL} M_0 : C,$$

then there is a decoration $t$ of $M_0$ with type $C^*$ and with free variables $x_1^{A_1^*}, \ldots, x_m^{A_m^*}, y_1^{B_1^*}, \ldots, y_n^{B_n^*}$ which is well-structured.

Proof. One can build a (possibly non-regular) decoration $M_0^+$ of $M_0$ by induction on the derivation. Depending on the last typing rule used (see Figure 2.1), $M_0^+$ takes one of the following forms:

(Id) $x^{A^*}$

(⇒ i) $\lambda x^{A^*}. M^+$

(⇒ e) $(M^+)^N$

(⇒ i) $\lambda x^{iA^*}. M^+$

(⇒ e) $(M^+)^N [\bar{z}^{iC^*}/z]$

(Weak) $M^+$

(Cntr) $M^+[x/x_1, x/x_2]$

(∀ i) $\Lambda \alpha. M^+$

(∀ e) $(M^+)^B^*$

(§ i) $\bar{z} M^+[\bar{z} x_{i}^{A_{i}^*}/x_{i}, \bar{z} y_{j}^{B_{j}^*}/y_{j}]$

(§ e) $M^+[N^+/x],$

where $M^+$ in (§ i) has free variables $x_1^{A_1^*}, \ldots, x_m^{A_m^*}, y_1^{B_1^*}, \ldots, y_n^{B_n^*}$.

It is easy to verify that $M_0^+$ admits Local typing with the output type $C^*$ and has the free variables $x_1^{A_1^*}, \ldots, x_m^{A_m^*}, y_1^{B_1^*}, \ldots, y_n^{B_n^*}$.

Moreover, one can show by induction on the derivation that $M_0^+$ satisfies the Bracketing, Bang and Λ-scope conditions. Let us just remark:

- The rules (⇒ i) and (⇒ i) introduce new abstraction terms $\lambda x^{A^*}, M^+$ and $\lambda x^{iA^*}. M^+$, respectively. The Bracketing condition (ii.b) for them follows from the Bracketing condition (i) for $M^+$.

- The rule (⇒ e) introduces a new bang term $\bar{z} M^+[\bar{z} x_{i}^{A_{i}^*}/x_{i}, \bar{z} y_{j}^{B_{j}^*}/y_{j}]$. It satisfies the Bang condition (i) because $N$ contains at most one linear variable $z$. The condition (ii) holds because $N^+$ satisfies the Bracketing condition, and thus we have $\text{doors}(N^+, u) \geq 0$ for any subterm $u$.

Observe also that the Bracketing condition is maintained because the $\bar{z}$ added before $N^+$ and the $\bar{z}$ added before the variable $z$ match each other, so $z$ remains well-bracketed, and condition (i) is preserved; since we add a $\bar{z}$ on $N$, condition (ii.a) is maintained as well; and as bounded variables of $N$ are left unmodified, (ii.b) is obviously still verified.

We also have to make sure that the substitution of $\bar{z} z$ for $z$ does not violate the Λ-scope condition. It follows from the eigenvariable condition for $N$, which ensures that $z$ does not depend on any bound type variable.

- The rule (Cntr) conforms to the Local typing condition (ii).

- The rule (∀ i) introduces a new type abstraction $\Lambda \alpha. M^+$. The Λ-scope condition for it follows from the Bracketing condition for $M^+$.

- The rule (§ i) clearly preserves the Bracketing condition. It is also clear that the substitution involved does not cause violation of the Bang condition (as $x_i$’s and $y_j$’s have linear types in $M^+$, and thus do not appear in any bang term), and the Λ-scope condition (as $x_i$’s and $y_j$’s do not depend on any bound type variable due to the eigenvariable condition).

- The rule (§ e) involves substitution. The term $M^+[N^+/x]$ satisfies the Λ-scope condition since substitution is capture-free, and thus no free type variable in $N^+$ becomes bound in $M^+[N^+/x]$. 
Finally, the required regular pseudo-term $t$ is obtained from $M^0_0$ by applying the following rewrite rules as many times as possible:

$$\frac{\bar{s}u}{u}, \frac{\bar{\bar{s}}u}{u}.$$  

It is clear that all the conditions are preserved by these rewrites.

To show the converse direction, the following Lemma plays a crucial role:

**Lemma 2.7 (Boxing).** If $\bar{s}t : \bar{s}A$ is a well-structured pseudo-term, then there exist pseudo-terms $v : A$, $u_1 : \bar{s}B_1$, ..., $u_n : \bar{s}B_n$, unique (up to renaming of $v$’s free variables) such that:

1. $\text{FV}(v) = \{x_1^{B_1}, \ldots, x_n^{B_n}\}$ and each $x_i$ occurs exactly once in $v$,
2. $\bar{s}t = \bar{s}v[\bar{s}u_1/x_1, \ldots, \bar{s}u_n/x_n]$ (substitution is assumed to be capture-free),
3. $v, u_1, \ldots, u_n$ are well-structured.

**Proof.** Given $\bar{s}t$, assign an index to each occurrence of $\bar{s}$ and $\bar{\bar{s}}$ in $\bar{s}t$ to distinguish occurrences (we assume that the outermost $\bar{s}$ has index 0). By traversing from the root of the syntactic tree, one can find closing brackets $\bar{s}_1, \ldots, \bar{s}_n$ that match the opening bracket $\bar{s}_0$ in $\bar{s}t$. Replace each $\bar{s}_i u_i : B_i$ with a fresh and distinct free variable $x_i^{B_i}$ ($1 \leq i \leq n$), and let $\bar{s}v$ be the resulting pseudo-term. This way one can obtain $v, u_1, \ldots, u_n$, such that condition (2) holds.

Strictly speaking, it has to be checked that the substitution does not cause capture of type or term variables. Let us consider the case of type variables: suppose that $u_i$ contains a subterm $s$ that depends on a bound variable $\alpha$ of $\bar{s}v$. Then $\bar{s}_0 t$ contains a subterm of the form $\bar{s}v[\bar{s}_i u_i[s]/x_i]$. However, doors$(\bar{s}v, s)$ with $v'' = v'[\bar{s}_i u_i[s]/x_i]$ cannot be weakly well-bracketed because $\bar{s}_i$ has to match the outermost opening bracket $\bar{s}_0$. This contradicts the $\Lambda$-scope condition for $\bar{s}_0 t$. Hence the case of type variable capture is solved. A similar argument using the Bracketing condition shows that the substitutions do no cause term variable capture either.

As to condition (1), we claim that $v$ does not contain a free variable other than $x_1, \ldots, x_n$. If there is any, say $y$, then it is also a free variable of $t$, thus the Bracketing condition for $\bar{s}_0 t$ implies that doors$(\bar{s}_0 t, y)$ is well-bracketed, and thus there is a closing bracket that matches $\bar{s}_0$ in the path from $\bar{s}_0 t$ to $y$. That means that $y$ belongs to one of $u_1, \ldots, u_n$, not to $v$. A contradiction.

Let us now check condition (3). As to the Bracketing condition (i) for $v$, we define $l_i = \text{doors}(\bar{s}_0 t, \bar{s}_i u_i)$ for each $1 \leq i \leq n$. Then we have $s(l_1) \geq 1$ for all $\epsilon \neq l \leq l_i$ and $s(l_i) = 1$, and the same is true of the list doors$(\bar{s}_0 v, x_i)$. Therefore, doors$(v, x_i)$ is well-bracketed for each $1 \leq i \leq n$. (ii.a) and (ii.b) are easy. As for $u_i$ ($1 \leq i \leq n$), notice that $s(\text{doors}(\bar{s}_0 t, u_i)) = 0$. This means that for any subterm occurrence $\bar{s}v'$ of $u_i$, we have $s(\text{doors}(u_i, u')) = s(\text{doors}(\bar{s}_0 t, u'))$. Therefore, the Bracketing condition for $u_i$ reduces to that for $\bar{s}_0 t$.

The $\Lambda$-scope condition for $v, u_1, \ldots, u_n$ easily reduces to that for $\bar{s}_0 t$.

As to the Local typing condition, the only nontrivial point to check is whether $v$ satisfies the eigenvariable condition. Suppose that $x_i$ depends on a variable $\alpha$ which is bound in $v$. Then $\bar{s}_0 t$ contains a subterm of the form $\bar{s}v'[\bar{s}_i u_i[x_i]]$ and $u_i$ depends on $\alpha$. However, doors$(v'', u_i)$ with $v'' = v'[\bar{s}_i u_i[x_i]]$ cannot be weakly well-bracketed because $\bar{s}_i$ should match the outermost opening bracket $\bar{s}_0$. This contradicts the $\Lambda$-scope condition for $\bar{s}_0 t$.

To show the Bang condition for $v$ (it is clear for $u_1, \ldots, u_n$), suppose that $v$ contains a bang subterm $\bar{s}v'$. We claim that $\bar{s}v'$ does not contain variables $x_1, \ldots, x_n$. If it contains any, say $x_i$, then $\bar{s}_0 t$ contains $v'' = v'[\bar{s}_i u_i/x_i]$ and the Bang condition for $\bar{s}_0 t$ implies that $s(\text{doors}(v'', x_i)) \geq 1$. On the other hand, we clearly have $s(\text{doors}(\bar{s}_0 t, v'')) \geq 1$ because $v''$ contains the closing bracket $\bar{s}_i$ that matches $\bar{s}_0$. As a consequence, we have $s(\text{doors}(\bar{s}_0 t, x_i)) \geq 2$. This means that $\bar{s}_i$ does not match $\bar{s}_0$, a contradiction. As a consequence, $v'$ does not contain $x_1, \ldots, x_n$. So $\bar{s}v'$ occurs in $\bar{s}_0 t$, and therefore satisfies the Bang condition.

}\]
2.3 LOCALISATION OF DLAL TYPE INFERENCES

Now we can prove:

**Theorem 2.8.** Let \( M \) be a system \( F \) term. Then

\[
x_1 : A_1, \ldots, x_m : A_m; y_1 : B_1, \ldots, y_n : B_n \vdash_{DLAL} M : C
\]

if and only if there is a decoration \( t \) of \( M \) with type \( C^* \) and with free variables \( x_1^{A_1}, \ldots, x_m^{A_m}, y_1^{B_1}, \ldots, y_n^{B_n} \) which is well-structured.

**Proof.** The 'only-if' direction has already been proved. As for the 'if' direction, we prove the following: if a pseudo-term \( t : C^* \) is well-structured and \( \text{FV}(t) = \{x_1^{A_1}, \ldots, x_m^{A_m}, y_1^{B_1}, \ldots, y_n^{B_n}\} \) for some DLAL types \( A_1, \ldots, A_m, B_1, \ldots, B_n \), then we have \( \Gamma; \Delta \vdash_{DLAL} t^\circ : C \), where \( \Gamma = x_1 : A_1, \ldots, x_m : A_m \) and \( \Delta = y_1 : B_1, \ldots, y_n : B_n \). The proof proceeds by induction on the size of \( t \).

- When \( t = x_i^{A_i} \) for some \( 1 \leq i \leq m \), \( C^* \) must be \( \alpha \hat{A}_i \) by Local typing, and we have \( \Gamma; \Delta \vdash_{DLAL} x_i : \hat{A}_i \). Likewise, if \( t = y_j^{B_j} \) for some \( 1 \leq j \leq n \), we have \( \Gamma; \Delta \vdash_{DLAL} y_j : B_j \).

- When \( t = \lambda z^{A_0\alpha}; u : A_0^\circ \rightarrow C_0^\circ, u : C_0^\circ \) is also well-structured; observe in particular that the Bracketing condition for \( t \) implies the same for \( u \). By induction hypothesis, we have \( z : A_0, \Gamma; \Delta \vdash_{DLAL} u^\circ : C_0 \), and hence

\[
\Gamma; \Delta \vdash_{DLAL} \lambda z^{A_0\alpha}.u^\circ : A_0 \Rightarrow C_0.
\]

The case when \( z \) has a linear type is similar.

- When \( t = \Lambda \alpha; u : \forall \alpha.C_0^\circ, u : C_0^\circ \) is also well-structured. Hence one can argue as above; notice in particular that the eigenvariable condition on \( t \) ensures that one can apply the rule \( (\forall) \) to \( u^\circ \).

- When \( t = (u)B^* : C_0^\circ[B^*/\alpha], u : \forall \alpha.C_0^\circ \) is well-structured, and the induction hypothesis yields \( \Gamma; \Delta \vdash_{DLAL} u^\circ : \forall \alpha.C_0 \). We therefore obtain \( \Gamma; \Delta \vdash_{DLAL} (u^\circ)B^* : C_0[B/\alpha] \).

- It is impossible to have \( t = \ll u \), because it clearly violates the Bracketing condition.

- When \( t = \ll t^\prime : C_0^\circ \), the Boxing Lemma gives us well-structured terms \( v : C_0^\circ, u_1 : \hat{C}_1^\circ, \ldots, u_k : \hat{C}_k^\circ \) such that
  1. \( \text{FV}(v) = \{z_1^{C_1}, \ldots, z_k^{C_k}\} \) and each \( z_i \) occurs exactly once in \( v \),
  2. \( t^\prime = v[u_1/z_1, \ldots, u_k/z_k] \).

By the induction hypothesis, we have

\[
\Gamma; \Delta_i \vdash_{DLAL} v^\circ, u_1^- : \hat{C}_i \text{ and } \Gamma; \Delta_i \vdash_{DLAL} v^\circ, u_k^- / z_k : \hat{C}_0
\]

for \( 1 \leq i \leq k \), where \( (\Delta_1, \ldots, \Delta_k) \) is a partition of \( \Delta \) such that each \( \Delta_i \) contains the free variables occurring in \( u_i \). Hence by rules (§i), (§e) and (Cntx), we obtain

\[
\Gamma; \Delta \vdash_{DLAL} v^\circ, [u_1^- / z_1, \ldots, u_k^- / z_k] : \hat{C}_0 .
\]

- When \( t = (t^\prime)t^\prime\) and \( t^\prime \) is not a bang subterm, one can argue as above. When \( t^\prime \) is a bang subterm, \( t^\prime \) and \( t^\prime\) are locally typed as \( t^\prime : A^* \rightarrow C^\circ \) and \( t^\prime : \alpha.A^* \). They are well-structured, and moreover:
(i) $t''$ contains at most one free variable $x_i^{A_i}$, which is among $\{x_1, \ldots, x_m\}$,
(ii) for any subterm $v$ of $t''$ such that $v \neq t''$ and $v \neq x_i$, $s(\text{doors}(u, v)) \geq 1$.

By the induction hypothesis on $t'$ (and by the fact that $t''$ does not contain any variable
of linear type), we have

$$\Gamma; \Delta \vdash_{DLAL} (t')^- : A \Rightarrow C.$$ 

On the other hand, the condition (ii) above entails that $t''$ is either the variable
$x_i$ or of the form $\$u$. In the former case, $A^* = A_i^*$ and we have:

$$\Gamma; \Delta \vdash (t')^- : A \Rightarrow C ; x_i : A \vdash x_i : A \quad \Gamma; \Delta \vdash (t')^-x_i : C .$$

In the latter case, we can apply the Boxing Lemma. Then the conditions (i) and (ii)
entail that there is a well-structured term $v : A^*$ with a free variable $z$
such that $t'' = \$u = \$v[\$x_i/z]$. Notice here that $z$ has a linear type $A_i^*$, and by renaming, one can assume
w.l.o.g. that $z = x_i$ in $v$. Therefore, we obtain:

$$\Gamma; \Delta \vdash (t')^- : A \Rightarrow C ; x_i : A_i \vdash v^- : A \quad \Gamma; \Delta \vdash (t')^-v^- : C .$$

As a consequence of Theorem 2.8 our DLAL typing problem (Problem 2.2) boils down to:

**Problem 2.9 (decoration).** Given a system F term $M$, determine if there exists a decoration
$t$ of $M$ which is well-structured.

### 2.4 Parameterisation and constraints generation

To solve the decoration problem (Problem 2.9), one needs to explore an infinite set of decorations.
This can be effectively done by introducing an abstract kind of types and terms with
symbolic parameters (in 2.4.1), and expressing the conditions for such abstract terms to be
materialised by boolean and integer constraints over those parameters (in 2.4.2 and in 2.4.3).

#### 2.4.1 Parameterised terms and instantiations

Let us begin with introducing a term syntax with parameters. We use two sorts of parameters:
integer parameters $n, m, \ldots$ meant to range over $\mathbb{Z}$, and boolean parameters $b_1, b_2, \ldots$
meant to range over $\{0, 1\}$. We also use linear combinations of integer parameters $c = n_1 + \cdots + n_k$
where $k \geq 0$ and each $n_i$ is an integer parameter. In case $k = 0$, it is written as $0$.

The set of parameterised types ($p$-types for short) is defined by:

$$F ::= \alpha | D \to A | \forall \alpha. A ,$$
$$A ::= \$^c F ,$$
$$D ::= \$^{b+c} F .$$

where $b$ is a boolean parameter and $c$ is a linear combination of integer parameters. Informally
speaking, the parameter $c$ in $\$^{b+c} F$ stands for the number of modalities ahead of the type, while
the boolean parameter $b$ serves to determine whether the first modality, if any, is $\$ or $!$. In
the sequel, $A, B, C$ stand for linear $p$-types of the form $\$^c F$, and $D$ for bang $p$-types of the form
$\$^{b+c} F$, and $E$ for arbitrary $p$-types.
2.4 Parameterisation and Constraints Generation

When $A$ is a linear p-type $\frac{\phi}{A}$, $B[A/\alpha]$ denotes a p-type obtained by replacing each $\frac{\phi}{x}$ in $B$ with $\frac{\phi}{x^{+}}$ and each $\frac{\phi}{b}$ with $\frac{\phi}{b^{+}}$. When $D = \frac{\phi}{b^{e}}$, $D^{e}$ denotes the linear p-type $\frac{\phi}{b^{e}}$.

We assume that there is a countable set of variables $x^{D}, y^{D}, \ldots$ for each bang p-type $D$. The parameterised pseudo-terms (p-terms for short) $t, u \ldots$ are defined by the following grammar:

$$u ::= x^{D} \mid \lambda x^{D}.t \mid (t)t \mid \Lambda \alpha.t \mid (t)A,$$

$$t ::= \frac{\phi}{m}u.$$

We denote by $par^{bool}(t)$ the set of boolean parameters of $t$, and by $par^{int}(t)$ the set of integer parameters of $t$. An instantiation $\phi = (\phi^{b}, \phi^{\tau})$ for a p-term $t$ is given by two maps $\phi^{b} : par^{bool}(t) \to \{0,1\}$ and $\phi^{\tau} : par^{int}(t) \to \mathbb{Z}$. The map $\phi^{b}$ can be naturally extended to linear combinations $c = n_{1} + \cdots + n_{k}$ by $\phi^{b}(c) = \phi^{b}(n_{1}) + \cdots + \phi^{b}(n_{k})$. An instantiation $\phi$ is said to be admissible for a p-type $E$ if for any linear combination $c$ occurring in $E$, we have $\phi^{b}(c) \geq 0$, and moreover whenever $\frac{\phi}{b^{e}}$ occurs in $E$, $\phi^{b}(c) = 1$ implies $\phi^{b}(\overline{c}) \geq 1$. When $\phi$ is admissible for $E$, a type $\phi(E)$ of Dlal* is obtained as follows:

$$\phi(\frac{\phi}{b^{e}}F) = \frac{\phi}{\phi}(\frac{\phi}{b^{e}}F), \quad \phi(\frac{\phi}{b^{e}}F) = \frac{\phi}{\phi}(\frac{\phi}{b^{e}}F) \phi(F) = \frac{\phi}{\phi}(\frac{\phi}{b^{e}}-1F) \text{ if } \phi^{b}(b) = 0,$$

and $\phi$ commutes with the other connectives. An instantiation $\phi$ for a p-term $t$ is said to be admissible for $t$ if it is admissible for all p-types occurring in $t$. If $\phi$ is admissible for $t$, a regular pseudo-term $\phi(t)$ can be obtained by replacing each $\frac{\phi}{m}u$ with $\frac{\phi}{\phi}(\frac{\phi}{m}u)$, each $x^{D}$ with $x^{D}$, and each $(t)A$ with $(t)\phi(A)$.

As for pseudo-terms there is an erasure map $(.)^{\perp}$ from p-terms with their p-types to system F terms consisting in forgetting modalities and parameters.

A free linear decoration (free bang decoration, resp.) of a system F type $T$ is a linear p-type (bang p-type, resp.) $E$ such that (i) $E = T$, (ii) each linear combination $c$ occurring in $E$ is a single integer parameter $m$, and (iii) the parameters occurring in $E$ are mutually distinct. Two free decorations $\overline{T}_{1}$ and $\overline{T}_{2}$ are said to be disjoint if the set of parameters occurring in $\overline{T}_{1}$ is disjoint from the set of parameters in $\overline{T}_{2}$.

The free decoration $\overline{M}$ of a system F term $M$ (which is unique up to renaming of parameters) is obtained as follows: first, to each variable $x^{T}$ we associate a parameterised variable $\overline{x}^{T} = \overline{x}^{D}$ such that (i) $D$ is a free bang decoration of $T$, and (ii) whenever $x^{T_{1}}$ and $y^{T_{2}}$ are distinct variables, the free bang decorations $D_{1}, D_{2}$ associated to them are disjoint. $\overline{M}$ is now defined by induction on the construction of $M$:

$$\overline{\lambda x}^{T}.M = \overline{\frac{\phi}{m}}\lambda x^{T}.\overline{M}, \quad \overline{(M)N} = \overline{\frac{\phi}{m}}(\overline{M})\overline{N}, \quad \overline{M}A = \overline{\frac{\phi}{m}}(\overline{M})A,$$

where all newly introduced parameters $m$ are chosen to be fresh, and the parameter $p$-type in the definition of $(\overline{M})\overline{T}$ is a free linear decoration of $T$ which is disjoint from all p-types appearing in $\overline{M}$.

The key property of free decorations is the following:

**Theorem 2.10.** Let $M$ be a system F term and $t$ be a regular pseudo-term. Then $t$ is a decoration of $M$ if and only if there is an admissible instantiation $\phi$ for $\overline{M}$ such that $\phi(\overline{M}) = t$.

**Proof.** We first prove that for any system F type $T$, any free bang decoration $D$ of $T$ and any Dlal* type $E$, we have $E^{\perp} = T$ if there is an admissible instantiation $\phi$ for $D$ such that $\phi(D) = E$. This statement, as well as a similar one with respect to free linear decorations and linear Dlal* types, can be simultaneously proved by induction on $T$. Then the Theorem can be shown by induction on $M$. \qed
and a unique set of constraints § for $t_{\text{subterm}}$

Problem 2.11 (instantiation) a system F typed term, we know that impossible instantiation

Lemma Proof.

will speak of linear inequations.

2.4.2 Local typing constraints

First of all, we need to express the unifiability of two p-types $E_1$ and $E_2$. We define a set $U(E_1, E_2)$ of constraints by

$$U(\alpha, \alpha) = \emptyset,$$

$$U(D_1 \rightarrow A_1, D_2 \rightarrow A_2) = U(D_1, D_2) \cup U(A_1, A_2),$$

$$U(\forall \alpha.A_1, \forall \alpha.A_2) = U(A_1, A_2),$$

$$U(\frac{E_1}{\alpha}, \frac{E_2}{\alpha}) = \{c_1 = c_2\} \cup U(F_1, F_2),$$

$$U(\frac{\beta_1}{\alpha}, \frac{\beta_2}{\alpha}) = \{b_1 = b_2, c_1 = c_2\} \cup U(F_1, F_2).$$

It is undefined otherwise. It is straightforward to observe:

Lemma 2.12. Let $E_1$, $E_2$ be two linear (bang, resp.) p-types such that $E_1^- = E_2^-$. Then $U(E_1, E_2)$ is defined. Moreover, when $\phi$ is an admissible instantiation for $E_1$ and $E_2$, we have $\phi(E_1) = \phi(E_2)$ if and only if $\phi$ is a solution of $U(E_1, E_2)$.

Proof. By induction on $E_1$. \hfill \square

For any p-type $E$, define

$$\text{Adm}(E) = \{c \geq 0 : c \text{ occurs in } E\} \cup \{b = 1 \Rightarrow c \geq 1 : \frac{\beta}{\alpha}F \text{ occurs in } E\}.$$

Then $\phi$ is admissible for $E$ if and only if $\phi$ is a solution of $\text{Adm}(E)$.

Now consider the free decoration $M$ of a system F typed term $M$. We assign to each subterm $t$ of $\overline{M}$ a linear p-type $B$ and a set $M$ of constraints (indicated as $t : B : M$) as on Figure 2.6. Notice that any linear p-type is of the form $\frac{E}{\alpha}$. Moreover, since $t$ comes from a system F typed term, we know that $t$ has a p-type $\frac{E}{\alpha}(D \rightarrow B)$ when $t$ occurs as $(t)u$, and $\frac{E}{\alpha}(\forall \alpha.B)$ when $t$ occurs as $(t)A$. In the former case, we have $(D^-)^- = \bigvee$, so that $U(D^-, A)$ used in the application rule is always defined. As a consequence, for any $M$ a unique p-type and a unique set of constraints $M(M)$ are obtained. Finally, observe that $M$ satisfies the eigenvariable condition.

Let $\text{Ltype}(M)$ be $M(M) \cup \{b = 1 : \frac{\beta}{\alpha}F \text{ occurs more than once in } M\}.$

Lemma 2.13. Let $M$ be a system F term and $\phi$ be an instantiation for $M$. Then $\phi$ is admissible for $M$ and $\phi(M)$ satisfies the Local typing condition if and only if $\phi$ is a solution of $\text{Ltype}(M)$. 
2.4.3 Boxing constraints

We consider the words over integer parameters \( m, n \ldots \), whose set we denote by \( L_p \). Let \( t \) be a \( p \)-term and \( u \) an occurrence of subterm of \( t \). We define, as for pseudo-terms, the word \( \text{doors}(t, u) \) in \( L_p \) as follows. If \( t = u \), let \( \text{doors}(t, u) = \epsilon \). Otherwise:

\[
\begin{align*}
\text{doors}(\lambda y^D, t, u) &= \lambda y^D \text{:: } \text{doors}(t, u), \\
\text{doors}(\Lambda \alpha, t_1, u) &= \text{doors}(\Lambda \alpha, t_1, u) = \text{doors}(t_1, u) = \text{doors}(t_1, u), \\
\text{doors}((t_1)t_2, u) &= \text{doors}(t_1, u) \text{ where } t_i \text{ is the subterm containing } u.
\end{align*}
\]

The sum \( s(l) \) of an element \( l \) of \( L_p \) is a linear combination of integer parameters defined by:

\[
s(\epsilon) = 0, \quad s(m :: l) = m + s(l).
\]

For each list \( l \in L_p \), define \( \text{wbracket}(l) = \{ s(l') \geq 0 \mid l' \leq l \} \) and \( \text{bracket}(l) = \text{wbracket}(l) \cup \{ s(l) = 0 \} \).

Given a system \( F \) term \( M \), we consider the following sets of constraints:

\textbf{Bracketing constraints.} \( \text{Bracket}(\overline{M}) \) is the union of the following sets:

(i) for each occurrence of free variable \( x \) in \( \overline{M} \), \( \text{bracket}(\text{doors}(\overline{M}, x)) \);

(ii) for each occurrence of an abstraction subterm \( \lambda x.v \) of \( \overline{M} \):

(ii.a) \( \text{wbracket}(\text{doors}(\overline{M}, \lambda x.v)) \),

(ii.b) for any occurrence of \( x \) in \( v \), \( \text{bracket}(\text{doors}(v, x)) \).

\textbf{Bang constraints.} A subterm \( u \) that occurs in \( \overline{M} \) as \( (t)u \) with \( t : \xi^c(\xi^b.c.F \rightarrow B) \) is called a \textit{bang subterm} of \( \overline{M} \) with the \textit{critical parameter} \( b \). Now \( \text{Bang}(\overline{M}) \) is the union of the following sets: for each bang subterm \( u \) of \( \overline{M} \) with a critical parameter \( b \),

(i) \( \{ b = 0 \} \) if \( u \) has strictly more than one occurrence of free variable, and

\( \{ b = 1 \Rightarrow b' = 1 \} \) if \( u \) has exactly one occurrence of free variable \( x^{b', c'} f' \).

(ii) \( \{ b = 1 \Rightarrow s(\text{doors}(u, v)) \geq 1 : v \text{ subterm of } u \text{ such that } v \neq u \text{ and } v \neq x \} \cup \{ b = 1 \Rightarrow s(\text{doors}(u, x)) = 0 \} \).

(where \( x \) is the free occurrence of variable in \( u \), if there is one, otherwise the second set in the union is removed)

\textbf{Remark 2.14.} Note that if \( t \) is a \( p \)-term and \( \phi \) is an instantiation, the pseudo-term \( \phi(t) \) might have more subterms than \( t \). In fact subterms of the \( p \)-term \( t \) are in correspondence with door-extreme subterms of the regular pseudo-term \( \phi(t) \).

For instance if \( t = \xi^m x \) and \( \phi(\xi^m x) = 2 \), the subterms of \( t \) and \( \phi(t) \) are respectively \( \{ \xi^m x, x \} \) and \( \{ \xi^2 x, \xi x, x \} \). The door-extreme subterms of \( \phi(t) \) are \( \{ \xi^2 x, x \} \). This is why we had to add \( \text{Bang}(\overline{M}) \) constraints (ii) the condition \( \{ b = 1 \Rightarrow s(\text{doors}(u, x)) = 0 \} \) (see Lemma 5.3).

\textbf{\( \Lambda \)-Scope constraints.} \( \text{Scope}(\overline{M}) \) is the union of the following sets:

(i) \( \text{wbracket}(\text{doors}(u, v)) \) for each subterm \( \Lambda \alpha, u \) of \( \overline{M} \) and for each subterm \( v \) of \( u \) that depends on \( \alpha \).

We denote \( \text{Const}(\overline{M}) = \text{Ltype}(\overline{M}) \cup \text{Bracket}(\overline{M}) \cup \text{Bang}(\overline{M}) \cup \text{Scope}(\overline{M}) \).
Remark 2.15. Instead of using p-terms, the parameters and constraints might be visualised on pseudo-terms graphs. Using our running example of Fig. 2.5, we can decorate it with parameters $m_i$: see Fig. 2.7. Each $m_i$ stands for a possible sequence of doors: if it is instantiated with $k \geq 0$ (resp. $k \leq -1$) then this will correspond to $k$ (resp. $(-k)$) opening (resp. closing) doors. Then, for instance, the Bracketing constraints correspond to conditions on the parameters occurring along certain paths of the graph (as indicated in Remark 2.3). As an example the Bracketing constraint (ii.b) for the binder $\lambda f$ and the two free occurrences of $f$ gives here \{ $m_3 \geq 0$, $m_3 + m_4 = 0$, $m_3 + m_5 \geq 0$, $m_3 + m_5 + m_6 = 0$ \}.

Figure 2.7: Example of parameterised pseudo-term graph.

Theorem 2.16. Let $M$ be a system F term and $\phi$ be an instantiation for $M$. Then: $\phi$ is admissible for $M$ and $\phi(M)$ is well-structured if and only if $\phi$ is a solution of $\text{Const}(M)$.

Moreover, the number of (in)equations in $\text{Const}(M)$ is quadratic in the size of $M$.

Proof. Clearly, the above constraints are almost direct parameterisations of the corresponding conditions given in the previous section. Let us just examine the Bang condition.

Suppose that $\phi(M)$ satisfies the Bang condition. For each (parameterised) bang subterm $u$ with the critical parameter $b$ in $M$, one of the following two happens:

- $\phi(u)$ is not a bang subterm of $\phi(M)$ (in the sense of the previous section). Namely, $\phi(b) = 0$. In this case, $\phi$ is a solution of the equation $b = 0$, and also of $b = 1 \Rightarrow b' = 1$ if $u$ has a free variable $x$.

- $\phi(u)$ is a bang subterm of $\phi(M)$. Namely, $\phi(b) = 1$. In this case, $u$ contains at most one variable by the Bang condition. Hence the equation $b = 0$ does not belong to $\text{Bang}(M)$. Moreover, if $u$ has a free variable $x$, $\phi(x)$ must have a bang type and so $\phi(b') = 1$. Hence $\phi$ is a solution of $b = 1 \Rightarrow b' = 1$.

It is straightforward to observe that $\phi$ is a solution of the equations in (ii), by using Lemma 2.5. Therefore, $\phi$ is a solution of $\text{Bang}(M)$.

Now suppose the converse and let $u$ be a (parameterised) bang subterm with the critical parameter $b$ in $M$. Suppose also that $\phi(u)$ is a bang subterm of $\phi(M)$. This means that
\( \phi(b) = 1 \). Since \( \phi \) is supposed to be a solution of \( \text{Bang}(\mathcal{M}) \), \( u \) must contain at most one free variable, say \( x^b \). Moreover, we have \( \phi(b') = 1 \), which means that \( \phi(x) \) has a bang type in \( \phi(M) \). Therefore, \( \phi(M) \) satisfies the Bang condition (i). As \( \phi \) satisfies the conditions in (ii) and \( \phi(b) = 1 \) we get that \( \phi(u) \) satisfies the condition in Lemma 2.5 hence by this Lemma we obtain that \( \phi(u) \) satisfies the Bang condition (ii).

\[ \square \]

### 2.5 Solving the constraints

Having described a way to collect a set of constraints from a given system \( F \) term, there just remains to give a fast algorithm to solve them. Our method proceeds as follows: first solve the boolean constraints, which corresponds to determine which \( ! \)-boxes are necessary (in 2.5.1), and then solve the integer constraints, which corresponds to complete the decoration by finding a suitable box structure (in 2.5.2).

#### 2.5.1 Solving boolean constraints

We split \( \text{Const}(\mathcal{M}) \) into three disjoint sets \( \text{Const}^b(\mathcal{M}), \text{Const}^i(\mathcal{M}), \text{Const}^m(\mathcal{M}) \):

- A **boolean constraint** \( s \in \text{Const}^b(\mathcal{M}) \) consists of only boolean parameters. \( s \) is of one of the following forms:

  \[
  \begin{align*}
  b_1 &= b_2 \quad (\text{in } \text{Ltype}(\mathcal{M})), \\
  b &= 0 \quad (\text{in } \text{Bang}(\mathcal{M})), \\
  b &= 1 \Rightarrow b' = 1 \quad (\text{in } \text{Bang}(\mathcal{M})).
  \end{align*}
  \]

- A **linear constraint** \( s \in \text{Const}^i(\mathcal{M}) \) deals with integer parameters only. A linear constraint \( s \) is of one of the following forms:

  \[
  \begin{align*}
  c &= c_2 \quad (\text{in } \text{Ltype}(\mathcal{M})), \\
  c &\geq 0 \quad (\text{in } \text{Ltype}(\mathcal{M}), \text{Bracket}(\mathcal{M}), \text{Scope}(\mathcal{M})), \\
  c &= 0 \quad (\text{in } \text{Ltype}(\mathcal{M}) \text{ and } \text{Bracket}(\mathcal{M})).
  \end{align*}
  \]

- A **mixed constraint** \( s \in \text{Const}^m(\mathcal{M}) \) contains a boolean parameter and a linear combination and is of the following form:

  \[
  \begin{align*}
  b &= 1 \Rightarrow c = 0 \quad (\text{in } \text{Bang}(\mathcal{M})), \\
  b &= 1 \Rightarrow c \geq 1 \quad (\text{in } \text{Ltype}(\mathcal{M}) \text{ and } \text{Bang}(\mathcal{M})).
  \end{align*}
  \]

We first try to find a solution of \( \text{Const}^b(\mathcal{M}) \), and then proceed to the other constraints. This does not cause loss of generality, because \( \text{Const}^b(\mathcal{M}) \) admits a minimal solution whenever solvable. Let us consider the set of instantiations on boolean parameters and the extensional order \( \leq \) on these maps: \( \psi^b \leq \phi^b \) if for any \( b \), \( \psi^b(b) \leq \phi^b(b) \).

**Lemma 2.17.** There is a polynomial time algorithm to decide whether \( \text{Const}^b(\mathcal{M}) \) has a solution or not. Moreover, the algorithm returns a minimal solution whenever there exists any.

**Proof.** Our algorithm is based on the standard resolution procedure. Let \( B := \text{Const}^b(\mathcal{M}) \). Apply repeatedly the following steps until reaching a fixpoint:

- if \( b_1 = b_2 \in B \) and \( b_1 = i \in B \) with \( i \in \{0, 1\} \), then let \( B := B \cup \{b_2 = i\} \);
- if \( b_1 = b_2 \in B \) and \( b_2 = i \in B \) with \( i \in \{0, 1\} \), then let \( B := B \cup \{b_1 = i\} \);
• if \((b = 1 \Rightarrow b' = 1) \in \mathcal{B}\) and \(b = 1 \in \mathcal{B}\), then let \(\mathcal{B} := \mathcal{B} \cup \{b' = 1\}\).

It is obvious that this can be done in a linear number of steps and that the resulting system \(\mathcal{B}\) is equivalent to \(\text{Const}^b(\overline{\mathcal{M}})\).

Now, if \(\mathcal{B}\) contains a pair of equations: \(b = 0, b = 1\), then it is inconsistent, and hence \(\text{Const}^b(\overline{\mathcal{M}})\) does not have a solution. Otherwise, define the boolean instantiation \(\psi^b\) by

\[
\psi^b(b) := 1 \quad \text{if } b = 1 \in \mathcal{B};
\]
\[
:= 0 \quad \text{otherwise}.
\]

It is clear that \(\psi^b\) is a solution of \(\mathcal{B}\). In particular, observe that any constraint of the form \((b = 1 \Rightarrow b' = 1)\) in \(\mathcal{B}\) is satisfied by \(\psi^b\). Moreover any solution \(\phi^b\) of \(\mathcal{B}\) satisfies \(\psi^b \leq \phi^b\). Therefore, \(\psi^b\) is a minimal solution of \(\text{Const}^b(\overline{\mathcal{M}})\).

\[\square\]

### 2.5.2 Solving integer constraints

When \(\phi^b\) is a boolean instantiation, \(\phi^b\text{Const}^m(\overline{\mathcal{M}})\) denotes the set of linear constraints defined as follows:

• for any constraint of the form \((b = 1 \Rightarrow I)\) in \(\text{Const}^m(\overline{\mathcal{M}})\), where \(I\) is a linear (in)equation (of the form \(c \geq 1\) or \(c = 0\)), \(I\) belongs to \(\phi^b\text{Const}^m(\overline{\mathcal{M}})\) if and only if \(\phi^b(b) = 1\).

Then we clearly have:

\((*)\) \((\phi^b, \phi^i)\) is a solution of \(\text{Const}(\overline{\mathcal{M}})\) if and only if \(\phi^b\) is a solution of \(\text{Const}^b(\overline{\mathcal{M}})\) and \(\phi^i\) is a solution of \(\phi^b\text{Const}^m(\overline{\mathcal{M}}) \cup \text{Const}^i(\overline{\mathcal{M}})\).

#### Lemma 2.18

\(\text{Const}(\overline{\mathcal{M}})\) admits a solution if and only if it has a solution \(\psi = (\psi^b, \psi^i)\) such that \(\psi^b\) is the minimal solution of \(\text{Const}^b(\overline{\mathcal{M}})\).

**Proof.** Suppose that \(\text{Const}(\overline{\mathcal{M}})\) admits a solution \((\phi^b, \phi^i)\). Then by the previous Lemma, there is a minimal solution \(\psi^b\) of \(\text{Const}^b(\overline{\mathcal{M}})\). Since \(\psi^b \leq \phi^b\), we have \(\psi^b\text{Const}^m(\overline{\mathcal{M}}) \subseteq \phi^b\text{Const}^m(\overline{\mathcal{M}})\). Since \(\phi^i\) is a solution of \(\phi^b\text{Const}^m(\overline{\mathcal{M}}) \cup \text{Const}^i(\overline{\mathcal{M}})\) by \((*)\) above, it is also a solution of \(\psi^b\text{Const}^m(\overline{\mathcal{M}}) \cup \text{Const}^i(\overline{\mathcal{M}})\). This means that \((\psi^b, \phi^i)\) is a solution of \(\text{Const}(\overline{\mathcal{M}})\).

Coming back to the proof-net intuition, Lemma 2.18 means that given a syntactic tree of term there is a most general (minimal) way to place \(l\)-boxes (and accordingly !-subtypes in types), that is to say: if there is a DLAL decoration for this tree then there is one with precisely this minimal distribution of \(l\)-boxes.

Now notice that \(\psi^b\text{Const}^m(\overline{\mathcal{M}}) \cup \text{Const}^i(\overline{\mathcal{M}})\) is a linear inequation system, for which a polynomial time procedure for searching a rational solution is known (\([\text{Kac79],[Kar84]}\)).

#### Lemma 2.19

\(\psi^b\text{Const}^m(\overline{\mathcal{M}}) \cup \text{Const}^i(\overline{\mathcal{M}})\) has a solution in \(\mathbb{Q}\) if and only if it has a solution in \(\mathbb{Z}\).

**Proof.** Clearly the set of solutions is closed under multiplication by a positive integer. \(\square\)

#### Theorem 2.20

Let \(M\) be a system \(F\) term. Then one can decide in time polynomial in the cardinality of \(\text{Const}(\overline{\mathcal{M}})\) whether \(\text{Const}(\overline{\mathcal{M}})\) admits a solution.

**Proof.** First decide if there is a solution of \(\text{Const}^b(\overline{\mathcal{M}})\), and if it exists, let \(\psi^b\) be the minimal one (Lemma 2.17). Then apply the polynomial time procedure to decide if \(\psi^b\text{Const}^m(\overline{\mathcal{M}}) \cup \text{Const}^i(\overline{\mathcal{M}})\) admits a solution in \(\mathbb{Q}\). If it does, then we also have an integer solution (Lemma 2.19). Otherwise, \(\text{Const}(\overline{\mathcal{M}})\) is not solvable. \(\square\)

By combining Theorems 2.18, 2.19, 2.16 and 2.20, we conclude that the DLAL typing problem (Problem 2.22) can be solved in polynomial time:

#### Theorem 2.21

Given a system \(F\) term \(M^T\), it is decidable in time polynomial in the size of \(M\) whether there is a decoration \(A\) of \(T\) such that \(\vdash_{\text{DLAL}} M : A\).


\[ N_F = \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha) \]  

We denote by \( k \) the Church integer for \( k \).

If we apply the type inference procedure to the Church integer 2, we obtain the following family of parameterised types with constraints as result:

\[
\begin{align*}
A &= \{ b_1 \geq 0, b_3 = b_7 = 0, n_4 = n_5 = n_8, n_3 + n_4 = n_6 + n_7, n_7 \geq n_4 \} \\
\end{align*}
\]

It is easy to check that conversely, any solution to this system gives a type suitable for all Church integers. We denote by \( N(A) \) this set of constraints. If \( D \) is a free bang decoration of \( N_F \), we define \( N(D) = N(D^\phi) \).

Observe that the type \( N_{DLAL} = \forall \alpha. (\alpha \to \alpha) \Rightarrow (\alpha \to \alpha) \) is obtained by a solution of this system \( \phi(n_3) = \phi(n_6) = 1, \phi(n_4) = 0 \) for \( i \neq 3, 6 \), \( \phi(b_3) = 1, \phi(b_4) = \phi(b_7) = 0 \) but it is not the only one. For instance the following types are also suitable \( DLAL \) types for Church integers:

- \( N_{DLAL}' = \forall \alpha. (\alpha \to \alpha) \Rightarrow (\alpha \to \alpha) \), obtained with \( \phi_0 \) defined as the previous \( \phi \), but for \( \phi_0(n_6) = 0, \phi_0(n_7) = \phi_0(n_8) = 1 \).

- \( \exists \forall \alpha. \exists^3 (\alpha \to \alpha) \Rightarrow (\exists \forall \alpha \to \exists \forall \alpha) \), obtained with \( \phi_1(n_4) \) for \( i = 1, 4, 5, 6 \); \( \phi_1(n_1) = 3 \) for \( i = 3, 7, 8 \); \( \phi_1(n_2) = 0, \phi_1(b_3) = 1, \phi_1(b_4) = \phi_1(b_7) = 0 \).

In the same way we can characterise the \( DLAL \) types for the Church representations of binary words, with a linear free decoration \( A \) of the system F type \( W_F \) and the following set of constraints \( W(A) \):

\[
\begin{align*}
A &= \{ b_3 = b_7 = 1, b_4 = b_8 = b_{11} = 0, n_4 = n_5, n_8 = n_9, n_{11} = n_{12}, n_7 + n_8 = n_{10} + n_{11}, n_{11} \geq n_8, n_{11} \geq n_4, n_1 \geq 0, n_j \geq b_j \text{ for } 1 \leq i \leq 12 \text{ and } j = 3, 7. \} \\
\end{align*}
\]
2.6.2 Typing with domain specification

Actually the DIAL typability of a term $M^{W_F \rightarrow W_F}$ of system F is not sufficient to ensure that $M$ is P-TIME computable. To illustrate this point, we consider for simplicity unary Church integers and terms of type $N_F \rightarrow N_F$. Observe that the following term of system F has type $N_F \rightarrow N_F$ and represents the exponentiation function ($2^n$) over unary integers:

$$\text{exp} = \lambda n. \Lambda \beta. (n \beta \rightarrow \beta)(2 \beta).$$

Thus the term $\text{exp}$ does not represent a P-TIME function but... it is typable in DIAL, with for instance the type:

$$\forall \alpha. [(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)] \rightarrow \forall \beta. [(\beta \rightarrow \beta) \rightarrow \beta(\beta \rightarrow \beta)].$$

The trick here is that this DIAL type does not allow the term $\text{exp}$ to be applied to all Church integers. Indeed the only closed terms of type $\forall \alpha. [(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)]$ are $\mathbf{0}$ and $\mathbf{1}$. So we do obtain a P-TIME term but over a restricted, finite domain...

In general we are therefore not just interested in mere typability but in typability with meaningful types. Indeed we generally want the terms to be typable in DIAL in such a way that they can be applied to arguments of certain data-types (unary integers, lists...). This can be enforced by adding some specification about the domain of the function.

Let $M$ be a system F term of type $T$. We call a domain specification of $M$ a list $\text{Dom} = \langle (x_1, s_1), \ldots, (x_k, s_k) \rangle$ such that for each $i$:

- $x_i$ is a bound variable of $M$,
- $s_i \in \{N, W\}$,
- if $s_i = N$ (resp. $s_i = W$), then $x_i$ is of type $N_F$ (resp. $W_F$) in $M$.

For instance for the previous example of term $\text{exp}$ we can take $\text{Dom} = \{(n, N)\}$.

Here we restrict to $N_F, W_F$ for simplicity, but this definition could be extended to other data-types of system F such as lists, binary trees...

Now we consider the free decoration $\overline{M}$. Let $\text{DomConst}(\overline{M}, \text{Dom})$ be the union of $\mathcal{N}(D_i)$ (resp. $\mathcal{W}(D_i)$) for all bound variables $x_i^{D_i}$ such that $(x_i, N)$ (resp. $(x_i, W)$) is in $\text{Dom}$.

Finding a DIAL type for $M$ such that, in the resulting DIAL typed term, each $x_i$ from $\text{Dom}$ can be instantiated with a Church integer or binary word, is thus equivalent to finding a solution of $\text{Const}(\overline{M})$ which also satisfies $\text{DomConst}(\overline{M}, \text{Dom})$. In the previous example of $\text{exp}$ and the domain specification $\text{Dom}$, there is not any such solution.

We have:

**Theorem 2.22.** Let $M$ be a System F term and $\text{Dom}$ be a domain specification. One can decide in time polynomial in the cardinality of $\text{Const}(\overline{M}) \cup \text{DomConst}(\overline{M}, \text{Dom})$ whether it admits a solution.

**Proof.** It is sufficient to observe that the constraints in $\mathcal{N}(D)$ (where $D$ is a decoration of $N_F$ or $W_F$) also satisfy the properties used to prove Lemma 2.17, Lemma 2.18 and Lemma 2.19.

Note that for Lemma 2.19 for instance the argument would not be valid anymore (at least in an obvious way) if we added constraints of the form $n = 1$ or $n \leq 1$.

Therefore one can perform DIAL decoration for system F terms in polynomial time even with domain specification.
2.7 Implementation and examples

2.7.1 Overview

We designed an implementation of the type inference algorithm. The program is written in functional CAML and is quite concise (less than 1500 lines). A running program not only shows the actual feasibility of our method, but is also a great facility for building examples, and thus might allow for a finer study of the algorithm.

Data types as well as functions closely follow the previous description of the algorithm: writing the program in such a way tends to minimise the number of bugs, and speaks up for the robustness of the whole proof development.

The program consists of several successive parts:

1. Parsing phase: turns the input text into a concrete syntax tree. The input is a system F typing judgement, in a Church style syntax with type annotations at the binders. It is changed into the de Bruijn notation, and parameterised with fresh parameters. Finally, the abstract tree is decorated with parameterised types at each node.

2. Constraints generation: performs explorations on the tree and generates the boolean, linear and mixed constraints.

3. Boolean constraints resolution: gives the minimal solution of the boolean constraints, or answers negatively if the set admits no solution.


We use a solver employing the simplex algorithm to solve the linear constraints. It runs in time $O(2^n)$, which comes in contrast with the previous result of polynomial time solving, but has proven to be the best in practice.

We now have to define the objective function that we will give to the solver. Basically, to minimise the resulting complexity bound, we should have an objective function which minimises the nesting depth of the boxes of the typed term. To achieve this, we would have to minimise the maximum of the sums of door parameters from the root to each node (this corresponds to the depth of the proof-net, which yields the bound of Theorem 2.1). This clearly calls for a minimax objective function. Unfortunately, this does not fit into the linear programming setting that we are currently using: our objective function can only be a ponderated sum of variables.

So, we chose to simply put as objective function the sum of door parameters. A little trick is needed in order to handle the case of variables which are not of positive domain, and could lead to the absence of an optimal solution. Once this special case is handled, the solver always gives sensible results in practice.

We put some extracts of the source code in Appendix A. The full program, together with some examples, is available at:

http://www-lipn.univ-paris13.fr/~atassi/

2.7.2 Two examples: reversing of list and predecessor

List reversing.

Let us consider the reversing function on binary words. It can be defined by a single higher-order iteration on the type $WF$, with the untyped term $\lambda w. \lambda s o. \lambda s i. (w) step_0 \ step_1 \ base$, with:

- base term: $base = \lambda z. z$,
• step terms: \( step_0 = \lambda a. \lambda x. (a)(so)x, \) \( step_1 = \lambda a. \lambda x. (a)(si)x \).

We obtain as system F term the following one, denoted \( \text{rev} \):

\[
\lambda l \Lambda \alpha. \lambda so. (l \rightarrow \alpha) \rightarrow (l \rightarrow \alpha). \Lambda so. (l \rightarrow \alpha) \rightarrow (l \rightarrow \alpha). \Lambda so. (l \rightarrow \alpha) \rightarrow (l \rightarrow \alpha).
\]

\[
\Lambda a. \lambda x. (a)(so)x
\]

\[
\Lambda a. \lambda x. (a)(si)x
\]

\[
\Lambda z. z
\]

As discussed in Section 2.6.2 to obtain a meaningful typing we need to force the domain of the term to be that of binary words. For that a simple way is to apply the term to a particular argument, for instance: \( \Lambda \alpha. \lambda so. (l \rightarrow \alpha) \rightarrow (l \rightarrow \alpha). \Lambda so. (l \rightarrow \alpha) \rightarrow (l \rightarrow \alpha). \Lambda so. (l \rightarrow \alpha) \rightarrow (l \rightarrow \alpha). \Lambda so. (l \rightarrow \alpha) \)

\[
\Lambda z. z
\]

\[
\Lambda z. z
\]

representing the word \( 1010 \). Since \( \text{rev} \) involves higher-order functionals and polymorphism, it is not so straightforward to tell, just by looking at the term structure, whether it works in polynomial time or not.

Given \( \text{rev}(1010) \) as input, our program produces 200 (in)equations on 76 variables. After constraint solving, we obtain the result:

\[
(\lambda l \Lambda \alpha. \lambda so. (l \rightarrow \alpha) \rightarrow (l \rightarrow \alpha). \Lambda so. (l \rightarrow \alpha) \rightarrow (l \rightarrow \alpha). \Lambda so. (l \rightarrow \alpha) \rightarrow (l \rightarrow \alpha). \Lambda so. (l \rightarrow \alpha) \)
\]

\[
\Lambda a. \lambda x. (a)(so)x \Lambda z. z
\]

It corresponds to the natural depth-1 typing of the term \( \text{rev} \), with conclusion type \( W_{DLAL} \rightarrow W_{DLAL} \). The solution ensures polynomial time termination, and in fact its depth guarantees normalisation in a quadratic number of \( \beta \)-reduction steps.

Predecessor on unary integers.

We now turn to another example which illustrates the use of polymorphism: the predecessor function on unary integers.

We consider a slight simplification of the term given by Asperti (Asp98). The simplification is not needed for typability, but is just chosen to facilitate readability.

For that we consider:

• pairs represented in the following way: \(< P, Q > : \lambda z. (z) P Q , \)

• terms for projection and an application combinator for pairs:

\[
\begin{align*}
\text{fst} & = \lambda x. \lambda y. x, \\
\text{snd} & = \lambda x. \lambda y. y, \\
\text{appl} & = \lambda x. \lambda y. (x)y.
\end{align*}
\]

We will do an iteration on type \( N_F \), with:

• base term: \(< I, x > \) (where \( I = \lambda x. x \)),

• step term: \( \lambda p. < f, (p) \text{appl} > \).

The untyped term will then be \( \lambda n. ((n) \text{step base}) \text{snd} \).

Let us specify the system F typing of the subterms:

• \(< P, Q > : \lambda z. (\beta \rightarrow \alpha) \rightarrow (l \rightarrow \alpha). (z) P^{\beta \rightarrow \alpha} Q^{\beta} : ((\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)) \rightarrow \beta , \)

• \( \text{snd, appl} : (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta , \)

• \( \text{fst} = \lambda x. \lambda y. x, \)

• \( \text{snd} = \lambda x. \lambda y. y, \)

• \( \text{appl} = \lambda x. \lambda y. (x)y . \)
2.7 Implementation and Examples

- \( \text{step} = \lambda p ((\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)) \rightarrow \beta \lambda z ((\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)) \rightarrow (\beta \rightarrow \beta) \),

- \( \text{base} = \lambda z ((\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)) . f (\beta \rightarrow \beta) x : ((\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)) \rightarrow \beta . \)

The overall F-typed term for predecessor, denoted \( \text{pred} \) is thus:

\[
\lambda n^\forall. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) . \lambda \beta . \lambda f^\beta \rightarrow \beta . \lambda x^\beta . \\
( (n ( (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)) \rightarrow \beta ) \lambda p^\beta \rightarrow \beta . \lambda z^\beta \rightarrow \beta . f (\beta \rightarrow \beta) (z) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta ) \\
\lambda x^\beta \rightarrow \beta . \lambda y^\beta . y .
\]

Observe that this term is linear (as Aserti’s original one). Again, to force a meaningful typing we apply the term \( \text{pred} \) to a Church integer argument, here the integer \( 2 \).

The program then produces 220 equations, for 130 parameters. The solver produces a solution, yielding the following type for the subterm \( \text{pred} \):

\[
(\forall \alpha . (\alpha \rightarrow \alpha) \Rightarrow \exists (\alpha \rightarrow \alpha)) \Rightarrow (\forall \alpha . (\alpha \rightarrow \alpha) \Rightarrow \exists (\alpha \rightarrow \alpha)) ,
\]

which corresponds to the \( N_{DLAL} \rightarrow N_{DLAL} \) type.

2.7.3 Experiments with larger examples: polynomials

In order to test our type inference program with larger examples it is interesting to consider a family of system F terms of increasing size. The family of terms representing polynomial functions over unary integers is a natural candidate for this goal, in particular it is important for the encoding of polynomial time Turing machines in the system (\cite{AR02, BT04}).

Therefore we wrote a CAML program which given a polynomial \( P \) outputs a system F term representing \( P \) and with type \( NF \rightarrow NF \), that can then be fed to the DLAL type inference program.

There is however a subtlety that needs to be stressed. Recall that in order to represent polynomial functions in LAL or DLAL with suitable types it is necessary to use type coercions (\cite{AR02, BT04}). These coercions are needed just for typing reasons, and not for computational ones. However, if we consider the system F terms underlying the LAL or DLAL terms for polynomials the coercions are still present and correspond to explicit subterms.

So if we want our system F terms representing polynomials to be typable in DLAL we need to anticipate on the need for coercions. Therefore our program generating system F terms for polynomials is guided by the encoding of polynomials in DLAL, in particular it takes into account the placement of subterms for coercions (even if the terms are not yet typed with modalities during this phase). It should be stressed that this increases considerably the size of the resulting term: in practice inside the resulting term the subpart accounting for the management of coercions is larger than the subpart performing a computational task... This makes however a good test for our type inference program, since the typing is not trivial and will put into use a large number of parameters and constraints.

In the following we will:

- describe the encoding of polynomials used,
- report on experiments of our type inference program on terms of this family.
CHAPTER 2. TYPE INFERENCE IN DUAL LIGHT AFFINE LOGIC

Encoding of polynomials.

We recall from [BT04] the rules for coercions on type \( N_{DLAL} \) derivable in DLAL:

\[
\begin{align*}
\frac{n : N_{DLAL}; \Delta \vdash t : A}{m : N_{DLAL}; \Delta \vdash C_1[t] : \hat{\hat{A}}} \quad \text{(Coerc-1)} \\
\frac{\Delta ; n : N_{DLAL} \vdash t : A}{\Gamma ; m : N_{DLAL} \vdash C_2[t] : A} \quad \text{(Coerc-2)}
\end{align*}
\]

where \( C_1[\_] \) and \( C_2[\_] \) are contexts, which contain as free variables some variables of the environments:

\[
C_1[x] = (m(\lambda g.\lambda p.(g (\text{succ} \ p))))(\lambda n.x)\emptyset,
C_2[x] = (\lambda n.x)(m \text{ succ} \emptyset).
\]

Here \( \text{succ} \) is the usual term for successor.

Similarly we define the term \( \text{coerc} = \lambda n.(n) \text{ succ} \emptyset \), which can be given in DLAL any type \( N_{DLAL} \Rightarrow \hat{\hat{N}}_{DLAL} \).

Multiplication can be represented by the (untyped) term \( \text{mult} = \lambda n.\lambda m.n \text{ succ} \) with \( u = ((m) \lambda k.\lambda x.(n) f(k) x)\emptyset \). It can be given in DLAL the type \( N_{DLAL} \Rightarrow \hat{\hat{N}}_{DLAL} \).

Now, in order to give multiplication any type: \( \hat{\hat{k}} N_{DLAL} \Rightarrow \hat{\hat{k}} N_{DLAL} \Rightarrow \hat{\hat{k}+2} N_{DLAL} \) with \( k \geq 0 \) we can use coercions:

\[
\begin{align*}
\frac{n : N_{DLAL}; m : N_{DLAL} \vdash u : \hat{\hat{k}} N_{DLAL}}{...} \quad \text{(Coerc-1)} \\
\frac{n_2 : N_{DLAL}; m_2 : N_{DLAL} \vdash C_1[u] : \hat{\hat{N}}_{DLAL}}{...} \quad \text{(Coerc-2)} \\
\frac{n_2 : N_{DLAL}; m_2 : N_{DLAL} \vdash C_2[C_1[u]] : \hat{\hat{N}}_{DLAL}}{...} \quad \text{(Coerc-3)}
\end{align*}
\]

Note that there is here a small abuse of notation as now the free variable of \( C_1[\_] \) is called \( n_2 \) (similarly for \( C_2[\_] \)).

We will associate to each polynomial \( P \) of \( \mathbb{N}[X] \) a system F term \( t_P \) of type \( N_F \rightarrow N_F \) representing it, and which is typable in DLAL. We first describe the encoding of monomials.

We define the term \( t_{X^n} \) by induction on \( n \):

\[
t_{X^0} = \lambda x.1, \quad t_{X^1} = \lambda x.x, \quad t_{X^{n+1}} = \lambda x.\lambda n.\lambda m_2.C_1[(\lambda n_2.\lambda m_2.C_2[C_1[u]]) \ (t_{X^n}) \ x \ (\text{coerc} \ x)],
\]

for \( n \geq 1 \).

The term \( t_{X^n} \) can be given in DLAL the type \( N_{DLAL} \Rightarrow \hat{\hat{N}}_{DLAL} \). Actually a better encoding of monomials could be given, with a lower depth, but we stick here to this one for simplicity. To show that \( t_{X^n} \) can be typed with \( N_{DLAL} \Rightarrow \hat{\hat{N}}_{DLAL} \) note that it is easy to observe for \( t_{X^0} \) and \( t_{X^1} \), and supposing it for \( t_{X^n} \), we get for \( t_{X^{n+1}} \) (\( N \) in this derivation stands for \( N_{DLAL} \)):
In the array $N$ (resp. $N'$) stands for $N_{DLAL}$ (resp. $N'_{DLAL}$).

Figure 2.8: Type inference for terms representing polynomials.

Now, once $t_{X^n}$ has been defined it is easy to represent monomials with coefficient, $\alpha_n X^n$ and, using the term for addition and coercions again, arbitrary polynomials: $\sum_{i=1}^j \alpha_i X^{\alpha_i}$.

Experiments of type inference.

We wrote a small program implementing this encoding, which, given a polynomial, produces a system F term representing it. Then we used it to test our DLAL type inference program. We give the results of the experiments on a few examples, in the array of Figure 2.8 where $t_P$ denotes the F term representing a polynomial. Again we stress that the large size of $t_P$, even for small polynomials, is due to the coercions (for instance the encoding of $x^5$ without coercions produces a term of size 322 — which is not typable however) and to the fact that all types are written explicitly in the term, since it is written in à la Church style syntax.

In the array the following quantities are reported:

- the size of $t_P$ is the number of symbols of the term;
- the column # Par stands for the number of parameters in the resulting parameterised term,
- the time (in seconds) for generating the set of linear constraints is divided in two parts:
  - GEN is the time taken by the program for parsing the input, generating the whole constraints and solving the boolean part,
  - SIMPL is the time taken to simplify the set of constraints (this is a preprocessing before using the solver).
- # Cons is the cardinality of the set of of linear constraints generated by the program,
- Sol is the time taken by the solver (LPsol) to solve the set of constraints.

Recall that $N'_{DLAL} = \forall \alpha.(\alpha \rightarrow \alpha) \Rightarrow \alpha \rightarrow \alpha$. We think that the fact that we obtain a $N'_{DLAL}$ type instead of $N_{DLAL}$ on the right-hand-side is not significative here: we could force obtention of a $N_{DLAL}$ type instead by techniques similar to that of domain specification of Section 2.6 (adding a constraint of the form $p = 0$).

Note that the type obtained is slightly smaller (containing fewer $\wedge$ and of smaller depth) than the one described above: we obtain the type $N_{DLAL} \rightarrow \wedge^{3n-3} N_{DLAL}$ for $t_{X^n}$, and it is possible to check that this is indeed a suitable type in general.
Observe that on these examples the respective times needed for generating the constraints and solving the boolean part (GEN) on the one hand, and for simplifying the linear constraints (SIMPL) on the other, are comparable. The time needed to solve the linear constraints (Sol) is comparatively smaller.

We also generated the system F terms representing the same polynomials but without subterms for coercions, and noted with our program that type inference in DLAL for these terms fails: coercions are indeed necessary. Even though the family of terms $t_X$ is a particular case, these examples illustrate the fact that our algorithm is manageable with lambda-terms of reasonable size, and gives results in a sensible time.

2.8 Discussion and further work for the case of propositional DLAL

It should be stressed that our method can be applied to type untyped lambda-terms in propositional (quantifier-free) DLAL. Indeed, propositional DLAL can naturally be seen as a subsystem of DLAL. Given an untyped term $t$, we can thus proceed in the following way (in the lines of previous works for EAL or LAL like [CM01, Bai02]):

1. search for the principal simple type of $t$,
2. using the principal simple type derivation of $t$, search for a valid DLAL decoration by using our method.

If we find a suitable decoration then it will give a derivation in propositional DLAL (simply because the underlying system F derivation does not use quantification). It can be checked that this method is complete (for instance by a simple adaptation of the argument in [Bai01]): if the term is typable in propositional DLAL, then a suitable decoration of the principal simple type decoration will be found.

However, the bound on this procedure given by Theorem 2.21 is polynomial w.r.t. the size of the principal simple type derivation of $t$, and not w.r.t. to the size $|t|$ of the untyped term $t$ itself.

Still, we strongly believe that our method can be adapted in order to give an algorithm performing type inference in propositional DLAL for an untyped term $t$ in time polynomial in $|t|$. The starting point is that it is known that simple type inference can be done in polynomial time by using a shared representation of types. If one designs an algorithm performing together simple type inference and decoration with parameters, one can presumably obtain, instead of a free decoration of $t$, a suitable decoration with a number of parameters polynomial in $|t|$ (by taking advantage of the shared representation of types) and a constraints system also polynomial in $|t|$. Hence in the end type inference would be polynomial w.r.t. $|t|$. We also believe that in this way we would obtain a notion of principal propositional DLAL type. This would be analogous to the work of [CRDR05] for EAL, but could give a single principal type scheme instead of a finite family of principal type schemes.

However in the present work we preferred to follow the approach starting with a system F typed term in order to be able to consider second-order DLAL typing, which is more interesting for expressivity reasons (propositional DLAL is not complete for polynomial time computation). The case of polynomial time type inference for propositional DLAL is left for future work.
2.9 Conclusion

We showed that typing of system F terms in D LAL can be performed in a feasible way, by reducing typability to a constraints solving problem and designing a resolution algorithm. This demonstrates a practical advantage of D LAL over LAL, while keeping the other important properties. We illustrated the manageability of our algorithm by implementing it in CAML and giving some examples of type inference. Note that other typing features could still be automatically inferred, like coercions (as in the next chapter for the case of EAL).

This work illustrates how Linear logic proof-net notions like boxes can give rise to techniques effectively usable in type inference, even with the strong boxing discipline of D LAL, which extends previous work on EAL.
Chapter 3

Type inference with Coercions

3.1 Introduction

The previous chapter dealt with linear type decoration for DLAL, starting from system F terms. Now we will move on to another light logic, Elementary affine logic EAL, and its extension.

**Background** EAL has been introduced in [Asp98, AR02], as a simplification of Elementary linear logic ([Gir98]). It might be seen as a subsystem of Linear Logic, in which box placement (ie promotion) follows a particular discipline. As a result, the cut-elimination of the proofs is bounded by an elementary function. Furthermore, one can encode elementary functions in it, as shown in [DJ03]. So, this can be seen as a logical characterisation of the elementary time complexity class.

When treated as a type system for pure lambda-calculus, the implicative fragment of EAL provides a way of building termination bounds for functional programs. Type inference provides a method to make the process of programming with EAL transparent, and has been now widely studied in [CM06, CRDR05, BT05, CDLRDR05].

Even though EAL is not interesting from a computational complexity point of view, it is often investigated for two reasons: the work done in EAL can often be adapted in LAL; lambda terms typable in EAL admit a simplified Optimal reduction procedure ([Asp98, BCDL07]).

**Motivations** Even with type inference, one has to deal with linear types by the mean of coercions, which really impedes programming naturally and modularly. When we speak of coercions in this system, it is about the transotyping $\iota \rightarrow !\iota$, where $\iota$ is the type of integers. $A \rightarrow !A$ is not true in general in EAL, but it is for integers or booleans: there exists a term, extensionally equivalent to the identity, which performs this transtyping. As soon as one starts programming with EAL, one is confronted to this typing issue. For instance the square function cannot be typed modularly: $\lambda x. (\text{mult}) x x$ is not typable, only $\lambda x. (\text{mult}) x (\text{coerc}) x$ is. The user must be aware of EAL boxes placement to write a typable term, which is not satisfactory for an implicit complexity characterisation where all the process should be transparent to the user. Any work done with light logics, starting with [Gir98], has to deal with coercions placement.

The previous chapter offers yet another example, of an occurrence when it was necessary to
place appropriate coercions to encode polynomials. Not only does it complicate the programming task, but it also results in serious overheads, since the size of a term without coercions — untypable but computationally equivalent — is more than two times smaller than the size of the term with coercions.

So, in the process of extending the ICC tools to reach a point where it will be possible to use them for functional programs static analysis, it seems unavoidable to automatise the coercions placement.

Outline We define in the next section EAL as a type system, equipped with integers and boolean constants. We recall here the principles of the type inference procedure introduced in [BT05]. Then we build the type relation on the top of coercive terms, and extend it to the arrow and bang types. We show that it can be consistently added to EAL with respect to bounded termination. Then we give a type inference algorithm, which starting from a pure lambda term \( t \), computes whether \( t \) admits a typing with implicit coercions or not, and gives its full type if it does.

3.1.1 Some Background

EAL Type System

As a source language, we take the usual simply typed \( \lambda \)-calculus, enriched with constants and corresponding reduction rules; we will use contexts for correctness properties (Note that for brevity, \( !\ldots!A \) is denoted \( !^n A \)):

\[
\begin{align*}
t, u &= \{ \lambda x.t \mid (u) \mid t \mid c, c \in C \} \\
C &= \{ t, f, ite, 0, 1, 2, \ldots, succ, iter \}
\end{align*}
\]

\[
\begin{align*}
T &= \{ \alpha \mid T \rightarrow T \mid ![T] \mid b, b \in B \}
\end{align*}
\]

\[
\begin{align*}
B &= \{ \iota, o \}
\end{align*}
\]

\[
\begin{align*}
\Upsilon &= \{(t, !^n o), (f, !^n o), (ite, !^n o \rightarrow A \rightarrow o A), (0, !^n \iota), (1, !^n \iota), \ldots, \\
&\quad (succ, !^n \iota \rightarrow o !^n \iota), (iter, ![A \rightarrow o] \rightarrow ![A \rightarrow o \iota \rightarrow ![A \rightarrow o \iota]])
\end{align*}
\]

\[
\begin{align*}
C, D &= \square \mid \lambda x.C \mid (C) D \mid x
\end{align*}
\]

For reduction, we use of course the usual \( \beta \)-reduction, but in addition, we have to define reduction rules for constants:

\[
\begin{align*}
\text{succ} \, n &\rightarrow n + 1 \\
\text{ite} \, t \, u_1 \, u_2 &\rightarrow u_1 \\
\text{ite} \, f \, u_1 \, u_2 &\rightarrow u_2 \\
\text{iter} \, f \, u \, n &\rightarrow (f \ldots (f \, u) \ldots)
\end{align*}
\]

The notation \( \rightarrow \) will denote the contextual closure of beta reduction and those rules, \( \rightarrow^* \) denotes the reflexive and transitive closure of \( \rightarrow \) and \( \rightarrow^+ \) its transitive closure. Figure 3.1 presents EAL as a type system, that is to say, in a natural deduction way. In the APPL rule, the two contexts must be disjoint, in the CNTR rule, \( n \) must superior to one and in the WEAK rule, the \( x \) variable must not appear in \( \Gamma \). The "*\+*" is for some limitations introduced to ease the type inference process: in the PROM rule, the \( x_i \) variables must appear exactly once in \( M \). This restricts the sharing to variables, as in the algorithms previously developed in [CM06] [BT05].
3.2 Subtyping Coercions

We define, in two steps, a subtyping relation on \(\text{eal}^*\) types. The first step consists in the design of this relation on (banged) base types, namely \(\iota\) and \(\omega\). We will show that it is correct w.r.t \(\text{eal}^*\), and has the expected properties of an order relation.

### 3.2.1 Coercions

### 3.2.2 On Base Types

In the sequel we will restrict the subtyping relation to the integers type, \(\iota\). Let us define an order relation on base banged types:

**Definition 3.1.** The \(\preceq\) relation is defined as follows:

\[
\forall n, m \text{ s.t. } n \geq 0, m \geq n, !^n \iota \preceq !^m \iota
\]

And enjoys the following properties

**Remark 3.2.** The relation \(\preceq\) is reflexive, transitive and constant-time decidable if one considers integer comparison as an unit operation.

**Correctness**

To show the relation’s correctness, we first exhibit a term performing the coercion: let \(\text{coerc} = \lambda x. \text{iter succ } 0 \ x\), this term can be given any type in the family \(!^n \iota \rightarrow !^{n+m+1} \iota\), with \(0 \leq n\) and \(0 \leq m\), as shown by the following derivation:
And then we show that adding this coercion in the term does not change its computational behaviour, namely that it is observationally equivalent to the term without coercions:

**Definition 3.3.** Given two λ-terms \( t_1 \) and \( t_2 \), we will say that \( t_1 \simeq t_2 \) iff for all \( C \) s.t. \( \vdash C[t_1] : !^k \), \( C[t_1] \mathbin{\to^*} c \Leftrightarrow C[t_2] \mathbin{\to^*} c \), where \( c \) is a constant.

**Lemma 3.4.** If \( t \to_n t' \) then \( t \simeq t' \).

**Lemma 3.5.** For all contexts \( C \), \( C[N] \mathbin{\to^*} c \) if and only if \( C[(\text{coerc} \ N)] \mathbin{\to^*} c \), given that \( C[N] \) and \( C[(\text{coerc} \ N)] \) are closed and of type \( !^k \).

**Proof.** (\( \Rightarrow \) direction) We prove that: if \( C[N] \mathbin{\to^*} c \), then \( C[(\text{coerc} \ N)] \mathbin{\to^*} c \). We prove it by induction on the length \( n \) of the reduction sequence:

- For \( n = 0 \): it means that \( C = [] \) and \( N = c \), and it is easy to check that \( (\text{coerc} \ c) \mathbin{\to^*} c \)

- For \( n' = n + 1 \), let us consider the first redex fired in the reduction:
  - If the redex is inside \( N \), let \( N' \) be the term obtained by firing it, \( C[N] \mathbin{\to^+} C[N'] \) and \( C[N'] \) reduces to \( c \) in \( m \) steps, with \( m \leq n \). We can reduce \( C[(\text{coerc} \ N)] \) to \( C[(\text{coerc} \ N')] \) and apply IH on it to show that it reduces to \( c \) as well
  - If the redex is in a subterm of \( C \), then there exists a term \( W \) s.t. \( C[N] \mathbin{\to} W \) and \( W = C'[N] \), so we can reduce \( C[(\text{coerc} \ N)] \) to \( C'[\text{coerc} \ N] \) and apply IH
  - If the redex is the application of a functional subterm to \( N \), \( C \) is of the form \( C'[t](N, N, \ldots, N) \):
    * If \( N \) is duplicated or just substituted once by the reduction \( C[(t)N, N, \ldots, N] \mathbin{\to} C'[N] \) then we can reduce \( C[(t) \text{coerc} \ N, (\text{coerc} \ N), \ldots, (\text{coerc} \ N)] \) to \( C'[\text{coerc} \ N] \) and apply IH
    * The case is similar if \( N \) is erased since \( C[(t)N, N, \ldots, N] \mathbin{\to} C'[N] \), and \( C[(t)(\text{coerc} \ N), (\text{coerc} \ N), \ldots, (\text{coerc} \ N)] \mathbin{\to} C'[\text{coerc} \ N] \)
  - If the variable substituted by the redex firing occurs free inside \( k \) (with \( k \geq 0 \)) of the \( N \) subterm(s) (remember that replacement of \( [] \) by \( N \) in \( C \) does not take care of variable capture), \( C[N_1, \ldots, N_k, N, \ldots, N] \mathbin{\to} C'[N_1, \ldots, N_k, N, \ldots, N] \), and \( C[(\text{coerc} \ N_1, \ldots, (\text{coerc} \ N_k, (\text{coerc} \ N), \ldots, (\text{coerc} \ N) \mathbin{\to} C'[\text{coerc} \ N_1, \ldots, (\text{coerc} \ N_k, (\text{coerc} \ N), \ldots, (\text{coerc} \ N)], and one can then apply IH

(\( \Leftarrow \) direction) idem:

- base case: \( C = [] \) and \( (\text{coerc} \ N) \mathbin{\to^*} c \) in \( c \) steps, so \( N = c \).

From this general lemma we can deduce:
Lemma 3.6 ($\preceq$ correctness). If $A \preceq B$, then there exists $t$ s.t. $x : A \vdash_{EAL} t : B$ is derivable, and $t \simeq x$.

Proof. $A$ must be of $!^m$ type and $B$ of $!^{m+n}$ type, as follows from the $\preceq$ definition. If $m = 0$ (and thus $A = B$), then $t = x$, and if $m > 0$, then $t$ is simply the (coerce) $x$ term, and according to the previous Lemma, $t \simeq x$. □

3.2.3 General Types

We give the extension of the previous definition to arrow types. First the base cases:

\[
\begin{align*}
\text{(base)} & \quad A \preceq B \\
\text{(var)} & \quad \alpha \preceq \alpha
\end{align*}
\]

The two following rules deal with bangs and arrows.

\[
\begin{align*}
\text{(prom)} & \quad A \preceq B \\
\text{(arrow)} & \quad A_1 \preceq B_1 \quad A_2 \preceq B_2
\end{align*}
\]

The following property is obvious:

Proposition 3.7. $\preceq$ is syntax-directed, deterministic and enjoys the subformula property.

Definition 3.8 (erasure). This is an erasure mapping $(\cdot)^-$ from EAL types to simple types.

- $\alpha^\prec = \alpha$
- $b^\prec = b$
- $(A \to B)^\prec = (A^\prec) \Rightarrow (B^\prec)$
- $(!A)^\prec = (A^\prec)$

Proposition 3.9. If $A \preceq B$ is provable, then $(A^\prec) = (B^\prec)$.

Remark 3.10. $\preceq$ is reflexive: for any type $A$, $A \preceq A$ is provable; $\preceq$ is transitive: if $A \preceq B$ and $B \preceq C$ are provable, then $A \preceq C$ is provable.

Now, we will show correctness of the subtyping relation with respect to EAL. Again, we will use the observational equivalence property.

Proposition 3.11 (correctness of $\preceq$). If $A \preceq B$ is provable, then there exists $t$ such that $x : A \vdash_{EAL} t : B$ derivable in EAL, and $t \simeq x$.

Proof. Induction on the $A \preceq B$ derivation:

- **VAR**: $x : \alpha \vdash : \alpha$ is derivable and obviously $x \simeq x$ ;

- **BASE**: If $A \preceq B$ is derived from the BASE rule, then $A \preceq B$, and so we can conclude by Lemma 3.6 ;

- **PROM**: We have a subtyping derivation $A \preceq B$. By induction hypothesis, there exists $t$ s.t. $x : A \vdash t : B$ is derivable and $x \simeq t$, and we can then apply the PROM rule which does not change the term ;

- **ARROW**: We have a derivation of $A_1 \to A_2 \preceq B_1 \to B_2$. As $B_1 \preceq A_1$ and $A_2 \preceq B_2$ are derivable, we have, by induction hypothesis, the $t_1$ and $t_2$ terms, and the proofs $\sigma_1$ and $\sigma_2$ of $x_1 : B_1 \vdash t_1 : A_1$ and $x_2 : A_2 \vdash t_2 : B_2$. Which allows us to build the following term:
General Proof.

By induction on the proof of $\Gamma \vdash M : A$, except for the coercion rules: $\text{cerc-l}$ can be dealt with in the same way.

We enrich now the eal type system with two coercion rules to obtain $\text{eal}_\leq$:

- **cerc-l**: $x : B, \Gamma \vdash M : A \quad C \leq B \quad \text{cerc-l : } \Gamma \vdash M : A \\
  x : C, \Gamma \vdash \leq M : A$

- **cerc-r**: $\Gamma \vdash M : B \quad B \leq A \quad \text{cerc-r : } \Gamma \vdash \leq M : A$

The main theorem of this part expresses the correctness of this extended type system:

**Theorem 3.12.** If $\Gamma \vdash \leq M_1 : A$, then there exists a term $M_2$ such that $\Gamma \vdash_{\text{eal}} M_2 : A$ is derivable and $M_1 \simeq M_2$.

**Proof.** By induction on the proof of $\Gamma \vdash \leq M_1 : A$. All the cases can be solved simply by applying induction hypothesis, except for the coercion rules:

1. For $\text{cerc-r}$: we have a proof $\sigma_1$ of $\Gamma \vdash \leq M_1 : A$, and a derivation $\sigma_2$ of $A \leq B$, which allows for building the following derivation:

   $$
   \text{cerc-r : } \Gamma \vdash \leq M_1 : B \quad B \leq A
   $$

   From Proposition 3.11 we can build a proof $\sigma'_2$ of $y : B \vdash_{\text{eal}} t : A$ s.t. $y \simeq t$ and by induction hypothesis, a proof $\sigma'_1$ of $\Gamma \vdash_{\text{eal}} M'_1 : B$ s.t. $M_1 \simeq M'_1$.

   $$
   \text{cerc-l : } \Gamma \vdash_{\text{eal}} (\lambda y.t) M'_1 : A
   $$

   So $M_2 = (\lambda y.t) M'_1$ and for all context $C$, $C[M_2] = C[(\lambda y.t)M'_1]$. Now, note that for all $C$, there is a $C_1$ s.t. $C[(\lambda y.t)M'_1] =_{\beta} C_1[t]$. And $C_1[t] =_{\beta} C_1[y] = C[(\lambda y.y)M'_1] =_{\beta} C[M'_1] =_{\beta} C[M_1]$. Hence, $M_1 \simeq M_2$.

2. $\text{cerc-l}$ can be dealt with in the same way:

   $$
   \text{cerc-l : } \Gamma \vdash \leq M_1 : A \quad C \leq B
   $$
3.3 TOWARDS TYPE INFERENCE

Figure 3.2: eal*₁ type assignment system

By Proposition 3.11, we have a term \( t \) s.t. \( x : C \vdash \text{eal} t : B \) is derivable with a proof \( \sigma'_2 \) and \( x \simeq t \) and by induction hypothesis a derivation \( x : B, \Gamma \vdash M'_1 : A \) s.t. \( M'_1 \simeq M_1 \)

\[
\begin{align*}
\sigma'_1 &\xrightarrow{\text{ABST}} x : B, \Gamma \vdash \text{eal} M'_1 : A \\
\sigma'_2 &\xrightarrow{\text{APPL}} \Gamma \vdash \text{eal} \lambda x.M' : B \rightarrow A \\
&\qquad x : A, \Gamma \vdash \text{eal} \ (\lambda x. M'_1) t : A
\end{align*}
\]

So \( M_2 = (\lambda x. M'_1) t \). The variable \( x \) appears bound in \( M'_1 \) and free in \( t \). And for all \( C \), there is a \( C_1 \) s.t. \( C[(\lambda x. M'_1)t] = C_1[t] =_\beta C_1[x] = C[(\lambda x. M'_1)x] = C[M'_1] = C[M_1] \). We can conclude that \( M_1 \simeq M_2 \).

\[\square\]

3.3 Towards type inference

3.3.1 Algorithmic system

Figure 3.2 describes the type assignment system eal*₁ that we will use for type inference (with the same conditions and restrictions as for eal* defined in figure 3.1). Note that there is no explicit COERC rule, and that subtyping is included in the other rules. We want now to show that it is equivalent to the system eal* \(\cup\) \{COERC-L\} \(\cup\) \{COERC-R\}. To achieve this we define:

\[
\begin{align*}
eal*_{\leq} &= \text{eal*} \cup \{\text{COERC-L}\} \cup \{\text{COERC-R}\} \\
eal*_{A-L-R} &= \text{eal*}_{A-L} \cup \{\text{COERC-L}\} \cup \{\text{COERC-R}\} \\
eal*_{A-R} &= \text{eal*}_{A} \cup \{\text{COERC-R}\}
\end{align*}
\]

We will prove that if a term \( M \) with a context \( \Gamma \) admits a type \( A \) in eal*₃, then it admits the same type in eal*₃\(\_\_\_\_\_L-R\), which implies its typability in eal*₃\(\_\_\_\_\_R\). And from the typability in eal*₃\(\_\_\_\_\_R\), we will prove that the term \( M \) can then be typed with type \( A' \) in eal*₃, with \( A' \leq A \). And finally, that this implies typability in eal*₃ with type \( A' \).
CHAPTER 3. TYPE INFERENCE WITH COERCIONS

3.3.2 Equivalences

Let us first define some useful notations:

- We denote \( \text{dom}(\Gamma) \) the set of variables to which \( \Gamma \) gives a type.
- We denote \( \Gamma \leq \Delta \) if \( \text{dom}(\Gamma) = \text{dom}(\Delta) \) and if for all \( (x, A) \in \Gamma \), there exists \( (x, A') \in \Delta \) such that \( A \leq A' \).
- We denote \( \Gamma \uplus \Delta \) the disjoint union of two sets \( \Gamma \) and \( \Delta \).

First, obviously:

**Lemma 3.13.** If \( \Gamma \vdash \leq M : A \) is derivable, then \( \Gamma \vdash_{A-L-R} M : A \) is also.

The following result is an intermediary lemma towards showing that the left coercion rule is redundant in \( \text{eAL}^{A-R}_L \):

**Lemma 3.14.** If \( \Gamma \vdash_{A-R} t : A \) is derivable with a derivation \( \sigma \), then for all \( \Gamma' \) s.t. \( \Gamma' \leq \Gamma \), \( \Gamma' \vdash_{A-R} t : A \) is derivable.

**Proof.** Induction on the derivation \( \sigma \):

- **VAR:** we have a derivation of \( x : A \vdash_{A-R} x : A \), so \( \Gamma = \{(x : A)\} \). For all \( \Gamma' = \{(x : B)\} \) s.t. \( B \leq A \), one can derive:

\[
\frac{B \leq A}{x : B \vdash_{A-R} x : A}
\]

- **ABS:** \( t = \lambda x. u \) and \( A = A_1 \rightarrow A_2 \). We have a derivation ending by:

\[
\frac{\Gamma, x : A_1 \vdash_{A-R} u : A_2}{\Gamma \vdash_{A-R} \lambda x. u : A_1 \rightarrow A_2}
\]

If \( \Gamma' \leq \Gamma \), \( \Gamma' \cup \{(x, A_1)\} \leq \Gamma \cup \{(x, A_1)\} \). We can apply induction hypothesis on the derivation of \( \Gamma' \vdash_{A-R} x : A_1 \vdash_{A-R} u : A_2 \), and then apply the rule **ABS**.

- **APPL:** \( t = (t_1) \) \( t_2 \):

\[
\frac{\Gamma_1 \vdash_{A-R} t_1 : B \rightarrow C \quad \Gamma_2 \vdash_{A-R} t_2 : D \quad D \leq B \quad C \leq A}{\Gamma_1, \Gamma_2 \vdash_{A-R} (t_1) t_2 : A}
\]

\( \Gamma' \leq \Gamma_1 \cup \Gamma_2 \). We define \( \Gamma'_1 \) and \( \Gamma'_2 \) s.t. their domain is equal, respectively to \( \Gamma_1 \) and \( \Gamma_2 \), with \( \Gamma'_1 \leq \text{Gamma}_1 \) and \( \Gamma'_2 \leq \Gamma_2 \).

One can apply induction hypothesis on the two premises and obtain \( \Gamma'_1 \vdash_{A-R} t_1 : B \rightarrow C \) et \( \Gamma'_2 \vdash_{A-R} t_2 : D \). Since \( D \leq B \) and \( C \leq A \), we can apply the rule **APPL**.

- **PROM:** \( t = t'\{u_i/x_i\} \), \( A = !A' \) and we have a derivation ending by:

\[
\frac{\Gamma_1 \vdash_{A-R} u_1 : !A_1 \ldots \Gamma_n \vdash_{A-R} u_n : !A_n \quad x_1 : A_1, \ldots, x_n : A_n \vdash_{A-R} t' : A'}{\Gamma_1, \ldots, \Gamma_n \vdash_{A-R} t'\{u_i/x_i\} : !A'}
\]

Let \( \Gamma'_1, \ldots, \Gamma'_n \) be s.t. \( \Gamma'_1 \leq \Gamma_1, \ldots, \Gamma'_n \leq \Gamma_n \).

By induction hypothesis, \( \Gamma'_1 \vdash_{A-R} u_i : !A_i \) are derivable, and we can apply the promotion rule with the same \( A_i \) for premises of the derivation of \( t' \).
3.3 TOWARDS TYPE INFERENCE

• **CNTR**: if we can contract with a context \( \Gamma, x : C \):

\[
\begin{align*}
\Gamma, x_1 : !B, \ldots, x_n : !B \vdash_{A \rightarrow R} t : C & \quad C \leq !B \\
\Gamma, x : C \vdash_{A \rightarrow R} t : A & \quad \text{CNTR}
\end{align*}
\]

Then one can also contract on the context \( \Gamma', x : D \) such that \( D \leq C \) in the following way:

\[
\begin{align*}
\Gamma', x_1 : !B, \ldots, x_n : !B \vdash_{A \rightarrow R} t : A & \quad D \leq !B \\
\Gamma', x : D \vdash_{A \rightarrow R} t : A & \quad \text{CNTR}
\end{align*}
\]

• **CST**: only applicable in empty context.

• **WEAK, COERC-R**: simply by I.H.

Now, we can state that the left coercion rule is redundant in \( \text{eal}_{A \rightarrow R}^* \):  

**Lemma 3.15.** If \( \Gamma \vdash_{A \rightarrow L \rightarrow R} M : A \) admits a derivation \( \sigma \), then \( \Gamma \vdash_{A \rightarrow R} M : A \) is derivable.

**Proof.** Using Lemma 3.14 on each application of the COERC-L rule.

**Lemma 3.16 (coercion lemma).** If \( \Gamma \vdash_{A} M : A \) has a derivation \( \pi \) and there exists \( B \) s.t. \( A \leq !B \), then \( \Gamma \vdash_{A} M : !B \) is derivable.

**Proof.** Induction on \( \pi \):

- **VAR, CST, APPL:** in those rules, we can coerce the conclusion type, so since \( A \leq !B \) we can conclude by \( !B \) rather than \( A \). For instance on the VAR rule:

  \[
  \text{VAR} \quad \frac{A' \leq A}{x : A' \vdash_{A} x : A}
  \]

  becomes

  \[
  \text{VAR} \quad \frac{A' \leq !B}{x : A' \vdash_{A} x : !B}.
  \]

- **WEAK, CNTR:** simply by applying induction hypothesis.

- **PROM:** if PROM is the last rule applied then \( A \) is of the form \( A = !C \) and \( !C \leq !B \), and we just have to apply the induction hypothesis.

- **ABS:** \( A = A_1 \leadsto A_2 \). There exists no \( B \) s.t. \( A_1 \leadsto A_2 \leq !B \), so the property is true.

**Lemma 3.17.** If \( \Gamma \vdash_{A \rightarrow R} M : A \) is derivable, then there exists \( A' \leq A \) s.t. \( \Gamma \vdash_{A} M : A' \) is derivable.

**Proof.** By induction on the term derivation, following the last rule applied:

- For the rules VAR and CST, we can simply keep them as they are.

- If (COERC-R) is the last rule applied:

\[
\begin{align*}
\Gamma \vdash_{A \rightarrow R} M : B & \quad B \leq A \\
\Gamma \vdash_{A \rightarrow R} M : A
\end{align*}
\]

There exists by LH a derivation \( \Gamma \vdash_{A} M : A' \), with \( A' \leq B \) and by transitivity \( A' \leq A \)

• **APPL:**
\[ \Gamma_1 \vdash_{A-R} M_1 : B \rightarrow C \quad \Gamma_2 \vdash_{A-R} M_2 : D \quad D \leq B \quad C \leq A \]

By induction hypothesis we can derive \( \Gamma_1 \vdash_A M_1 : B' \rightarrow C' \) and \( \Gamma_2 \vdash_A M_2 : D' \) s.t. \( D' \leq D, B \leq B' \), and \( C' \leq C \), which implies by transitivity that \( C' \leq A \) and \( D' \leq B' \), so we can build the following proof:

\[ \Gamma_1, \Gamma_2 \vdash_{A-R} (M_1) M_2 : A \]

- PROM, \( M = M'[M_i/x_i] \) and \( A = !A_1 \):

\[ \Gamma_1 \vdash_{A-R} M_1 : !B_1 \ldots \Gamma_n \vdash_{A-R} M_n : !B_n \]

\[ x_1 : B_1, \ldots, x_n : B_n \vdash_{A-R} M' : A_1 \]

By I.H., for \( 1 \leq i \leq n \) we have derivations \( \Gamma_i \vdash_A M_i : !B_i \), with \( B_i \leq !B_i \). Using Lemma \( 3.16 \) we can turn those derivations in \( \Gamma_i \vdash_A M_i : !B_i \) derivations.

Then one can apply I.H on the derivation of \( M' \) and thus obtain a proof of \( x_1 : B_1, \ldots, x_n : B_n \vdash_{A-R} M' : A_1' \) with \( A_1' \leq A_1 \). Finally we can apply the PROM rule to conclude: \( \Gamma_1, \ldots, \Gamma_n \vdash_A M'[M_i/x_i] : !A_1' \).

- If the last rule applied is WEAK or CNTR, we just have to apply induction hypothesis.

- For the abstraction rule, \( M = \lambda x.M' \) and \( A = A_1 \rightarrow A_2 \), we have a derivation of the form:

\[ \frac{\Gamma, x : A_1 \vdash_{A-R} M : A_2}{\Gamma \vdash_{A-R} \lambda x.M' : A_1 \rightarrow A_2} \]

By I.H we obtain a derivation of \( \Gamma, x : A_1 \vdash_A M' : A_2' \), with \( A_2' \leq A_2 \) and we can apply the ABS rule to derive \( \Gamma \vdash_A \lambda x.M' : A_1 \rightarrow A_2' \) and \( A_1 \rightarrow A_2' \leq A_1 \rightarrow A_2 \).

To get the full equivalence of the different systems the only missing result is:

**Lemma 3.18.** If \( \Gamma \vdash_A M : A \) is derivable, then \( \Gamma \vdash \leq M : A \) also.

**Proof.** Simply observe that rules APPL, CNTR, CST, and VAR of EAL\( ^*_A \) can be decomposed with the former rules of EAL\( _\leq \) and the COERC-L and COERC-R rules.

And so we are now able to state the main result of this part:

**Proposition 3.19.** \( \Gamma \vdash \leq M : A \) is derivable iff \( \Gamma \vdash A M : A' \), with \( A' \leq A \).

**Proof.** By combining previous Lemmas:

- \((\Rightarrow)\) Using Lemmas 3.13, 3.15, 3.17

- \((\Leftarrow)\) Using Lemma 3.18
3.3 TOWARDS TYPE INFERENCE

3.3.3 Pseudo-terms, typability

In this section, we define intermediary criteria to ensure typability. This is quite in the spirit of the technique developed in \cite{BT05} and in chapter 2.

Introduction

To proceed towards type inference, a new syntax is introduced, defining pseudo-terms, which puts in evidence applications of the promotion rule:

\[ t, u = x | \lambda x.t | (t) | !t | \top | c, c ∈ C \]

The type assignment system of Figure 3.2 for lambda terms gives a corresponding type assignment system (also called \( \text{EAL}_A^* \) for pseudo-terms, by keeping all the rules unchanged, except the PROM rule which is changed into:

\[
\frac{\Gamma_1 \vdash M_1 : !A_1 \quad \ldots \quad \Gamma_n \vdash M_n : !A_n \quad x_1 : A_1, \ldots, x_n : A_n \vdash M : B}{\Gamma_1, \ldots, \Gamma_n \vdash ![M_i/x_i] : !B} \quad \text{(PROM)}
\]

We will define a simple erasure map \((\_)^-\) from pseudo-terms to \(\lambda\)-terms: \((x)^- = x\), \((!t)^- = (\top)^- = t^-\) and \((\_)^-\) commutes with other connectives. And whenever \(t^- = t'\), we’ll say that \(t\) is a pseudo-decoration of \(t'\). When there is ambiguity, we will denote usual terms by \(t, u\) and pseudo-terms by \(t^+, u^+\).

The type inference problem is now to find a syntax decoration of the original term actually representing a derivation and ensuring types consistency. Those conditions can be expressed by criteria on the chains of “! ” and “ ! ” for the one hand, and unification constraints on types which include unification on the number of modalities on the other hand.

Local typing

We define a criterion which partially checks the typability of a pseudo-term in \(\text{EAL}_A^*\), which only considers local conditions, of type consistency. Here, we move from sequent calculus to a more abstract setting. First, we need to build a context which assigns a type to all the variables of a term, free or bound. We will call such contexts assignations and denote them by \(\Delta\). Starting from this, we show how to build inductively a function giving a type to all subterms of a pseudo-term and ensuring types consistency. We will call such functions extensions and denote them by \(\Gamma\). This is defined as follows:

\[
\begin{align*}
\Gamma(c) &= B \text{ if } (c, A) ∈ \mathcal{U} \text{ and } A ≤ B \\
\Gamma(x) &= A \text{ if } Δ(x) ≤ A \text{ and } x \text{ has only one occurrence in } t \\
\Gamma(x) &= !A \text{ if } Δ(x) ≤ !A \text{ and } x \text{ has several occurrences in } t \\
\Gamma(!u) &= !A \text{ if } Γ(u) = A \\
\Gamma(\top u) &= B \text{ if } Γ(u) = !B \\
\Gamma(λx.u) &= A → B \text{ if } Δ(x) = A \text{ and } Γ(u) = B \\
Γ((u_1) u_2) &= D \text{ if } Γ(u_1) = A → B, Γ(u_2) ≤ A \text{ and } B ≤ D
\end{align*}
\]

Criterion 3.20 (local typing). A pseudo-term \(t\), an assignation \(Δ\), and an extension \(Γ\) satisfy the local typing criterion if \(Γ(u)\) is defined for all subterm \(u\) of \(t\).
CHAPTER 3. TYPE INFERENCE WITH COERCIONS

Note that the \( \Gamma \) construction follows the form of the rules of the \( \text{eal}_A \) type system, except for the \text{prom} rule. And obviously:

**Lemma 3.21.** If \( t \) satisfies the local typing criterion for an assignation \( \Gamma \) and an extension \( \Delta \), then all its subterms do.

**Boxing**

We present now the criterion ensuring the well-formedness of the \text{prom} rule application. As announced before, it is a criterion on the chain of "\( \! \)" and "\( \! \)" in a term. So, we are led to consider words over the language \( \{ \! \}, \! \} \)∗, equipped with the prefix ordering denoted \( \leq \). Let \( t \) be a pseudo-term and \( u \) be one of its subterms, \( \text{doors}(t, u) \) is the function which gives the word formed by the \( \! \) and \( \! \) occurring in the path from the root of \( t \) to \( u \) and is formally defined as follows:

\[
\text{doors}(t, u) = \epsilon, \text{if } t = u \]
\[
\text{doors}(t', u) = \! :: \text{doors}(t', u) \\
\text{doors}(\! t', u) = \! :: \text{doors}(t', u) \\
\text{doors}(\lambda x.t', u) = \text{doors}(t', u) \\
\text{doors}((t_1) t_2, u) = \text{doors}(t_i, u), \text{where } t_i \text{ is the subterm containing } u
\]

We define a map \( s: \{\!\!,\!\!\}^* \to \mathbb{Z} \) by:

\[
s(\epsilon) = 0, \quad s(\! l) = 1 + s(l), \quad s(\! l) = -1 + s(l).
\]

A word \( l \) is **weakly well-bracketed** if for all \( l' \leq l, s(l') \geq 0 \), and is **well-bracketed** if this condition holds and moreover \( s(l) = 0 \): intuitively, \( \! \) and \( \! \) are resp. opening and closing brackets. It is the same notions as in the previous chapter, except that we considered words overs \( \{\text{§}, \text{¯}\}^* \).

**Criterion 3.22 (bracketing).** Let \( t \) be a pseudo-term. We say that \( t \) satisfies the bracketing criterion if:

(i) for any occurrence of free variable \( x \) in \( t \), \( \text{doors}(t, x) \) is well-bracketed;

(ii) for any occurrence of an abstraction subterm \( \lambda x. v \) of \( t \):

(ii.a) \( \text{doors}(t, \lambda x. v) \) is weakly well-bracketed, and

(ii.b) for any occurrence of \( x \) in \( v \), \( \text{doors}(v, x) \) is well-bracketed.

**3.3.4 Correctness and completeness of criteria**

In this section, we relate formally the fact for a pseudo-term to satisfy the criteria with typability in \( \text{eal}_A^* \).

**Proposition 3.23.** Let \( \sigma \) be a \( \text{eal}_A^* \) derivation of \( x_1 : A_1, \ldots, x_n : A_n \vdash t : A \), where \( t \) is a lambda term, then there exists a pseudo-term \( t^+ \), pseudo-decoration of \( t \), an assignation \( \Delta \) and an extension \( \Gamma \) s.t. \( \Gamma(t^+) = A \) which satisfy the local typing and bracketing criteria.

**Proof.** \( \Delta \) is built as follows:

- \( (x_1 : A_1), \ldots, (x_n : A_n) \in \Delta \)
3.3 TOWARDS TYPE INFERENCE

For each $\lambda$ subterm of $t$ of the form $\lambda y.t' : A \to B$, $(y : A) \in \Delta$

We now define $t^+$ by induction on the $\sigma$ derivation:

- **VAR.** We have a derivation $\frac{B \leq A}{x : B \vdash_A x : A}$
  Then $t^+ = x$ and $\Gamma(x) = A$.

- **CST.** $t = c$, $(c, A) \in \mathcal{U}$ and $A \leq B$, so $t^+ = c$ and $\Gamma(c) = B$

- **WEAK.** $\frac{\Gamma' \vdash_A t : A}{\Gamma', x : B \vdash_A t : A}$
  By I.H, one can build $t^+$, a pseudo-term which satisfies the criteria.

- **CONTR.** We have a derivation $\frac{z_1 : !C, \ldots, z_n : !C, \Pi \vdash t' : A \quad C' \leq !C}{z : C', \Pi \vdash t\{z/z_i\} : A}$
  By I.H., there exists $t'^+$ satisfying both criteria. This rule application preserves the bracketing criterion and ensures that $\Delta(z) = C'$ and that $C' \leq !C$. So $t'^+\{z/z_i\}$ satisfies local typing.

- **ABS.** $A = A' \to A''$, $t = \lambda z.t'$: by I.H., there exists $t'^+$, which satisfies the bracketing and local typing criteria and s.t $\Gamma(t'^+) = A''$. Since $t'^+$ satisfies the bracketing criterion, then $t^+$ also. By definition, $(z : A') \in \Delta$, so $\Gamma(\lambda z.t'^+) = A' \to A'' = A$.

- **APPL.** $t = (t_1) t_2$: Simply apply I.H.: $\Gamma(t_1^+) = C_1 \to C_2$, $\Gamma(t_2^+) = D$. By hypothesis there exists $A$ and $C_1$ in the APPL rule application s.t. $D \leq C_1$ and $C_2 \leq A$, so $t^+ = (t_1^+ t_2^+)$ and $\Gamma(t^+) = A$.

- **PROM.** we have an application of the rule:
  $$\Pi_1 \vdash_A t_1 : !D_1, \ldots, \Pi_n \vdash_A t_n : !D_n \quad y_1 : D_1, \ldots, y_n : D_n \vdash_A t' : A'$$
  $\Pi_1, \ldots, \Pi_n \vdash_A !t'\{t_i/y_i\} : !A'$
  by I.H we can define $t_1^+, \ldots, t_n^+$ and $t'^+$ pseudo terms, satisfying the local typing criterion, which is preserved by the application of the PROM rule. By I.H, for all $i$, $\Gamma(t_i^+) = D_i$, so by definition $\Gamma(!t'^+) = D_i$.

The introduction of $!$ and of the several $\Gamma$s preserve the bracketing criterion:

- (i) : since all the free variables of $t$ belong to the $t_i$’s, which are well-bracketed by induction hypothesis, and adding a $!$ and a $!$ on the path from the root of the term to the free variable preserves this.

- (ii.a) : If there is an abstraction subterm of $t$ which does not belong to one of the $t_i$ subterms, then adding a $!$ preserves the weak well-bracketing criterion.

- (ii.b) : The added $!$ and $\Gamma$s never alter the path between a free variable and its binder: since all free variables of $t$ belong to the $t_i$’s, the path from the binder to the variable crosses the added $!$ and exactly one of the added $\Gamma$s. There is no $\lambda$ subterm of $t$ whose path to one occurrence of its bound variable would cross one of the added $\Gamma$s.

So we can build $t^+ = !t'^+\{t_i^+/y_i\}$ satisfying both criteria.
To show that the bracketing criterion ensures the well-formedness of the PROM rule application, we introduce the matching $!$'s of a term, and the matching subterms of a pseudo-term:

**Definition 3.24 (matching $!$).** For a pseudo-term $!t$ satisfying the bracketing criterion, let us build the words of all doors from the root of this term to each of its leaves (i.e. the variable occurrences of the pseudo-term). Suppose that $!t$ has $n$ variable occurrences, then we have $n$ words of $\{!, !\}^*$, denoted $w_1, ..., w_n$.

It is easy to see that the bracketing criterion implies that for all $i$ s.t. $1 \leq i \leq n$, there is at most one $!$ in $w_i$ matching the root $!$, which is the last $!$ in the smallest word $w'$ s.t. $w' \leq w_i$ and $s(w') = 0$.

The set of those $!$ positions in $!t$ is the set of matching $!$'s of $!t$.

**Lemma 3.25 (boxing).** If $!t$ is a pseudo-term satisfying the bracketing criterion, then there exists a pseudo term $t'$ and $n$ pseudo-terms $t_1, ..., t_n$ s.t.:

- $FV(t') = \{x_1, ..., x_n\}$ and each of these variables occurs exactly once in $t'$
- $!t = !t'[!t_i/x_i]$
- $t'$ and each of the $t_i$'s satisfy bracketing criterion

*Proof.* $t$ has a $!$ at the root. Let us denote $!_1, ..., !_n$ the matching $!$'s of $!t$ and $!_1, ..., !_n$ its matching subterms, following previous definition. By definition, no $t_i$ is a subterm of $t_j$, for $i \neq j$.

To obtain the term $t'$, we replace each matching subterm $!_i$ of $t$ by a distinct fresh variable $x_i$. So, by definition, $!t = !t'[!t_i/x_i]$, and each $x_i$ appears exactly once in $t'$.

Now, suppose that there exists a free variable $z$ in $t'$ s.t. for all $x_i, z \neq x_i$. Then $z$ also occurs free in $t$ and so in $!t$. Since by hypothesis $!t$ satisfies the bracketing criterion, $s\langle doors(!!z) \rangle = 0$.

So there must be a $^1$" in the path from the root of $!t$ to $z$, so $z$ belongs to a $t_i$, which is a contradiction. Therefore, $FV(t') = \{x_1, ..., x_n\}$ and each of the variables occurs exactly once.

To show that $t'$ satisfies the bracketing criterion, observe that for each $t_i$, $s\langle doors(!!t_i) \rangle = 0$, by definition of the matching subterm. And $s\langle doors(!!t_i) \rangle = s(! :: doors(t'_i, x_i) :: !) = 1 + s\langle doors(t'_i, x_i) \rangle - 1 = s\langle doors(t'_i, x_i) \rangle$, so bracketing criterion (i) is verified. For criterion (ii.a), just observe that for all subterms of $t$ of the form $\lambda y. u$ if $doors(!!t, \lambda y. u) = 0$, then $\lambda y. u$ is in one of the $t_i$'s, and if $doors(!!t, \lambda y. u) \geq 1$, then it is still weakly well bracketed in $t'$, so criterion (ii.a) is met. It is straightforward to see that criterion (ii,b) is preserved in $t'$ if it was in $t$.

The bracketing criterion for the $t_i$'s follows from the fact that $t$ itself satisfies it and that for all $t_i$, $s\langle doors(t_i, t_j) \rangle = 0$ by definition of the matching subterms.

**Lemma 3.26.** If a pseudo-term $t$, an assignation $\Delta$ and a context $\Gamma$ satisfy the local typing criterion, we have:

- $t_1, ..., t_n$ be subterms of $t$, with $\Gamma(t_i) = A_i$
- $t'$ s.t. $t = t'[t_i/x_i]$, for $1 \leq i \leq n$, and s.t. the $x_i$ are the only free variables of type $A_i$

We denote:

- $\Delta' = \Delta \cup \{(x_1 : A_1), ..., (x_n : A_n)\}$
- $\Gamma'$ is the union of the restriction of $\Gamma$ to the subterms of $t$ that appear in $t'$, and the set $\{(x_1 : A_1), ..., (x_n : A_n)\}$

Then the pseudo term $t'$ associated with the assignation $\Delta'$ and the context $\Gamma'$ satisfy the local typing criterion.
Proof. By definition of $t'$, $\Delta'$, $\Gamma'$ and of the local typing criterion. \hfill \square

Proposition 3.27. If a pseudo-term $t^+$, an assignation $\Delta$ and its extension $\Gamma$ satisfy the local typing and bracketing criteria, then there exists a derivation $\sigma$ of $x_1 : A_1, \ldots, x_n : A_n \vdash t^+ : A$ in $\text{EAL}_A^\ast$, where $x_1 : A_1, \ldots, x_n : A_n$ is the restriction of $\Delta$ to the free variables of $t^+$.

Proof. Induction on $t^+$:

- $t^+ = c$: $\Gamma(t^+)$ is defined, so there exists $A$ such that $(c, A) \in \mathcal{U}$ and $\Gamma(t) = A$ and $B$ s.t. $A \leq B$. We can build the inference tree:

\[
\begin{array}{c}
\text{cst} \quad \vdash_A c : B \\
\text{VAR} \quad \vdash_A x : A \\
\text{app} \quad \vdash_A x : B
\end{array}
\]

- $t^+ = x$: by hypothesis, $\Gamma(x) = A, \Delta(x) = B, B \leq A$, so one can derive:

\[
\begin{array}{c}
\text{VAR} \quad \vdash_A x : A \\
\text{app} \quad \vdash_A x : B
\end{array}
\]

- $t^+ = (t_1^+) t_2^+$: if $t_1^+$ and $t_2^+$ share a common free variable $x$, we can rename them, to obtain the terms $t'^1$, $t'^2$ and $t'^3$, s.t. $t^+ = t'^1 \{x/x_1\} t'^2 = (t_1^+) t_2^+$. Then by applying the induction hypothesis on $t'^1$ and $t'^2$, we can derive $\Pi_1 \vdash_A t'^1 : A \rightarrow B$ and $\Pi_2 \vdash_A t'^2 : C$ such that $C \leq A$ where $\Pi_1$ and $\Pi_2$ are respected the restriction of $\Delta$ to the free variables of $t'^1$ and $t'^2$, which permits to apply the APPL rule.

For the common free variables, since $t^+$ satisfies the local typing criterion, $\Delta(x) \leq !C$ and we apply the CNTR rule on $t'^-$, and thus to obtain $t^-$.

- $t^+ = \lambda x. u^+$: by I.H. $x_1 : A_1, \ldots, x_n : A_n \vdash_A u^- : A$ is derivable. If there exists $j$ s.t. $x = x_j$ then we can apply the ABS rule. If not, then $x$ does not appear free in $u^-$, so we have to apply the WEAK rule and only then the ABS rule.

- $t^+ = !u^+$: since $t^+$ satisfies the bracketing criteria we can apply the boxing Lemma to obtain a term $t'^+ \quad \text{s.t.} \quad t^+ = t'^+ \{t_1^+/x_i\}$, with $1 \leq i \leq n$, where the $x_i$ are the only free variables of $t'^+$, and where the $t_i^+$s, for $1 \leq i \leq n$, satisfy the bracketing criterion.

Since the subterms of $t^+$ satisfy the local typing criterion, $\Gamma(\{t_i^+\})$ is defined and is of the form $C_i$

So by I.H one can derive $\Pi_i \vdash t_i^- : !C_i$ for $1 \leq i \leq n$. The cases of common free variables between two subterms $t_j^+$ and $t_k^+$ can be solved by the same argument as for the APPL case.

By the boxing Lemma 3.26 $t'^+$ satisfies the bracketing criterion, and by Lemma 3.26 it also satisfies the local typing criterion, so we can apply the induction hypothesis, and obtain $t^- = t'^+ \{t_i^- /x_i\}$ with $x_1 : C_1, \ldots, x_n : C_n \vdash_A t^-$ derivable. This allows the application of the PROM rule.

- $t^+ = !u^+$: $t^+$ does not satisfies the bracketing criterion \hfill \square

Theorem 3.28. A term $t$ is derivable in $\text{EAL}_A^\ast$ under hypothesis $x_1 : A_1, \ldots, x_n : A_n$ if there exists a pseudo decoration $t^+$ of $t$, an assignation $\Delta$ and an extension $\Gamma$ such that:

- $\Gamma(u^+)$ is defined for every subterm $u^+$ of $t^+$

- $t^+$ satisfies the bracketing criterion

Proof. By Proposition 3.27 for the if direction, and by Proposition 3.28 for the only if direction. \hfill \square
3.4 Type Inference

To perform actual type inference we will restrict to a certain subclass of \( \text{eal}^{*}_{A} \) derivations, namely *restricted* derivations, and show that this subclass is sufficient to recognise all the terms typable in \( \text{eal}^{*}_{A} \).

3.4.1 Restricted derivations

**Definition 3.29 (restricted derivation).** An \( \text{eal} \) derivation is said to be restricted iff for any application of the rule \( \text{prom} \), the premises of the form \( \Gamma_{i} \vdash M_{i} : !A_{i} \) (see fig. 3.2) have for last rule \( \text{var} \), \( \text{cst} \) or \( \text{appl} \).

**Remark 3.30.** Lemma 3.16 does not affect the restriction property: if one applies it on a restricted derivation, the derivation obtained is still restricted.

**Lemma 3.31 (substitution lemma).** If \( \Gamma \vdash_{A} M : A \) admits a derivation \( \pi \) and \( x : A, \Delta \vdash_{A} N \), then \( \Gamma, \Delta \vdash_{A} N\{M/x\} : C \) admits a derivation, with \( C \leq B \).

**Proof.** Induction on \( \sigma \).

- **var, \( \sigma \) is of the form:**

  \[
  \frac{A \leq B}{x : A \vdash_{A} x : B}
  \]

  So, \( N = x, \Delta = \emptyset, N\{M/x\} = M \) and by hypothesis, \( \Gamma \vdash_{A} M : A \) is derivable

- **weak:**

  - if the weakening is performed on \( x \), then \( \sigma \) is of the form:

    \[
    \frac{\sigma'}{x : \Delta \vdash_{A} N : B}
    \]

    Then \( N\{M/x\} = N \) (since \( x \notin \Delta \)) and \( \Gamma, \Delta \vdash_{A} N\{M/x\} : B \) is derivable by performing weakening on the variables of \( \Gamma \) and then following the \( \sigma' \) derivation.

  - if the weakening is performed on a variable \( z \) of \( \Delta \), let \( \Delta' \) be s.t. \( \Delta = (z : D) \cup \Delta' \), \( \sigma \) is of the form:

    \[
    \frac{\sigma'}{x : A, \Delta' \vdash_{A} N : B}
    \]

    By I.H. one can derive \( \Gamma, \Delta' \vdash_{A} N\{M/x\} : C \), with \( C \leq B \), and apply the weak rule to obtain \( \Gamma, \Delta \vdash_{A} N\{M/x\} : C \).

- **cntr:** by hypothesis, the contraction is not performed on \( x \). Let \( \Delta' \) be s.t. \( \Delta = (z : D') \cup \Delta' \) we have a derivation of last rule:

  \[
  \frac{x : A, \Delta', z_{1} : !D_{1}, \ldots, z_{n} : !D \vdash_{A} N : B}{x : A, \Delta \vdash_{A} N : B}
  \]

  We can apply I.H to derive \( \Gamma, \Delta', z_{1} : !D_{1}, \ldots, z_{n} : !D \vdash_{A} N\{M/x\} : C \), with \( C \leq D \) and then apply again the \( \text{cntr} \) rule and obtain a proof of \( \Gamma, \Delta \vdash_{A} N\{M/x\} : C \).
3.4 TYPE INFERENCE

- **ABS**: \( B = B_1 \rightarrow B_2, N = \lambda z.N' \). Since \( x \) is a free variable of \( N \) and by \( \alpha \)-conversion one can suppose without loss of generality that \( z \neq x \). We obtain by I.H a derivation of \( \Gamma, x, \Delta \vdash A \, N'(M/x) : D \), with \( D \leq B_2 \) and we can apply the ABS rule to obtain a derivation of \( \Gamma, \Delta \vdash A \, N(M/x) : B_1 \rightarrow D \).

- **APPL**: \( N = (N_1) \, N_2 \). If \( x \) does not appear free in \( N_1 \) nor in \( N_2 \), we can apply the WEAK rule to conclude. Otherwise, \( x \) might appear in \( N_1 \) or in \( N_2 \) (but not both), so we distinguish the cases (with \( \Delta = \Delta_1 \uplus \Delta_2 \)):
  - \( x \in \text{FV}(N_1) \): then we have a derivation of \( x : A, \Delta_1 \vdash_A N_1 : D \rightarrow E \), and a derivation of \( \Delta_2 \vdash_A N_2 : F \), s.t. \( F \leq D \) and \( E \leq B \). By induction hypothesis one can derive \( \Gamma, \Delta_1 \vdash_A N_1(M/x) : D' \rightarrow E' \), s.t. \( E' \leq E \), and \( D \leq D' \), which implies that \( E' \leq B \) and \( F \leq D' \), so we can apply the APPL rule:
    \[
    \Gamma, \Delta_1, \Delta_2 \vdash_A (N_1(M/x)) \, N_2 : B
    \]
    And since \( x \) does not appear free in \( N_2 \), \( (N_1(M/x)) \, N_2 = [(N_1) \, N_2][M/x] \).
  - \( x \in \text{FV}(N_2) \): then we have a derivation of \( \Delta_1 \vdash_A N_1 : D \rightarrow E \), and a derivation of \( x : A, \Delta_2 \vdash_A N_2 : F \), s.t. \( F \leq D \) and \( E \leq B \). By induction hypothesis one can derive \( \Gamma, \Delta_2 \vdash_A N_2(M/x) : F' \), s.t. \( F' \leq F \), which implies that \( F' \leq D \), so we can apply the APPL rule:
    \[
    \Gamma, \Delta_1, \Delta_2 \vdash_A (N_1 N_2(M/x)) : B
    \]
    And since \( x \) does not appear free in \( N_2 \), \( (N_1 N_2(M/x)) = [(N_1) \, N_2][M/x] \).

- **PROM**: \( N = !N'[t_i/y_i], B = !B' \), let \( \Delta = \Delta_1 \uplus \ldots \uplus \Delta_n \), since by hypothesis \( x \) has only one occurrence in \( N \), the last rule is of the form, for some \( k \):

  \[
  \begin{array}{c}
  \Delta_1 \vdash_A t_1 : !D_1 \ldots \Delta_k, x : A \vdash_A t_k : !D_k \ldots \Delta_n \vdash_A t_n : !D_n \\quad y_1 : D_1, \ldots, y_n : D_n \vdash_A N' : B' \\
  \end{array}
  \]

  \[
  x : A, \Delta \vdash_A N'[t_i/y_i] : !B'
  \]

  By I.H, there exists an \( E \) s.t \( \Gamma, \Delta_k \vdash_A t_k(M/x) : E \) is derivable, and \( E \leq !D_k \). And so, using Lemma 3.16 we can derive:

  \[
  \begin{array}{c}
  \Delta_1 \vdash_A t_1 : !D_1 \ldots \Gamma, \Delta_k \vdash_A t_k(M/x) : !D_k \ldots \Delta_n \vdash_A t_n : !D_n \\quad y_1 : D_1, \ldots, y_n : D_n \vdash_A N' : B' \\
  \end{array}
  \]

  \[
  \Gamma, \Delta \vdash_A N'[t_1/y_1, \ldots, t_k(M/x)/y_k, \ldots, t_n/y_n] : !B'
  \]

- **CST**: not applicable in non-empty context.

\[\square\]

**Lemma 3.32.** If \( \Gamma \vdash_A M : A \) has a derivation \( \sigma \), then there exists a restricted derivation of conclusion \( \Gamma \vdash_A M : B \) with \( B \leq A \).

**Proof.** By induction on \( \sigma \), the only involved case is the one of PROM:

We have a proof of the form:

\[
\Gamma_1, \ldots, \Gamma_n \vdash_A M'_n \vdash_A M_n \vdash_A M_n : !C_n \quad x : C_1, \ldots, x_n : C_n \vdash_A M' : A'
\]

\[
\Gamma_1, \ldots, \Gamma_n \vdash_A M'[M_n/x_1] : !A'
\]
With $\Gamma = \Gamma_1 \uplus \ldots \uplus \Gamma_n$, $M = M'[M_i/x_i]$ and $A = !A'$.

Let $\sigma_1, \ldots, \sigma_n$ be the proofs of the $\Gamma_i \vdash A M_i : !C_i$, and $\tau$ be the proof of $x : C_1, \ldots, x_n : C_n \vdash A M' : A'$.

By I.H, there exists restricted proofs $\sigma'_1, \ldots, \sigma'_n$ (by Lemma 3.16 and Remark 3.30) we can suppose them to be of type $!C_i$ and $\tau'$ (with a final type $B' \vdash B' \leq A'$) of those sequents. We reason by case on the last rule of the $\sigma'_i$ derivations:

- **weak and cntr**: we can postpone the application of those rules, ie perform them only after the promotion.
- **abs**: impossible, since the derivation would end up with an arrow type which cannot be in a subtyping relation with a bang type.
- **prom**: we have a proof of the form:

$$
\Gamma_1 \vdash A M_1 : !C_1 \ldots \Gamma_n \vdash A M_n : !C_n \quad \Theta \vdash A M'_1 : C_k
$$

$$
\frac{\Delta_i \vdash A N_i : !C'_i}{\Gamma_1, \ldots, \Gamma_n \vdash A M'_1 \{N_i/z_i\} : !C_k \ldots \Gamma_n \vdash A M_n : !C_n} \quad \Theta \vdash A M' : B' \quad \Pi \vdash A B'
$$

With $\Theta = z_1 : C'_1, \ldots, z_m : C_m$, $\Pi = x_1 : C_1, \ldots, x_n : C_n$.

We define $\Pi'$ to be s.t $\Pi = \{(x_k : C_k) \cup \Pi'$, so we can apply the substitution Lemma (since the $x_i$ appear exactly once in $M'$, because of the “*” restriction) on $x_k : C_k, \Pi' \vdash A M' : B'$ and $\Theta \vdash A M'_k : C_k$ to obtain a derivation of $\Theta, \Pi' \vdash A M'_k \{M_i' / x_k\} : B''$, where $B'' \leq B'$.

We then substitute the $\Gamma_k \vdash A M'_k \{N_i/z_i\} : !C_k$ derivation by its premises $\Delta_i \vdash A N_i : !C'_i$ and thus can apply the PROM rule.

$$
\frac{\Gamma_1 \vdash A M'_1 \{M'_k / x_k\} \{M'_1 / x_1, \ldots, M'_k / x_k, \ldots, M'_n / x_n\} : !B''}{\Gamma_1, \ldots, \Gamma_n \vdash A M'_1 \ldots \Gamma_n \vdash A M'_n \Theta \vdash A M' \{M'_k / x_k\} : B''}
$$

By the substitution Lemma, $B'' \leq B'$, and by induction hypothesis, $B' \leq A'$ and so $!B'' \leq !A'$, and $M'\{M_1/x_1, \ldots, M'_k[N_i/z_i]/x_k, \ldots, M_n/x_n\}$ is equal to $M'\{M_i/x_i\}$. Then it is easy to observe that $M'\{M'_k/x_k\}\{M'_1 / x_1, \ldots, M'_{k-1}/x_{k-1}, N_i/z_i, M'_k/x_{k+1}, \ldots, M'_n/x_n\}$ is equal to $M'\{M_1/x_1, \ldots, M'_k[N_i/z_i]/x_k, \ldots, M_n/x_n\}$ since the $z_i$ variables appear only in $M'_k$.

- the cases of rules appl, var and cst do not contradict the restriction condition.

There is a counterpart of restricted derivations on the pseudo-terms side: a pseudo-term yield by a restricted derivation has the property of having no subterm of the form $!!t'$ or $!!t$. This will come in handy for the type inference itself allowing for the use of parametrised pseudo-terms.

### 3.4.2 Parametrised types and pseudo-terms

**Preliminary definitions and notations**

Performing type inference for a lambda-term $t$ now amounts to finding a correct (i.e. respecting the local typing and bracketing criteria) pseudo-term $t^+$ being a pseudo-decoration of $t$. It is easy to see that the set of pseudo-decorations is infinite (even in the frame of restricted
derivations). That is why we will introduce integer parameters in terms: variables \( n, n_1, n_k, \ldots \) that will denote parameters ranging on \( \mathbb{Z} \). We also introduce similar parameters for types, denoted by \( p, p_1, p_k, \ldots \) and that will range on \( \mathbb{N} \). The linear combinations of those parameters (which can include integer constants) \( p_1 + n_1 + 1 + \ldots \) will be denoted by \( c, c_1, c_k, \ldots \). Then we will use those linear combinations to express linear (in)equations which will be denoted \( c \geq 1, c \equiv 0 \) or more concretely \( n + p \geq 1 \).

Using those integer variables, we introduce parametrised types and terms:

\[
B = \alpha \mid A \to A \mid C, C \in \mathcal{B} \\
A = !^n B \\
t = x \mid \lambda x. u \mid (u) u \mid c, c \in \mathcal{C} \\
u = !^n t
\]

The types defined by the \( B \) line of the grammar will be used for instance in substitutions and be called head-free parametrised types. We denote them by \( B_1, B_k, \ldots \). The \( A \) line represent regular parametrised terms, with a parameter in head of the type, and we will denote them by \( A_1, A_k, \ldots \).

The free decoration, from lambda-terms and simple types to parametrised terms and parametrised types is the map denoted by \( \overline{\Gamma} \) and defined as follows:

\[
\begin{align*}
\overline{x} &= !^n x \\
\overline{\lambda x. t} &= !^n \lambda x. \overline{t} \\
\overline{(t_1) t_2} &= !^n (\overline{t_1}) \overline{t_2} \\
\overline{c} &= !^n c \\
\overline{\alpha} &= !^p \alpha \\
\overline{A \to B} &= !^p (\overline{A} \to \overline{B}) \\
\overline{c} &= !^p c, c \in \mathcal{B}
\end{align*}
\]

Where all the parameters chosen to decorate terms and types are chosen to be all distinct from each other.

An instantiation will be a map from parametrised types (resp. parametrised terms) to restricted linear types (resp. restricted pseudo-terms), meaning that we give an integer value to each parameter. Instantiations will be denoted by \( \psi \) or \( \phi \). \( A\phi \) (resp. \( t\phi \)) is the type (resp. restricted pseudo-term) obtained by replacing each integer parameter of \( A \) (resp. \( t \)) by its value in \( \phi \). We can extend instantiations to typing contexts: given a context \( \Gamma \), \( \Gamma\phi \) is the context obtained by instantiating the types of all variables of \( \Gamma \).

Substitutions will be maps from type variables to head-free parametrised types, will be denoted by \( \sigma \) or \( \tau \). Most of the time, we note \( \sigma(A) \) the application of the substitution \( \sigma \) to the type \( A \), but when a concrete notation is needed, we write \( A \bullet \{ \alpha_1 \leftarrow B_1; \ldots; \alpha_n \leftarrow B_n \} \) for the application of the \( \{ \alpha_1 \leftarrow B_1; \ldots; \alpha_n \leftarrow B_n \} \) substitution to the parametrised type \( A \), where the \( B_i \)'s are head-free parametrised types. This concrete notation is used for instance in the definition:
Again, we extend this notion to typing contexts: given a context $\Gamma$, $\sigma(\Gamma)$ is the context obtained by applying the substitution $\sigma$ to the types of all variables of $\Gamma$.

We extend the forgetful map of types on substitutions: if $\sigma = \{\alpha_1 \leftarrow B_1; \ldots; \alpha_n \leftarrow B_n\}$, $\sigma^-$ is the substitution obtained by applying the forgetful map to all types of $\sigma$: $\sigma^- = \{\alpha_1 \leftarrow B_1^-; \ldots; \alpha_n \leftarrow B_n^-\}$.

Finally we will rely on first-order unification constraints production and resolution to perform type inference, we will denote the subtyping unification relation by $\leq$, and a constraint by, for instance, $A_1 \leq A_2$.

**Unification**

To compute the most general unifier, we need a unification procedure, which taking two types, produces the necessary substitutions and linear constraints. This procedure takes in input two parametrised types, and produces in output $\text{false}$ if the unification fails, or a set of linear constraints and substitutions, respectively denoted by $C$ and $\sigma$ if it succeeds. The set is noted $\{C; \sigma\}$. In practice we will denote it concretely for instance by $\{p_1 \leq p_2, p_k \leq p; \alpha \leftarrow B\}$. A set containing $\text{false}$ is said to be equal to $\text{false}$.

The following definition does not closely follow the inductive grammar of parametrised types. However, it is easy to check that all the possible cases are handled, despite this presentation. The last line describes the failure case, and the two lines before the particular cases in which subtyping actually takes place.

**Definition 3.33 (unification).**

\[
\begin{aligned}
U(\text{!}^{c_1}B_1 \leq \text{!}^{c_2}B_2) &= \{c_1 \leftarrow c_2\} \cup U(B_1 \leq B_2), B_1 \neq \iota, B_2 \neq \iota \\
U(\alpha \leq \alpha) &= U(\alpha \leq \alpha) = \emptyset, \emptyset \\
U(\alpha \leq \alpha) &= U(\alpha \leq \alpha) = \{; \alpha \leftarrow \alpha\} \\
U(\alpha \leq \beta) &= \{; \alpha \leftarrow \beta\} \\
U(\alpha \leq (A_1 \rightarrow A_2)) &= \{; \alpha \leftarrow \text{!}^{c_1}\alpha_1 \rightarrow \text{!}^{c_2}\alpha_2\} \cup U(A_1 \leq \text{!}^{c_1}\alpha_1) \\
&\quad \cup U(\text{!}^{c_2}\alpha_2 \leq A_2), \alpha \notin (A_1 \rightarrow A_2), \text{!}^{c_1}\alpha, \text{!}^{c_2}\alpha \text{ fresh} \\
U((A_1 \rightarrow A_2) \leq \alpha) &= \{; \alpha \leftarrow \text{!}^{c_1}\alpha_1 \rightarrow \text{!}^{c_2}\alpha_2\} \cup U(\text{!}^{c_1}\alpha_1 \leq A_1) \\
&\quad \cup U(A_2 \leq \text{!}^{c_2}\alpha_2), \alpha \notin (A_1 \rightarrow A_2), \text{!}^{c_1}\alpha, \text{!}^{c_2}\alpha \text{ fresh} \\
U(A_1 \rightarrow A_2 \leq A_3 \rightarrow A_4) &= U(A_3 \leq A_1) \cup U(A_2 \leq A_4) \\
U(\text{!}^{c_1}\iota \leq \text{!}^{c_2}\iota) &= \{c_1 \leftarrow c_2; \} \\
U(\text{!}^{c_1}\iota \leq \text{!}^{c_2}\iota) &= \{c_1 \leftarrow c_2; \alpha \leftarrow \iota\} \\
U(A_1 \leq A_2) &= \text{false otherwise}
\end{aligned}
\]

Now we define a unification procedure on simple types, in the style of the classical unification procedure for simple types. One can check that it has the same shape as the subtyping unification procedure, except that the cases involving modalities have disappeared. With
respect to the classical unification procedure, the only difference is in the case of the unification of an arrow type with a variable \((\text{Unif}(\alpha, (A_1 \rightarrow A_2)))\). In this case, in contrast with the classical definition, we do not substitute the whole type to \(\alpha\), but we carry on deconstructing it. This produces many small substitutions where only one big one would be needed and is unnecessary in the case of simple types. However, it is necessary when subtyping is involved. One can easily check that even if the resulting substitution is different, its effect on a type is the same as in the classical case.

**Definition 3.34.**

\[
\begin{align*}
\text{Unif}(\alpha, \alpha) &= \text{Unif}(\alpha, \alpha) = \emptyset \\
\text{Unif}(\alpha, \alpha) &= \text{Unif}(\alpha, \alpha) = \{\alpha \leftarrow \alpha\} \\
\text{Unif}(\alpha, \beta) &= \{\alpha \leftarrow \beta\}
\end{align*}
\]

\[
\text{Unif}(\alpha, (A_1 \rightarrow A_2)) = \text{Unif}((A_1 \rightarrow A_2), \alpha) = \{\alpha \leftarrow (\alpha_1 \rightarrow \alpha_2)\} \cup \text{Unif}(A_1, \alpha_2)
\]

\[
\cup \text{Unif}(\alpha_2, A_2), \alpha \notin (A_1 \rightarrow A_2), \alpha_1, \alpha_2 \text{fresh}
\]

\[
\text{Unif}(A_1 \rightarrow A_2, A_3 \rightarrow A_4) = \text{Unif}(A_3, A_1) \cup \text{Unif}(A_2, A_4)
\]

\[
\text{Unif}(\iota, \iota) = \emptyset
\]

\[
\text{Unif}(\iota, \alpha) = \text{Unif}(\alpha, \iota) = \{\alpha \leftarrow \iota\}
\]

\[
\text{Unif}(A_1, A_2) = \text{false otherwise}
\]

We can express the correctness of the subtyping unification w.r.t simple types unification:

**Lemma 3.35.** If \(U(A_1 \leq A_2) = \{\mathcal{C}; \sigma\}\) then \(\text{Unif}(A_1^{-}, A_2^{-}) = \sigma^{-}\).

**Proof.** Induction on \(A_1\). Let us look at the cases where \(U\) and \(\text{Unif}\) differ:

- \(A_1 = !\mathcal{C} B_1\): then subtyping unification proceeds with \(U(B_1 \leq B_2) = \{\mathcal{C}'; \sigma\}\) and leaves the substitution \(\sigma\) unchanged. Note that \(A_1^{-} = B_1^{-} \). By I.H \(\text{Unif}(B_1^{-}, B_2^{-}) = \sigma^{-}\).

- If the cases where \(\iota\) appears, one can check in the definitions that the substitution definitions are the same in both cases.

And can be extended to linear types:

**Lemma 3.36.** If \(U(A_1 \leq A_2) = \{\mathcal{C}; \sigma\}\) and \(\mathcal{C}\) admits a solution \(\phi\), then \((\sigma(A_1))\phi \leq (\sigma(A_2))\phi\).

**Proof.** By previous Lemma we know that if \(U(A_1 \leq A_2) = \{\mathcal{C}; \sigma\}\), then \((\sigma(A_1))^{-} = (\sigma(A_2))^{-}\). Let us note \((\sigma(A_1))\) and \((\sigma(A_2))\) resp. \(A_1^{'}\) and \(A_2^{'}\).

By induction on \(A_1^{'}\), we can reconstruct a type derivation of \(A_1^{'}\phi \leq A_2^{'}\phi\).

- If \(A_1^{'} = !^n C\) and \(A_2^{'} = !^k D\), with \(C\) and \(D\) not equal to \(\iota\), by I.H we have a subtyping derivation of \(\mathcal{C}\phi \leq D\phi\), and since the produced constraint is \(n \equiv k\), so we can apply \(n\) times the PRM rule of the \(\leq\) definition to obtain a derivation of \(A_1^{'}\phi \leq A_2^{'}\phi\).

- If \(A_1^{'} = \alpha\) and \(A_2^{'} = \alpha\) we can apply the base rule;

- If \(A_1^{'} = \iota\) and \(A_2^{'} = \iota\) we can apply the base rule.

- If \(A_1^{'} = C \rightarrow D\) and \(A_2^{'} = E \rightarrow F\), by I.H we can derive \(E\phi \leq C\phi\) and \(D\phi \leq F\phi\) and then apply the arrow rule;

- \(A_1^{'} = !^n \iota\) and \(A_2^{'} = !^k \iota\) by definition the produced constraint is a linear inequality \(j \leq k\), so, by definition 3.11 (page 53), \(A_1^{'}\phi \leq A_2^{'}\phi\).
And conversely, we begin by stating that the subtyping unification produces the same result as the classical one in the sense of simple types:

**Lemma 3.37.** If \( \text{Unif}(A_1^\top, A_2^\top) = \sigma \), then \( U(A_1 \leq A_2) = \{ C; \sigma' \} \) for some \( C \), and \( \sigma = \sigma'^\top \).

**Proof.** Induction on \( A_1 \). Again, we simply look at the cases where the subtyping and the classical definition differ:

- \( A_1 = \lnot^c \) and \( A_2 = \lnot^c \). Then \( A_1^\top = B_1^\top \) and \( A_2^\top = B_2^\top \). If \( \text{Unif}(A_1^\top, A_2^\top) = \text{Unif}(B_1^\top, B_2^\top) = \sigma \), then by I.H \( U(B_1 \leq B_2) = \{ C'; \sigma' \} \) for some \( C' \) and \( \sigma = \sigma'^\top \). By definition of subtyping unification \( U(A_1 \leq A_2) = \{ C; \sigma' \} \), for some \( C \).

- In the cases where \( \iota \) appears, the substitutions produced are the same in both definitions.

And we can extend it to linear types:

**Lemma 3.38.** Let \( C \) and \( D \) be real types, and let \( C^- \) and \( D^- \) be their respective free decorations (with disjoint sets of parameters). If \( C \leq D \), and if \( \phi \) is an instantiation s.t. \( C^-\phi = C \) and \( D^-\phi = D \), then \( \phi \) is a solution of the constraints generated by \( U(C^- \leq D^-) \).

**Proof.** By induction on \( C \). Note that since \( C \leq D \), \( C^- = D^- \).

- \( C = \lnot^c \_ \ldots \lnot^c \_ \), with \( c \neq \iota \). Then \( D = \lnot^c \_ \ldots \lnot^c \_ \). Then \( C^- = \lnot^c C^- \) and \( D^- = \lnot^c D^- \). By I.H \( \phi \) is a solution of the constraints produced by \( U(C^- \leq D^-) \). \( U(\lnot^c C^- \leq \lnot^c D^-) \) produces the constraints \( \{ c_1 \leq c_2 \} \cup U(C^- \leq D^-) \). Since \( C^-\phi = C \) and \( D^-\phi = D \), \( \phi \) must contain instantiation \( c_1 \leq c_2 \) and \( c_2 = n \), so \( \phi \) is a solution of \( \{ c_1 \leq c_2 \} \cup U(C^- \leq D^-) \).

- If \( C = \lnot^c \ldots \lnot^c \iota \), then, since \( C \leq D \), \( D = \lnot^c \ldots \lnot^c \iota \), with \( n \leq m \) (by definition of the \( \leq \) relation). \( C^- = \lnot^c \_ \ldots \lnot^c \_ \) and \( D^- = \lnot^c \_ \ldots \lnot^c \_ \), so by definition of the subtyping unification, the constraint produced will be \( c_1 \leq c_2 \). \( \phi \) satisfies this constraint since \( n \leq m \), \( C^-\phi = C \) and \( D^-\phi = D \).

- The other cases do not imply linear constraints and can be solved by I.H only.

Furthermore, it is easy to see that the unification procedure always terminates, possibly by a failure.

### 3.4.3 Constraints generation

**Local typing**

First, we have to build a context \( \Theta_\iota \) which associates to each variable, free or bound, of a parametrised term \( t \) a fresh — by “fresh”, we mean not appearing anywhere else before — parametrised type variable of the form \( \forall^p \alpha \) where \( p \) and \( \alpha \) are both fresh. We call such contexts *generalised context*. We need to ensure that the parameters combinations occurring in the type of the terms are correct, in the sense that they have a positive value. To this purpose, we define the function \( \text{adm} \) on parametrised types: \( \text{adm}(A) = \{ c \geq 0, c \text{ occurs in } A \} \).

We present constraints generation for the local typing criterion in figure 3.13, which amounts to defining a function on parametrised terms that we will call \( \text{cstr} \). This function takes in input a parametrised term, typically a free decoration of a pure lambda-term, and a generalised context \( \Theta_\iota \) associated to it, and returns:
3.4 TYPE INFERENCE

\[
\begin{align*}
\Theta(x) &= A \text{ and } \forall \alpha \text{ fresh} \\
x &: \{ (x : \forall \alpha) ; \{ A \leq \forall \alpha \} ; \text{adm}(A) \} \quad \text{(VAR)} \\
M &: \{ \Psi ; U ; C \} \quad \Theta(x) = A_1 \quad \Psi(M) = A_2 \\
\lambda x. M &: \{ \Psi \cup \{ \lambda x. M : ^{1\alpha}(A_1 \Rightarrow A_2) \} ; U ; C \cup \text{adm}(A_1) \} \quad \text{(ABS)} \\
M_1 &: \{ \Psi_1 ; U_1 ; C_1 \} \quad M_2 &: \{ \Psi_2 ; U_2 ; C_2 \} \quad \Psi_1(M_1) = A_1 \quad \Psi_2(M_2) = A_2 \\
(M_1) M_2 &: \{ (M_1) M_2 : ^{1\alpha} \} \cup \Psi_1 \cup \Psi_2 ; U_1 \cup U_2 \cup \{ A_1 \leq ^{1\alpha}(A_2 \Rightarrow ^{1\alpha}) \} ; C_1 \cup C_2 \quad \text{(APPL)} \\
(t, A) &: \{ t : ^{1\alpha} \} \text{ fresh} \quad \Psi(t) = ^{1\alpha}B \\
\forall \tau &: \{ \{ \tau \in U \} \cup \{ \tau \in U \} \} \quad \text{(1)} \\
\end{align*}
\]

Figure 3.3: cstr function of constraints generation

- a generalised context \( \Psi \) assigning to each subterm of \( t \) a parametrised type
- a set of first-order unification constraints \( U \)
- a set of linear (in)equations \( C \)

So the result of \( \text{cstr}(t, \Theta_t) \) is of the form \( \{ \Psi ; U ; C \} \).

The rules are to be understood in the following way: we choose the rule to apply depending on the shape of the term. We deconstruct the term in a bottom-up fashion, without producing any constraint at this stage, by performing recursive calls until we reach the leaves of the term, and then proceed top-down in the term to generate constraints and increases the triple \( \{ \Psi ; U ; C \} \). This defines an algorithm in the sense that all the rules are syntax-directed and deterministic. Note that in the \( \text{APPL} \) case of the algorithm, the \( ^{1\alpha} \) is fresh as in the \( \text{VAR} \) and \( \text{CST} \) cases.

We define a function \( \text{cntr} \), taking in input a parametrised term and an extended context \( \Psi \), \( \text{cntr}(t, \Psi) \) is the union of the sets \( \{ c \geq 1 \} \), for all \( x \) occurring more than once in \( t \) and s.t. \( \Psi(x) = ^{1\alpha}B \).

We define now \( \text{LTyping}(t, \Theta_t) \) as the union of the constraints built by \( \text{cntr}(t, \Psi) \) and \( \text{cstr}(t, \Theta_t) \). \( \text{LTyping}(t, \Theta_t) \) also takes in input a parametrised term \( t \) and an associated generalised context \( \Theta_t \) and produces an extended context \( \Psi \), a set of first-order unification constraints and one of linear (in)equations. In the following we will note \( \text{LTyping}(t, \Theta_t) = \{ \Psi ; U ; C \} \), \( \Psi \) being the extended context, \( U \) the set of first-order unification constraints, and \( C \) the set of linear (in)equations.

Bracketing

We consider the words over integer parameters that we will note \( m, n \ldots \). Let \( t \) be a parametrised term and \( u \) a subterm of \( t \). We define, as for pseudo-terms, the word \( \text{doors}(\vec{t}, u) \) as follows:

\[
\begin{align*}
doors(t, u) &= \epsilon, \text{ if } t = u \\
doors(\forall^\alpha \alpha', u) &= n :: doors(t', u) \\
doors(\lambda x.t', u) &= doors(t', u) \\
doors((t_1 t_2), u) &= doors(t_1, u) \text{, where } t_i \text{ is the subterm containing } u \\
\end{align*}
\]
We define a map \( s \) from the words over integer parameters to linear combination of parameters:

\[
  s(\epsilon) = 0, \quad s(m::l) = m + s(l),
\]

And now we can define two functions from parameters words to linear constraints:

- \( \text{wbrack}(l) = \{ s(l') \geq 0 \mid l' \leq l \} \)
- \( \text{brack}(l) = \text{wbrack}(l) \cup \{ s(l) = 0 \} \)

And now we define a constraints generation function on parametrised terms \( \text{Bracketing}(t) \) as the union of the following sets:

(i) for any occurrence of free variable \( x \) in \( t \), \( \text{brack}(\text{doors}(t, x)) \);
(ii) for any occurrence of an abstraction subterm \( \lambda x.v \) of \( t \):
   (ii.a) \( \text{wbrack}(\text{doors}(t, \lambda x.v)) \)
   (ii.b) and for any occurrence of \( x \) in \( v \), \( \text{brack}(\text{doors}(v, x)) \).

Note that a constraint system generated by the \( \text{Bracketing} \) function can have several (possibly infinitely many) solutions, and thus a parametrised term can admit several correct instantiations. This corresponds to the fact that given a term, there are several correct box placements.

### 3.4.4 Constraints resolution

#### Unification

The \text{LTyping} procedure gives a set of unification constraints, and a set of linear (in)equations. To solve this system, we have to find a most general unifier for the first order unification constraints, and then solve the linear constraints. The \( \text{Mgu} \) function will thus take in input a set of unification constraints, and produce in result a pair \( \{ \mathcal{C}; \sigma \} \) where \( \mathcal{C} \) is a set of linear constraints and \( \sigma \) a substitution when it succeeds, or \( \text{false} \) when it fails.

\[
  \text{Mgu}(\emptyset) = \{ \emptyset; \emptyset \}
\]

\[
  \text{Mgu}(\{ A_1 \leq A_2 \} \cup \mathcal{U}) = \{ \mathcal{C}_1 \cup \mathcal{C}_2; \sigma_2 \circ \sigma_1 \}
  \quad \text{if} \ U(A_1 \leq A_2) = \{ \mathcal{C}_1; \sigma_1 \}
  \quad \text{and} \ \text{Mgu}(\sigma_1(\mathcal{U})) = \{ \mathcal{C}_2; \sigma_2 \}
\]

\[
  \text{Mgu}(\{ A_1 \leq A_2 \} \cup \mathcal{U}) = \text{false}, \ \text{if} \ U(A_1 \leq A_2) = \text{false}
\]

Note that the \( \text{Mgu} \) function obviously terminates (possibly failing to unify), since the unification function \( U \) does.

**Lemma 3.39.** If \( \text{Mgu}(\mathcal{U}) \) succeeds with the result \( \{ \mathcal{C}; \sigma \} \), then for all unification constraints \( A_1 \leq A_2 \) present in \( \mathcal{U} \), \( U(A_1) \sigma (\mathcal{A}_1) = (\sigma(A_2))^{-} \).

**Proof.** By induction on the size of \( \mathcal{U} \), then using Lemma 3.35 \( \square \)

**Lemma 3.40 (correctness of the \( \text{Mgu} \) function).** If \( \text{Mgu}(\mathcal{U}) \) succeeds with result \( \{ \mathcal{C}; \sigma \} \), and if \( \mathcal{C} \) admits a solution \( \phi \), then for all unification constraints \( A_1 \leq A_2 \) present in \( \mathcal{U} \), \( \sigma(A_1) \phi \leq \sigma(A_2) \phi \).

**Proof.** For all unification \( U(A_1 \leq A_2) = \{ \mathcal{C}'; \sigma' \} \) done by the \( \text{Mgu} \) function:
By Lemma 3.39, \((\sigma(A_1))^- = (\sigma(A_2))^-\)

By definition of \(\text{Mgu}\), \(C\) is the union of all the \(C'\) generated by the unifications, so the global solution \(\phi\) satisfies the (in)equations of all \(C'\).

Hence, we can conclude that for all unification constraints \(A_1 \preceq A_2\) present in \(U\), \(\sigma(A_1)\phi \preceq \sigma(A_2)\phi\) by applying Lemma 3.36 on each of them.

**Lemma 3.41 (completeness of the \(\text{Mgu}\) function).** Let \(U\) be a set of unification constraints, if a substitution \(\tau\) and an instantiation \(\phi\) are such that for each constraint \(A_1 \preceq A_2\) in \(U\), \(\tau(A_1)\phi \preceq \tau(A_2)\phi\), then \(\text{Mgu}(U)\) succeeds with result \(\{C;\sigma\}\), \(\tau = \tau' \circ \sigma\) for some \(\tau'\) and \(\phi\) is a solution of \(C\).

**Proof.** By induction on the size of \(U\) and using Lemma 3.38.

**Linear (in)equations**

We have described the way to produce unification and linear constraints, and showed how to solve unification constraints. It remains only to show how to resolve linear constraints.

As in DIAL, notice that for a parametrised term \(\bar{t}\), the linear constraints of \(\text{Bracketing}(\bar{t}) \cup \text{LTyping}(\bar{t}, \Theta_{\bar{t}})\) constitute a linear inequation system, for which a polynomial time procedure for searching a rational solution is known (\([\text{Kac79, Kar84}]\)):

**Lemma 3.42.** For any parametrised term \(\bar{t}\), let \(C\) be the linear constraints of \(\text{Bracketing}(\bar{t}) \cup \text{LTyping}(\bar{t}, \Theta_{\bar{t}})\). It is decidable whether \(C\) admits a solution in \(\mathbb{Q}\), in time polynomial in the cardinality of \(C\).

**Lemma 3.43.** For any parametrised term \(\bar{t}\), let \(C\) be the linear inequations of \(\text{Bracketing}(\bar{t}) \cup \text{LTyping}(\bar{t}, \Theta_{\bar{t}})\). The linear inequation system \(C\) admits a solution in \(\mathbb{Q}\), if and only if it admits a solution in \(\mathbb{Z}\).

**Proof.** The produced constraints are always of the form \(c \geq 1\) or \(c \geq 0\), or \(c \equiv 0\), so the set of solution is closed by multiplication by a positive integer. Therefore, one can solve the constraints on \(\mathbb{Q}\) and multiply the solution to obtain only integers.

**3.4.5 Correctness and completeness of the constraints**

In this section we will denote pure lambda-terms by \(t, u\), (restricted) pseudo-terms by \(t^+, u^+\) when they are pseudo-decorations of lambda terms \(t, u\) and free decorations by the application of the \((\cdot)\) map on pure lambda-terms.

First, the correctness and completeness of the bracketing constraints generation w.r.t the bracketing criterion is easy to check:

**Lemma 3.44.** Given a parametrised term \(\bar{t}\), and an instantiation \(\phi\), \(\bar{t}\phi\) satisfies the bracketing criterion iff \(\phi\) is a solution of \(\text{Bracketing}(\bar{t})\).

**Proof.** By the definitions of the bracketing criterion and constraints generation.

Now, we show the correctness of the \(\text{LTyping}\) constraints generation procedure:

**Lemma 3.45.** Given a lambda-term \(t\) and its free decoration \(\bar{t}\) and a context extension \(\Theta_{\bar{t}}\) of \(\bar{t}\), let \(\text{LTyping}(\bar{t}, \Theta_{\bar{t}}) = \{\Psi; U; C_1\}\). If \(\text{Mgu}(U)\) succeeds with a substitution \(\sigma\) and produces a set of (in)equations \(C_2\), and if \(C_1 \cup C_2\) admits a solution \(\phi\), then \((\bar{t}\phi, \sigma(\Theta_{\bar{t}})\phi, \sigma(\Psi)\phi)\) satisfies the local typing criterion.
Proof. Induction on \( t^+ = \bar{t} \phi \), we define \( \Delta = \sigma(\Theta_t)\phi \) and \( \Gamma = \sigma(\Psi)\phi \):

- \( t^+ = x \): by definition of \( \text{LTyping} \) and by the correction of the \( \text{Mgu} \) function and of the linear constraints resolution \( \Delta(x) \leq \Gamma(x) \) and \( \Gamma(x) \) is of the form \( !A \) if \( x \) has several occurrences, thanks to the \( \text{cntr} \) constraint, which fits the local typing criterion

- The case where \( t^+ \) is a constant is treated in the same way.

- \( t^+ = \lambda x.t'^+ \): by I.H \( \Delta(x) = A_1 \) and \( \Gamma(t'^+) = A_2 \), by definition of \( \text{LTyping} \), \( \Gamma(\lambda x.t'^+) = A_1 \rightarrow A_2 \), so \( (t^+, \Delta, \Gamma) \) satisfies the local typing criterion

- \( t^+ = (t_1^+) t_2^+ \): by I.H \((t_1^+), \Delta, \Gamma) \) and \((t_2^+), \Delta, \Gamma)\) satisfy the local typing criterion; by definition of \( \text{LTyping} \) \( \Gamma(t_1^+) \leq \Gamma(t_2^+) \rightarrow A \) where \( \Gamma((t_1^+) t_2^+) = A. \) Hence \( (t^+, \Delta, \Gamma) \) satisfies local typing

- \( t^+ = !t'^+ \): by I.H \((t'^+), \Delta, \Gamma) \) satisfies local typing and \( \Gamma(t'^+) = A \), by definition of \( \text{LTyping} \) and correction of the linear constraints resolution, \( \Gamma(!t'^+) = !A \) and so \( (t'^+, \Delta, \Gamma) \) also satisfies local typing

- \( t^+ = !t'^+ \): by I.H \((t'^+), \Delta, \Gamma) \) satisfies the local typing criterion. By definition of \( \text{LTyping} \) and \( \text{adm} \), \( \Gamma(t'^+) \) must be of the form \( \Gamma(t'^+) = !A \), for some \( A \). So \( \Gamma(!t'^+) = A \) and \( (t^+, \Delta, \Gamma) \) satisfies the local typing criterion

- \( t^+ = \bar{t} \phi \): by I.H \((t'^+), \Delta, \Gamma) \) satisfies the local typing criterion. If \( (t'^+), \Delta, \Gamma) \) satisfies the local typing criterion, then:

\[
\begin{align*}
\text{LTyping}(\bar{t}, \Theta_t) & \text{ succeeds with result } \{\Psi; U; C_1\} \\
\text{Mgu}(\bar{t}) & \text{ succeeds with result } \{C_2; \sigma\} \sigma \\
C_1 \cup C_2 & \text{ admits a solution } \phi
\end{align*}
\]

Proof. By induction on \( t^+ \):

- If \( t^+ \) is a variable or a constant, it is easy to see that resolution and constraints solving cannot fail on those cases

- If \( t^+ = \lambda x.t'^+ \): solved simply by I.H

- \( t^+ = (t_1^+) t_2^+ \) and so \( \bar{t} = (\bar{t}_1 \bar{t}_2) \): by hypothesis \( t_1^+ \) is of the type \( A_1 \), and \( t_2^+ \) of the types \( A_2 \) s.t. \( A_1 \leq A_2 \rightarrow A_3 \). Therefore by I.H and by applying Lemma 3.31 unification and linear (in)equations solving will succeed

- \( t^+ = !t'^+ \) or \( t^+ = !t'^+ \): By induction hypothesis, the result holds for \( (t'^+, \Delta, \Gamma) \). It is easy to check on Figure 5.3 that the \( \text{cntr} \) constraints generation function does not prevent from adding or removing !’s at the global type of the term.
Main result

We can now state the main results of type inference:

**Proposition 3.47.** Let \( \pi \) be the free decoration of a lambda-term \( u \) and \( \Theta_\pi \) a generalised context, if:

- \( \text{LTyping}(\pi, \Theta_\pi) = \{ \Psi; U; C_1 \} \)
- \( \text{Bracketing}(\pi) = C_2 \)
- \( \text{Mgu}(U) = \{ C_3; \sigma \} \)
- \( C = C_1 \cup C_2 \cup C_3 \) admits a solution \( \phi \)

Then \( u_\phi \) satisfies the bracketing criterion, and \( (u_\phi, \sigma(\Theta_\pi)_\phi, \sigma(\Psi)_\phi) \) satisfies the local typing criterion.

*Proof.* Using Lemmas 3.44 and 3.45.

**Proposition 3.48.** Consider a pseudo-term \( t^+ \), \( t \) is the underlying lambda term, satisfying the bracketing criterion and a context \( \Delta \) and an extension \( \Gamma \) s.t. \((t^+, \Delta, \Gamma)\) satisfies the local typing criterion, then:

- \( \text{LTyping}(t, \Theta_t) \) succeeds with \( \{ \Psi; U; C_1 \} \)
- \( \text{Bracketing}(t) = C_2 \)
- \( \text{Mgu}(U) \) succeeds with \( \{ C_3; \sigma \} \)

And then \( C = C_1 \cup C_2 \cup C_3 \) admits a solution \( \phi \) such that \( t_\phi = t^+ \) and \( (t_\phi, \sigma(\Theta_t)_\phi, \sigma(\Psi)_\phi) \) satisfies the local typing criterion.

*Proof.* Using Lemmas 3.44 and 3.46.

This allows us to relate directly the type inference procedure and typability in \( \text{eal}^* \leq \):

**Theorem 3.49.** Let \( \pi \) be the free decoration of a lambda-term \( u \) and \( \Theta_\pi \) a generalised context. Let \( \text{LTyping}(\pi, \Theta_\pi) = \{ \Psi; U; C_1 \} \), and \( \text{Bracketing}(\pi) = C_2 \). The lambda-term \( u \) is typable in \( \text{eal}^*_\leq \) if and only if:

- \( \text{Mgu}(U) \) succeeds with result \( \{ C_3; \sigma \} \)
- \( C = C_1 \cup C_2 \cup C_3 \) admits a solution \( \phi \)

*Proof.* First, using theorem 3.28 which states an equivalence between the typing criteria and the derivability in \( \text{eal}^*_\leq \), and secondly by Propositions 3.47 and 3.48 which state the equivalence of the linear and unification constraints generation and the typing criteria.
3.5 Conclusion

We have proved that the problem of coercions placement in EAL can be treated with subtyping and thus internalised in the type system. Then we showed that it was possible to infer the necessary coercions placement along the EAL type inference. So, our work shows, on the one hand that coercions placement is not a fundamental obstacle to programming in EAL and, on the other hand, that subtyping can be accommodated in this type system. Our work is also the illustration of a fully-fledged type inference algorithm for a Light logic: we start from a pure lambda term, and the procedure is not in two phases.

One might consider adapting our approach to other Light logics, and particularly to DLAL, which already admits an efficient type inference procedure, as we saw in chapter 2. We leave open the question of reducing efficiently the lambda terms or the Proof-nets obtained with our extended type assignment system. In the case of Proof-net reduction, one could consider adding nodes to perform coercions in constant time. In the case of lambda-calculus beta reduction however, the term can be equivalently reduced without the coercions, and the number of beta steps in its reduction is obviously inferior to the number of beta steps in the reduction of the term with coercions.

Another question left open is about the complexity of the whole type inference procedure. Even though we do not have a clear argument, we believe that the overhead due to linear typing is polynomial. However, it does not imply that the whole procedure could be easily made polynomial.

In a more long-term prospect, if one ever wants to perform real static analysis on programming languages using light logics, we believe that it will be important to be able to infer coercions as well as box placement. Subtyping has proven to be a convenient solution to do so. However, there are still obstacles, such as the handling of recursively defined functions. Designing a variant of light logics in which one can handle first-order recursive functions and implicit coercions placement could lead to prototypes of static analysis for programming languages like CAML. However, we believe that the approach taken in the previous chapter (linear decoration) is better than the current one. It seems more elegant, and has the benefit of clearly separating the qualitative (usual typing) and the quantitative (linear typing) problems.
Chapter 4

Optimal reduction for Soft Linear logic

4.1 Soft Sharing Graphs

4.1.1 Introduction

**Background** The Optimal Reduction algorithm (OR), first introduced by Lamping in \[Lam90\], implements the sharing of redex families as developed in the labelled lambda-calculus of Lévy (\[Lév80\]). The algorithm is usually divided into two parts: the *abstract* algorithm and the *oracle*. Even though Optimal reduction is Optimal in the sense that redexes are not duplicated, it does not mean that it is *efficient*, if we take for efficiency the number of elementary steps required to reduce a lambda-term. It is commonly admitted that, roughly, the abstract part of the algorithm carries out the useful part of the work, while the oracle is there to ensure that it is done correctly. Thus, it is in the cost of the oracle that lie the possible inefficiencies of Optimal reduction. In fact, in [LM96], authors have exhibited a family of terms for which the cost of the oracle is exponential in w.r.t cost of the abstract algorithm. Besides this result, many studies regarding the complexity of the Optimal reduction algorithm have been carried out.

The correspondence between Optimal reduction and Linear logic, and Proof-nets of Linear logic, notably via Geometry of Interaction (GoI), developed in [Gir89, Gir90, DR93, DR95] has been known for a long time, even if part of this knowledge is still folklore and is not considered well-established. In a recent work, [BCDL07], Baillot, Coppola and Dal Lago gave a proof of the correctness of the reduction of Elementary affine logic typable terms with the abstract algorithm of OR, using an innovative proof methodology. So, this provides a huge set of terms (Elementary affine logic typable terms) for which the oracle is not needed in the reduction. This result had already been mentioned in [Asp98, CM06]. Besides, polynomial type inference algorithms have been developed for Elementary affine logic, using integer constraints resolution,
Soft Linear logic (SLL) is a Light logic characterising polynomial time, as Light Linear logic. However, it is based on a completely different mechanism to enforce the complexity bound. There is no definitive result regarding the comparison of the typing powers of SLL and LLL. It has been introduced by Lafont in [Laf04], and further studied in [GRDR07], where a fragment of SLL is defined, that fits well with lambda-calculus typing.

Motivations The aim of this work is to show that lambda-terms typable in Soft linear logic can be evaluated in an ad-hoc Optimal reduction algorithm. It is in the spirit of [BCDL07], for its goal and the global proof method, but it differs on the logic and the semantics style adopted. In particular, Elementary affine logic relies on stratification to control duplication and thus ensure complexity properties, while Soft Linear logic rather uses a kind of “potential” on boxes, via the use of the multiplexor.

The relations between Proof-nets cut-elimination and Optimal reduction are intricate, and even though this topic has been studied in [GAL92, GMM03, DL06], there are still questions left open, and maybe general results on Optimal reduction to draw from a Linear logic intuitions. We believe that the proof technique and the semantics we formulate could be used to show the correctness of Optimal reduction for lambda-terms typable in Intuitionistic linear logic, for all the possible typings of a lambda-term rather than for the initial translation.

Outline First we present our restricted version of Optimal reduction, and prove the complexity bound, with a condition close to the Soft Linear logic typing discipline. Then we equip those sharing graphs with a GoI traces semantics, that we also define on Intuitionistic Soft Proof-nets. With this tool, we can state a correctness and completeness result for our limited Sharing graphs.

4.1.2 Soft sharing graphs

Figure 4.1 presents the nodes of what we we will call Soft sharing graphs (SSG), which are indexed by integers of \( \mathbb{N} \) representing their depth. It is directly inspired of the Proof-nets of Soft Linear logic. The lambda and application nodes (called the multiplicative nodes) are as in usual Optimal reduction Sharing graphs, and are the counterpart of \( \otimes-r \) and \( \otimes-l \) nodes of Soft linear logic Proof-nets (see Chapter 1). Then we have a multiplexor, which is quite different from the OR fan:

- It has \( n \), and not two, auxiliary ports (\( n \) will be the arity of a multiplexor).
- In particular, \( n \) can be equal to 0 (weakening) or 1 (dereliction).
- Reduction rules will show that it does not only duplicate the other nodes but also changes their depth.
- Intuitively, it has the same function as the multiplexor of Soft linear logic Proof-nets [Laf04].

Finally the indexes correspond to the box depth in Proof-nets. Those nodes are the basic language of SSG. There is one particular port of each node that carries an arrow, which is the principal port of the node. The top port of the application node is its context node, the last one its argument node. For the \( \lambda \) node, its left port is its variable port, and the right one the subterm port.

Simple lines will denote the free ports of the Sharing graph. One of those free ports is labelled with \( \perp \) an is the root of the Sharing graphs, and the other free ports, called free variable ports will carry labels ranging over \( z, z_k, v_j \).

The Sharing graphs rewriting relation we design is the contextual closure of the reduction rules given in Figure 4.2 and we will denote one step of reduction by \( \rightarrow \). The reflexive and transitive closure of \( \rightarrow \) will be denoted by \( \rightarrow^* \).
4.1 SOFT SHARING GRAPHS

The $\beta$ and $\delta_4$ rules are very close to the usual rules of OR: $\beta$ reduction and multiplexor annihilation. However in the case of multiplexor annihilation, we have another condition stating that the arity of the two annihilating multiplexors must be equal. The $\delta_1$ and $\delta_2$ rules describe the action of the multiplexor on multiplicative nodes: it creates $n$ copies of the node at depth $j - 1$ and makes the original node of depth $j$ disappear. The $\delta_3$ rule obeys the same principle: the multiplexor of lowest depth ($i$ on the figure) creates $n$ copies of the other one, but at depth $j - 1$ only; the original node is deleted. The only difference between the $\delta_{1,2}$ rule and the $\delta_3$ rule is finally the number of auxiliary ports of the duplicated node.

4.1.3 Main properties

Confluence

Since the system we defined is an Interaction system, in the sense of [La90], it enjoys the diamond property:

**Proposition 4.1.** For all Soft sharing graphs $S_1, S_2, S_3$ such that $S_1 \rightarrow S_2$ and $S_1 \rightarrow S_3$, there exists $S_4$ such that $S_2 \rightarrow S_4$ and $S_3 \rightarrow S_4$.

And so:

**Corollary 4.2 (confluence).** For all Soft sharing graphs $S_1, S_2, S_3$ such that $S_1 \rightarrow^* S_2$ and $S_1 \rightarrow S_3^*$, there exists $S_4$ such that $S_2 \rightarrow^* S_4$ and $S_3^* \rightarrow S_4$.

Bound on the reduction length

Let $S$ be a sSG Sharing graph. We denote by $S^i$ the set of its nodes at depth $i$, and so $|S^i|$ is the number of nodes at depth $i$. We denote by $d$ the maximum depth. We define the tuple $W(S) = (|S^d|, \ldots, |S^0|)$, and denote by $>_\text{lex}$ the lexicographic ordering on those tuples.

**Lemma 4.3.** If $S_1 \rightarrow S_2$, then $W(S_1) >_\text{lex} W(S_2)$.

**Proof.** Looking at the rules:

- $\beta$: If the interaction is at depth $i$, then $|S_2^i| = |S_1^i| - 2$ and for all $j$ different from $i$, $|S_2^j| = |S_1^j|$. So $W(S_1) >_\text{lex} W(S_2)$.

- $\delta_{1,2}$: If the multiplexor node interacting is at depth $i$ and the duplicated node at depth $j$, then $|S_2^i| = |S_1^i| - 1$. Since $j > j - 1$ and $j > i$, then $W(S_1) >_\text{lex} W(S_2)$.

- $\delta_3$: This time there are $n$ duplicated nodes at depth $j - 1$ and $m - 1$ duplicated nodes at depth $i$. Still, the number of nodes at depth $j$ (the highest depth appearing) decreases.

- $\delta_4$: there are only deleted nodes.
CHAPTER 4. OPTIMAL REDUCTION FOR SOFT LINEAR LOGIC

\[
\begin{align*}
\beta \text{ reduction rule} & : \quad i = j \\
\delta_{1,2} \text{ reduction rule } (\alpha = \lambda \text{ or } \alpha = \odot) & : \\
\delta_3 \text{ reduction rule} & : \\
\delta_4 \text{ reduction rule} & : \quad i = j \text{ and } m = n
\end{align*}
\]

Figure 4.2: ssg reduction rules
Now, we would like to have a tighter bound on the reduction length. We will denote by $\text{Ar}(S)$ the maximal arity of all the multiplexor nodes of a Sharing graph $S$.

**Remark 4.4.** If $S_1 \rightarrow S_2$, then $\text{Ar}(S_1) \geq \text{Ar}(S_2)$. The only case where the maximal arity changes is when two multiplexors annihilate each other, if they had the maximal arity of the Sharing graph.

We define the polynomial $P_S = \sum_{i=0}^{n} |S_i|X^i$.

**Lemma 4.5.** If $S_1 \rightarrow S_2$, then $P_{S_1}(n) > P_{S_2}(n)$ for all $n > 2 \times \text{Ar}(S_1)$.

**Proof.** Looking at the rules:

- $\beta$: If the interaction is at depth $i$, then $|S_1| = |S_i| - 2$ and for all $j$ different from $i$, $|S_j| = |S_j'|$. So $P_{S_1}(n) > P_{S_2}(n)$ for all $n$.

- $\delta_{1,2}$: If the multiplexor node interacting is at depth $i$ and the duplicated node at depth $j$, then $|S_2'| = |S_1| - 1$, $|S_1'| = |S_1| - 1$ and $|S_2'| = |S_1| - 1 + n$, where $n$ is the arity of the multiplexor node. The worst case appears if $n = \text{Ar}(S_1)$ and $i = j - 1$. Then $P_{S_2} = P_{S_1} - X^j + (\text{Ar}(S_1) + 1)X^{j-1}$.

  $$P_{S_1}(n) > P_{S_2}(n) \Leftrightarrow n^j > (\text{Ar}(S_1) + 1)n^{j-1} \Leftrightarrow n > (\text{Ar}(S_1) + 1)$$

  Which is satisfied since by hypothesis we have that $n > 2 \times \text{Ar}(S_1)$.

- $\delta_3$: it is roughly the same case as for $\delta_{1,2}$, except that this time there are at most $\text{Ar}(S_1)$ nodes created at depth $j - 1$ and $\text{Ar}(S_1) - 1$ nodes created at depth $i$. So in the worst case (where $i = j - 1$) $P_{S_2} = P_{S_1} - X^j + (2 \times \text{Ar}(S_1) + 1)X^{j-1}$.

  $$P_{S_1}(n) > P_{S_2}(n) \Leftrightarrow n^j > (2 \times \text{Ar}(S_1) + 1)n^{j-1} \Leftrightarrow n > 2 \times \text{Ar}(S_1) - 1$$

  Which is satisfied since by hypothesis we have that $n > 2 \times \text{Ar}(S_1)$.

- $\delta_4$: idem as for $\beta$.

**Corollary 4.6.** If a family of Sharing graphs $F$ has a constant depth $k$ and a maximal multiplexor arity of $n$, then each element $S$ of the family reduces in time $O((|S| \times n)^k)$.

Note that even if Soft linear logic is the inspiration for this, there is no need of typing whatsoever, or even of boxes, to establish the termination bound.

Although we are in the OR setting, in which duplication is done node by node and not box-wise, the polynomial bound has the same shape as in SLL. Only, the minimal argument needed for the polynomial variable is here $2 \times \text{Ar}(s)$ while it is only $\text{Ar}(s)$ is SLL.

The previous result carries a paradox: indeed, any lambda-term has a translation in SSG. You can keep the lambda-term skeleton and put any indexes on the nodes to obtain a SSG graph admitting a polynomial bound. . . We necessarily have to identify a subset of SSG graphs s.t. their reduction is correct: the readback (to be defined) of the normal form must be the normal form of the lambda-term that had been originally translated.

In the sequel, we will show that SLL-typable lambda-terms indeed satisfy this, given that their initial translation correspond to the SLL typing, and exhibit an example of a non-typable lambda-term which can be correctly reduced in the soft sharing graphs discipline.
4.2 GoI semantics

To relate Soft sharing graphs Optimal reduction and beta reduction on lambda-terms of SLL (called further on soft lambda-terms), we introduce a Geometry of Interaction (GoI) semantics.

Most literature on Optimal reduction use as GoI semantics the so-called context semantics first used in [GAL92]. In the sequel however, we will rather use GoI traces following [DR93, DR95]. The algebraic structure it provides it certainly less computational than context semantics, but is more informative, which fits better our objectives.

4.2.1 Traces

We equip our sharing graphs with a trace semantics based on GoI, and inspired by the dynamic algebra of [DR93] and Interaction combinators ([Laf97]). Building the GoI denotation of a Sharing graph will be called an execution. This is done by travelling in the graph (following certain rules) and pushing tokens on a stack, called a trace, along the way. Tokens are of three different kinds:

- Free ports tokens, $\perp, z, v_j, \ldots$, that start or end traces.
- Two multiplicative tokens $p, q$ for multiplicative nodes, indexed by the depth of the traversed node.
- One Multiplexor token $\delta$ for multiplexor nodes, also indexed by the depth of the traversed node and carrying as exponent an integer characterising the auxiliary port by which we travel.

And we also have a reversing operation on traces, denoted by $(\cdot)^*$. A token or a trace carrying a star will be said to be starred, and if not, they will be said to be non-starred. The traces will be denoted by $w_1, w_2, z \ldots$ or in a concrete way by $p_j q_i \ldots$ and are to be read from the right to the left: the leftmost element is the last one to have been pushed. We have an empty trace $\epsilon$, and the concatenation of $w_1$ and $w_2$ is denoted by $w_1 \cdot w_2$ or simply by $w_1 w_2$. Figure 4.3 gives the travelling rules, and their effect on a trace.

The principle for the travelling is the following: if one enters a node by an auxiliary port, it goes out by the principal port — and adds in the trace a non-starred token. Otherwise, if one enters by a principal port of a node, it goes out by any auxiliary port, chosen in an non-deterministic way — and then adds in the trace a starred token. A travel that follows the rules given in Figure 4.3 is said to be correct thus yielding a correct trace. A correct travel that starts and stops at free ports of a graph is a complete travel, and the associated trace is a complete trace, it starts and ends with free ports tokens.

4.2.2 Algebra of traces

We define here a free involutive monoid algebra overs those traces, which will be useful to define a word rewriting theory. We have two constants $0$ (absorbing for multiplication and neutral for addition) and $1$ (neutral for multiplication), an involutive mapping denoted by $(\cdot)^*$ and s.t:

$$(uv)^* = v^*u^* \quad (0)^* = 0 \quad (1)^* = 1$$

We also have axioms on the multiplicative generators:

$$p_i^* p_i = q_i^* q_i = 1 \quad p_i^* q_i = 0$$

And on the exponential generators:

$$\delta_i^k \delta_i^k = 1 \quad \delta_i^k \delta_i^{k'} = 0 \text{ if } k \neq k'$$
4.2 GOI SEMANTICS

Then we have commutation equations:

\[ \delta^k_i t_j = t_{j-1}\delta^k_i \] if \( i < j \), \( t_j = p_j, q_j \)
\[ t^*_j \delta^k_i = \delta^k_i t^*_j \] if \( i < j \), \( t_j = p^*_j, q^*_j \)
\[ \delta^{k*}_i \delta^{k'}_j = \delta^{k'}_{j-1} \delta^{k*}_i \] if \( i < j \)
\[ \delta^{k*}_j \delta^{k'}_i = \delta^{k'}_{i-1} \delta^{k*}_j \] if \( i < j \)

In the following we will note \( w_i \approx w_j \) if \( w_i \) and \( w_j \) are syntactically equal or provably equal with the previous equations. The execution of a Soft sharing graph will be denoted by \([S]\), the set of non provably null complete traces in \( S \). For an arbitrary SSG \( S \), \([S]\), may be infinite. We will restrict the next results to the cases where \([S]\) is finite.

**Lemma 4.7.** If \( w \) is a non null trace in a Sharing graph \( S \), then \( w^* \) is also.

**Proof.** Induction on \( w \). \[ \square \]

There is a **direct deadlock** in a sharing graph \( S \) is two nodes face each other by their principal port and are unable to interact — so because of bad indexes. There is a **deadlock** in a sharing graph \( S \) if \( S \rightarrow^* S' \) and \( S' \) contains a direct deadlock. \( S \) is in **normal form** if it is irreducible and contains no direct deadlock.

### 4.2.3 Word rewriting

As in general GoI, we orient the equations of Section 4.2.2 from left to right to obtain a rewriting relation, and we denote it by \( w \rightarrow w' \), as usual \( \rightarrow^* \) denotes the contextual, reflexive and symmetric closure of \( \rightarrow \). We extend this notation to sets of traces and write \( \mathcal{P} \rightarrow \mathcal{P'} \) if for all \( w \in \mathcal{P}, w \in \mathcal{P'} \) or \( w \rightarrow w' \) and \( w' \in \mathcal{P'} \).

**Proposition 4.8** (invariance by reduction). If \( S_1 \rightarrow S_2 \) then for all \( w \in [S_1] \) there is a \( w' \in [S_2] \) s.t. \( w = w' \) or \( w \rightarrow w' \).
Lemma 4.11. \( w \) is a trace in normal form iff it is of the form \( w = w_1w_2 \), with \( w_1 \) and \( w_2 \) containing no starred symbols.

Proof. (only if part) Looking at (commutation) rewriting rules: all starred symbols tend to go to the right and non-starred ones to the left.

Proof. By looking at each interaction rule: reduction preserves the non null traces.

We say that a trace \( w \) contains a direct deadlock if one of the following equations hold:

\[
\begin{align*}
  w &= w_1t_i^*_i t'_i w_2 \text{ with } i \neq j, t_i = p_i, q_i, t'_j = p_j, q_j \\
  w &= w_1t_j^* t_j w_2 \text{ with } i \geq j, t_j = p_j, q_j \\
  w &= w_1t_j^* t_j^* w_2 \text{ with } i \geq j, t'_j = p'_j, q'_j
\end{align*}
\]

And a trace \( w \) contains a deadlock if \( w \rightarrow^* w', w' \) containing a direct deadlock. A trace \( w \) is in normal form if it is irreducible and contains no direct deadlock. These notions can be naturally extended to sets of traces. We have the property that:

Proposition 4.9. \( S \) contains a deadlock iff \( [S] \) contains a deadlock.

Proof. By definition of \( [.] \), \( S \) contains a direct deadlock iff \( [S] \) contains a direct deadlock.

And by Proposition 4.13 if \( S \rightarrow^* S' \), \( S' \) containing a direct deadlock, then \( [S] \rightarrow [S'] \), and \( [S'] \) contains a direct deadlock.

For the converse, observe that if \( w \rightarrow w', w \in [S] \) with \( w' \) containing a direct deadlock, then there exists \( S' \) s.t \( S \rightarrow S' \) and \( S' \) contains a direct deadlock.

We give usual properties on GoI traces, adapted from the general case ([Reg92], [DR93], [DR95]):

Lemma 4.10. The obtained word rewriting system is terminating and confluent.

Proof. Termination is easy: the axioms on generators make the number of symbols decrease, and the commutation equations decrease a natural index.

For the confluence, we can prove a diamond lemma, let us suppose that a trace \( w \) rewrites to a trace \( w_1 \) with a rule \( r_1 \) and to a trace \( w_2 \) with a rule \( r_2 \). You can observe by the form of the rules, and notably by the importance of the position of the starred generators, that a generator cannot be at the origin of two different rule applications.

Lemma 4.12. If \( S \) is a Sharing graph in normal form then for all \( w \in [S] \), \( w \) is of the form \( w_1w_2 \), with \( w_1 \) and \( w_2 \) contain no starred symbols.

Proof. If it is not the case, there is a \( w \) of the form \( w'x^*yw''' \), which corresponds to a redex in the sharing graph, or to a deadlock, which contradicts the normal form hypothesis.

Corollary 4.13. If \( S \) is a Sharing graph in normal form then for all \( w \in [S] \), \( w \) is in normal form.

Lemma 4.14. If \( S \) is a Sharing graph in normal form and \( w = w_1w_2 \in [S] \), \( w' = w'_1w'_2 \in [S] \) then if \( w_2 = w'_2 \) then \( w_1 = w'_1 \).
Proof. Looking at the way tokens are collected in Figure 4.3 when we are in the $w_1$ part of the trace, we enter a node always by its auxiliary port, and the travel followed as well as the tokens collected are deterministic.

We will write $[S]_{nf}$ for the set of non null traces of a sharing graph $S$ put in normal form using the oriented axioms.

**Definition 4.15.** $|P|$ is the cardinality of a set of traces, and $|w|$ is the size of a trace, that is, simply the number of tokens it contains. For a set of traces $P$ we define the function giving the size of its longest trace: $\max(P) = \max(|w|, w \in P)$.

And so:

**Remark 4.16.** If $S \rightarrow S'$, then $\max([S]) \leq \max([S'])$ and $|[S]| \leq |[S']|$.

### 4.2.4 Multiplicative traces and exponential stacks

We define an operation $(.)^-$ on traces in the following way:

\[
(1)^- = 1, \\
(0)^- = 0, \\
(x,w')^- = x(w')^-, \text{ with } x = p,q \\
(x^i,w')^- = x^i(w')^-, \text{ with } x = p,q \\
(\delta^k_i,w')^- = (\delta^k_i)^*(w') = (w')^-
\]

This operation erases the exponential tokens and the depth informations to leave only the unindexed multiplicative structure of a trace. It can be naturally extended to sets of traces. For instance $(p_1\delta_0^q)p_2\delta_1^3p_0)^- = pqp^*p^*$.

Then we define exponential stacks, built on of exponential tokens and denoted by $\sigma$, $\tau$, $\sigma_1$, ... Due to the peculiarities of Soft sharing graphs, and thus of Soft GoI, such a stack can be represented as an infinite list of tokens in which only a finite number of tokens are not null.

**Definition 4.17.** The set of exponential stacks $\Sigma$ is the set of infinite lists built over the set of symbols $\{\square\} \cup \{\delta^k, k \in \mathbb{N}^+\}$, with only a finite number of symbols different from $\square$. $\emptyset$ will denote the infinite list of $\square$.

Adding an element $\delta^k$ at the beginning of a stack $\sigma$ will be denoted by $\delta^k:\sigma$ or simply by $\delta^k\sigma$, and the concatenation of two exponential stacks will be denoted by $\sigma_1 \cdot \sigma_2$ or just $\sigma_1\sigma_2$. $\sigma_k$ is the stack formed by the $k$ first elements of $\sigma$ and $\sigma_k\sigma$ the stack formed by removing the $k$ first elements of $\sigma$. An exponential stack can be written for instance $\delta^1\delta^2$ to denote the stack formally defined as $\delta^1::\delta^2::\emptyset$.

We define inductively the operations push() and pop() on exponential stacks. Those operations take in input a token $\delta^k$ and a stack $\sigma$, and are indexed by an integer $n$. The $\text{push}_n(\delta^k, \sigma)$ operation outputs a new stack, where the token $\delta^k$ is added in $\sigma$ in the $n$-th position. The $\text{pop}_n(\delta^k, \sigma)$ operation outputs a new stack, where the $\delta^k$ token has been removed from the $n$-th position of $\sigma$ if it was present. If not, the operation is undefined.
CHAPTER 4. OPTIMAL REDUCTION FOR SOFT LINEAR LOGIC

Mind that those operations are not defined on the position of higher position, and conversely removing a token decreases the position of higher tokens.

Remark 4.18. We may reformulate the definition of push and pop:

- \( \text{push}_i(\delta^k, \sigma) = \sigma_{i+1} \cdot \delta^{k+1} \)
- If \( \text{pop}_i(\delta^k, \sigma) \) is well-defined, then \( \text{pop}_i(\delta^k, \sigma) = \sigma_{|i+1|} \).

Remark 4.19. push and pop are converse operations:

- \( \text{pop}_i(\delta^k, \text{push}_i(\delta^k, \sigma)) = \sigma \), for all \( \sigma \).
- \( \text{push}_i(\delta^k, \text{pop}_i(\delta^k, \sigma)) = \sigma \), if \( \text{pop}_i(\delta^k, \sigma) \) is well-defined.

Now the following definitions will relate general traces to exponential stacks:

Definition 4.20 (l \& m). We define two functions on traces \( l \) and \( m \):

- Let \( w \) be a trace, \( l(w) \) is the trace obtained by increasing by one all indexes in \( w \).
- Let \( w \) be a trace, \( m(w) \) is the trace obtained by decreasing by one all indexes in \( w \).

Those two functions can be naturally extended to sets of traces and to Sharing graphs: given a sharing graph \( S, l(S) \) (resp. \( m(S) \)) is the sharing graph obtained by increasing (resp. decreasing) all indexes of the nodes by one.

We have left to define an operation that we will call application of a trace on an exponential stack, noted \( w(\sigma) \) for the application of the trace \( w \) on the stack \( \sigma \), by induction on traces:

\[
\begin{align*}
\epsilon(\sigma) &= \sigma \\
(w'\delta_i^k)(\sigma) &= w'(\text{push}_i(\delta^k, \sigma)) \\
(w'\delta^k)(\sigma) &= w'(\sigma') \text{ if } \text{pop}_i(\delta^k, \sigma) = \sigma', \text{ undefined else} \\
(w't_i)(\sigma) &= w'(\sigma), \text{ for } t_i = p_i, q_i, r_i^*, q_i^*
\end{align*}
\]

For instance:

\[
p_1\delta_0^2q_0p_2^*\delta_3^1p_0^*\delta_1^1::\delta^4 = \text{push}_0(\delta^2, \text{pop}_3(\delta^1, \delta^1::\delta^1::\delta^4)) = \delta^2::\delta^1::\delta^4
\]

Lemma 4.21. This procedure enjoys some properties:

1. If \( w \rightarrow w' \), then \( w(\sigma) = w'(\sigma) \).
2. \( w(\sigma) = \sigma' \) implies that \( w^*(\sigma') = \sigma \).
Proof. We prove the previous statements one by one:

1. By induction on the length of the reduction of \( w \), looking at the last rewriting rule applied:
   
   - in the case where \( w \) is of the form \( w = w_1 \delta^k w_2 \):
     
     * If \( i = j \) and \( k = k' \), \( w(\sigma) = w_1(\text{pop}_i(\delta^k, w_2(\sigma))) = w_1(w_2(\sigma)) \) and \( w \approx w_1 w_2 \).
     
     * If \( i < j \) (\( j = i + l, l > 0 \)), then \( w \approx w_1 \delta^k_{j-1} \delta^*_k w_2 = w' \).

     \[
     w(\sigma) = w_1(\text{pop}_i(\delta^k, \text{push}_j(\delta^{k'}, w_2(\sigma))))
     \]
     
     \[
     = w_1(\text{pop}_i(\delta^k, w_2(\sigma)_{\mid i+1} \cdot \text{push}_{l-1}(\delta^{k'}, \mid_{i+1} w_2(\sigma)))) \text{ by def. of push}
     \]
     
     \[
     = w_1(\text{pop}_i(\delta^k, w_2(\sigma)_{\mid i} \cdot \text{push}_{l-1}(\delta^{k'}, \mid_{i+1} w_2(\sigma)))) \text{ or undefined, by Remark 4.15}
     \]

     \[
     w'(\sigma) = w_1(\text{push}_{i+l-1}(\delta^{k'}, \text{pop}_i(\delta^k, w_2(\sigma))))
     \]
     
     \[
     = w_1(\text{push}_{i+l-1}(\delta^{k'}, w_2(\sigma)_{\mid i} \cdot \mid_{i+1} w_2(\sigma)))
     \]
     
     \[
     = w_1(\text{push}_{i+l-1}(\delta^{k'}, w_2(\sigma)_{\mid i} \cdot \mid_{i+1} w_2(\sigma)), \text{ or undefined}
     \]

   - The other reduction rules do not change the exponential structure of the trace.

2. By induction on \( w \):
   
   - \( w = \epsilon: \epsilon(\sigma) = \sigma = \epsilon^*(\sigma) \).
   
   - \( w = tw' \): Induction hypothesis is that \( w'(\sigma) = \tau' \) implies that \( w'^*(\tau') = \sigma' \). By case on \( t \):
     
     * \( t = \delta^k \): By definition, \( (\delta^k w')(\sigma) = \text{push}_i(\delta^k, w'(\sigma)) \). And if we take \( w'(\sigma) = \tau' \), then \( \text{push}_i(\delta^k, w'(\sigma)) = \text{push}_i(\delta^k, \tau') \). And finally, by remark 4.19, we have that \( (w'^* \delta^*_k)(\text{push}_i(\delta^k, \tau')) = w'^*(\tau') \), and by I.H \( w'^*(\tau') = \sigma \).

     * \( t = \delta^k \): By definition, \( (\delta^*_k w')(\sigma) = \text{pop}_i(\delta^k, w'(\sigma)) = \text{pop}_i(\delta^k, \tau') \). And \( (w'^* \delta^k)(\text{pop}_i(\delta^k, \tau')) = w'^*(\tau') = \sigma \) by I.H.
CHAPTER 4. OPTIMAL REDUCTION FOR SOFT LINEAR LOGIC

By induction on $w$:

- $w = c$: direct.
- $w = \delta^k w'$:

$$l(\delta^k :: w')(\delta^k :: \sigma) = \delta^k_{t+1}(l(w')(\delta^k :: \sigma)) \quad \text{(IH)}$$

$$= \text{push}_{t+1}(\delta^k, \delta^k :: w'(\sigma))$$

$$= \delta^k :: \text{push}_{t}(\delta^k', w'(\sigma))$$

$$= \delta^k :: w(\sigma)$$

- $w = \delta^k w'$: same reasoning.
- $w = tw', t = p_i, q_i, q_i^*$: $l(tw')(\delta^k :: \sigma) = l(w')(\delta^k :: \sigma) = \text{IH} \delta^k :: w'(\sigma) = \delta^k :: w(\sigma)$.

**COROLLARY 4.22.** Three easy consequences of the previous lemma will be of some use:

- $l(w) \text{push}_0(\delta^k, \sigma) = \text{push}_0(\delta^k, w(\sigma))$.
- $\text{push}_0(\delta^k, w(\text{pop}_0(\delta^k, \delta^k :: \sigma))) = \delta^k :: w(\sigma)$.
- $\text{pop}_0(\delta^k, l(w)(\delta^k :: \sigma)) = w(\sigma)$.

We define an ordering relation on exponential stacks:

**DEFINITION 4.23 ($\sigma \sqsubseteq \sigma'$).** We will say that $\sigma$ is less defined than $\sigma'$ and denote is by $\sigma \sqsubseteq \sigma'$, defined by induction on $\sigma$:

$$\square :: \sigma' \sqsubseteq t :: \tau' \iff \sigma' \sqsubseteq \tau', \text{ for any } t \quad \delta^k :: \sigma' \sqsubseteq t :: \tau' \iff \sigma' \sqsubseteq \tau' \text{ and } t = \delta^k$$

This relation enjoys some basic properties:

**PROPOSITION 4.24.** $\sqsubseteq$ defines a partial ordering on exponential stacks: it is reflexive, transitive and anti-symmetric.

**REMARK 4.25.** By definition, for all $\sigma$, $\emptyset \sqsubseteq \sigma$.

Now, we show that the ordering is preserved by application of a trace:

**PROPOSITION 4.26.** If $\sigma \sqsubseteq \sigma'$, then for all $w$ s.t $w(\sigma)$ is well-defined, $w(\sigma) \sqsubseteq w(\sigma')$.

**Proof.** By induction on $w$:

- $w = c$: $\epsilon(\sigma) = \sigma$, $\epsilon(\sigma') = \sigma'$ and $\sigma \sqsubseteq \sigma'$.
- $w = t :: w'$, with $t_i = p_i, q_i, q_i^*$. $t :: w'(\sigma) = w'(\sigma) = t :: w'(\sigma') = w'(\sigma')$, and by IH, if $w'(\sigma) = \tau$, then $w'(\sigma') = \tau'$ and $\tau \sqsubseteq \tau'$.
- $w = \delta^k :: w'$: By IH, if $w'(\sigma) = \tau_1$ then $w'(\sigma') = \tau'_1$ et $\tau_1 \sqsubseteq \tau'_1$. And it is easy to check that if $\tau_1 \sqsubseteq \tau'_1$, then $\text{push}_{i}(\delta^k, \tau_1) \sqsubseteq \text{push}_{i}(\delta^k, \tau'_1)$. So, if $w(\sigma) = \tau$, then $w(\sigma') = \tau'$ and $\tau \sqsubseteq \tau'$.
- $w = \delta^{k*} :: w'$: By IH, if $w'(\sigma) = \tau_1$ then $w'(\sigma') = \tau'_1$ et $\tau_1 \sqsubseteq \tau'_1$. And it is easy to check that if $\tau_1 \sqsubseteq \tau'_1$, then $\text{pop}_i(\delta^k, \tau_1)$ is well-defined, $\text{pop}_i(\delta^k, \tau'_1)$ is also, and $\tau_{1_i+1} \sqsubseteq \tau'_{1_i+1}$. So, if $w(\sigma) = \tau$, then $w(\sigma') = \tau'$ and $\tau \sqsubseteq \tau'$.
4.3 ReadBack

The ReadBack is the operation that consists in producing a lambda-term from a sharing graph in normal form. This is not a trivial operation. Here, we will define the readback on the traces of a Sharing graph.

4.3.1 Preliminary definitions

We will readback pure lambda-terms in normal form. We will divide a term in normal form by level (independent of the Sharing graphs depth). The root of the term is at level $\varepsilon$, and so are the binders chains and the applications chain. The $m$ arguments of the applications chain of level $\varepsilon$ are $m$ head-subterms, with level $1, \ldots, m$.

Each head-subterm will be characterised by the number of its lambda binders (the length of the lambda chain), the number of applications (the length applications chain) and its head variable. Each lambda and each application of a global term belong to a unique head-subterm. We will represent levels by lists of integers denoted by $1, 1', 1_1, 1_2, \ldots$ and equipped with a prefix ordering $\leq$. The constant $\varepsilon$ will correspond to the empty list. In general, if a head-subterm of level $l$ has $m$ application nodes, then it has $m$ argument head-subterms, of level $1::l, \ldots, m::l$.

The first head-subterm corresponds to the first argument.

We will impose a certain form on the names of variables. Bound variables will range over $x_{l,k}$ where $l$ is an integer list (representing the level of the binder chain in the term) and $k$ an integer (representing the number of the lambda in the lambda chain, counting top-down and starting from 1), and so our trees will respect Barendregt’s convention (that all bound variables have different names).

Now, let us properly define the set of Böhm Trees:

**Definition 4.27 (Böhm Trees BT).** The set $BT$ is the least set s.t. :

- $(\lambda x_{l,1} \cdots \lambda x_{l,n} \cdot y, []) \in BT$.
- If $b_1 \in BT, \ldots, b_m \in BT$, then $(\lambda x_{l,1} \cdots \lambda x_{l,n} \cdot y, [b_1, \ldots, b_m]) \in BT$.

We use bracket notation for subterms to stress the fact that the sons of a node are completely ordered.

Take a look at the lambda-term drawn in Figure 4.4, it is the graphical representation of, up to alpha-renaming, church integer 2: $\lambda sz.(s)(s)z$. It is in normal form, so it corresponds to a Böhm tree. It has three head-subterms. The head-subterm of level $\varepsilon$ has two binders and one application, and its head variable is the bound variable $s$. So its representation in our Böhm tree syntax is $(\lambda x_{z,1} x_{z,2} x_{z,1}[b_1])$, where $b_1$ is its subterm. The head-subterm of level 1 contains no lambdas and only one application, and has for head-variable $s$, again. So,
The size of a tree will be denoted by $|b|$ and correspond to the number of nodes of the tree. To preserve Barendregt’s convention, we will make use of alpha-renaming and denote by $b\{l_1 \to l_2, k\}$ the renaming of the $x_{i,k}$ by $x_{i,k}$ in the binders and variable occurrences names. We note simply $b\{l \to l\}$ for $b\{l_1, k \to l_2, k\}$. We can also define (restricted) substitution on those trees:

**Definition 4.28.** We restrict substitution to those particular cases which preserve the normal form property.

- $b\{z/z'\}$:
  
  - $(\lambda x_{i,1} \cdots x_{i,n} \cdot [b_1, \ldots, b_m])\{z/z'\} = (\lambda x_{i,1} \cdots x_{i,n} \cdot [b_1\{z/z'\}, \ldots, b_m\{z/z'\}])$
  
  - $(\lambda x_{i,1} \cdots x_{i,n} \cdot y, [b_1, \ldots, b_m])\{z/z'\} = (\lambda x_{i,1} \cdots x_{i,n} \cdot y, [b_1\{z/z'\}, \ldots, b_m\{z/z'\}]), y \neq z'$

- $b\{(y) b'/z\}$:
  
  - $(\lambda x_{i,1} \cdots x_{i,n} \cdot z, [b_1, \ldots, b_m])\{(y) b'/z\} = (\lambda x_{i,1} \cdots x_{i,n} \cdot y, [b_1\{(y) b'/z\}, \ldots, b_m\{(y) b'/z\}])$
  
  - $(\lambda x_{i,1} \cdots x_{i,n} \cdot y, [b_1, \ldots, b_m])\{(y) b'/z\} = (\lambda x_{i,1} \cdots x_{i,n} \cdot y, [b_1\{(y) b'/z\}, \ldots, b_m\{(y) b'/z\}]), y \neq z$

Substitutions and alpha-conversion notations can be naturally extended to lists of trees $[b_1, \ldots, b_m]$. An important characteristic of a Böhm tree is the set of levels (integer lists) appearing in it. For instance the set of levels of the example of Figure 14 is $\{\varepsilon, 1, 11\}$. More formally, let us define the Levels function, from Böhm Trees to sets of integer lists:

$$\text{Levels}(\lambda x_{i,1} \cdots x_{i,n} \cdot y, [b_1, \ldots, b_m]) = \{\varepsilon\} \cup \{i : l, l \in \text{Levels}(b_i), \forall i.s.t 1 \leq i \leq m\}$$

### 4.3.2 Characteristic traces

The readback will rely on particular traces, proper to a head-subterm, that we will call the characteristic trace of a head-subterm, which will allow to retrieve the number of lambdas and applications of a head subterm, together with its head variable.

The ReadBack procedure will take in input a set of traces in normal form and produce a $\lambda$-term in normal form that we will directly identify to its Böhm tree representation. Intuitively, each trace of the semantics of a sharing graph in normal form corresponds to one or several head-subterms in the corresponding lambda-term (Section 14.3 will provide examples). The point in the readback procedure is to find which trace corresponds to which head-subterm, and to understand how a trace indeed defines the number of lambdas and applications of the head-subterm, and identifies the head-variable.

**Trace of level $\varepsilon$** Looking at a Sharing graph in normal form, one can see that there is a travel which starts from the root and goes through lambda nodes, entering each node by the principal port and going out by the subterm port (adding $q^n$ to the trace). Then it will then go down application or multiplexor nodes, entering by an auxiliary port an thus going out by the principal port (adding $q$’s and $\delta^k$’s to the trace) and finally reach a variable, which is identified according to the end of the trace: it can be a free variable port and then the trace stops immediately with free port token $z$, or the travel can go back to a binder (adding a $p$ token to the trace), go up the lambda chain to finally end up at the root of the sharing graph crossing for instance $j$ lambdas (thus adding $j$ times the $q$ tokens). This completely
4.3 READBACK

characterises the ε head-subterm. At level ε, we cannot enter a multiplexor by its principal port, so there is no ambiguity.

Since the Sharing graph is by definition in normal form, the characteristic trace of level ε, further on called \( \text{carac}_ε \), is of the form \( w_1 w_2^* \). One can observe that the \( w_2 \) part corresponds to the initial crossing of lambda’s, while the \( w_1 \) part corresponds to the crossing of applications, multiplexors and to the recovery of the head-variable. We will divide the trace accordingly, so \( \text{carac}_ε \) is such that:

- \( \text{for}(\text{carac}_ε) \) is of the form \( \text{for}(\text{carac}_ε) = \bot q^n \)
- \( \text{back}(\text{carac}_ε^-) \) is of the form \( \text{back}(\text{carac}_ε^-) = \text{esc}_ε^- q^m \), with \( \text{esc}_ε^- \) of the form:
  - (1) \( \text{esc}_ε^- = z \), or
  - (2) \( \text{esc}_ε^- = \bot q^p \)

Note that we impose that the \( \text{for}(\text{carac}_ε) \) contains no exponential token, so it implies that \( \text{carac}_ε(\emptyset) \) is well-defined. We will say that \( \sigma_ε = \emptyset \), and that \( \text{carac}_ε(\emptyset) = \tau \) for some \( \tau \). This resulting exponential stack will be useful the the argument head-subterms.

So, there are \( n \) lambdas in the lambda’s chain, \( m \) applications in the applications chain. In case (1), the head-variable is the free variable \( z \), and in case (2) it is \( x_{ε, j+1} \) (with our convention on bound variables name).

**Head-subterm of level i:I** Given the characteristic trace \( \text{carac}_l \) of the head-subterm of level \( l \), how do we find the characteristic trace of the head-subterm of level \( i : l ? \) Again, looking at a sharing graph in normal form, one can see that if one takes \( \text{carac}_l \) backward, one enters all the applications of the \( l \) head-subterm by the principal port. So, to access the root of the \( i : l \) head-subterm, one has to choose the right moment to go out of an application by the argument port rather than by the context port. Therefore, to the head-subterm of level \( i : l \), we will associate a (multiplicative) prefix denoted by \( \text{acc}_{i,l} \), and defined as follows: \( \text{acc}_{i,l} = \text{esc}_l q^{i-1} p \). The \( \text{esc}_l \) part reaches the applications chain, the \( q^{i-1} \) makes going out \( i - 1 \) applications by their context port, and the final \( p \) makes going out the application by its argument port, which corresponds to accessing the root of the \( i \)-th argument. There is also a result exponential stack associated to the trace of the previous head-subterm, \( \text{carac}_l \), that we will reuse under the name of \( \sigma_{i,l} \).

So the characteristic trace of level \( i : l \) \( \text{carac}_{i,l} \) must be such that:

- \( \text{carac}_{i,l}(\sigma_{i,l}) = \tau \), for some \( \tau \)
- \( \text{for}(\text{carac}_{i,l}^-) \) is of the form \( \text{for}(\text{carac}_{i,l}^-) = \text{acc}_{i,l} q^n \)
- \( \text{back}(\text{carac}_{i,l}^-) \) is of the form \( \text{back}(\text{carac}_{i,l}^-) = \text{esc}_{i,l}^- q^m \), with \( \text{esc}_{i,l}^- \) of the form:
  - (1) \( \text{esc}_{i,l}^- = z \), or
  - (2) \( \text{esc}_{i,l}^- = \text{acc}_{i,l} q^p \), \( l' \leq l \) and \( \sigma_{i,l'} \sqsubseteq \tau \)

It is only a slight generalisation of the case of the head-subterm of level \( \varepsilon \), except to retrieve the head-variable, in the case where it is bound. Indeed, we now have to identify not only the position of the binder in its lambda’s chain, but also the head-subterm to which it belongs. This is why \( \text{esc}_{i,l}^- \) must be of the form \( \text{esc}_{i,l}^- = \text{acc}_{i,l} q^p \), with \( l' \leq l \). It must be the access trace of a head-subterm whose position in the term is “higher”: thus, \( l' \leq l \). However, this is not enough to ensure that the correct head-subterm for the binder is uniquely identified: there can be several head-subterms whose access traces are equal, then they differ only by their exponential stacks. So, the relation \( \sigma_{i,l} \sqsubseteq \tau \) must also hold to disambiguate.

**Example** Figure 1.5 is a shared representation of church integer 2, indeed, the only application node is shared by the two multiplexors. The dashed line represents the travel of the
characteristic trace of the head-subterm of level 1, we will call it \( \text{carac}_1 \). Following the arrow, it starts from the root, exits the first lambda by its variable port and enters the application by its principal port. We call this part of the characteristic trace the access trace, because it leads to the root of the head-subterm that we are targeting. Then it enters a multiplexor by its principal port, goes out by the auxiliary port numbered 1 and goes in the other multiplexor by the auxiliary port numbered 2. Then it goes back through the application node, this time by its context port. Finally, it goes back in the lambda node by its variable port and reaches the root. The part after the application node is called the escape trace and is denoted by \( \text{esc}_1 \): it is the part of the travel done after the applications chain. The trace associated to the travel is \( \text{carac}_1 = \perppq\delta^1p^*p^*\perp. \) It is indeed of the form \( w_1w_2^* \) for \( (\text{carac}_1^-) = \perppp and \( \text{back}(\text{carac}_1^-) = \perppq. \) And if we forget the exponential informations: for \( (\text{carac}_1^-) = \perp and \( \text{back}(\text{carac}_1^-) = \perpq. \) We can deduce from this that the head-subterm of level 1 has no lambda, one application, and that its head-variable is \( x_{\varepsilon,1}. \)

### 4.3.3 ReadBack algorithm

The following intermediary procedure \( \text{RB}(\cdot) \), equipped with a context \( P \), takes in input a trace \( \text{carac}_l \), which is the characteristic trace of the head-subterm of level \( l \), an exponential stack \( \sigma \) and the level of the current head-subterm. After possible recursive calls it produces in output a Böhm tree which is the tree corresponding to the lambda-term. When there is no ambiguity, we will omit to put the context explicitly.

**Definition 4.29 (\( \text{RB}_P \)).** Let us first define the \( \text{RB}_P \) procedure, an intermediary function:  
\[
\text{RB}_P(\text{carac}_l, \sigma_l, l) = (\lambda x_{1,1} \cdots x_{l,n}, y_1, \ldots, y_m)[b_1, \ldots, b_m]) \text{ if }
\]

- \( \text{for}(\text{carac}_l^-) = \text{acc}_q^n \)
- \( \text{back}(\text{carac}_l^-) = \text{esc}^{-q^n}, \text{ with: } \)
  \[
  \begin{align*}
  y &= z, & \text{if } \text{esc}^{-z} = z \\
  y &= x_{l',j+1}, & \text{if } \text{esc}^{-z} = \text{acc}_{l'q^n}, l' \leq l \text{ and } \\
  \sigma_{l'} &\subseteq \text{carac}(\sigma_l)
  \end{align*}
\]
- \( \text{for } 1 \leq i \leq m : \)
  - \( \sigma_{i;1} = \text{carac}_i(\sigma_l) \)
  - \( \text{carac}_{i;1} \in P \text{ is s.t. } \text{for}(\text{carac}_{i;i}) = \text{esc}^{-q^{i-1}}pq^n \text{ and } \text{carac}_{i;1}(\sigma_{i;1}) \text{ is well-defined.} \)
  - \( b_i \text{ is s.t. } \text{RB}_P(\text{carac}_{i;1}, \sigma_{i;1}, i;i) = b_i \)

Now we can define the readback procedure on a Sharing graph \( S \) in normal form:
DEFINITION 4.30. We define $\text{carac}_e \in [S]$ s.t. for($\text{carac}_e$) $= \bot q^n$ and $\text{carac}_e(\emptyset)$ is well-defined. Then $\text{ReadBack}([S]) = b$ if $\text{RB}(\text{carac}_e, \emptyset, \varepsilon) = b$

In the sequel, we will use $\text{esc}(\text{carac}_e)$ to denote $\text{esc}_1$, and $\text{acc}(\text{carac}_e)$ to denote $\text{acc}_1$.

We might need to examine the result $b$ of a $\text{ReadBack}$. Each node of level $l$ of the Böhm tree produced has a characteristic trace $\text{carac}_e$, and an associated exponential stack $\sigma_i$. We will write $\forall (l, \text{carac}_e, \sigma_i) \in b$ even though those informations do not appear explicitly in the Böhm tree.

The following proposition states a kind of (limited) independence of the $\text{RB}()$ procedure w.r.t to an extension of the context. It will be of use for Proposition 4.50.

PROPOSITION 4.31. Let $S$ and $S'$ be two Soft sharing graphs in normal form s.t $FV(S) = \{v, v_1, \ldots, v_n\}$, $FV(S') = FV(S) \cup \{z\}$:

\[
\begin{align*}
\bot w \bot \in [S] &\iff z w p^r w z \in [S'] \\
v w \bot \in [S] &\iff v w p^r w z \in [S'] \\
\bot w v \in [S] &\iff z p v w \in [S'] \\
v_2 w v_1 \in [S] &\iff v_1 w v_2 \in [S']
\end{align*}
\]

Intuitively, the Sharing graph $S'$ is obtained by applying a fresh free variable $z$ to $S$.

This defines an bijection $\text{Inj}(\cdot)$ between traces of $[S]$ and a subset of those of $[S']$. $S'$ may contain other traces, but not of the form of $\text{Inj}(w), w \in [S]$. In particular, all traces starting by $p^r z$ or ending by $z p$ are images of $\text{Inj}(\cdot)$, and free variables of $S'$ do not appear in $[S]/\text{Inj}([S'])$. Note that $\text{Inj}(\cdot)$ does not alter the exponential structure of a trace, so $w(\sigma) = \tau$ iff $\text{Inj}(w)(\sigma) = \tau$.

If $\text{ReadBack}([S]) = \text{RB}([S] \text{carac}_e, \emptyset, \varepsilon) = b$, then:

1. For all $(\text{carac}_e, \sigma_i, l) \in b$, for any $l'$, $(\text{Inj}(\text{carac}_e), \sigma_i', l' \cdot l') \in \text{RB}([S] \text{Inj}(\text{carac}_e), \sigma_i', l')$ with $\sigma_i \subseteq \sigma_i'$.

2. For all $l \in \text{Levels}(b)$, $\text{RB}([S] \text{Inj}(\text{carac}_e), \sigma_i', l') = b[l \rightarrow l' \cdot l']$

The first statement expresses that for any exponential stack, and for any base level, each step of $\text{RB}(\text{Inj}(\text{carac}_e), \sigma_i', l' \cdot l')$ will identify $\text{Inj}(\text{carac}_e)$ as the characteristic trace of the head-subterm of level $l' \cdot l'$.

The second statement expresses that the result of $\text{RB}([S] \text{Inj}(\text{carac}_e), \sigma_i', l')$ is $b$, up to alpha-renaming.

Proof. First we prove statement 1 by induction on $l$:

- $l = \varepsilon$: by definition, $(\text{Inj}(\text{carac}_e), \sigma_i', l') \in \text{RB}(\text{Inj}(\text{carac}_e), \sigma_i', l')$ and by Remark 4.26, $\emptyset \subseteq \sigma_i'$.

- $l = i \cdot i_1$: By I.H $\text{RB}([S] \text{Inj}(\text{carac}_{i_1}), \sigma_{i_1'}, l_1 \cdot l') \in \text{RB}([S] \text{Inj}(\text{carac}_{i_1}), \sigma_i, l')$, and $\sigma_{i_1} \subseteq \sigma_{i_1'}$.

By definition $\text{carac}_{i_1}(\sigma_{i_1}) = \tau_{i_1} \cdot \sigma_{i_1}$ and $\text{Inj}(\text{carac}_{i_1})(\sigma_{i_1'}) = \sigma_{i_1' \cdot i_1}$, and by Proposition 4.26, $\sigma_{i_1} \subseteq \sigma_{i_1' \cdot i_1}$.

By case on the shape of $\text{esc}(\text{carac}_{i_1})$:

- $\text{esc}(\text{carac}_{i_1}) = \bot w'$: $\text{carac}_{i_1 \cdot i_1}$ is the unique trace of $[S]$ s.t. for($\text{carac}_{i_1 \cdot i_1}$) $= \text{esc}(\text{carac}_{i_1}) \cdot q^{i_1 \cdot p q^n} = \bot w' q^{i_1 \cdot p q^n}$ and $\text{carac}_{i_1 \cdot i_1}(\sigma_{i_1' \cdot i_1}) = \tau$.

By definition, $\text{esc}(\text{Inj}(\text{carac}_{i_1}))$ is of the form $\text{esc}(\text{Inj}(\text{carac}_{i_1})) = z p w'$, so $\text{Inj}(\text{carac}_{i_1 \cdot i_1})$ is the only trace of $S'$ s.t. for($\text{Inj}(\text{carac}_{i_1 \cdot i_1})$) $= z p w' q^{i_1 \cdot p q^n}$.

- Since $\sigma_{i_1} \subseteq \sigma_{i_1' \cdot i_1}$, $\text{Inj}(\text{carac}_{i_1 \cdot i_1})(\sigma_{i_1' \cdot i_1}) = \tau'$ for some $\tau'$.
- \text{esc(carac}_l) = vw', with \( v \) being a free variable of \([S]\). \text{carac}_l is the unique trace of \([S]\) s.t. for(carac}_l) = wv'q^{1-p}q^\tau and carac}_l(\sigma_{l,i}) = \tau for some \( \tau \).

By induction hypothesis and by definition, the escape trace \text{esc}(Inj(carac}_l)) is of the form \text{esc}(Inj(carac}_l)) = zvq', so Inj(carac}_l) is the only trace of \( S' \) s.t \text{for}(Inj(carac}_l)) = zvq'q^{-1}p^\tau, and, since \( \sigma_{l,i} \subseteq \sigma_{l,i}' \) and carac}_l(\sigma_{l,i}) = \sigma_{l,i}, Inj(carac}_l)(\sigma_{l,i}) = \tau' for some \( \tau' \).

Hence, \((\text{Inj(carac}_l), \sigma_{l,i}'ll, ll') \in \text{RB}_{S'1}(\text{Inj(carac}_l), \sigma_{l,i}, ll')\).

Now, we prove statement 2 by induction on \( b \) and using statement 1:

- \( b = (\lambda x_{1,1} \cdots x_{1,n}, y, []). \) For(carac}_l) = w'q^n and back(carac}_l) = w''q^0. By statement 1, \((\text{Inj(carac}_l), \sigma_{l,i}, ll') \in \text{RB}_{S'1}(\text{Inj(carac}_l), \sigma_{l,i}'ll, ll')\), and \text{for}(\text{Inj(carac}_l)) = w'q^n, and back(\text{Inj(carac}_l)) = w''q^0. Now, focusing on head variable \( y \):
  - if \( y = x_{1,i} \): then esc\( y = acc_i q^{-1}p, l_i \leq t \) and so esc\( \text{Inj(carac}_l) = acc_i q^{-1}p. \)
  - if \( y = v, v \in \text{FV}[\([S]\): then esc\( v = v \), and so Inj(esc\( v) = v. \)

So \( \text{RB}_{S'1}(\text{Inj(carac}_l), \sigma_{l,i}'ll, ll') = (\lambda w_{1,1} \cdots w_{1,n}, y(l \rightarrow ll'), []) = b(l \rightarrow ll'), \) for all \( l \in \text{Levels}(b). \)

- \( b = (\lambda x_{1,1} \cdots x_{1,n}, y, [b_1, \ldots, b_m]): \) we can apply the same reasoning as in the base case and apply induction hypothesis on the \( b_i \) subterms to conclude.

\( \square \)

\textbf{Proposition 4.32.} If \text{ReadBack}([S]) = b then there is less than \|b\| calls to the \text{RB} procedure, if you take the size of \( b \) as the size of the corresponding lambda-term, counting contractions.

\section*{4.4 Examples}

\subsection*{4.4.1 A high complexity function in ssg}

In our framework, it is delicate to determine whether or not a lambda-term admits an indexing s.t its reduction in Soft sharing graph is correct. Indeed, one can make indexes vary as one wishes, without any boxing condition, and furthermore, place arbitrary derelictions (unary multiplexor).

Our general purpose is to show that at least the terms typable in SLL admit a correct reduction. In this section we simply give an example of a lambda-term not typable in any SLL type system, be it equipped with second order types or recursive types\(^1\).

We denote the Church integer \( n \) by \( n = \lambda sz. (s) \cdots (s) z. \) Let \( H_2 = ((2) 2) 2. \) By beta reduction, \( H_2 \rightarrow^* 16. \) This term is hyper-exponential, and in general, let \( H_n = ((n) 2) 2. \) \( H_n \rightarrow^* 2^{2^n}. \)

Now, we can slightly generalise this example: let \( H_n = (n 2) 2. \) We will give a translation of \( H_n \) in SSG, and show that it is correct, that is, that the readback of the normal form of the Sharing graph produces the expected normal lambda term. We define several translations of integer 2 and a translation of integer \( n \) in SSG, given in Figure 1.

\textbf{Definition 4.33.} We define:

- \( 2_n, \) the natural translation of church integer 2 at level \( n. \)

\(^1\)Personal communication of Marco Gaboardi
4.4 Examples

• $\gamma_n$, a translation of integer $n$ with added derelictions: both head binders are at level 0; the top application is at level 0, and there is $n - 1$ derelictions on the principal port before reaching the multiplexor. The second application is at level 1 and has $n - 2$ derelictions on its principal port, and so on until the last application, at level $n - 1$, which has no dereliction on its principal port.

• $\tilde{\gamma}_{2,k}$. We have $k$ multiplexors after the binders, descending towards two application nodes, and again $k$ others, met by their principal port, going out the last application by its argument port. Note that for $k = 0$; it is a decoration of church integer 2.

Now, we can define the global term:

$$S_{H_n} = ((\gamma_n) 0 \ 2_n) 0 \tilde{\gamma}_{2,0}$$

This is indeed a decoration of $(n \ 2) 2$. We put zero in indice of the parenthesis since those applications are at level zero.

We have a few intermediary lemmas to show that $S_{H_n}$ reduces correctly.

**Lemma 4.34.** $(\gamma_n) 2_n \rightarrow^* \lambda x. (2_n) \cdots (2_{n-1}) x$

**Proof sketch.** The first beta redex being fired, $2_n$ is connected by its root to the principal port of the $n$-ary multiplexor of $\gamma_n$. $2_n$ is closed and in normal form, so its copy does not let residuals of the multiplexor or of the derelictions, therefore we are left with $n$ copies of $2_{n-1}$. The first copy, eventually reaching the argument port of the top application, undergoes $n - 1$ derelictions, so has level 0 once it reaches it, and so on until the last application, whose copy does not go through any dereliction and thus keeps level $n - 1$.

**Lemma 4.35.** $(\tilde{\gamma}_{2,n}) 2_{n+1} \rightarrow^* \tilde{\gamma}_{2,n,2k+1}$

**Proof sketch.** Observe that the sharing graph reduces to $\lambda z. (\tilde{\gamma}_{2,n,k}) (\tilde{\gamma}_{2,n,k}) z$. The first copy, in the course of duplicating the second one, creates two beta redexes (with correct indexes on application and lambda nodes) and then, once the redex fired, duplication is stopped, and the...
Lemma 4.36. $\text{ReadBack}(\lceil \frac{n}{2} \rceil, k) = 2^{k+1}$

Proof sketch. For $k = 0$, the readback produces church integer 2. Then, each time one adds a multiplexor in the chains, it doubles the result, so in general we end up with reading back $2^{k+1}$.

So this family of terms is actually an economic (in number of sharing graph nodes) representation of powers of 2. Actually, a very close family of Sharing graphs had been used by Asperti in [Asp96].

Proposition 4.37. $S_{H_n}$ reduces to a normal form $S'_{H_n}$ and $\text{ReadBack}(\lceil S'_{H_n} \rceil) = 2^{2^n}$.

Proof. $S_{H_n} = ((\lambda x)(n_2)0_{2n,0} \to (\lambda_0 z_2)(n_2-1)z_0 \to (2_0) \cdots (2_{n-1})2_{n,0} \to (2_0) \cdots (2_{n-1})2_{n,0} \to (2_0, 2^{n-1})$, and $\text{ReadBack}(\lceil 2_{2^n, 2^{n-1}} \rceil) = 2^{2^n-1+1} = 2^{2^n}$.}

The sizes of $2_{2n,k}$ and of $2_n$ remain constant for all $n$. Note that the there is only one node of depth $n + 1$, $n + 2$ and $n + 3$, 7 nodes of degree $n$.

However, the size of $\frac{n}{2^n}$ increases by a quadratic factor. It is easy to check that $|\frac{n}{2^n}| = 6$: two lambda nodes, two application nodes, a binary multiplexor and a unary one. Then, we have that $|\frac{n}{2^n}| = |\frac{n - 1}{2^{n-1}}| + n$: indeed, we add 1 application node, and $n - 1$ unary multiplexors. So, in general $|\frac{n}{2^n}| = 3 + \frac{n^2 + n}{2}$, $|S_{H_n}| = 14 + \frac{n^2 + n}{2} = O(n^2)$. But this increase is only for depth 0: the depth of the multiplexors. There is only one node at the depths 1 to $n - 1$, the corresponding application node of $\frac{n}{2^n}$.

So, on the whole, one obtains for $S_{H_n}$ a polynomial $P(S_{H_n}) = \Sigma_{i=0}^{n+3}|S_{H_n}|X^i = X^{n+3} + X^{n+2} + X^{n+1} + 7 \times X^n + X^{n-1} + \cdots + X + 3 + \frac{n^2 + n}{2}$.

Therefore, the bound we obtain on the length of any reduction sequence is that $S_{H_n}$ reaches a normal form in $O(n^{n+3})$ steps. Actually, one could use an alternate encoding to enforce a bound $O(4^{n+3})$. Instead of having $n$-ary multiplexors, together with unary multiplexors, one could encode the $n$ duplications with $n - 1$ binary multiplexors, and one unary multiplexor one the topmost application node. This would also cause the creation of $n$ copies of level 0..$n - 1$. In this case, $|\frac{n}{2^n}| = 2 \times n + 2$, and for all $n$, the maximum multiplexor degree of the Sharing graph is 2, and the complexity bound is indeed of the form $O(4^{n+3})$. However, this slight encoding variation is arguably not formally speaking a decoration of $(n/2)2$ in ssg, because of the tree of multiplexors.

This reduction bound comes in contradiction with the theoretical complexity of this function. However, the ReadBack runs in $O(2^{2^{n+1}})$, which is the number of steps required to produce the church integer $2^{2^{n+1}}$.

This suggests that the readback operation is not to be overlooked in the study of the complexity of Optimal reduction w.r.t the lambda calculus reduction. However, with respect to its output, in this example, the ReadBack procedure runs in reasonable time. It is unclear to us whether it would be possible to enhance the ReadBack from a complexity point of view, to get a reasonable bound, not on the size of the result, but on the size of its input, that is, a sharing graph in normal from.
4.4 EXAMPLES

4.4.2 An involved ReadBack run

With our ReadBack procedure, one can easily follow the traces to get the number of lambdas and application of a node, but has to be careful to identify correctly the head variable. With $\Delta = \lambda x. (x) x$, the normal form of the term $E = (\Delta) \lambda z. (f) (x) z$ is

$$(f) \lambda z_1. (f) \lambda z_2. ((f) \lambda z_3. (z_1) (z_3)) ((f) \lambda z_4. (z_1) (z_1) z_4) z_2$$

A correct translation of this term, and the corresponding normal form is given in Figure 4.7. It has been necessary to get a correct reduction to place a dereliction behind the left auxiliary port of $\Delta$, and to put ad-hoc indexes. In the result Sharing graph, all the nodes have index 0. Observe that in the Sharing graph normal form, we have only one lambda node, while there are 4 binders in the lambda-term in normal form.

Now, the reader can check that $\text{ReadBack}([S'_{nf}])$ follows the steps given in the array of Figure 4.8. In the first column, we put the level $l$ of the node currently being read back, the multiplicative stack $\sigma_l$ that it is given, and for which it must be correct, and the access trace $\text{acc}_l$. The second column gives the $\text{for}()$ and the $\text{back}()$ parts of the characteristic trace. Mind that we left the multiplicative tokens appear, and that we omit the indexes (all at 0).

The last column shows the information we can draw from $\text{carac}_l$: the number $n$ of binders, the number $m$ of applications and the head variable. When needed, we denote by $\tau$ the stack $\text{carac}_l(\sigma_l)$; finally we give the pair $(\lambda x_{1,1} \cdots x_{1,m}. y[b_{1,1}, \ldots, b_{m,1}])$, where the $b_i$’s refer to subsequent calls to $\text{RB}$.

Regarding the order of the lines, we present the array as if the readback was doing a depth-first search to build the Böhm tree, and we represented by double lines the point where readback goes back to a different subterm. The head-subterm of level 11 has two argument

Figure 4.7: Initial translation $S$ of $E$ and its normal form $S'_{nf}$
head-subterms, the first one, of level 111 is shown in the line just below, and the second one, of level 211, is put only after the head-subterm of level 1111.

For the first line of the array, we start at the root, with an empty exponential stack. The for() part of the trace is empty: there is no lambda node. Then back(carac) = fδ1δ1: it is the only correct travel starting from the root. Therefore, we have one application, with head-variable f, and one argument.

For the head-subterm of level 11, we have that σ_{11} = δ^1δ^1, and acc_{11} = fp. Indeed, esc(carac) = f, and we need to explore the first argument of the chain.

This example illustrates, that a given trace may be used several times: it is the case for the two subterms 1211 and 1111:

\[ carac_{1211} = carac_{1111} = fppδ^1δ^1qδ^2q*p^*f \]

The two subsequent head-subterms have also the same trace, carac_{1111} = carac_{1211}, carac_{11111} = carac_{11121}. But note that even though their characteristic traces are equal, the head-subterms of level 1111 and of level 111211 have different head-variables, since their exponential stack is different.

We have involved bindings to solve, since their is one lambda in the normal form of the sharing graph, representing in fact 4 lambda nodes in the lambda term. In the subterm 111, you can see that the resulting stack τ is equal to (δ^1)^4, and that the binder of the head variable has for incoming stack δ^1δ^1. This illustrates the need for an inequality constraint.
4.5 Proof of the main result

4.5.1 Soft Proof-nets

In this section, we will present Soft Proof-nets and Soft sequent calculus, which types soft lambda-terms. Then we will prove a limited invariance property on the GoI: that Proof-net reduction preserves ReadBack, so even if Proof-net reduction alters the GoI of a Proof-net, it leaves it essentially unchanged regarding our purpose.

In Chapter 1, Figure 1.14 gives the inductive rules for building Intuitionistic Soft Proof-nets, together with the cut-elimination rules. One can see that it is very close indeed to Sharing graphs Optimal reduction rules, except for the duplication operation: in Proof-nets cut-elimination, duplication is performed box-wise, while it is done node-wise in Sharing graphs Optimal reduction.

In the following we will write $\Pi : \Gamma \vdash t : A$ for “$\Gamma \vdash t$ has a derivation in sll and $\Pi$ is the Soft Proof-net corresponding to the derivation.”

We give a translation of Soft Proof-nets into Soft sharing graphs:

**Definition 4.38 ($\text{Tr}(P)$).** Let $P$ be a Soft Proof-net, $\text{Tr}(P)$ is obtained by turning Axioms and Cuts into simple wires, $\rightarrow r$ nodes into $\lambda$ nodes, $\rightarrow l$ nodes into $!$. The Box doors do not appear anymore, and each node is indexed by its depth in the Proof-net.

We extend the $\llbracket \rrbracket$ notation to Proof-nets: $\llbracket P \rrbracket = \llbracket \text{Tr}(P) \rrbracket$.

**Lemma 4.39.** If $P$ is a Soft Proof-net, $\llbracket P \rrbracket$ is deadlock-free.

**proof sketch.** The proof relies on the two following arguments:

1. For any Soft Proof-net $P_{nf}'$ associated with a cut-free Soft derivation, by definition $\llbracket P_{nf}' \rrbracket$ is deadlock free.
2. If $P_1 \rightarrow P_2$ and $\llbracket P_2 \rrbracket$ is deadlock-free, then $\llbracket P_1 \rrbracket$ is deadlock-free.

**Corollary 4.40.** If $P$ is a soft Proof-net, then $\text{Tr}(P)$ is deadlock-free.

**Proof.** Since, $\llbracket P \rrbracket = \llbracket \text{Tr}(P) \rrbracket$, we apply Proposition 4.9 to conclude that $S$ is deadlock-free.

**Lemma 4.41.** If $P : \Gamma \vdash t : A$ is a Soft type derivation, and $P \rightarrow^* P_{nf}'$, then $P_{nf}' : \Gamma \vdash t_{nf}' : A$ and $t \rightarrow^* t_{nf}'$.
Now, we want to get correctness for the Optimal reduction of lambda terms typed in sll. We will mimic the proof of [BCDL07]. So, to achieve this, we will need a few intermediary steps: We already have that:

- Proof-net cut-elimination simulates beta-reduction (Lemma 4.41).
- Translation of a Proof-net in a Soft sharing graph does not change GoI (by Definition 4.38).
- GoI is invariant by Optimal reduction (by Lemma 4.8).

We have left to prove that:

- The ReadBack on the normal form of the GoI of a Proof-net is preserved by cut-elimination.
- ReadBack of the GoI of a Proof-net corresponding to an sll cut-free derivation is the underlying lambda-term.

The first point is the subject of the next section.

### 4.5.2 Preservation of ReadBack by cut-elimination

The first result that we want to prove is an invariance of the ReadBack by the Proof-nets reduction, that ReadBack([P_{nf}]) = b iff ReadBack([P]_{nf}) = b. Proof-net reduction alters the GoI semantics only in the case of duplication of a box comporting auxiliary doors, i.e in the case of a Box-Multiplexor cut-elimination step. This feature is also present in general Linear Logic.

We will start by an utilitarian lemma, showing that, the GoI traces of a Proof-net P₁ are essentially preserved by its reduction in a Proof-net P₂ by a Box-Multiplexor cut-elimination step. Figure 4.10 shows a step of Box-Multiplexor cut-elimination.

First, we define families of partial traces of P₁ and P₂, parametrised by an integer n, which will correspond to trace that cross several times the cut considered.
\textbf{Definition 4.42.} A partial trace \( w \) of \( \text{cross}_1(n) \) is a trace that starts at the auxiliary port \( k_1 \) of the multiplexor, goes \( n \pi' \), reaches again the multiplexor — this time by its principal port — goes in \( \Pi \), goes back to an auxiliary port \( k_1 \) of the multiplexor, and so on.

\( \text{cross}_2(n) \) is the image of \( \text{cross}_1(n) \) in \( P_2 \). It does the same travels in the Proof-nets \( \Pi' \) and \( \Pi \), and thus doesn’t cross the multiplexor anymore.

\begin{itemize}
  \item \( w \in \text{cross}_1(n) \) if \( w \) is of the form \( w = w_{\Pi_1} \delta^{k_1} l(w_{\Pi_1}') \delta^{k_1} \cdots w_{\Pi_1} \delta^{k_1} l(w_{\Pi_1}') \delta^{k_1} \)
  \item \( w \in \text{cross}_2(n) \) if \( w \) is of the form \( w = w_{\Pi_1} \delta^{k_1} \cdots w_{\Pi_1} \delta^{k_1} l(w_{\Pi_1}') \delta^{k_1} \).
\end{itemize}

\( s.t. \)

\begin{itemize}
  \item \( \forall i, 1 \leq i < n, y_{k_{i+1}}, y_{k_i} \in [\Pi] \) and \( w_{\Pi_n} y_{k_n} \) is a partial trace of \( \Pi \).
  \item \( \forall i, 1 \leq i \leq n, \bot_{\Pi'} w_{\Pi'} \bot_{\Pi'} \in [\Pi'] \).
\end{itemize}

By convention, \( \text{cross}_3(0) \) and \( \text{cross}_2(0) \) contain only the empty trace. Note that in those families, there can be from zero to several different traces for a given \( n \).

\textbf{Lemma 4.43.} Now, we prove some properties on those families:

1. If \( w \in \text{cross}_1(n) \), then \( w \) is a partial trace of \( P_1 \), and if \( w \in \text{cross}_2(n) \), then \( w \) is a partial trace of \( P_2 \).

2. There exists \( w_1 \in \text{cross}_1(n) \) iff there exists \( w_2 \in \text{cross}_2(n) \) and \( w_1 \approx w_2 \).

\textit{Proof.} Statement (1) follows of the families definition. For statement (2), by definition of the families, if \( w_1 \in \text{cross}_1(n) \), then \( w_1 \) is a partial trace of \( P_1 \). Now, let us build the trace \( w_2 \) by turning the sequences \( w_{\Pi_1} \delta^{k_1} l(w_{\Pi_1}') \delta^{k_1} \) of \( w_1 \) in \( w_{\Pi_1} w_{\Pi_1}' \) for \( 1 \leq j \leq n \), by definition, it is in \( \text{cross}_2(n) \). So, \( w_1 \in \text{cross}_1(n) \) implies that there exists \( w_2 \in \text{cross}_2(n) \). Reasoning is the same for the converse: we turn the sequences \( w_{\Pi_1} w_{\Pi_1}' \) into \( w_{\Pi_1} \delta^{k_1} l(w_{\Pi_1}') \delta^{k_1} \).

Moreover, it is easy to check that \( \delta l(w_{\Pi_1}) \delta^{k_1} \to^* z_{1} z_{2} \) if \( w_{\Pi_1} \to^* z_{1} z_{2} \). So, \( w_1 \approx w_2 \).

Therefore, there exists \( w_1 \in \text{cross}_1(n) \) iff there exists \( w_2 \in \text{cross}_2(n) \) and \( w_1 \approx w_2 \). \( \square \)

Now, building on the two families we just defined, we will divide the complete traces of \( P_1 \) and \( P_2 \) in five classes, depending on the beginning and ending free port, and the number of Multiplexor’s crossing. In this definition, when \( w \in \text{cross}_1(n) \) appears, it can also mean that \( w = \varepsilon \), for the case where \( n = 0 \). Traces denoted by \( w_{\Pi_1}, w_{\Pi_2}, \ldots \) are traces of \( \Pi \), and those denoted by \( w_{\Pi_1'}, w_{\Pi_2'}, w_{\Pi_3'}, \ldots \) are traces of \( \Pi' \).

\textbf{Definition 4.44.} We divide the complete traces of \( P_1 \) in five classes:

\begin{itemize}
  \item (1.i) \( v_1 w_1 w_{\Pi_1} v_2 \), with \( v_1, v_2 \in \Gamma \cup \{ \bot_\Pi \} \), \( w_1 \in \text{cross}_1(n) \).
  \item (1.ii) \( v w_1 w_{\Pi_1} \delta^{k_0} l(w_{\Pi_1}') x_i \), with \( v \in \Gamma \cup \{ \bot_\Pi \} \), \( \bot_{\Pi'}, w_1 \in \text{cross}_1(n) \).
  \item (1.iii) \( x l(w_{\Pi_1}') \delta^{k_0} \delta^{k_1} w_{\Pi_1} v \), with \( v \in \Gamma \cup \{ \bot_\Pi \} \), \( w_1 \in \text{cross}_1(n) \).
  \item (1.iv) \( x l(w_{\Pi_1}') \delta^{k_0} \delta^{k_i} w_{\Pi_1} \delta^{k_i} l(w_{\Pi_1}') x_j \), with \( w_1 \in \text{cross}_1(n) \).
  \item (1.v) \( x l(w_{\Pi_1}') x_j \)
\end{itemize}

And we do the same for the traces of \( P_2 \):

\begin{itemize}
  \item (2.i) \( v_2 w_2 w_{\Pi_2} v_2 \), with \( v_1, v_2 \in \Gamma \cup \{ \bot_\Pi \} \), \( w_2 \in \text{cross}_2(n) \).
  \item (2.ii) \( v w_2 w_{\Pi_2} \delta^{k_0} x_i \), with \( v \in \Gamma \cup \{ \bot_\Pi \} \), \( \bot_{\Pi'} w_{\Pi_2} x_i \in [\Pi'] \), \( y_k, w_{\Pi_2}, y_k \in [\Pi] \), \( w_2 \in \text{cross}_2(n) \).
\end{itemize}
\( (2.iii) x_1 \delta^{k_0} w^\Pi_1 w^2 w^\Pi_1 v, \text{ with } v \in \Gamma \cup \{ \bot_\Pi \}, y_{k_1} w^\Pi_1 v \in [\Pi], x_i w^\Pi_1 \bot_\Pi \in [\Pi'], w_2 \in \text{cross}_2(n) \).

\( (2.iv) x_1 \delta^{k_0} w^\Pi_1 w^2 w^\Pi_1 \delta^{k_0} \cdot x_j, \text{ with } \bot_\Pi w^\Pi_1 x_j \in [\Pi'], y_{k_1} w^\Pi_1 y_{k_0} \in [\Pi], y_{k_0} w^\Pi_1 y_{k_0} \in [\Pi], x_i w^\Pi_1 \bot_\Pi \in [\Pi'], w_2 \in \text{cross}_2(n) \).

\( (2.v) x_i \delta^k w^\Pi_1 \delta^k x_j, \text{ for } 1 \leq k \leq m, \text{ with } x_i w^\Pi_1 x_j \in [\Pi']. \)

The next lemma states a limited invariance of the GoI interpretation for \( P_1 \) and \( P_2 \).

**Lemma 4.45.** The classes we defined enjoy those properties:

- For \( k = i, ii, iii, iv, v \), there exists \( w_1 \) of class 1.k \( \in [P_1] \) iff there exists \( w_2 \) of class 2.k \( \in [P_2] \) and \( w^1_{2nf} = w^2_{2nf} \).

- For \( k = i, ii, iii, iv \), there exists \( w_1 \) of class 1.k \( \in [P_1] \) iff there exists \( w_2 \) of class 2.k \( \in [P_2] \) and for all exponential stack \( \sigma \), \( w^1_{2nf}(\sigma) = w^2_{2nf}(\sigma) \).

- There exists \( w_1 \) of class 1.v \( \in [P_1] \) iff there exists \( w_2 \) of class 2.v \( \in [P_2] \) and for all exponential stack \( \sigma \), for all \( p \leq m \) \( w^1_{2nf}(\delta^p::\sigma) = w^2_{2nf}(\delta^p::\sigma) = \delta^p::\tau \) for some \( \tau \).

**Proof.** The two first claims are easy to check with the rewriting rules of GoI and Lemma 4.43. In particular, observe that for any partial trace \( u, (\delta^{k_0}l(u))^{-} = (\delta^{k_0}u)^{-} \) and \( (\delta^{k_0}l(u))(\sigma) = (\delta^{k_0}u)(\sigma) \) (see Corollary 4.22).

For the last claim, observe that for one trace in the class 1.v, one can associate \( k \) traces in class 2.v. So, for any \( p \leq m, x_i l(w^\Pi_1) x_j (\delta^p::\sigma) = \delta^p::w^\Pi_1(\sigma) \), and there is a trace \( w_2 \) of the form \( x_i \delta^k w^\Pi_1 \delta^k x_j \) in the class (2.v) s.t \( k = p \), and thus \( w_2(\delta^p::\sigma) = \delta^p::w^\Pi_1(\sigma) \).

Looking at Figure 4.10, the next lemma states that at each step of the \textit{ReadBack()} execution on the traces of \([P_1]\) and \([P_2]\), we will find characteristic traces having the same multiplicative structure and the same behaviour w.r.t exponential stacks.

**Lemma 4.46.** If \( P \) reduces to \( P' \), by a Box-Multiplexor cut-elimination step, then there exists \( (\text{carac}_1, \sigma'_1, l) \in \text{ReadBack}([P_1]_{nf}) \) iff there exists \( (\text{carac}_1, \sigma_1, l) \in \text{ReadBack}([P_1]_{nf}) \) with \( \text{carac}_1 = \text{carac}_1^- \) and \( \sigma'_1 = \sigma_1 \).

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

- \( l = \varepsilon \): By Lemma 4.45 there exists a unique \( \text{carac}_c \in [P] \) s.t for \( (\text{carac}_c) = \bot q^\alpha \) in class (i) or (ii) \( \text{フト exists a unique } \text{carac}_c \in [P] \) s.t for \( (\text{carac}_c) = \bot q^\alpha \).

By Lemma 4.45 if \( \text{carac}_c \) is not correctly defined, or if it is not uniquely defined, then \( \text{carac}_c \) isn’t either, and conversely, if \( \text{carac}_c \) isn’t correctly and uniquely defined, then \( \text{carac}_c \) isn’t either.

- \( l = i::l \): by induction hypothesis, there exists \( (\text{carac}_{i_1}, \sigma_{i_1}, l_1) \in \text{ReadBack}([P_1]_{nf}) \) iff there exists \( (\text{carac}_{i_1}, \sigma_{i_1}, l_1) \in \text{ReadBack}([P_1]_{nf}) \) with \( \text{carac}_{i_1} = \text{carac}_{i_1}^- \) and \( \sigma_{i_1} = \sigma_{i_1} \).

So, in particular, \( \text{esc}_{i_1}^- = \text{esc}_{i_1} \). By Lemma 4.45 there exists \( \text{carac}_{i_1::l_1} \in [P_1] \) s.t for \( \text{carac}_{i_1::l_1} = \text{esc}_{i_1}^{-1} q_{i_1}^{-1} p_{i_1} \) and \( \text{carac}_{i_1::l_1}(\sigma_{i_1}) = \tau \) for some \( \tau \) and if only if there exists \( \text{carac}_{i_1::l_1} \in [P_1] \), s.t \( \text{carac}_{i_1::l_1} = \text{carac}_{i_1::l_1}^- \).

If it is a trace of class (i), (ii), or (iii), \( \text{carac}_{i_1::l_1}(\sigma) = \text{carac}_{i_1::l_1}^- \) for any \( \sigma \).

If it is a trace of class (iv) or (v), then it implies that \( \text{carac}_{i_1} \) and \( \text{carac}_{i_1} \) are in class (iii), (iv) or (v), since it must end with an \( x_i \) free port token. Since those traces ends are of the form \( l(w) \delta^k \in [P_1] \) and of the form \( \delta^k \) in \([P_2]\), for some \( k \leq m, \sigma_{i_1} \) and \( \sigma_{i_1} \) are of the form \( \sigma_{i_1} = \sigma_{i_1} = \delta^k::\tau \) for some \( \tau \).
Remark 4.47. From the previous Lemmas, we can deduce a little quantitative result: if \( P \rightarrow^* P' \), then \( \max(\|P\|_n) \leq \max(\|P'\|_n) \) and \( \|P\|_n \leq \|P'\|_n \).

In other words, reduction by cut-elimination produces possibly more and larger traces than GoI execution.

Now, building on previous results, we are ready to show that cut-elimination in Proof-nets preserves the readback of the normal form of their associated GoI:

Proposition 4.48. If \( P \rightarrow^* P'_1 \rightarrow P_1 \), then \( \text{ReadBack}(\|P\|_n) = b \) iff \( \text{ReadBack}(\|P'_1\|_n) = b \).

Proof. By induction on the reduction \( P \rightarrow^* P'_1 \rightarrow P_1 \), by case on the last cut link reduced:

- Ax-Ax, Box-Box: \( \|P'_1\| = \|P_1\| \), so we can conclude simply by applying induction hypothesis.

- \( \langle \rangle \rightarrow \langle \rangle \rightarrow \langle \rangle \): \( \|P'_1\| \approx \|P_1\| \), so again, we can conclude simply by applying induction hypothesis.

- Box-Multiplexor: We want to show that \( \text{ReadBack}(\|P\|_n) = b \) iff \( \text{ReadBack}(\|P'_1\|_n) = b \). Using Lemma 4.46, we can deduce that \( \text{ReadBack}(\|P\|_n) \) is well-defined if and only if \( \text{ReadBack}(\|P'_1\|_n) \) is well-defined. Now, we prove equality by induction on \( b \):

\[
\begin{align*}
\text{RB}(&\text{carac}_i', \sigma_i', l) = (\lambda x_1 \cdots x_{n-1}. x_{n+1}', [\,]) \Leftrightarrow \\
&\text{for}(\text{carac}_i') = wq^n \text{ and back}(\text{carac}_i') = acc_q q^{-1} p, \text{ with } l' \leq l \text{ and } \\
&\sigma_i' \subseteq \text{carac}_i(\sigma_i) \\
&\Leftrightarrow \text{for}(\text{carac}_i') = wq^n \text{ and back}(\text{carac}_i') = acc_q q^{-1} p, \text{ with } l' \leq l \text{ and } \\
&\sigma_i' \subseteq \text{carac}(\sigma) \\
&\Leftrightarrow \text{RB}(\text{carac}_i, \sigma_i, l) = (\lambda x_1 \cdots x_{n-1}. x_{n+1}', [\,])
\end{align*}
\]

- \( \text{RB}(\text{carac}_i', \sigma_i', l) = (\lambda x_1 \cdots x_{n-1}. z, [\,]) \), \( z \) being a free variable \( \Leftrightarrow \)

\[
\begin{align*}
&\text{for}(\text{carac}_i') = wq^n \text{ and back}(\text{carac}_i') = z \\
&\Leftrightarrow \text{for}(\text{carac}_i') = wq^n \text{ and back}(\text{carac}_i') = z \\
&\Leftrightarrow \text{RB}(\text{carac}_i, \sigma_i, l) = (\lambda x_1 \cdots x_{n-1}. z, [\,])
\end{align*}
\]

- \( \text{RB}(\text{carac}_i', \sigma_i', l) = (\lambda x_1 \cdots x_{n-1}. y, [b_1, \ldots, b_m]) \)

* if \( y = x_{n+1}' \):

\[
\begin{align*}
&\forall i, 1 \leq i \leq m, \text{RB}(\text{carac}_i', \sigma_i', l) = b_i \text{ and } \\
&\text{RB}(\text{carac}_i', \sigma_i', l) = (\lambda x_1 \cdots x_{n+1} y, [b_1, \ldots, b_m]) \\
&\text{iff } \\
&\text{for}(\text{carac}_i') = wq^n \text{, back}(\text{carac}_i') = acc_q q^{-1} p, \text{ and } \\
&l' \leq l, \sigma_i' \subseteq \text{carac}_i(\sigma_i) \\
&\forall i, 1 \leq i \leq m, \text{RB}(\text{carac}_i', \sigma_i', l) = b_i \text{, by IH and } \\
&\text{iff } \\
&\text{for}(\text{carac}_i') = wq^n \text{, back}(\text{carac}_i') = acc_q q^{-1} p, \text{ and } \\
&l' \leq l, \sigma_i' \subseteq \text{carac}_i(\sigma_i) \\
&\text{RB}(\text{carac}_i, \sigma_i, l) = (\lambda x_1 \cdots x_{n+1}. x_{n+1}', [b_1, \ldots, b_m])
\end{align*}
\]
The next Lemma states that the ReadBack on the GoI of the Soft Proof-net the underlying lambda term.

Proposition 4.50 does not alter the result of the ReadBack w.r.t to Sharing graphs Optimal reduction.

By induction on 4.49 Remark 106 CHAPTER 4. OPTIMAL REDUCTION FOR SOFT LINEAR LOGIC

We can make an observation on the variation of the number of traces in Proof-nets: reduction and Soft sharing graphs Optimal reduction: \( l_1 \cdot z \leq l_2 \cdot z \) if.

In order to state the final result, we have left to prove that the result of the ReadBack (Multiplexor):

\( \Pi \rightarrow \Pi' \) by definition of the \( \Pi \) map on Soft Proof-nets:

\[ z_i w_i z_j \in [\Pi'] \iff z_i w_j z_j^* z \in [\Pi] \]
\[ z_i w_j \in [\Pi'] \iff z_i w_j y \in [\Pi], \text{ for } y \neq z_i, y \neq z \]
\[ y w z_i \in [\Pi'] \iff y w z_i \in [\Pi], \text{ for } y \neq z_i, y \neq z \]
\[ y_1 w y_2 \in [\Pi'] \iff y_1 w y_2 \in [\Pi], \text{ for } y_1, y_2 \neq z_i, y_1, y_2 \neq z \]

This defines a bijection between traces of \([\Pi']\) and those of \([\Pi]\). We will note \( Mux(w) \) the application from \([\Pi']\) to \([\Pi]\).

4.5.3 GoI associated to a cut-free soft derivation

The next Lemma states that the ReadBack on the GoI of the Soft Proof-net \( \Pi \) associated to a cut-free Soft derivation corresponds to the underlying lambda term.

**Proposition 4.50.** If \( \Pi : \Gamma \vdash b : A \) is a cut-free soft derivation, then ReadBack([\Pi]) = b.

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

- **(Ax):** \( \Pi : z : A \vdash z : A \). \([\Pi] = \{ z, \bot, \bot z \}, \) so by definition \( \text{ReadBack}([\Pi]) = z. \)
- **(Box):**

\[
\Pi' : \Gamma \vdash b : A \\
\Pi : \Gamma \vdash b : !A
\]

By definition \([\Pi] = l([\Pi']). \) It is easy to check that \( \text{ReadBack}([\Pi]) = \text{ReadBack}([\Pi']). \)

So by I.H \( \text{ReadBack}([\Pi]) = b. \)

- **(Multiplexor):**

\[
\Pi' : \Gamma z_1 : A, \ldots, z_n : A \vdash b : A \\
\Pi : \Gamma, z : !A \vdash b(z/z_1) : A
\]

We can define the shape of \([\Pi]\), w.r.t the one of \([\Pi']\) by definition of the \([\Pi]\) map on Soft Proof-nets:

This defines a bijection between traces of \([\Pi']\) and those of \([\Pi']\). We will note \( Mux(w) \) the application from \([\Pi']\) to \([\Pi]\).
First, we prove that if we have that \((\text{carac}_1', \sigma_1', l) \in \text{ReadBack}(\Pi')\), then \((\text{carac}_1, \sigma_1, l) \in \text{ReadBack}(\Pi)\), with \(\text{carac}_1 = \text{Mux}(\text{carac}_1')\), \(\sigma_1 = \delta^k \cdot \sigma_1'\) if \(\text{carac}_1'\) starts with \(z_k\) and \(\sigma_1 = \sigma_1'\) else. By induction on \(l\):

- \(l = \varepsilon\): \(\text{carac}_1'\) is s.t \(\text{for}(\text{carac}_1') = \perp q^0\), and \(\text{carac}_1'(\emptyset) = \tau'\) for some \(\tau'\). By definition, \(\text{for}(\text{Mux}(\text{carac}_1')) = \perp q^0\) and \(\text{Mux}(\text{carac}_1')(\emptyset) = \tau\) for some \(\tau\) — since \(\text{carac}_1'\) starts with \(\perp\). If there is another trace \(w \in [\Pi]\) s.t \(\text{for}(w) = \perp q^0\) and \(w(\emptyset)\) is well-defined, then by definition, \(\text{carac}_1'\) is not uniquely determined in \([\Pi']\), which contradicts the induction hypothesis on \(\Pi'\).

So, \(\text{Mux}(\text{carac}_1'), \theta, \varepsilon \in \text{ReadBack}(\Pi)\).

- \(l = i : l_1\): by induction hypothesis, \(\text{carac}_{i_1} = \text{Mux}(\text{carac}_{i_1}')\) and \(\sigma_{i_1} = \delta^k \cdot \sigma_{i_1}'\) if \(\text{carac}_{i_1}'\) starts with \(z_k\) or \(\sigma_{i_1} = \sigma_{i_1}'\) else.

We reason by case on the form of \(\text{carac}_{i_1}'\):

1. \(\text{carac}_{i_1}' = z_k w z_j\): By definition of the ReadBack, \(\text{carac}_{i_1}'\) is the only trace of \([\Pi']\) s.t \(\text{for}(\text{carac}_{i_1}') = w q^0\) for some \(w\), and \(\text{carac}_{i_1}'(\sigma_{i_1}) = \tau'\) for some \(\tau'\).

   By induction hypothesis, \(\text{Mux}(\text{carac}_{i_1})\) is the characteristic trace of level \(l_1\) and \(\text{Mux}(\text{carac}_{i_1}') = z_k w \delta_k z_j\).

   By definition, \(\text{Mux}(\text{carac}_{i_1}')\) is s.t \(\text{for}(\text{carac}_{i_1}') = z w q^0\), \(\sigma_{i_1} = \delta^k \cdot \sigma_{i_1}'\), so \(\text{Mux}(\text{carac}_{i_1}')(\sigma_{i_1}) = \tau\) for some \(\tau\).

   Therefore, \((\text{Mux}(\text{carac}_{i_1}'), \delta^k \cdot \sigma_{i_1}' + i : l_1) \in \text{ReadBack}(\Pi)\).

2. \(\text{carac}_{i_1}' = z w y z_j, y \neq z_j\): By definition, \(\text{carac}_{i_1}'\) is the only trace of \([\Pi']\) s.t \(\text{for}(\text{carac}_{i_1}') = w q^0\) for some \(w\) and \(\text{carac}_{i_1}'(\sigma_{i_1}) = \tau'\) for some \(\tau'\).

   By I.H, \(\text{Mux}(\text{carac}_{i_1})\) is the characteristic trace of level \(l_1\) and \(\text{Mux}(\text{carac}_{i_1}') = z w y \delta_0 z_j\).

   By definition \(\text{Mux}(\text{carac}_{i_1}')\) is s.t \(\text{for}(\text{carac}_{i_1}') = w q^0\), \(\sigma_{i_1} = \delta^k \cdot \sigma_{i_1}'\), so \(\text{Mux}(\text{carac}_{i_1}')(\sigma_{i_1}) = \tau\) for some \(\tau\).

   Therefore, \((\text{Mux}(\text{carac}_{i_1}'), \delta^k \cdot \sigma_{i_1}' + i : l_1) \in \text{ReadBack}(\Pi)\).

We show that \(\text{ReadBack}(\Pi') = b\) implies that \(\text{ReadBack}(\Pi) = b(z/z_1)\), by induction on \(b\).

- \(b = (\lambda x_{1,1} \cdots x_{1,n'}, y', []): \text{carac} = \text{Mux}(\text{carac}'_1)\), so \(n = n', m = m' = 0\). Now, for head variable \(y\), we reason by case on \(y'\):

  1. \(y' = z_i\): \(\text{esc}(\text{carac}') = z_i\), so \(\text{esc}(\text{carac}) = z\), thus \(y = z\).

  2. \(y' = v, v \neq z_i\): \(\text{esc}(\text{carac}') = v\), so \(\text{esc}(\text{carac}) = v\), thus \(y = v\).

  3. \(y' = x_{1,j}\): \(\text{esc}(\text{carac}') = \text{acc}_v q^{j-1} p\). By definition, \(\text{acc}_v = \text{acc}(\text{Mux}(\text{carac}_1'))\) so \(\text{esc}(\text{carac}') = \text{acc}_v q^{j-1} p\). If \(\text{for}\) \(\delta^k \cdot \sigma_{i_1}'\), then \(\text{acc}_v = z \delta_0^k\) thus \(\tau = \delta^k \cdot \tau'\) and \(\sigma_1 \subseteq \tau\), so \(y = x_{1,j}\).
So, if \( \text{ReadBack}([\Pi']) = b \), then \( \text{ReadBack}([\Pi]) = b\{z/\_z\} \).

\[- b = (\lambda x_1 \cdots x_{t', n'}, y', [b_1, \ldots, b_{m'}]): \]
We reason as in the base case to retrieve the number of binders, the number of applications and the head variable. Next, we apply the induction hypothesis on the \( b'_1 \) subterms to obtain that \( b_i = b'_1\{z/\_z\} \) to conclude that \( \text{ReadBack}([\Pi]) = b\{z/\_z\} \).

- \( /-\): The last rule of the derivation is of the form:

\[
\Pi' : \Gamma, z : A_1 \vdash (\lambda x_{z, 1} \cdots x_{z, n}, y, [b_1, \ldots, b_{m}]) : A_2
\]

\[
\Pi : \Gamma \vdash (\lambda x_{z, 1} \cdots x_{z, n+1}, y, [b_1, \ldots, b_{m}])\{x_{z, 1}/z, (\_z, i + 1) \rightarrow (\_z, i)\} : A_1 \rightarrow A_2
\]

We can define the shape of \( [\Pi] \), w.r.t the one of \( [\Pi'] \) by definition of the \( [\_] \) map on Soft Proof-nets:

\[
zwz \in [\Pi'] \iff \perp pwp \perp \in [\Pi]
\]

\[
wz \in [\Pi'] \iff \perp qwp \perp \in [\Pi]
\]

\[
zw \perp \in [\Pi'] \iff \perp pqw \perp \perp \in [\Pi]
\]

\[
vwz \in [\Pi'] \iff vwp \perp \perp \in [\Pi], v \in FV([\Pi])
\]

\[
wz \perp \in [\Pi'] \iff \perp qwp \perp \perp \in [\Pi]
\]

\[
vw \perp \in [\Pi'] \iff vwp \perp \perp \in [\Pi], v \in FV([\Pi])
\]

\[
zw \perp \in [\Pi'] \iff \perp qwp \perp \perp \in [\Pi]
\]

\[
vwv1 \in [\Pi'] \iff v2wv1 \in [\Pi], v_1, v_2 \in FV([\Pi])
\]

This defines a bijection between traces of \( [\Pi'] \) and those of \( [\Pi'] \). We note \( \text{Abs}(w) \) the application from \( [\Pi'] \) to \( [\Pi] \). Note that for all \( \sigma \in \Sigma, w \in [\Pi'], w(\sigma) = \text{Abs}(w)(\sigma) \).

First, we show that \( \forall (\text{carac}_t', \sigma_t', t) \in b, (\text{carac}_t, \sigma_t, t) \in \text{ReadBack}([\Pi]), \) with \( \text{carac}_t = \text{Abs}(\text{carac}_t'), \sigma_t(\sigma_t') = \text{carac}_t(\sigma_t') \), by induction on \( l \):

- \( l = \_z : \text{carac}_t' \) is the unique trace of \( [\Pi'] \) s.t \( \text{for} (\text{carac}_t') = \perp q^n \), for some \( n' \), and \( \text{carac}_t(\emptyset) = \tau' \) for some \( \tau' \).

By definition of \( \text{Abs}, \text{carac}_0 = \text{Abs}(\text{carac}_0') \) is the only trace of \( [\Pi] \) s.t \( \text{for} (\text{carac}_0') = \perp q^n \) — with \( n = n'+1 \) —, and \( \text{carac}_0(\emptyset) = \text{carac}_0(\emptyset) = \sigma_0' \).

- \( l = i::l_1 \): By induction hypothesis, \( \text{carac}_{l_1} = \text{Abs}(\text{carac}_{l_1}') \) and \( \sigma_{l_1} = \sigma_{l_1}' \). By definition, \( \text{carac}_{l_1}' \) is the unique trace of \( [\Pi'] \) s.t \( \text{for} (\text{carac}_{l_1}') = \text{esc}_{l_1}' \perp q^{n' - 1} p^a \), and \( \text{carac}_{l_1}(\sigma_{l_1}') = \tau' \) for some \( \tau' \). We reason by case on the form of \( \text{esc}_{l_1}' \):

* \( \text{esc}_{l_1}' = zw \): by definition, \( \text{esc}_{l_1}' = \perp pw. \text{for} (\text{Abs}(\text{carac}_{l_1}')') = \perp pwp \perp p^n \), and since \( \text{carac}_{l_1}'(\sigma_{l_1}') = \tau' \), \( \text{Abs}(\text{carac}_{l_1}'(\sigma_{l_1}')) = \tau' \).

* \( \text{esc}_{l_1}' = \perp w \): by definition, \( \text{esc}_{l_1}' = \perp qw. \text{for} (\text{Abs}(\text{carac}_{l_1}')') = \perp pw \perp q^n \), and since \( \text{carac}_{l_1}'(\sigma_{l_1}') = \tau' \), \( \text{Abs}(\text{carac}_{l_1}'(\sigma_{l_1}')) = \tau' \).

* \( \text{esc}_{l_1}' = vw, v \in FV([\Pi']), v \neq z \): by definition, we have that \( \text{esc}_{l_1}' = vw \) and \( \text{for} (\text{Abs}(\text{carac}_{l_1}')') = \perp pq^n \), and since \( \text{carac}_{l_1}(\sigma_{l_1}') = \tau' \), we have that \( \text{Abs}(\text{carac}_{l_1}(\sigma_{l_1}')) = \tau' \).

So \( (\text{Abs}(\text{carac}_{l_1}'), \sigma_{l_1}', i::l_1) \in \text{ReadBack}([\Pi]) \)
Now, we have left to prove that if \( \text{ReadBack}(\Pi') = b' = (\lambda x_1, \ldots, x_n, y', [b_1', \ldots, b_m']) \), then \( \text{ReadBack}(\Pi) = b = (\lambda x_1, \ldots, x_n, y', [b_1, \ldots, b_m]) \{ x_{z_1}/z, (e, i + 1) \mapsto (e, i) \} \). We proceed by induction on \( b' \):

\[-b' = (\lambda x_1, \ldots, x_n, y', [\Pi']) \] if \( l = e \), \( \text{for}(\Pi') = \bot q' \), and by definition, \( \text{for}(\text{carac}^{-}) = \text{for}(\text{Abs}(\text{carac}^{-})) = \bot q'^{+1} \). Else, for \( l \neq e \), \( \text{for}(\text{carac}^{-}) = wq'^{d} \), and by definition, \( \text{for}(\text{carac}^{-}) = \text{for}(\text{Abs}(\text{carac}^{-})) = w'q'^{d}, \) and \( m = m' = 0 \). Now for the head variable of \( b \), we reason by case on \( y' \):

* \( y' = z \): so, \( \text{esc}(\text{carac}^{-}) = z \), and \( \text{esc}(\text{carac}^{-}) = \text{esc}(\text{Abs}(\text{carac}^{-})) = \bot p \), so \( y = x_{z_1} (\emptyset \sqsubseteq \tau \text{ for any } \tau) \).

* \( y' = x_{z_1} \): so, \( \text{esc}(\text{carac}^{-}) = \bot q'^{-1}p \), and \( \text{esc}(\text{carac}^{-}) = \text{esc}(\text{Abs}(\text{carac}^{-})) = \bot q'^{1}p, \emptyset \sqsubseteq \tau \text{ for any multiplicative stack } \tau \) so \( y = x_{z_1+1} \).

* \( y' = x_{z_1} \): therefore, \( \text{esc}(\text{carac}^{-}) = \text{esc}(\text{Abs}(\text{carac}^{-})) = \bot q'^{-1}p, \sigma_1 = \sigma' \), and \( \text{esc}(\text{carac}^{-}) = \text{esc}(\text{Abs}(\text{carac}^{-})) = \bot q'^{1}p, \sigma_1 = \sigma' \), and \( \text{for}(\text{carac}^{-}) = (\text{for}(\text{carac}^{-})) \), so \( l \sqsubseteq \text{carac}(\sigma_1) \), so \( y = x_{z_1} \).

* \( y' = v, v \in \text{FV}(\Pi') \), \( v \neq z \): so, \( \text{esc}(\text{carac}^{-}) = v \), and \( \text{esc}(\text{carac}^{-}) = \text{esc}(\text{Abs}(\text{carac}^{-})) = v \), so \( y = v \).

\[-b' = (\lambda x_1, \ldots, x_n, y', [b_1', \ldots, b_m]) \] : We reason as in the base case to retrieve the number of binders, the number of applications and the head variable. Then, we apply the induction hypothesis on the \( b'_1 \) subterms to conclude.

\( (-o-1) \): The last rule of the derivation is of the form:

\[
\Pi_1 : \Gamma_1 \vdash t_1 : B \quad \Pi_2 : \Gamma_2, z' : C \vdash t_2 : A \quad \Pi : \Gamma_1, \Gamma_2, z : B \rightarrow o C \vdash t_2 (z/t_1/z') : A
\]

With the domains of \( \Gamma_1 \) and \( \Gamma_2 \) being disjoints. We can define the shape of \( \Pi \), w.r.t the ones of \( \Pi_1 \) and \( \Pi_2 \), by definition of the \( \llbracket \) interpretation of Soft Proof-nets:

\[
\bot w \bot \in \llbracket \Pi_1 \] \iff \( zwp^*z \in \llbracket \Pi \]
\[
v w v \bot \in \llbracket \Pi_1 \] \iff \( v wp^*z \in \llbracket \Pi, v \in \text{FV}(\Pi_1) \]
\[
\bot w v \in \llbracket \Pi_1 \] \iff \( zpvw \in \llbracket \Pi \]
\[
v_1 w v_2 \in \llbracket \Pi_1 \] \iff \( v_1 w v_2 \in \llbracket \Pi_1, \text{ and } v_1, v_2 \in \text{FV}(\Pi_1) \]
\[
z' w z' \in \llbracket \Pi_2 \] \iff \( zqwq^*z \in \llbracket \Pi \]
\[
z' w y \in \llbracket \Pi_2 \] \iff \( zqw y \in \llbracket \Pi, y \in \text{FV}(\Pi_2) \], \( y \neq z' \]
\[
y w z' \in \llbracket \Pi_2 \] \iff \( y w q^*z \in \llbracket \Pi, y \in \text{FV}(\Pi_2) \], \( y \neq z' \]
\[
y_1 w y_2 \in \llbracket \Pi_2 \] \iff \( y_1 w y_2 \in \llbracket \Pi, y_1, y_2 \in \text{FV}(\Pi_2) \], \( y_1, y_1 \neq z' \]

This defines a bijection between traces of \( \llbracket \Pi_1 \cup \llbracket \Pi_2 \] and those of \( \llbracket \Pi \]. We note \( \text{App}(w) \) the application from \( \llbracket \Pi_1 \cup \llbracket \Pi_2 \] to \( \llbracket \Pi \].

We define \( \mathcal{L}_z \), the set of the levels of the head-subterms of \( b_2 \) which have \( z' \) for head variable:

\[
\mathcal{L}_z = \{ l \in \text{Levels}(b_2) \mid \text{esc}_l = z' \}
\]

Now, we extend the \( \text{App} \) application to integer lists:

\[
\text{App}(z) = z \quad \text{App}(i,l') = (i + 1) :: \text{App}(l'), \text{ if } l' \in \mathcal{L}_z \quad \text{App}(i::l') = i :: \text{App}(l'), \text{ else}
\]
Intuitively, for any head-subterm in $t_2$ of level $l'$ whose head-variable is $z'$, then $l' \in \mathcal{L}_{z'}$, and all its head-subterms of level $i:E_{l'}$ must be shifted in the result: $1:E_{l'}$ is now the level of the root of a copy of $t_2$.

Now we rephrase the $\neg \cdot l$ deduction rule:

$\Pi_1 : \Gamma_1 \vdash b_1 : B \quad \Pi_2 : \Gamma_2, z' : C \vdash b_2 : A$

$\Pi : \Gamma_1, \Gamma_2, z : B \rightarrow C \vdash b_2 \{t_2 \rightarrow \text{App}(t_2), (z) b_1 \{l \rightarrow l \cdot 1:E_{l'} / z'_l\} : A$ 

For all $t_2 \in \text{Levels}(b_2)$, for all occurrences $z'_l$ in $b_2$ and forall $l \in b_1$.

First, we show:

1. For all $(carac''_l, \sigma''_l, l) \in b_2$, $(carac_{\text{App}(l)}, \sigma''_l, \text{App}(l)) \in \text{ReadBack}([\Pi])$, with $\text{carac}_{\text{App}(l)} = \text{App}(\text{carac}''_l)$

2. That if $l \in \mathcal{L}_{z'}$, then for all $(carac''_{l'}, \sigma''_{l'}, l') \in b_2$, $(\text{App}(carac''_{l'}), \tau', l' \cdot 1::\text{App}(l)) \in \text{ReadBack}([\Pi])$, with $\sigma''_{l'} \subseteq \tau'$

Finally, we prove by induction on $b_2$ the desired result, i.e. that $\text{ReadBack}([\Pi_1]) = b_1$ and $\text{ReadBack}([\Pi_2]) = b_2$ imply that $\text{ReadBack}([\Pi]) = b_2 \{t_2 \rightarrow \text{App}(t_2), (z) b_1 \{l \rightarrow l \cdot 1:E_{l'} / z'_l\}$.

We will note $\text{carac'}_l$ (resp. $\text{carac}''_l$) the characteristic trace of level $l$ of $b_1$ (resp. $b_2$).

The proof of statement (1) is by induction on $l$:

1. $l = \varepsilon$: $(\text{carac}''_l, \emptyset, \varepsilon) \in b_2$, so by definition, $\text{carac}''_l$ is the only trace of $[\Pi_2]$ s.t. $\text{for}((\text{carac}''_l) = \bot q^n$ and $\text{carac}''_l(\emptyset) = \sigma''_l$. By definition of $\text{App}$, $(\text{App}(\text{carac}''_l))$ is the only trace of $[\Pi]$ s.t. $\text{for}((\text{App}(\text{carac}''_l))) = \bot q^n$, and $(\text{App}(\text{carac}''_l))$ is well-defined. So $\text{carac}_\varepsilon = \text{App}(\text{carac}''_l)$ and $(\text{App}(\text{carac}''_l), \emptyset, \varepsilon) \in \text{ReadBack}([\Pi_\varepsilon])$.

2. $l = i:E_{l_1}$: By induction hypothesis, $(\text{App}(\text{carac}''_{l_1}), \sigma''_{l_1}, \text{App}(l_1)) \in \text{ReadBack}([\Pi])$. We show that $(\text{App}(\text{carac}''_{i:E_{l_1}}), \sigma''_{i:E_{l_1}}, \text{App}(i:E_{l_1})) \in \text{ReadBack}([\Pi])$ by case on the head variable $y$ of $b_2$:

* $y = v, \; v \in \text{FP}([\Pi_2]), v \neq z'$: then $l_1 \notin \mathcal{L}_{z'}$, and $\text{esc}(\text{carac}''_{l_1}) = w$, for some $w$. So $\text{carac}''_{i:E_{l_1}}$ is s.t. $\text{for}((\text{carac}''_{i:E_{l_1}})) = w^{-q^i-1}pq^n$ and $\text{carac}''_{i:E_{l_1}}(\sigma''_{i:E_{l_1}})$ is well-defined.

By I.H. $(\text{App}(\text{carac}''_{l_1}), \sigma''_{l_1}, \text{App}(l_1)) \in \text{ReadBack}([\Pi])$ and by definition of $\text{App}$, $\text{esc}(\text{App}(\text{carac}''_{l_1})) = w$.

$l_1 \notin \mathcal{L}_{z'}$, so by definition $\text{App}(i:E_{l_1}) = i:E_{\text{App}(l_1)}$, $(\text{carac}''_{i:E_{l_1}})$ is such that $\text{for}((\text{carac}''_{i:E_{l_1}})) = w^{-q^i-1}pq^n$ and $\text{App}(\text{carac}''_{i:E_{l_1}})(\sigma''_{i:E_{l_1}})$ is well-defined.

So, $(\text{App}(\text{carac}''_{i:E_{l_1}}), \sigma''_{i:E_{l_1}}, \text{App}(i:E_{l_1})) \in \text{ReadBack}([\Pi])$.

* $y = z'$: then $l_1 \in \mathcal{L}_{z'}$, and $\text{esc}(\text{carac}''_{l_1}) = z'$. So $\text{carac}''_{i:E_{l_1}}$ is s.t. $\text{for}((\text{carac}''_{i:E_{l_1}})) = z'q^i-1pq^n$ and $\text{carac}''_{i:E_{l_1}}(\sigma''_{i:E_{l_1}})$ is well-defined.

By I.H. $(\text{App}(\text{carac}''_{l_1}), \sigma''_{l_1}, \text{App}(l_1)) \in \text{ReadBack}([\Pi])$ and by definition of $\text{App}$, $\text{esc}(\text{App}(\text{carac}''_{l_1})) = z$.

$l_1 \in \mathcal{L}_{z'}$, so by definition $\text{App}(i:E_{l_1}) = (i + 1):\text{App}(l_1)$, $(\text{carac}''_{i:E_{l_1}})$ is s.t.

$\text{for}((\text{carac}''_{i:E_{l_1}})) = zq^iq^jpq^n$ and $\text{App}(\text{carac}''_{i:E_{l_1}})(\sigma''_{i:E_{l_1}})$ is well-defined. So $\text{carac}_{i:E_{l_1}:\text{App}(l_1)}(\text{App}(i:E_{l_1})) \in \text{ReadBack}([\Pi])$.

Now for point (2): for any $l \in \mathcal{L}_{z'}$, $\text{back}(\text{carac}''_{l'})) = z'q^m$, so by definition, we have that $\text{back}(\text{App}(\text{carac}''_{l'})) = zq^{m+1}$, and that $(\text{App}(\text{carac}''_{l'}))$ is the only trace of $[\Pi]$ such that $\text{for}((\text{App}(\text{carac}''_{l'}))) = zpq^n$ and $\text{App}(\text{carac}''_{l'})(\sigma)$ is well-defined for any $\sigma$. Since $\text{ReadBack}([\Pi_1])$ and $\text{App}(w) = \text{Inf}(w)$ for $w \in \Pi_1$, we can apply the first statement of
Proposition 4.31 to conclude that \( \forall (\text{carac}_{l'}', \sigma', l') \in b_1, (\text{App}(\text{carac}_{l'}'), \sigma_{l':1:l}, l' \cdot 1:l) \in \text{RB}_{[l]}(\text{App}(\text{carac}_{l'}'), \sigma_{l':1:l}, 1:l) \), with \( \sigma' \subseteq \sigma_{l':1:l} \).

Now, using statements (1) and (2), we are ready to show that if \( \text{ReadBack}(\Pi_1) = b_1 \) and \( \text{ReadBack}(\Pi_2) = b_2 \) then \( \text{ReadBack}(\Pi_3) = b_2 \{ l_2 \rightarrow \text{App}(l_2), (z) b_1 \{ l_1 \rightarrow l_1 \cdot 1: \text{App}(l') \} / z'_{l'} \} \) for all \( l_2 \in \text{Levels}(b_2), l' \in \mathcal{L}_z, l_1 \in \text{Levels}(b_1) \) by induction on \( b_2 \):

\[-\ b_2 = (\lambda x_1 \cdots \lambda x_n. y_2, []): \text{So, using statement (1), } (\text{App}(\text{carac}_{l'}'), \sigma', \text{App}(l)) \in \text{ReadBack}(\Pi_3), \text{so } \text{RB}_{[l]}(\text{carac}_{\text{App}(l)}, \sigma_{\text{App}(l)}, \text{App}(l)) = \text{RB}_{[l]}(\text{App}(\text{carac}_{l'}'), \sigma', \text{App}(l)). \text{It is easy to check that if } \text{carac}_{l'} \equiv \text{acc}_{l'} q^n, \text{then } \text{carac}_{\text{App}(l)} = \text{acc}_{l'} q^n. \]

In the following, we define \( b_2 = \text{App}(l) \), and we reason by case on the head variable \( y_2 \):

\* \( y_2 = v \): by definition \( \text{back}(\text{carac}_{l'}') = v \), since there is no application in \( b_2 \), so \( \text{back}(\text{App}(\text{carac}_{l'}')) = v, \text{RB}_{[l]}(\text{App}(\text{carac}_{l'}'), \sigma', l_b) = (\lambda x_{l_b,1} \cdots x_{l_b,n}. v, []) = b_2 \{ l \rightarrow l_b \} = b_2 \{ l \rightarrow l_b, (z) b_1 \{ l_1 \rightarrow l_1 \cdot 1:l_b \} / z'_{l'} \} \).

\* \( y_2 = x_{l_b,j} \): \( \text{back}(\text{carac}_{l'}') = \text{acc}_{l_b} q^j \) by statement (1), \( \text{acc}(\text{App}(\text{carac}_{l'}')) = \text{acc}(\text{App}(l_b)) \), and so \( \text{RB}_{[l]}(\text{App}(\text{carac}_{l'}'), \sigma', l_b) = (\lambda x_{l_b,1} \cdots x_{l_b,n}. x_{l_b,j}, []) = b_2 \{ l \rightarrow l_b \} = b_2 \{ l \rightarrow l_b, (z) b_1 \{ l_1 \rightarrow l_1 \cdot 1:l_b \} / z' \} \).

\* \( y_2 = z' \): \( \text{back}(\text{carac}_{l'}') = z' \), so \( l \in \mathcal{L}_z, \text{back}(\text{App}(\text{carac}_{l'}') = zq. \) By definition of the RB procedure, by statement (2) and Proposition 4.31

\[ \text{RB}(\text{App}(\text{carac}_{l'}'), \sigma_{l_b}, l_b) = (\lambda x_{l_b,1} \cdots x_{l_b,n}. z, \text{RB}_{[l]}(\text{App}(\text{carac}_{l'}'), \sigma_{l':1:l_b}, 1:l_b)) = (\lambda x_{l_b,1} \cdots x_{l_b,n}. z, \{ l_1 \rightarrow l_1 \cdot 1:l_b \}), \]

\[ \forall l_1 \in \text{Levels}(b_1) \]

\[ = b_2 \{ l \rightarrow l_b, (z) b_1 \{ l_1 \rightarrow l_1 \cdot 1:l_b \} / z'_{l_b} \}, \forall l_1 \in \text{Levels}(b_1). \]

\[ - b_2 = (\lambda x_{l_b,1} \cdots x_{l_b,n}. y_2, [b_2, \ldots, b_m]): \text{again, } l_b \text{ will stand for } \text{App}(l). \text{We can retrieve the number of binders and the head variable of } \text{RB}_{[l]}(\text{carac}_{l_b}, \sigma_{l_b}, l_b) \text{with the same reasoning as in base case.} \]

If \( y_2 = z' \), then \( \text{back}(\text{carac}_{l'}') = z'q^n \), so \( \text{back}(\text{App}(\text{carac}_{l'}')) = zq^{n+1} \). By definition, \( \text{carac}_{l_b} \) must be the unique trace of \( \Pi \) s.t \( \text{for}(\text{carac}_{l_b}) = zq^n \) and \( \text{carac}_{l_b} \) is well-defined. By definition, \( \text{App}(\text{carac}_{l'}') \) is the only trace of \( \Pi \) fitting this definition, so \( b_{1:l} = \text{RB}_{[l]}(\text{App}(\text{carac}_{l'}'), \sigma_{l':1:l_b}, 1:l_b) \) and by Proposition 4.31

\[ b_{1:l} = b_1 \{ l_1 \rightarrow l_1 \cdot 1:l_b \}, \forall l_1 \in \text{Levels}(b_1). \]

Then we can apply the induction hypothesis on the \( b_2 \), subterms. Since \( l_b \in \mathcal{L}_z, \text{App}(i:l) = (i + 1):l_b \), so \( b_{1:i+1:l} = \text{RB}_{[l]}(\text{App}(\text{carac}_{l'}'), \sigma_{l':1:l_b}, 1:l_b) = b_2 \{ l_2 \rightarrow \text{App}(l_2), (z) b_1 \{ l_1 \rightarrow l_1 \cdot 1:l_b \} / z'_{l'} \}, \forall l_2 \in \text{Levels}(b_2), \forall l_1 \in \text{Levels}(b_1). \)

So the result produced is

\[ \text{RB}_{[l]}(\text{carac}_{l_b}, \sigma_{l_b}, l_b) = (\lambda x_{l_b,1} \cdots x_{l_b,n}. z, \{ l_1 \rightarrow l_1 \cdot 1:l_b \}) = b_2 \{ l \rightarrow l_b, (z) b_1 \{ l_1 \rightarrow l_1 \cdot 1:l_b \} / z'_{l'} \}, \forall l \in \text{Levels}(b_1) \]

If on the contrary, \( y_2 \neq z' \), then we simply have to apply the induction hypothesis to conclude.
We restricted this correctness result to the case of propositional SLL. However, it is easy to add polymorphic and recursive types. Indeed, those features correspond to Proof-net nodes that are forgotten in the translation of a Proof-net in a Sharing graph, and thus do not alter the GoI interpretation of a Proof-net. Their cut-elimination does not alter GoI either.

We didn’t mention weakening and garbage collection. The proof of Proposition 4.50 also works if the derivation contains Weakening cuts. So a way to accommodate weakening is to postpone any weak-box cut-elimination step.

Finally, we wish to bound the size of the GoI associated to a cut-free Soft Proof-net:

**Proposition 4.51**. If \( \Gamma \vdash t : A \) is a cut-free Soft derivation, and \( \Pi \) is the associated Proof-net, then we have:

1. On the size of \( \Pi \): \( |\Pi| \leq |t| \) (counting contractions in \( t \)).

2. On the number of traces: \( |\Pi| \leq 2|\Pi| \).

3. On the length of traces: \( \max(|\Pi|) \leq 2|\Pi| \).

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

- **(Ax):** \( \Pi : z : A \vdash z : A \). \( |\Pi| = |t| = 1 \), and since \( |\Pi| = \{ z , z \perp \} \), \( |\Pi| \leq 2|\Pi| \) and \( \max(|\Pi|) \leq 2|\Pi| \).

- **(Box):** \( \Pi' : \Gamma \vdash b : A \rightarrow A \)

  By IH, \( |\Pi'| = |t'|, |\Pi'| \leq 2|\Pi'| \) and \( \max(|\Pi'|) \leq 2|\Pi'| \). And by definition, \( |\Pi| = |\Pi'| = |t|, |\Pi| = |\Pi'| \) and \( \max(|\Pi|) = \max(|\Pi'|) \).

- **(Multiplexor):** \( \Pi' : \Gamma_{z_1} : A_1, \ldots, z_n : A \vdash t : A \)

  By IH, \( |\Pi'| \leq 2|\Pi'| \) and \( \max(|\Pi'|) \leq 2|\Pi'| \). And by definition, \( |\Pi| = |\Pi'| \), \( |\Pi| \leq 2|\Pi'| \).

  However, it is easy to check that the number of non null traces is not increased: a trace \( z \delta_0^k w \delta_0^k z \) of \( \Pi \) is non null iff \( z w z \delta_0^k z \) is non null in \( \Pi' \).

  For the maximum length, for all \( w \in \Pi \), \( w \leq \max(|\Pi'|) + 2 \leq 2|\Pi| \).

- **(→r):** \( \Pi' : \Gamma z : A_1 \vdash t : A_2 \)

  By IH, \( |\Pi'| \leq 2|\Pi'| \) and \( \max(|\Pi'|) \leq 2|\Pi'| \). And by definition, \( |\Pi| = |\Pi'| + 1, |\lambda x.t| = |t| + 1 \).

  Again, the number of non null traces of \( \Pi \) is equal to the number of non null traces of \( \Pi' \), and a trace can increase by at most 2 (if it starts and ends at the root port).

- **(→i):** \( \Pi_1 : \Gamma_1 \vdash t_1 : B \rightarrow C \)

  By IH, \( |\Pi_1|--2|\Pi_1| \), \( \max(|\Pi_1|) \leq 2|\Pi_1| \), \( |\Pi_2|--2|\Pi_2| \), \( \max(|\Pi_2|) \leq 2|\Pi_2| \). And by definition, \( |\Pi| = |\Pi_1| + |\Pi_2| + 1, |t_2\{(z) t_1/z'\}| = |t_2| + k|t_1| \), where \( k \) is the number of occurrences of \( z' \) in \( t_2 \). So, \( |\Pi| \leq |t_2\{(z) t_1/z'\}| \).

  The number of traces in \( \Pi \) is exactly equal to \( |\Pi_1| + |\Pi_2| \). By definition of GoI interpretation, there cannot be a trace in \( \Pi \) that would be the concatenation of a trace of \( \Pi_1 \) and another of \( \Pi_2 \). At most, traces of \( \Pi_1 \) and \( \Pi_2 \) can increase by 2. So, \( \max(|\Pi|) = \max(\max(|\Pi_1|), \max(|\Pi_2|)) + 2 = \max(2|\Pi_1|, 2|\Pi_2|) + 2 \leq 2|\Pi| \).
To get a reduction system completely polynomial for soft lambda-terms, we have to state an upper complexity bound on the readback procedure:

**Proposition 4.52.** Let $\Gamma \vdash t : A$ be a soft derivation, $\Pi$ the associated Proof-net, $S$ the sharing graph s.t $S = \text{Tr}(\Pi)$, $\Pi_{nf}'$ and $S_{nf}'$ respectively the normal forms of $\Pi$ and of $S$.

If $\text{ReadBack}(\llbracket S_{nf}' \rrbracket) = t_{nf}'$, then $\text{ReadBack}(\llbracket S_{nf}' \rrbracket)$ runs in time polynomial w.r.t the size of $t$.

**Proof.** If $\text{ReadBack}(\llbracket \Pi \rrbracket) = t_{nf}'$, then it runs in $O(|t_{nf}'|)$ steps by Proposition 4.32: a step being (i) the identification of the correct carac trace in $\llbracket \Pi \rrbracket$; (ii) the reading of the number of nodes; and (iii) the retrieving of the head-variable, for a node in position $l$ in $t_{nf}'$.

Let $S$, be the sharing graph s.t $S = \text{Tr}(\Pi)$. We can bound the number of traces in $\llbracket S_{nf}' \rrbracket$ and their length w.r.t $t_{nf}'$, using Remarks 4.16 and 4.49: $max(|\llbracket S_{nf}' \rrbracket|) \leq 2|\Pi_{nf}'|$ and $|\llbracket S_{nf}' \rrbracket| \leq 2|\Pi_{nf}'|$, so $max(|\llbracket S_{nf}' \rrbracket|) \leq 2|t_{nf}'|$, and $|\llbracket S_{nf}' \rrbracket| \leq 2|t_{nf}'|$, by Proposition 4.51. So, one step can be performed in time $O(|t_{nf}'|^2)$, and the whole procedure in time $O(|t_{nf}'|^3)$.

Now, by a result on SLL (GRDR07) we can bound the size of $t_{nf}'$ w.r.t the size of $t$: $|t_{nf}'| = O(|t|^{d+1})$, where $d$ is the depth of the soft derivation of $t$.

So, $\text{ReadBack}(\cdot)$ on the input $\llbracket S_{nf}' \rrbracket$ runs in time polynomial in the size of $t$.

This seems to contradict the previous result on the complexity of the $\text{ReadBack}(\cdot)$ for the example of Section 4.4, but note that this holds only for the terms of SLL, which admit a bound on the size of their normal form. On the whole, this mainly states that the complexity of the readback is reasonable w.r.t the size of the lambda-term produced. And in the case of SLL typable terms, this happens to be polynomial in the size of the input term.

Nevertheless, the Readback procedure we defined was not meant to be efficient in any way. We believe that it could be significantly improved from this point of view.

**Correctness result**

We can deduce from previous lemmas the main result of this chapter:

**Theorem 4.53.** Let $\Gamma \vdash t : A$ be a Soft type derivation, $\Pi$ the associated Soft Proof-net, and $S$ the Sharing graph s.t $S = \text{Tr}(\Pi)$. If $t \rightarrow^* t_{nf}'$, then $S \rightarrow^* S_{nf}'$ and $\text{ReadBack}(\llbracket S_{nf}' \rrbracket) = t_{nf}'$.

**Proof.** Let $\Pi_{nf}'$ be the normal form of $\Pi$. Since Proof-net reduction simulates beta reduction (Lemma 4.41), $\Pi_{nf}'$ is the Proof-net associated with the derivation $\Gamma \vdash t_{nf}'$. By Proposition 4.50, $\text{ReadBack}(\llbracket \Pi_{nf}' \rrbracket) = t_{nf}'$. So, by Proposition 4.48 — stating that Proof-net reduction preserve the Readback of the GoI in normal form — $\text{ReadBack}(\llbracket \Pi_{nf}' \rrbracket) = \text{ReadBack}(\llbracket S_{nf}' \rrbracket) = t_{nf}'$.

By definition 4.38, $\llbracket \Pi \rrbracket = \text{Tr}(\Pi)$, so $\text{ReadBack}(\llbracket S_{nf}' \rrbracket) = \text{ReadBack}(\llbracket \Pi \rrbracket) = t_{nf}'$, and by Proposition 4.51 Soft sharing graph reduction correspond to GoI execution, so $\llbracket \text{Tr}(\Pi) \rrbracket_{nf} = \llbracket S_{nf}' \rrbracket$.

Hence, $\text{ReadBack}(\llbracket S_{nf}' \rrbracket) = t_{nf}'$.

And then it derives from results on Soft linear logic (GRDR07):

**Corollary 4.54.** Soft sharing graphs is correct and complete for FPTIME.

**Proof.** Since we can evaluate and readback correctly (by Theorem 4.53) all the terms of SLL, in polynomial time (by Corollary 4.6 and Proposition 4.52).
4.6 Conclusion

We have proved that we could design a slight variant of Optimal reduction abstract algorithm in which terms typable in Soft linear logic can be reduced correctly, with a bound approximately equal to the one of Soft linear logic Proof-nets. The proof technique which we have used relies on Geometry of Interaction traces, and on typing in Soft Proof-nets.

The complexity bound on Soft sharing graphs does not depend on Soft typing, but on a combinatorial criterion on the indexes of the nodes. We believe that this indexing criterion could be well adapted to provide an implicit polynomial complexity bound on general Interaction nets, while it seems difficult to adapt Light linear logic discipline to another framework.

Furthermore, it seems that there are strictly more reducible terms within this criterion that within Soft linear logic. The question of finding a correctness criterion for Soft sharing graphs other than Soft typing remains open. It may be interesting in the scope of ICC, but also in the scope Optimal reduction, since lambda terms translated in Sharing graphs with Soft typing are more efficiently reduced than with the initial translation.

We think that our formulation of Geometry of Interaction, and the fact of being outside the usual scope of initial encodings helps to explicit some internal mechanisms of Optimal reduction, even in this restricted case, and, together with the general proof technique, could be well adapted to recover full lambda-calculus with Intuitionistic linear logic typing. What would be needed then is to explicit the link between the cost of Proof-net cut-elimination and the one of Optimal reduction, to get a quantitative correctness result.

Furthermore, the type inference techniques developed for the different Light logics could be adapted so as to provide a translation of lambda calculus in Optimal reduction Sharing graphs that would be more minimal than the existing generic translations, and thus minimise the overheads due to the oracle in the Optimal reduction algorithm. Establishing bridges between Linear logic Intuitionistic Proof-nets cut-elimination and Sharing graphs Optimal reduction, with the help of Geometry of Interaction, seems to us to be a good way to analyse Optimal reduction complexity and thus enhance the efficiency of, eventually, functional programs execution.

Finally, the question of the intrinsic cost of the ReadBack operation would deserve to be explored.
Conclusion

The present work is a contribution to the study of lambda calculus and more precisely, of the computational complexity of its evaluation.

Summary We presented a first result stating that it is possible to efficiently delineate an extensionally complete set of lambda-terms which is correct and complete for the $\text{P-Time}$ complexity class. This method is based on an elegant type system, Dual light affine logic, which allowed for the design of a polynomial time linear decoration algorithm.

The second contribution, the extension of EAL, targets at another aim: extending the typing power of Light logics, to, eventually, reach a state where it can become a tool for static analysis of functional programs. We have internalised the placement of type coercions, which, in the scope of Light type systems, are a real impediment to programming.

Finally, our last work is about the lambda calculus Optimal reduction, in the framework of Light logics: we developed a slight variant of Optimal reduction which correctly reduces lambda-terms typed in Soft linear logic, within the same complexity bound as in Soft proof-nets. Optimal reduction is arguably a good alternative for the reduction of lambda terms: it keeps the nice complexity properties of the Proof-nets reduction, often lost in beta reduction, but is closer to a real implementation.

Further works There is still much to be done in the area of ICC and lambda-calculus: characterise new complexity classes, enhance the existing type inference procedures, etc. A long term goal should be to reach a unified type system and a unified type inference algorithm, which could efficiently and modularly semi-decide the complexity class of a given function. The type inference method developed in chapter 2 for DLAL is elegant — $\text{P-Time}$ and intuitive —, and relies, we think, on a sensible global procedure — the decoration of system F term. It might be of interest to add subtyping to this algorithm and, eventually make it modular to infer both elementary and polynomial bounds in presence of higher-order types.

In the last chapter, we introduced a variant of Optimal reduction inspired by Soft linear logic. It has interesting properties independently of Soft linear logic. The question of its expressive power, and thus of a correctness criterion that would be more permissive than Soft linear logic typability would deserve to be investigated. The combinatorial nature of the indexes discipline which is the reason of the good complexity properties of Soft linear logic and of Soft proof nets is put forward, and might be adapted to other computational models, such as Interaction nets. Finally, we also believe that the proof method used for chapter 4 could be well adapted to provide a formal proof of the correctness of Optimal reduction for any linear
typing, and that using techniques from light logics type inference rather than the usual initial translations may lead to a finer understanding of Optimal reduction complexity.
Bibliography


Appendix A

Source code extracts

A.1  lambda_expr.ml

(* SYNTAX DEFINITIONS *)

(* Parsing types: concrete syntax for types *)

(type t_expr_type =
  ETVAR of string
  | ETFORALL of string * t_expr_type
  | ETFLE of t_expr_type * t_expr_type;;

(* and for terms *)

(type t_expr_term =
  | ENOM of string
  | ELAMBDA of string * t_expr_type * t_expr_term
  | EAPPLY of t_expr_term * t_expr_term
  | ELAMBDA_T of string * t_expr_term
  | EAPPLY_T of t_expr_term * t_expr_type;;

(* Printing for parser debugging for types *)

let rec string_of_expr_type t = match t with
  ETVAR(s) -> s
  | ETFORALL(s,t) -> 'forall ' ^ s ^ '.'^
let rec string_of_expr_term t = match t with
| ENOM(s) -> s
| ELAMBDA(s,expr_type,expr_term) ->
  ("\\"(\s^\" = ^ (string_of_expr_type expr_type) ^ \").
  (string_of_expr_term expr_term))
| EAPPLY(e1,e2) -> "(" ^ (string_of_expr_term e1) ^ ")" ^ (string_of_expr_term e2)
| ELAMBDA_T(s,e) -> "/\" ^ s ^ ": " ^ (string_of_expr_term e)
| EAPPLY_T(e_term,e_type) -> "(" ^ (string_of_expr_term e_term) ^ ")" ^ (string_of_expr_type e_type);

(* Linear combinations — only additions *)
type t_lincomb =
| CCST of int (* Constants : 0,1,... *)
| CPAR of int (* ie CPAR(4) = n_4 : term parameters *)
| CTPAR of int (* ie CTPAR 4 = p_4 : type parameters *)
| CADD of t_lincomb * t_lincomb
| CSUB of t_lincomb * t_lincomb;;

(* Boolean parameter *)
type t_bool =
| CBOOL of int (* ie CBOOL(3) = b_3 : boolean parameter *)
| CTRUE
| CFALSE;;

(* printing combinations *)
let string_of_cbool i = match i with
| CBOOL i -> "b" ^ (string_of_int i)
| CTRUE -> "T" | CFALSE -> "F";;

let rec string_of_lincomb f =
let rec aux f signe = match f with
  | CCST i -> (string_of_int i)
  | CPAR i -> "n" ^ (string_of_int i)
  | CTPAR i -> "p" ^ (string_of_int i)
  | CADD(e1,e2) -> ((aux e1 signe)
      ^ (if signe then " + " else " - ")
      (aux e2 signe))
  | CSUB(e1,e2) -> ((aux e1 signe)
      ^ (if signe then " - " else " + ")
      (aux e2 (not signe)))
in aux f true;;
(***********************************************************************
(* PARAMETRIZED TYPES AND TERMS *)
(* *)
(* Section 4.1 *)
(***********************************************************************
(*
  DLAL enriched F types
  See Section 4.1 *)
)

type t_ftype = (* Line F ::= ... *)
  | TVAR of string
  | TFORALL of string * t_lintype
  | TFLE of t_bangtype * t_lintype
(* parametrized linear : NO BANG for sure *)
and t_lintype = (* Line A ::= ... *)
  | TPRG of t_lincomb * t_ftype
(* parametrized bang : maybe bangs... *)
and t_bangtype = (* Line D ::= ... *)
  | TPRGNG of t_bool * t_lincomb * t_ftype
;;

(* Printing types *)
let rec string_of_lin_type t = match t with
  | TPRG(lincomb,ftype) -> "\^ (string_of_lincomb lincomb) ~ ^ (string_of_ftype ftype) ~ ^"
and string_of_bang_type t = match t with
  | TPRGNG(cbool,lincomb,ftype) -> "\^ (string_of_cbool cbool) ~ ^ (string_of_lincomb lincomb) ~ ^ (string_of_ftype ftype) ~ ^"
and string_of_ftype ft = match ft with
  | TVAR(s) -> s
  | TFORALL(s,lin) -> "(forall ~ s ~ ^ . (~ (string_of_lin_type lin) ~ ^ )"
  | TFLE(s1,s2) -> (string_of_bang_type s1) ~ ^ -> ~ (string_of_lin_type s2);;

(* Extracts respective components of an arrow type *)
let lolli_left tt = match tt with
  | TFLE(1,_) -> 1
  | _ -> failwith "Extraction of left component failed.";;

let lolli_right tt = match tt with
  | TFLE(_,r) -> r
  | _ -> failwith "Extraction of right component failed.";;

(* Substitutes t to v in term and adds combination c to the box parameter. Bang parameter is unchanged. *)
CHAPTER A. SOURCE CODE EXTRACTS

term : btype, ltype or ftype;
v : string
t : ftype
c : lincomb

*)
let rec subst_ltype term v t c = match term with
  TPRG(c2,ft) ->
    (match ft with
      | TVAR(nom) ->
        if nom = v
        then TPRG(CADD(c,c2),t)
        else TPRG(c,ft)
      | x -> TPRG(c2,(subst_ftype ft v t c)))
and subst_btype term v t c = match term with
  TPRGENG(b,c2,ft) ->
    (match ft with
      | TVAR(nom) ->
        if nom = v
        then TPRGENG(b,CADD(c,c2),t)
        else TPRGENG(b,c,ft)
      | x -> TPRGENG(b,c2,(subst_ftype ft v t c)))
and subst_ftype term v t c = match term with
  | TVAR(nom) -> failwith "Incorrect substitution function..."
  | TFORALL(nom,sub) as old_t ->
    if nom = v
    then old_t
    else TFORALL(nom,(subst_ltype sub v t c))
  | TFLE(sub_b,sub_l) -> TFLE((subst_btype sub_b v t c),
                             subst_ltype sub_l v t c);

(* Application of a type to another : then substitutes *)
let appl_ltype t1 t2 = match t1,t2 with
  | TFORALL(nom,t),TPRG(c,ft) -> subst_ltype t nom ft c
  | _ -> failwith "Type application : typing error";;

(* Parametrized system F terms a la Church with parametrized type
decoration : a derivation."

First parameter is always the door parameter.

Last parameter of the constructor is always a parametrized type
decoration. It is the type BELOW the possible door.

An addition is therefore necessary to obtain the parametrized
type above the door.

Corresponds to definition of parametrised terms section 4.1
of chapter 2.

*)
t_type =
let extr_param_pterm pt = match pt with
  | VAR(n, _, _) | ABS(n, _, _, _) | APPL(n, _, _, _)
  | ABST(n, _, _, _) | APPLT(n, _, _, _) -> n;;

let rec string_of_pterm t = match t with
  | VAR(_, idx , b_typ) ->
    '\n' ("^ (string_of_lincomb (CPAR n)) ^ ')" ["^ (string_of_int idx) ^ ':' ^
    (string_of_bang_type b_typ) ^ '"]'
  | ABS(n, param, sub, l_typ) ->
    '\n' ("^ (string_of_lincomb (CPAR n)) ^ ')" [\"(" ^
    (string_of_bang_type param) ^ "]", '" ^ (string_of_pterm sub) ^
    " (abs n" ^ (string_of_int n) ^ "] );
    (string_of_lintype l_typ) ^ ']")'
  | APPL(n, e1, e2, l_typ) ->
    '\n' ("^ (string_of_lincomb (CPAR n)) ^ ')" ["^ (string_of_pterm e1) ^ ')
    " ^ (string_of_pterm e2) ^
    '\n' n @( "^ (string_of_lincomb (CPAR n)) ^ ' : " ^
    (string_of_lintype l_typ) ^ ']")'
  | ABST(n, nom, e1, l_typ) ->
    '\n' ("^ (string_of_lincomb (CPAR n)) ^ ')" [\"(" ^
    " ^ (string_of_pterm e1) ^ " \n \ : " ^
    (string_of_lintype l_typ) ^ ']")'
  | APPLT(n, sub, l_type_arg, l_typ) ->
    '\n' ("^ (string_of_lincomb (CPAR n)) ^ ')" ["^ (string_of_pterm sub) ^ ')
    " (string_of_lintype l_type_arg) ^ " \n \@ : " ^
    (string_of_lintype l_typ) ^ ']");

let rec no_occ t i = match t with
  | VAR(_, j , _) -> if j = i then 1 else 0
  | ABS(_, _, sub , _) -> no_occ sub (i + 1)
  | APPL(_, t1, t2 , _) -> (no_occ t1 i) + (no_occ t2 i)
  | ABST(_, _, sub , _) -> no_occ sub i
  | APPLT(_, sub , _, _) -> (no_occ sub i);;
VAR(_, j, _) -> if j > i then 1 else 0
| ABS(_, _, sub, _) -> aux sub (i + 1)
| APPL(_, t1, t2, _) -> (aux t1 i) + (aux t2 i)
| ABST(_, _, sub, _) -> aux sub i
| APPLT(_, sub, _, _) -> (aux sub i)
in aux t (-1);;

(* Returns the type of the unique free variable of a term. If there are several, the function returns the leftmost one. If there is none the function fails *)
let rec get_fv_btype t =
    let rec aux t i = match t with
    | VAR(_, j, t) -> if j > i then [t] else []
    | ABS(_, _, sub, _) -> aux sub (i + 1)
    | APPL(_, t1, t2, _) ->
        let t1 = (aux t1 i) in
        if t1 = [] then (aux t2 i)
        else t1
    | ABST(_, _, sub, _) -> aux sub i
    | APPLT(_, sub, _, _) -> (aux sub i)
in List.hd (aux t (-1));;

(* Forgets the boolean parameter *)
let bang_to_lin tt = match tt with
    | TPRGNG(b, c, t) -> TPRG(c, t);;

(* Adds a fresh boolean parameter *)
let lin_to_bang tt fr = match tt with
    | TPRG(c, t) -> TPRGNG(CBOOL fr, c, t);;

(* Forgets the initial type decoration for linear types *)
let lin_to_ftype t = match t with
    | TPRG(c, t) -> t ;;

(* Forgets the initial type decoration for bang types *)
let lin_to_ftype t = match t with
    | TPRG(c, t) -> t ;;

(* Extracts types parametrization of terms *)
let extr_type_bang t = match t with
    | VAR(_, t) -> t
    | ABS(_) | ABST(_) | APPLT(_) | APPL(_) -> failwith "No linear type." ;;

let extr_type_lin t = match t with
    | VAR(_, t) -> failwith "No bang type"
    | ABS(_, _, t) | ABST(_, _, t) | APPL(_, _, t)
    | APPL(_, _, t) -> t ;;

(* Coerces to a linear type (even if it was a bang) *)
let extr_type_to_lin t = match t with
A.1 LAMBDA_EXPR.ML

| VAR(_, _, t) -> (bang_to_lin t) | x -> extr_type_lin x;;

(* Picks only the subtype F *)
let extr_type_to_f t = lin_to_ftype (extr_type_to_lin t);;

(*
Extracts and coerces linear type, added with
term parameter (n_i)
*)
let extr_type_to_lin_parametrized t =
match (extr_type_to_lin t) with
| TPRG(CCST 0, tt) -> TPRG(CPAR (extr_param_pterm t), tt)
| TPRG(c, tt) -> TPRG(CADD(c, CPAR (extr_param_pterm t)), tt);;

(* On contexts : decreases all variable indexes, needed
when crossing a lambda. *)
let dec_ctx_extr_zero l = (List.hd l, List.tl l);;

(* Merging of ordered lists elminiating duplicates *)
let rec merge_ctx c1 c2 = match c1,c2 with
| [], _ -> c2
| _, [] -> c1
| (i1, t1)::tl1, (i2, t2)::tl2 ->
  if i1 = i2 then (i1, t1)::(merge_ctx tl1 tl2)
  else (if i1 < i2 then (i1, t1)::(merge_ctx tl1 c2)
    else (i2, t2)::(merge_ctx c1 tl2));;

let rec dec_ctx c1 = match c1 with
| [] -> []
| (i, t)::tl -> (i-1, t)::(dec_ctx tl);;

(* From the concrete syntax tree to the decorated F term
enriched of types.

  t is the concrete type, fb and fi fresh parameters for,
  respectively the boolean and integer parameters.

val decore_type_lin : t_expr_type -> int -> int ->
  t_lintype * int * int
val decore_type_bang : t_expr_type -> int -> int ->
  t_bangtype * int * int
*)
let rec decore_type_lin t fi fb = match t with
| ETVAR(nom) -> (TPRG(CPAR fi, TVAR(nom)), fi + 1, fb)
| ETFORALL(nom, t') ->
  let nouv, fil, fb1 = decore_type_lin t' (fi + 1) fb in
  (TPRG(CPAR fi, TFORALL(nom, nouv))), fil, fb1
| ETFLE(l, r) ->
  let nouv1, fil, fb1 = decore_type_bang l (fi + 1) fb in
let nouv2, fi2, fb2 = decore_type_lin r fi1 fb1 in 
(TPRG(CTPAR fi, TFLE(nouv1, nouv2)), fi2, fb2) 
and decore_type_bang t fi fb = match t with 
| ETVAR(nom) -> 
  TPRGNG(CBOOL fb, CTPAR fi, TVAR(nom)), fi + 1, fb + 1 
| ETFORALL(nom, t') -> 
  let nouv, fi1, fb1 = decore_type_lin t' (fi + 1) (fb + 1) in 
  TPRGNG(CBOOL fb, CTPAR fi, TFORALL(nom, nouv)), fi1, fb1 
| ETFLE(l, r) -> 
  let nouv1, fi1, fb1 = decore_type_bang l (fi + 1) (fb + 1) in 
  let nouv2, fi2, fb2 = decore_type_lin r fi1 fb1 in 
  TPRGNG(CBOOL fb, CTPAR fi, TFLE(nouv1, nouv2)), fi2, fb2 ;;

(* Index of a variable in an environment *)
let index_of id env = 
  let rec search n env = match env with 
  | name::names -> if name = id then n else (search (n+1) names) 
  | [] -> failwith 'Unbound identifier' in 
  search 0 env ;;

(*
From a concrete term, creates a pterm.

Translates term variables into De Bruijn indices.

Parametrizes each node with a fresh parameter,

Parametrized types with the appropriate free decoration:
bang for variables and lambda arguments, linear anywhere else.

Puts the right type annotations at the leaf
(from the lambdas).

In the sequel, we will need a function that decorates
the whole term and thus turns it into a representation
of the F derivation.

Implements the definition of free decoration (section 4.1) of
chapter 2.

val decore_term : t_expr_term -> int -> int -> int -> 
  string list -> 
  (string * t_bangtype) list -> 
  t_pterm * int * int * int
*)
let rec decore_term t fi_term fi_type fb env type_env = 
  let bidon_l = TPRG(CCST 0,TVAR('bidon')) in match t with 
  | ENOM(s) -> 
    VAR(fi_term, index_of s env, List.assoc s type_env), 
    fi_term + 1, fi_type, fb
| E L A M B D A(s, expr_type, expr_term) ->
  let d_type, fi_type, fb =
    decor_type_bang expr_type fi_type fb in
  let nouv, fi_term2, fi_type, fb =
    decor_term expr_term (fi_term + 1) fi_type fb (s::env)
    ((s, d_type):::type_env) in
  ABS(fi_term, d_type, nouv, bidon_l), fi_term2, fi_type, fb
| E A P P L Y(e1, e2) ->
  let nouv1, fi_term2, fi_type, fb =
    decor_term e1 (fi_term + 1) fi_type fb env type_env in
  let nouv2, fi_term2, fi_type, fb =
    decor_term e2 fi_term2 fi_type fb env type_env in
  APPL(fi_term, nouv1, nouv2, bidon_l), fi_term2, fi_type, fb
| E L A M B D A_T(s, e) ->
  let nouv, fi_term2, fi_type, fb =
    decor_term e (fi_term + 1) fi_type fb env type_env in
  let fi_type, fi_type, fb =
    decor_term_lin e_type fi_type fb in
  APPLT(fi_term, nouv, d_type, bidon_l), fi_term2, fi_type, fb;;

let add_ltype_param t par = match t with
| T P R G(CCST 0, ft) -> T P R G(par, ft)
| T P R G(x, ft) -> T P R G(CADD(x, par), ft);;

(*
  Rebuilds the full derivation from the a la Church term and the F type.

  Entry and output have the same type (t_pterm) but the output is more decorated than the input...

  - term : the term. Must have correct and parametrized type annotations consistent with the type parametrization at the leaves and the lambdas.
  - ltype : its F type. Must be correctly parametrised.
  - ctx : context for free variables.
  - fresh_fparam : context for types.
*)

let rec build_tt term fresh_fparam = match term with
  | VAR(n, j, pf) as x -> (x, [(j, pf)], fresh_fparam)
  | ABS(n, pf_var, sub, pf) ->
    let nouv, ctx2, ff = build_tt sub fresh_fparam in
    let extr_type =
      add_ltype_param
      (extr_type_to_lin nouv)
      (CPAR (extr_param_pterm nouv)) in
      ...
(ABS(n, 
    pf_var, 
    nov,
    TPRG(CCST 0,(TFLE(pf_var,extr_type)))),
    dec_ctx ctx2, ff)
| ABST(n,nom,sub, pf) ->
  let sub2, ctx, ff = build_tt sub fresh_fparam in
  let extr_type =
    add_type_param
    (extr_type_to_lin sub2)
    (CPAR (extr_param_pterm sub2)) in
  (ABST(n,nom,sub2,TPRG(CCST 0,TFORALL(nom,extr_type))),
    ctx, ff)
| APPL(n,t1,t2,plin) ->
  (try
    let n1,ctx1, ff = build_tt t1 fresh_fparam in
    let n2,ctx2, ff = build_tt t2 ff in
    (APPL(n,n1,n2,(lolli_right (extr_type_to_f n1))),
      (merge_ctx ctx1 ctx2), ff)
  with
    Failure(s) ->
      (print_string ((string_of_pterm t1) ^
        '\n\napplique a:\n' ^
        (string_of_pterm t2) ^
        's' ^
        (string_of_pterm t)));
  failwith s)
| APPLT(n,sub,lintype, pf) ->
  let sub2, ctx, ff = build_tt sub fresh_fparam in
  (APPLT(n,sub2,lintype,
    (appl_ltype (extr_type_to_f sub2) lintype)),
    ctx, ff)
|)

let rec decore_contexte l fi fb = match l with
  [[] -> [], fi, fb
  | (nom, t)::tl ->
    let t2, fi2, fb2 = (decore_type_bang t fi fb) in
    let t2, fi3, fb3 = (decore_contexte tl fi2 fb2) in
    (t::tl2, fi3, fb3);

let parse (t, ctx) =
  let ctx_typage, fi2, fb2 = decore_contexte ctx l l in
  let n, f_terme, f_type, fb =
    (decore_term t l fi2 fb2
    (fst (List.split ctx)) ctx_typage) in
  let t, c, c2 = (build_tt n f_type) in t;;

(* Lambda terms normalization *)
(* *************************************************************************)
(*************************************************************************)
(*
1. Lift: useful for defining substitution on De Bruijn indices.
*)
let rec lifti nb t k = match t with
  | VAR(n, idx, pf) ->
  |    if idx < k
  |    then t
  |    else VAR(n, idx + nb, pf)
  | ABS(n, t_var, st, pf) ->
  |    ABS(n, t_var, lifti nb st (k + 1), pf)
  | APPL(n, t1, t2, pf) ->
  |    APPL(n, lifti nb t1 k, lifti nb t2 k, pf)
  | APPLT(n, sub, lt, pf) ->
  |    APPLT(n, lifti nb sub k, lt, pf)
  | ABST(n, nom, sub, pf) ->
  |    ABST(n, nom, lifti nb sub k, pf);

let lift nb t = lifti nb t 0;;

(*
2. substitution of the variable of index 0 by sst in t.
*)
let rec substi sst t acc = match t with
  | VAR(n,idx, pf) ->
  |    if idx = acc
  |    then lift acc sst
  |    else (if idx < acc
  |      then t
  |      else VAR(n, idx - 1, pf))
  | ABS(n, t_var, sub, pf) ->
  |    ABS(n, t_var, substi sst sub (acc + 1), pf)
  | APPL(n, s1, s2, pf) ->
  |    APPL(n, substi sst s1 acc, substi sst s2 acc, pf)
  | APPLT(n, sub, lt, pf) ->
  |    APPLT(n, substi sst sub acc, lt, pf)
  | ABST(n, nom, sub, pf) ->
  |    ABST(n, nom, substi sst sub acc, pf);

let subst sst t = substi sst t 0;;

(*
3. Puts a lambda-term in head-normal form
*)
let rec hnf t = match t with
  | VAR(_) -> t
  | ABS(n, t_var, sub, pf) -> ABS(n, t_var, (hnf sub), pf)
  | APPL(n, t1, t2, tt) ->
(match (hnf t1) with
  | ABS(n1,t_var1,sub1,pf1) -> hnf (subst t2 sub1)
  | t1' -> APPL(n,t1', t2, tt))
| APPLT(n,sub,lintype,pf) ->
  let ss' = hnf sub in
  (match ss' with
       | ABST(n,nom,sub_abs,pf) -> sub_abs
         | _ -> APPLT(n,ss', lintype, pf))
| ABST(n, nom, sub, pf) -> ABST(n, nom, hnf sub, pf);;

(* 4. Puts a lambda term t in normal form *)

let rec nf t = match t with
  | VAR(_) -> t
  | ABS(n,t_var,sub,pf) -> ABS(n, t_var, (nf sub), pf)
  | APPL(n,t1,t2,tt) ->
      (match (hnf t1) with
           | ABS(n1,t_var1,sub1,pf1) -> nf (subst t2 sub1)
           | t1' -> APPL(n, nf t1', nf t2, tt))
  | APPLT(n,sub,lintype,pf) ->
      let ss' = hnf sub in
      (match ss' with
           | ABST(n,nom,sub_abs,pf) -> nf sub_abs
             | _ -> APPLT(n, nf ss', lintype, pf))
  | ABST(n, nom, sub, pf) -> ABST(n, nom, nf sub, pf);;

A.2  params.ml

open Lambda_expr;;
open Utils;;

(* type for linear constraints *)
type t_cstr =
      | SUPEG of t_lincomb * t_lincomb
      | EGAL of t_lincomb * t_lincomb;;

(* for boolean constraints *)
type t_bcstr =
      | EQ of t_bool * t_bool
      | IMPLIES of t_bool * t_bool;;

(* and finally when a boolean implies
 a set linear constraints *)
type t_condit =
      | COND_IMPL of t_bool * t_cstr list;;

(* Turns a list of term parameter (ie integers)
 into a sum of linear combination. *)
let rec sigma_doors lst_params = match lst_params with
  | [] -> CCST 0
A.2 PARAMS.ML

| h::[] -> CPAR h |
| h::t -> CADD(CPAR h,(sigma_doors t)); |

(* Extracts the linear parameter of a type *)
let rec extr_param_lin_btype t = match t with |
  | TPRGENG(_,c,_) -> c |
and extr_param_lin_ltype t = match t with |
  | TPRG(c,_) -> c;; |

(* Idem for the boolean parameter *)
let extr_bool_param bt = match bt with TPRGENG(b,_,_) -> b;;

*****************************************************************************
(* FOR LOCAL TYPING CONSTRAINTS *)
*****************************************************************************
(* Section 4.2 *)
*****************************************************************************

(* Generates the constraints suited on the variables types. Thus expects a Bang ptype. *)

Generates constraints of Adm() definition

Returns three sets : linear constraints, boolean constraints, mixed .. *)
let rec cstr_btype_var t = match t with |
  | TPRGENG(b,c,ft) -> |
    | let lc,bc,mc = cstr_ftype_var ft in |
    | (SUPEG(c,CCST 0)::lc, bc, |
    | COND_IMPL(b, [SUPEG(c,CCST 1)]):mc) |
and cstr_ltype_var t = match t with |
  | TPRG(c,ft) -> |
    | let lc,bc,mc = cstr_ftype_var ft in |
    | SUPEG(c, CCST 0)::lc, bc,mc |
and cstr_ftype_var t = match t with |
  | TVAR _ -> [[]],[[[]] | |
    | TFORALL(_,lt) -> cstr_ltype_var lt |
    | TFLE(bt,lt) -> |
      | let a,b,c = cstr_btype_var bt |
      | and a',b',c' = cstr_ltype_var lt in |
      | ((merge_ss_doub a a'), |
       (merge_ss_doub b b'), |
       (merge_ss_doub c c'));; |

let box_cstr_btype n t = |
  SUPEG(CADD(extr_param_lin_btype t,CPAR n), CCST 0));;
let box_cstr_ltype n t = |
  SUPEG(CADD(extr_param_lin_ltype t,CPAR n), CCST 0));;

(* Unification of type parameters according to 4.2 *)
let rec unif_ftypes t1 t2 = match t1, t2 with
  TVAR(_, TVAR(_)) -> [], [], []
| TFLE(l1, r1), TFLE(l2, r2) ->
  let lc, bc, mc = unif_btypes l1 l2 in
  let lc2, bc2, mc2 = unif_ltypes r1 r2 in
  (merge_ss_doub lc lc2),
  merge_ss_doub bc bc2,
  merge_ss_doub mc mc2
| TFORALL(_, t1), TFORALL(_, t2) -> unif_ltypes t1 t2
| _ -> failwith "Unification error: types are not consistent."
and unif_ltypes t1 t2 = match t1, t2 with
  TPRG(c1, ft1), TPRG(c2, ft2) ->
  let lc, bc, mc = unif_ftypes ft1 ft2 in
  EGAL(CSUB(c1, c2), CCST 0)::lc, bc, mc
and unif_btypes t1 t2 = match t1, t2 with
  TPRGNG(b1, c1, ft1), TPRGNG(b2, c2, ft2) ->
  let lc, bc, mc = unif_ftypes ft1 ft2 in
  EGAL(CSUB(c1, c2), CCST 0)::lc, EQ(b1, b2)::bc, mc;;

(* Local typing constraints generation (cf Figure 2.6) *)
let rec cstr_ltyping t = match t with
  | VAR(n, idx, t) ->
    let lc, bc, mc = cstr_btype_var t in
    ((box_cstr_btype n t)::lc, bc, mc)
  | ABS(n, tpar, sub, t) ->
    let lc, bc, mc = cstr_btype_var tpar in
    let lc2, bc2, mc2 = cstr_ltyping sub in
    let lc3, bc3, mc3 =
      (merge_ss_doub (lc1::lc) lc2),
      merge_ss_doub bc bc2,
      merge_ss_doub mc mc2)
  | APPL(n, t1, t2, t) ->
    let nc1 =
      EGAL(CADD((extr_param_lin_ltype (extr_type_to_lin t1)),
                 CPA (extr_param_pterm t1)), CCST 0) in
    let lc, bc, mc = cstr_ltyping t1 in
    let lc2, bc2, mc2 = cstr_ltyping t2 in
    let lc3, bc3, mc3 =
      (merge.ss_doub (nc1::lc) lc2),
      merge_ss_doub bc bc2,
      merge_ss_doub mc mc2)
    in
    let lc4, bc4, mc4 =
      unif_ltypes
      (bang_to_lin (lolli_left (extr_type_to_f t1)))
      (extr_type_to_lin_parametrized t2) in
    let lc5, bc5, mc5 = (merge.ss_doub lc3 lc4),
                       merge_ss_doub bc3 bc4,
                       merge.ss_doub mc3 mc4) in
    (box_cstr_ltype n t)::lc5, bc5, mc5
  | ABST(n, _, sub, t) ->
    let a = (box_cstr_ltype n t) in
    let lc, bc, mc = cstr_ltyping sub in
(merge_ss_doub [a] lc), bc, mc
| APPLT(n,sub,lt,t) ->
| let nc1 =
| EGAL(CADD((extr_param_lin_ltype (extr_type_to_lin sub)),
| CPAR (extr_param_pterm sub)), CCST 0) in
| let a = (box_cstr_ltype n t) in
| let lc,bc,mc = cstr_ltype_var lt in
| let lc2,bc2,mc2 = cstr_ltyping sub in
| a::(nc1::(lc@lc2)),(bc@bc2),(mc@mc2);;

(* Contraction constraints (cf. Section 4.2)
*** DOESN'T WORK FOR FREE VARIABLES ***
val contr_cstr : t_pterm -> t_bestr list *)
let rec contr_cstr t = match t with
| VAR(n,idx,t) -> []
| ABS(n,tpar,sub,t) ->
| let e = cstr_contr sub in
| let nb_occ = no_occ sub 0 in
| if nb_occ > 1
| then (EQ((extr_bool_param tpar), CTUE))::e
| else e
| APPL(n,t1,t2,t) ->
| (cstr_contr t1)@(cstr_contr t2)
| ABST(n,_,sub,t) -> cstr_contr sub
| APPLT(n,sub,lt,t) -> cstr_contr sub;;

*****************************************************************************
(* FOR BOXING CONSTRAINTS *)
(* *)
(* Section 4.3 *)
*****************************************************************************

*****************************************************************************
(* FOR BANG CONSTRAINTS *)
*****************************************************************************

(* Set of constraints ensuring opened door in a the subterm t (except to the free variables if any)
val cstr_opened_door_fv : t_pterm -> t_cstr list *)
let rec cstr_opened_door_fv t =
| let rec aux t i acc = match t with
| VAR(n,idx,_) ->
| if idx > i then [EGAL(sigma_doors (n::acc), CCST 0)]
| else [SUPEG(sigma_doors (n::acc), CCST 1)]
| | ABS(n,_,sub,_) ->
| | let a = SUPEG(sigma_doors (n::acc), CCST 1) in
| | a::(aux_sub (i + 1) (n::acc))
| | APPL(n,t1,t2,___) ->
| | |)
let a = SUPEG(sigma_doors (n::acc), CCST 1) in
let b = aux t1 i (n::acc) in
let c = aux t2 i (n::acc) in
a ::= (b@c)
| ABST(n,_,sub,_) | APPLT(n,sub,_,_) ->
let a = SUPEG(sigma_doors (n::acc), CCST 1) in
a ::= (aux sub i (n::acc))
in aux t (-1) [];

(* Bang constraints (cf 4.4) *)
let rec cstr_bang t = match t with
| APPL(n,t1,t2,_) -> (* Bang subterm *)
  let no_fv = (no_occ_fv t2) in
  let b =
    (extr_bool_param (lolli_left (extr_type_to_f t1)))
  in
  match no_fv with
  0 -> let set = cstr_opened_door_fv t2 in
           let bc1,mc1 = (cstr_bang t1) in
           let bc2,mc2 = (cstr_bang t2) in
           let bc3 = (merge_ss_doub bc1 bc2) and
                     mc3 = (merge_ss_doub mc1 mc2) in
                     (bc3,(COND_IMPL(b,set))::mc3)
| 1 ->
  let b' = (extr_bool_param (get_fv_btype t2))
  in
  let c = IMPLIES(b, b')
  in
  let op_door =
  COND_IMPL(b,cstr_opened_door_fv t2) in
  let bc1,mc1 = (cstr_bang t1) in
  let bc2,mc2 = (cstr_bang t2) in
  let bc3 = (merge_ss_doub bc1 bc2) and
            mc3 = (merge_ss_doub mc1 mc2) in
            c ::= bc3,op_door::mc3
| x ->
  let a = EQ(b,CFALSE) in
  let bc1,mc1 = (cstr_bang t1) in
  let bc2,mc2 = (cstr_bang t2) in
  let bc3 = (merge_ss_doub bc1 bc2) and
            mc3 = (merge_ss_doub mc1 mc2) in
            a ::= bc3,mc3
| VAR(_) -> [] , []
| ABS(_,_,sub,_) -> cstr_bang sub
| ABST(_,_,sub,_) -> cstr_bang sub
| APPLT(_,_,_,_) -> cstr_bang sub ; ;

(**********************************************************************)
(* FOR BRACKETING CONSTRAINTS *)
(**********************************************************************)

(* Adds the given element in head of the head list
head_add : 'a -> 'a list list -> 'a list list

head_add x [] = [x]
head_add x (h::t) = (x::h) :: t
let head_add deco lst = match lst with
| [] -> [[][deco]]
| head::tail -> (deco::head)::tail;;

(* Weak brack *)
let rec w_brack params =
  let rec aux l = match l with
  | [] -> []
  | h::t -> (SUPEG(sigma_doors l, CCST 0))::(aux t)
in aux params;;

(* Strong brack *)
let rec s_brack params = match params with
| [] -> []
| l ->
  (EGAL(sigma_doors l, CCST 0))::(w_brack (List.tl l));;

(* Flattens the n first elements of lst in one flat list *)
let rec aplatit_n lst n =
  try
    if n = 0 then (List.hd lst)
  else (List.hd lst)@(aplatit_n (List.tl lst) (n-1))
  with
  _ -> failwith "Aplatit_n : list trop petite";;

(* Generates bracket constraints, witheout free variables
val cstr_lambda_signi : t_pterm -> t_cstr list
*)
let cstr_brack terme =
  let rec aux t i accu = match t with
  | VAR(n,idx,_) ->
    let nouv = (head_add n accu) in
    let les_params =
      if idx > i (* free variable *)
      then List.flatten nouv
      else aplatit_n nouv idx
    in s_brack les_params
  | APPL(n,t1,t2,_) ->
    let nouv = (head_add n accu) in
    let lst1 = aux t1 i nouv and lst2 = aux t2 i nouv in
    merge_ss_doub lst1 lst2
  | ABS(n,ss_t,_) ->
    let nouv = ([]:(head_add n accu)) in
    let wb_lambda = w_brack (List.flatten nouv) in
    let lst = aux ss_t (i+1) nouv in
    merge_ss_doub wb_lambda lst
  | APPLT(n,sub,_) | ABST(n,sub,_) ->
    aux sub i (head_add n accu)
in aux terme (-1) [];;
(* FOR /\-SCOPE CONSTRAINTS *)

(* Wether a type depends on a type variable *)

let rec depends_ftype tt alpha = match tt with
  | TVAR(name) -> if name = alpha then true else false
  | TFORALL(name,subtype) ->
    (name = alpha) or (depends_ltype subtype alpha)
  | TFLE(sub1,sub2) ->
    (depends_btype sub1 alpha) or (depends_ltype sub2 alpha)
and depends_ltype tt alpha = match tt with
  | TFORALL(name,sub) ->
    (name = alpha) or (depends_ltype sub alpha)
  | TFLE(sub1,sub2) ->
    (depends_btype sub1 alpha) or (depends_ltype sub2 alpha)
and depends_ltype tt alpha = match tt with
  | TPRG(_,ft) -> depends_ftype ft alpha
  | TFLENG(_,_,_,ft) -> depends_ftype ft alpha;;

(* /\ scope for subterms of type abstraction. *)

Must be provided with the variable name
to look for. *)

let cstr_lscope_subterm tt name =
  let rec aux tt accu = match tt with
    | VAR(name,idx,t) ->
      if depends_btype t name then w_brack (n::accu)
      else []
    | APPL(n,t1,t2,t) ->
      let nouv_accu = n::accu in
      let s1 = aux t1 nouv_accu in
      let cstrs = merge_ss_doub (aux t2 nouv_accu) s1 in
      if depends_ltype t name
      then merge_ss_doub cstrs (w_brack nouv_accu)
      else cstrs
    | ABS(n,par,ss_t,t) ->
      let nouv_accu = n::accu in
      let cstrs = aux ss_t nouv_accu in
      if depends_ltype t name
      then merge_ss_doub cstrs (w_brack nouv_accu)
      else cstrs
    | APPLT(n,sub,lt,t) ->
      let nouv_accu = n::accu in
      let cstrs = aux sub nouv_accu in
      if depends_ltype lt name or depends_ltype t name
      then merge_ss_doub cstrs (w_brack nouv_accu)
      else cstrs
    | ABST(n,name_abs,sub,t) ->
      let nouv_accu = n::accu in
      let cstrs = aux sub nouv_accu in
      if name_abs = name
then failwith 'We don't handle type variable capture'
else (  
    if depends_ltype t name  
    then merge_ss_doub cstrs (w_brack nouv_accu)  
    else cstrs  
  )
in aux tt [];

(* Looks for */\ abstractions *)
let rec cstr_lscope tt = match tt with
  VAR(_) -> []  
| APPL(_,t1,t2,_) ->  
    (merge_ss_doub (cstr_lscope t1) (cstr_lscope t2))  
| ABS(_,sub,_) | APPLT(_,sub,_,_) -> (cstr_lscope sub)  
| ABST(_,name_abs,sub,_) ->  
    let cstr1 = cstr_lscope_subterm sub name_abs in  
    let cstr2 = cstr_lscope sub in  
    merge_ss_doub cstr1 cstr2;;

A.3 equs.ml

open Lambda_expr;;
open Utils;;
open Params;;

{******************************
(* PRINTING (IN)EQUATIONS *)
{******************************

(* Prints linear constraints *)
let rec string_of_lcstr l = match l with  
  SUPEG(c1,c2) ->  
    (string_of_lincomb c1) ^ " >= " ^ (string_of_lincomb c2)  
| EGAL(c1,c2) ->  
    (string_of_lincomb c1) ^ " = " ^ (string_of_lincomb c2);;

(* and linear constraints lists *)
let rec string_of_lcstr_list l = match l with  
  [] -> ""  
| h::t ->  
    (string_of_lcstr h) ^ "\n" ^ (string_of_lcstr_list t);;

(* Prints boolean constraints *)
let rec string_of_bcstr b = match b with  
  EQ(b1,b2) ->  
    (string_of_cbool b1) ^ " = " ^ (string_of_cbool b2)  
| IMPLIES(b1,b2) ->  
    (string_of_cbool b1) ^ " => " ^ (string_of_cbool b2);;

(* and boolean constraints lists *)
let rec string_of_bestr_list l = match l with
  | [] -> ""
  | h::t ->
    (string_of_bestr h) ^ '\n' ^ (string_of_bestr_list t);

(* Prints implications : b_n => \{ c => 1 \} *)

let string_of_condit cb =
  (* useful alias for the ^ operator partial application *)
  let string_conc a b = a ^ (string_of_lcstr b) ^ ' ' ; ' in
  match cb with
  | COND_IMPL(tb,cstr_l) ->
    (string_of_cbool tb) ^ ' ' => '{'
    ^ (List.fold_left string_conc '' cstr_l) ^ '}' ; ;

(* Prints implications lists *)

let rec string_of_condit_list l = match l with
  | [] -> ""
  | h::t ->
    (string_of_condit h) ^ '\n' ^ (string_of_condit_list t);

(***************)

(* REDUCING BOOLEAN EQUATIONS *)

(* Section 5.1 *)

(***************)

(*
Splits the constraints list in two :
- list of variables known to be = 1
- Equalities and implications.

in O(n)
*)

let rec splits_bestr lst = match lst with
  | [] -> [],[],[]
  | h::t ->
    let tail_true,tail_false,tail_misc = splits_bestr t in
    (match h with
    | EQ(CBOOL i, CTRUE) ->
      ((i::tail_true),tail_false,tail_misc)
    | EQ(CBOOL i, CFALSE) ->
      (tail_true,((i::tail_false),tail_misc)
    | IMPLIES(_) as c ->
      (tail_true,tail_false,(c::tail_misc))
    | EQ(CBOOL _, CBOOL _) as c ->
      (tail_true,tail_false,(c::tail_misc))
    | _ -> failwith
      'splits_bestr : incorrect constraint form' ) ; ;

(* Set insertion : inserts in a list without duplicates *)
let rec app_ss_doub elem lst =
let rec aux lst = match lst with
  | [] -> [elem]
  | h::t -> if h = elem then lst else h:(aux t)
in aux lst ;;

(* Takes one constraint of the form $b_i = b_j$ or $b_i \Rightarrow b_j$
and, based on the known values sets ($lst\_true$ and
$lst\_false$) tries to apply a reduction rule.

If we can reduce the third member of the returned tuple
is set to true. The two first are the sets of known
values, modified if necessary.
*)
let test_and_set l_true l_false e =
  (match e with
    EQ(CBOOL i, CBOOL j) ->
      let l_true_nouv, modif =
        (if List.mem i l_true
         then ((app_ss_doub j l_true),true)
         else (if List.mem j l_true
         then ((app_ss_doub i l_true),true)
         else l_true,false))
in let l_false_nouv, modif2 =
      (if List.mem i l_false
         then ((app_ss_doub j l_false),true)
         else (if List.mem j l_false
         then (app_ss_doub j l_false),true)
         else (l_false,false)))
in l_true_nouv, l_false_nouv,(modif or modif2)
| IMPLIES(CBOOL i, CBOOL j) ->
    (if List.mem i l_true
       then (app_ss_doub j l_true),l_false,true
       else l_true,l_false,false)
| _ ->
failwith "test_and_set: incorrect inequation form"
);

(*
Applies the above reductions on each constraint. When a
constraint is reduced, we take it away from the
constraint set, for it doesn’t have to be reduced again.
This allows for testing fixpoint by comparing the size
of the set before and after this reduction.

If no constraint in the set has been removed, then none
was reducible, and we have reached fixpoint.
*)
let rec reduce_once lst_true lst_false lst_misc =
    match lst_misc with [] -> lst_true,lst_false,[]
| c::t ->
let nouv_t, nouv_f, modif = 
test_and_set lst_true lst_false c in
let nouv_t, nouv_f, cstrs =
(reduce_once nouv_t nouv_f t) in
if modif then (nouv_t, nouv_f, cstrs)
else (nouv_t, nouv_f, c::cstrs);

(* Calls the above function as many times as necessary to reach fixpoint *)
let reduce_fixpoint lst_true lst_false lst_misc =
let rec aux lst_true lst_false lst_misc taille =
    let res_t, res_f, res_m = reduce_once lst_true lst_false lst_misc in
    let taille2 = List.length res_m in
    if taille = taille2 then (res_t, res_f, res_m)
    else aux res_t res_f res_m taille2
in aux lst_true lst_false lst_misc (List.length lst_misc);

(* Reduces a set of boolean constraints. Returns the result as an equations list b_i = T|F *)
let reduce_bcstr_list t =
    let lst_true, lst_false, lst_misc = splits_bcstr t in
    let res_t, res_f, res_m =
        reduce_fixpoint lst_true lst_false lst_misc in
        (List.map (fun x -> EQ(CBOOL x, CTRUE)) res_t)
       @(List.map (fun x -> EQ(CBOOL x, CFALSE)) res_f);

(* Returns true if lst_true \inter lst_false = \emptyset *)
let rec consistent lst_true lst_false =
    List.fold_left
    (fun x -> fun y -> x && (not (List.mem y lst_false)))
   true
   lst_true;

(* Reduces the set mc of implications (b_i => \{ c >= n \}) according to the boolean constraints set bc (Not necessarily in normal form). *)
let reduce_mcestr list mc bc =
    let lst_true, lst_false, lst_misc = splits_bcstr bc in
    let res_t, res_f, _ =
        reduce_fixpoint lst_true lst_false lst_misc in
    let aux implic = match implic with
        COND_IMPL(CBOOL i, l) ->
            if List.mem i res_t
            then l
            else []
        _ -> failwith 'Incorrect implication form' in
(List.flatten (List.map aux mc)), consistent res_t res_f;;

(* Counts the terms of an inequation *)
let rec counts_terms lin_comb = match lin_comb with
    CPAR _ | CCST _ | CTPAR _ -> 1
  | CADD(t1,t2) | CSUB(t1,t2) ->
    (counts_terms t1) + (counts_terms t2);;

(* True if the given linear combination is a constant *)
let is_constant_term lin_comb = match lin_comb with
    CPAR _ | CTPAR _ -> false
  | CCST _ -> true
  | CADD(_) | CSUB(_) -> false;;

(* True if the equation represents a bound *)
let is_bound lc = match lc with
    SUPEG(t1,t2) | EGAL(t1,t2) ->
    (counts_terms t1) = 1 && (is_constant_term t2)

(* True if b1 is a more restrictive bound than b2 raises exceptions if constraints are incompatible (and thus incomparable) *)
let more_restr b1 b2 = match b1 with
    SUPEG(CPAR i, CCST x) | SUPEG(CTPAR i, CCST x) ->
    (match b2 with
        SUPEG(CPAR i , CCST y) | SUPEG(CTPAR i , CCST y) ->
        x > y
      | EGAL(CPAR i , CCST y) | EGAL(CTPAR i , CCST y) ->
        if y >= x
        then false
        else failwith ('1: Incompatible constraints on p' ^
            (string_of_int i))
      | _ ->
        failwith "more_restr : incorrect constraints form")
  | EGAL(CPAR i, CCST x) | EGAL(CTPAR i, CCST x) ->
    (match b2 with
        SUPEG(CPAR i , CCST y) | SUPEG(CTPAR i , CCST y) ->
        if y > x
        then failwith "Incompatible constraints2"
        else true
      | EGAL(CPAR i , CCST y) | EGAL(CTPAR i , CCST y) ->
        if y = x
        then false
        else failwith "Incompatible constraints3"
| _ -> failwith "more_restr : incorrect constraints form"
| _ -> failwith "more_restr : inequation is no bound";;

(* Adds a bound (in)equation to an existing list. If more restrictive than an existing one, replace it. If less forget it, if no bound is on the variable add it. *)

let rec add_bound lst b = match lst with 
| [] -> [b]
| h::tl ->
  (match h with 
   SUPEG(t, _) | EGAL(t, _) ->
   (match b with 
    SUPEG(t2, _) | EGAL(t2, _) ->
    if t = t2 then (if more_restr b h
                     then (b::tl)
                     else lst)
    else h::(add_bound tl b)));;

(* Splits the equations in two sets: those representing bounds (n_i >= k or n_i = k) and "real equations". *)

let rec splits_bounds_equations lc_list = match lc_list with
| [] -> [],[]
| h::t ->
  let rec_bounds,rec_ineqs = (splits_bounds_equations t) in
  if is_bound h
  then (add_bound rec_bounds h,rec_ineqs)
  else (rec_bounds,h::rec_ineqs);;

(* Eliminates zeros from a linear combination whenever possible. *)

let rec elim_zeros lc = match lc with
  CADD(CCST 0, c) | CADD(c, CCST 0) -> c
| CSUB(c, CCST 0) -> c
| CSUB( _ ) -> lc
| CADD(c1,c2) -> CADD(elim_zeros c1, elim_zeros c2)
| CPAR _ | CCST _ | CTPAR _ -> lc;;

(* Idem on an (in)equation *)
let elim_zeros_ineq ineq = match ineq with
  EGAL(c1,c2) -> EGAL(elim_zeros c1,elim_zeros c2)
| SUPEG(c1,c2) -> SUPEG(elim_zeros c1,elim_zeros c2);;

(* Idem on an (in)equations list *)
let elim_zeros_ineq_lst = List.map elim_zeros_ineq;;
extract term parameters from a combination.

let rec extr_term_parameters lc = match lc with
  CPAR n -> [n]
| CCST _ | CTPAR _ -> []
| CADD(c1,c2) | CSUB(c1,c2) -> (sorted_merge
    (extr_term_parameters c1)
    (extr_term_parameters c2));;

idem on an (in)equality

let extr_term_params_ineq ineq = match ineq with
  EQL(c1,c2) | SUPEG(c1,c2) ->
    sorted_merge
    (extr_term_parameters c1)
    (extr_term_parameters c2));;

extracts the term parameters of a linear (in)equations list

let rec extr_t_params_lst lc_lst =
  List.fold_left
    ((fun x-> fun y-> (sorted_merge (extr_term_params_ineq y) x)))
  [] lc_lst ;;

extract type parameters from a combination *

let rec extr_type_parameters lc = match lc with
  CPAR n -> [n]
| CCST _ | CTPAR _ -> []
| CADD(c1,c2) | CSUB(c1,c2) -> (sorted_merge
    (extr_type_parameters c1)
    (extr_type_parameters c2));;

idem on an (in)equality *

let extr_type_params_ineq ineq = match ineq with
  EQL(c1,c2) | SUPEG(c1,c2) ->
    sorted_merge
    (extr_type_parameters c1)
    (extr_type_parameters c2));;

extract from a list *

let rec extr_type_params_lst lc_lst =
  List.fold_left
    ((fun x -> fun y -> (sorted_merge (extr_type_params_ineq y) x)))
  [] lc_lst ;;

prints an equation in cplex format *

let cplex_string_of_equ e i =
  "c" ^ (string_of_int i) ^ ":^ ^ (string_of_lcstr e) ^ "\n";;
let cplex_string_of_lst l =
    let rec aux 1 i = match l with
        [] -> "
        | h::t -> (cplex_string_of_equ h i) ^ aux t (i+1)
    in aux 1 1;;

let extr_bounded_param w = match w with
    SUPEG(x,_) | EGAL(x,_) -> x;;

let rec extr_unbound_term l1 l2 =
    let lst_bounded = List.map extr_bounded_param l2 in
    let rec aux lst = match lst with
        [] -> []
        | x::tl -> (if List.mem (CPAR x) lst_bounded
            then aux tl
            else (CPAR x)::(aux tl))
    in aux l1;;

let rec extr_unbound_type l1 l2 =
    let lst_bounded = List.map extr_bounded_param l2 in
    let rec aux lst = match lst with
        [] -> []
        | x::tl -> (if List.mem (CTPAR x) lst_bounded
            then aux tl
            else (CTPAR x)::(aux tl))
    in aux l1;;

let rec cstr_unbounds lst idx_deb_type idx_deb_term =
    match lst with
        [] -> []
        | (CTPAR x)::t ->
            (let cstr_eq =
                (EGAL(CADD(CTPAR x,
                    CSUB(CTPAR idx_deb_type,
                        CTPAR (idx_deb_type + 1))),
                    CCST 0))
            in
            let cstr_sup1 = SUPEG(CTPAR idx_deb_type, CCST 0) in
            let cstr_sup2 =
                SUPEG(CTPAR (idx_deb_type + 1), CCST 0) in
                [cstr_eq;cstr_sup1;cstr_sup2]@
                (cstr_unbounds t (idx_deb_type + 2) idx_deb_term))
        | (CPAR x)::t ->
            (let cstr_eq = (EGAL(CADD(CPAR x,
CSUB(CPAR idx_deb_term,
    CPAR (idx_deb_term + 1))
)
in
let cstr_sup1 =
    SUPEG(CPAR idx_deb_term, CCST 0) in
let cstr_sup2 =
    SUPEG(CPAR (idx_deb_term + 1), CCST 0) in
    [cstr_eq; cstr_sup1; cstr_sup2] @
    (cstr_unbounds t idx_deb_type (idx_deb_term + 2))
| _ -> failwith "Badly formed unbounded variables list"

(* Prints LP problem in cplex lp format *)
let cplex_string_of_lcs_lst lst_lst =
  (*let t1 = Sys.time () in *)
let titre = "\Problem Name : typepage DLAL\n" in
let lst_term_params = (extr_t_params_lst lst_lst) in
  (*let t1' = Sys.time () in
let _ = print_string
  ('Tp0:* ^ (string_of_float (t1' -. t1)) ^ '\n') in*)
let lst_type_params = (extr_type_params_lst lst_lst) in
  (*let t2 = Sys.time () in
let _ = print_string
  ('Tp1:* ^ (string_of_float (t2 -. t1')) ^ '\n') in*)
let norm = elim_zeros_ineq_lst lst_lst in
  (*let t3 = Sys.time () in
let _ = print_string
  ('Tp2:* ^ (string_of_float (t3 -. t2')) ^ '\n') in*)
let bounds, ineqs = splits_bounds_equations norm in
  (*let s_bounded = List.map extr_bounded_param bounds in *)
let s_term_unbounded =
    extr_unbound_term lst_term_params bounds in
  (*let t4 = Sys.time () in
let _ = print_string
  ('Tp3:* ^ (string_of_float (t4 -. t3)) ^ '\n') in*)
let s_type_unbounded =
    extr_unbound_type lst_type_params bounds in
let unbounds_bounds, unbounds_ineqs =
    splits_bounds_equations
    (cstr_unbounds
      (s_term_unbounded@s_type_unbounded) 100000 100000
    ) in
  (*let t5 = Sys.time () in
let _ = print_string
  ('Tp4:* ^ (string_of_float (t5 -. t4)) ^ '\n') in*)
let objective =
    sigma_doors (extr_t_params_lst (bounds@unbounds_bounds)) in
let ligne_min = 'Minimize\nobj:' ^
    (string_of_lincomb objective) ^ '\n\n' in
let s_equis = 'Subject To\n' ^
    (cplex_string_of_lst (ineqs@unbounds_ineqs)) ^ '\n' in
let s_bounds = 'Bounds\n' ^
(string_of_lcstr_list (bounds@unbounds_bounds)) ^
(List.fold_left (fun x -> fun y -> x ^ (string_of_lincomb y) ^
  " free\n") '')
  (s_term_unbounded@s_type_unbounded)) ^
'\n' in
(*let t6 = Sys.time () in
  let _ = print_string ('Tps4:' ^
    (string_of_float (t6 -. t5)) ^ '\n') in*)
let s_int = 'integer\n'
  (List.fold_left
    (fun x -> fun y -> (x ^ "n" ^ (string_of_int y) ^ "\n"))
     ** lst_term_params)
  (List.fold_left
    (fun x -> fun y -> (x ^ "p" ^ (string_of_int y) ^ "\n"))
     ** lst_type_params)
  '\n' in
(titre ^ ligne_min ^ s_equa ^ s_bounds ^ s_int ^ "end\n");;
Typage et Réduction Optimale pour le lambda-calcul
dans les variantes de la Logique Linéaire pour la
complexité implicite

Résumé

Le lambda-calcul a été introduit pour étudier les fonctions mathématiques d’un point de vue calculatoire. Il a ensuite servi de fondement au développement des langages de programmation fonctionnels. Savoir si il existe une méthode prouvablement la plus efficace pour réduire les lambda-termes, et connaître la complexité intrinsèque de cette opération en général sont toujours des questions ouvertes.

Dans cette thèse, nous utilisons les outils du typage, de l’inférence de type, de la Logique linéaire et de la Réduction optimale pour explorer ces questions.

Nous présentons un algorithme d’inférence de type pour Dual light affine logic (DLAL), un système de type qui caractérise la classe de complexité polynomiale. L’algorithme prend en entrée un lambda-terme typé dans le système F et renvoie un typage dans DLAL si il en existe un. Une implémentation est fournie.

Puis, nous étendons un système de type fondé sur Elementary affine logic avec du sous-typage, afin d’automatiser le placement des coercitions. Nous montrons que le sous-typage capture bien les coercitions, et nous donnons un algorithme d’inférence complet pour ce système étendu.

Enfin, nous adaptons l’algorithme de Réduction optimale de Lamping pour les lambda-termes typables dans Soft linear logic (SLL), une logique qui caractérise le temps polynomial. Nous montrons qu’une borne polynomial existe pour tous les graphes de partage ainsi réduits, et que les lambda termes typables dans SLL sont réduits correctement.

Abstract

Lambda-calculus has been introduced to study the mathematical functions from a computational point of view. It has then been used as a basis for the design of functional programming languages. Knowing whether there exists a provably most efficient method to reduce lambda-terms, and evaluate the complexity of this operation in general are still open questions.

In this thesis, we use the tools of typing, of Linear logic, of type inference and of Optimal reduction to explore those questions.

We present a type inference algorithm for Dual light affine logic (DLAL), a type system which characterises the polynomial time complexity class. The algorithm takes in input a system F typed lambda-term, and outputs a typing in DLAL if there exists one. An implementation is provided.

Then, we extend a type system based on Elementary affine logic with subtyping, in order to automatise the coercions placement. We show that subtyping captures indeed the coercions, and we give a fully-fledged type inference algorithm for this extended system.

Finally, we adapt Lamping’s Optimal reduction algorithm to the lambda-terms typable in Soft linear logic (SLL), also characterising polynomial time. We prove a complexity bound on the reduction of any Sharing graph, and that lambda-terms typable in SLL can be correctly reduced with our ad-hoc Optimal reduction algorithm.