Alma Mater Studiorum – Università di Bologna

Dottorato di Ricerca in Informatica
Ciclo XXIII
Settore scientifico disciplinare di afferenza: INF/01

Theoretical and Implementation Aspects in the Mechanization of the Metatheory of Programming Languages

Presentata da: Wilmer Ricciotti

Coordinatore Dottorato:
Simone Martini

Relatore:
Andrea Asperti

Esame finale anno 2011

Abstract

Interactive theorem provers are tools designed for the certification of formal proofs developed by means of man-machine collaboration. Formal proofs obtained in this way cover a large variety of logical theories, ranging from the branches of mainstream mathematics, to the field of software verification. The border between these two worlds is marked by results in theoretical computer science and proofs related to the metatheory of programming languages. This last field, which is an obvious application of interactive theorem proving, poses nonetheless a serious challenge to the users of such tools, due both to the particularly structured way in which these proofs are constructed, and to difficulties related to the management of notions typical of programming languages like variable binding.

During our PhD, we worked as part of the development team of the Matita interactive theorem prover, a light-weight system based on the Calculus of (Co)Inductive Constructions developed at the University of Bologna under the supervision of Andrea Asperti. In this period of time, Matita underwent a large reimplementation effort, whose aim was to deliver a more compact and maintainable system, based on well-reasoned design choices. We devoted a substantial part of our work to the implementation of user level, general purpose tactics, which are particularly used in formalizations that involve the metatheory of programming languages.

This thesis is composed of two parts, discussing our experience in the development of Matita and its use in the mechanization of the metatheory of programming languages. More specifically, part I covers:

- the results of our effort in providing a better framework for the development of tactics for Matita, in order to make their implementation and debugging easier, also resulting in a much clearer code;

- a discussion of the implementation of two tactics, providing infrastructure for the unification of constructor forms and the inversion of inductive predicates; we point out interactions between induction and inversion and provide an advancement over the state of the art.

In the second part of the thesis, we focus on aspects related to the formalization of programming languages. We describe two works of ours:

- a discussion of basic issues we encountered in our formalizations of part 1A of the Poplmark challenge, where we apply the extended inversion principles we implemented for Matita;

- a formalization of an algebraic logical framework, posing more complex challenges, including multiple binding and a form of hereditary substitution; this work adopts, for the encoding of binding, an extension of Masahiko Sato's canonical locally named representation we designed during our visit to the Laboratory for Foundations of Computer Science at the University of Edinburgh, under the supervision of Randy Pollack.

Acknowledgements

Writing acknowledgements is always a difficult task. No matter how hard you try, you will always end up forgetting somebody whose support was very important. This is particularly true of a PhD dissertation, marking the end of nine years of study. For all those who won't find their name on this page, please forgive me, I really didn't mean it.

My first thanks goes to my advisor, Prof. Andrea Asperti, for creating a truly enjoyable and stimulating research environment. Every former PhD student knows how much this is important.

Needless to say, I'm also in debt with the rest of the Matita team I had the pleasure to work with: Claudio Sacerdoti Coen, Enrico Tassi and Stefano Zacchiroli were always willing to share with me their knowledge. This was of invaluable help, especially at the beginning of this trip, when I still felt a bit lost.

During my visit to Edinburgh, Dr. Robert Pollack was a constant presence. The experience was truly enlightening. I'm really grateful to him.

I also want to thank all the friends with whom I spent all these years in Bologna. I'm still in contact with most of them, even those who now live far away. Thanks to them, all this time was a lot of fun.

Last, but absolutely not least, a deep thanks goes to my parents and my sister. Even though they understand very little of it, this thesis is also the result of their work.

Contents

Abstract iii

Acknowledgements v

List of Tables xiii

List of Figures xv

1 Introduction 1

Common notation used in the thesis 6

I Development of tactics for Matita 9

2 Tactics in the Matita interactive theorem prover 11

2.1 Architecture of Matita 12

2.1.1 Syntax and notation of CIC terms 14

Inductive types 15

Case analysis 17

2.2 LCF tactics 18

2.3 Tactics in Matita 0.x 20

2.3.1 Tinycals 22

2.3.2 Limitations 24

Poor implementation of declarative languages 25
Unclassified goals 26
2.4 A new type for tactics 27
Branching 29
Shift 30
Positioning 31
Merging 32
Skipping closed tasks 32
2.4.1 LCF-like tactics 35
2.4.2 Final remarks 38

3 Unification of constructor forms in Matita 43
3.1 Overview 44
3.2 Two notions of equality 44
3.2.1 Leibniz's equality 44
3.2.2 John Major equality 45
3.3 Rewriting dependent types with Leibniz equality 46
3.3.1 Leibniz telescopes 47
3.3.2 Simultaneous rewriting principles 48
3.4 The `destruct` tactic 52
3.4.1 Unification steps 54
Streicher's K property 54
Substitution 55
Cascade generalization 55
Injectivity and discrimination 57
Introduction of hypotheses 60
3.4.2 Analysis of the equations 60
3.4.3 The unification algorithm 61

3.4.4 Proof of termination 62
3.4.5 Advantages of our implementation 64

4 Inversion revisited **67**
4.1 Backwards reasoning 68
4.1.1 Case analysis 69
4.1.2 Structural induction 70
4.1.3 Inversion 71
4.2 Proving inversion principles 73
4.3 Mixing induction and inversion 74
4.3.1 Defining induction/inversion principles 79
4.4 Implementation of inversion principles in Matita 84

II Formal metatheory of programming languages **89**

5 Some concrete encodings of binding structures **91**
5.1 Introduction 92
5.2 Concrete encodings of variable bindings 93
5.3 Formalization 97
5.3.1 The type system 97
5.3.2 Proof principles 102
Induction/inversion principles 103
5.3.3 Proofs 104
Locally nameless 104
De Bruijn nameless encoding 107
Named representation 111
5.4 Adequacy of strong typing rules 116
5.5 Conclusions 119

6 Canonical locally named encoding **121**

6.1 Symbolic expressions . . . 122

6.2 Well-formed lambda terms . . . 125

6.2.1 An excellent height function . . . 127

6.3 Multiple binding . . . 131

6.3.1 The multivariate λ-calculus . . . 131

6.3.2 Representing multi-binders . . . 132

An iterative approach . . . 132

A simultaneous approach . . . 134

6.3.3 Multivariate symbolic expressions . . . 134

6.3.4 Well-formed multivariate terms . . . 139

6.3.5 Excellent multivariate heights . . . 139

6.3.6 β-reduction . . . 147

7 A formalization of an algebraic logical framework **151**

7.1 Informal syntax . . . 153

7.1.1 Type system . . . 157

7.1.2 Summary of DMBEL . . . 160

7.2 Symbolic expressions in DMBEL . . . 160

7.3 Well formed expressions . . . 167

7.3.1 An excellent height for DMBEL . . . 170

(DHE) . . . 175

(DHF) . . . 176

(DHP) . . . 176

7.4 Hereditary substitution . . . 180

7.5 Formalization of the type system . . . 188

7.6 Strengthened induction . . . 193

7.7 Conclusions . . . 194

8 Conclusions **195**

References **199**

List of Tables

2.1 CIC terms syntax 14

3.1 The type of n-th order rewriting principles 50
3.2 n-th order rewriting principles 51

6.1 `vclosed` predicate for the λ-calculus (Sato) 125
6.2 $\mathbb{L}$ predicate for the λ-calculus 126
6.3 Well-formedness predicate for the multivariate λ-calculus 140
6.4 β-reduction rules for the multivariate λ-calculus 148

7.1 Syntax of DMBEL 156
7.2 Substitutions in DMBEL 157
7.3 Typing rules of DMBEL 159
7.4 Canonical DMBEL expressions 169
7.5 Typing rules of DMBEL (as formalized) 189

List of Figures

2.1 CIC rule for case analysis 18
2.2 Sample implementation of context stacks 28
2.3 Sample implementation of the branching tinycal 30
2.4 Sample implementation of the shift tinycal 31
2.5 Sample implementation of positioning tinycals 33
2.6 Sample implementation of the merge tinycal 34
2.7 Sample implementation of the skip tinycal 34
2.8 Sample implementation of distribute tactic 40
2.9 Sample implementation of exec 41

4.1 Definining an inductive predicate in Matita 69
4.2 Example: a definition of less-or-equal 77

5.1 Syntax of the type sublanguage of $F_{<:}$ 93
5.2 Well-formedness and subtyping rules of $F_{<:}$ 94
5.3 Syntax of $F_{<:}$ (de Bruijn): types 95
5.4 Syntax of $F_{<:}$ (locally nameless): types 96
5.5 Some rules of the de Bruijn-style formalization of $F_{<:}$. 98
5.6 Some rules of the locally nameless formalization of $F_{<:}$. 100

7.1 DMBEL hereditary substitution (as formalized) 185

Chapter 1

Introduction

This thesis describes issues related to the mechanization of the metatheory of programming languages in the Matita interactive theorem prover [37], discussing on one hand the implementation of the tool, and on the other hand the use of the system in some formalizations.

During our PhD, we worked as part of the Matita development team. Matita is a relatively new tool, developed at the University of Bologna under the supervision of prof. Andrea Asperti during the last ten years. It is based on type theory, using the Calculus of Inductive Constructions as its foundational language ([69, 46, 22]).

One of the main goals of the Matita team, in the last few years, has been a deep revisitation of the whole system, in order to have a more compact tool. This was motivated by our experience telling us that the size of the code of an interactive theorem prover is influenced by a number of minor implementative choices whose impact is not completely clear until very late during the development of the tool. Since correcting these issues by means of patches did not seem a viable option, it was decided that the system would benefit from a reimplementation of its most important components. This would ultimately result in a simpler, more documented design, reducing the number of bugs, improving the maintainability of the system and allowing for a faster training for new developers. Furthermore, a lighter system is also expected to make it easier to experiment with new functionality.

This effort ultimately resulted in a complete rewriting of the kernel of Matita, implementing the reduction machinery and the typechecker of a slightly different version of CIC, which was described in a journal paper we coauthored with the rest of the Matita team [4]. This was followed by a new implementation of the refiner, which is the component of the system responsible for bridging the gap between the logic and the user, implementing a type inference algorithm that allows the system to deal with the partially specified proofs typical of user interaction. As part of the unification algorithm of the new refiner, we cooperated in the development of a new technique (called *unification hints* and described in [5]) that generalizes existing mechanisms like canonical structures and type classes ([63, 67]).

The last step in the revisitation of the system, which is described in this thesis,

concerns the outer layer of Matita, responsible for the interaction with the user by means of proof building statements known as *tactics*, in the style made popular by the LCF theorem prover [24]. Our work in the implementation of the tactics for the new version of Matita focused on those tactics that, while being useful in many formalizations, are particularly critical in the mechanization of the metatheory of programming languages.

This thesis is divided in two parts, describing our experience in the implementation of Matita and its use as a tool for the formalization of properties of programming languages.

The first part is composed of three chapters. Chapter 2, which is based on our work with Asperti, Sacerdoti and Tassi [6], describes the approach to the implementation of tactics that we followed in the new version of Matita. We identify some limitations of the LCF representation for tactics, which is still used as the basis for many current theorem provers. We propose a new type for tactics, which we implemented as part of the new version of Matita. We also describe the basic infrastructure that we provide for the implementation of tactics for Matita.

Chapter 3 discusses our implementation of the `destruct` tactic, whose job is to perform first-order unification of constructor forms involved in equational hypotheses, popularized by C. McBride [39]. The chapter starts by recalling some notions about inductive definitions of equality, then proceeds with an abstract discussion of unification of constructor forms. We then present our implementation of the `destruct` tactic, providing a relatively uniform treatment for equations based on Leibniz's equality or on John Major's equality. This means that, when we dealing with Leibniz's equality on a type for which the Uniqueness of Identity Proofs has been proved, we are able to perform first order unification of constructor forms using almost the same technique that we would use if we were assuming John Major's equality together with its elimination rule.

In chapter 4, we discuss inversion principles [15, 38, 40], providing the equivalent of the informal notion of inversion of an inductive rule that is common especially in logic, and used in many proofs related to theoretical computer science. While inver-

sion principles are commonplace in interactive theorem provers, to our knowledge, nobody has ever pointed out their relation with induction principles. Inversion principles bear a faint resemblance to induction principles, essentially because they are both backwards reasoning techniques; however, somewhat counterintuitively, they do not provide access to induction hypotheses; on the other side, induction principles are not a substitute for inversion. When a mixed notion of *induction/inversion* is needed, the most common strategy is to understand this complex operation as an induction on the height of a proof tree, followed by inversion on the last rule used in the proof; such proof techniques are quite inelegant and also require a relatively important effort by the user to give a formal justification of what is, in the informal proof, a simple proof step. In the chapter, we explain why inversion principles do not provide induction hypotheses, and we identify a class of inductive predicates for which, in certain situations, it is possible to obtain mixed induction/inversion hypotheses; these induction/inversion principles are automatized in Matita, thus relieving the user of the burden of justifying their use.

The second part of the thesis is devoted to formalizations involving the mechanization of the metatheory of programming languages. In recent years, this has been a particularly lively research field, especially for what concerns the study of encodings of binding structures that are suitable to formalization. Benchmarks, such as the POPLmark challenge [7], have also been proposed in order to evaluate the progress of tools and techniques.

Chapter 5 describes different formalizations of part 1A of the PoplMark challenge, involving the type sublanguage of System $F_{<:}$. The formalizations are carried on using three basic encodings of binding structures: de Bruijn's nameless encoding [44], the locally nameless encoding [23, 13], and a fully named representation, which is very close to the informal syntax. The three formalizations allow a comparison of the three approaches to encoding of variable binding. In the formalizations, we exploit the induction/inversion principles that we identified and justified in chapter 4.

Chapters 6 and 7, which conclude the dissertation, are devoted to a more recent,

yet very promising, technique for the representation of binding, which was introduced by Masahiko Sato [61] and formalized the first time by Randy Pollack [62] in the context of pure λ-calculus. This technique provides a local encoding of binding (essentially meaning that the syntax distinguishes between bound variables and free variables); in contrast with the locally nameless encoding, in the Sato encoding bound variables are identified by names instead of indices. This makes the Sato encoding a relative of the locally named encoding used by McKinna and Pollack in [41, 42]: however, the Sato encoding is canonical, meaning that α-equivalence is syntactical equality: this is made possible by a well-formedness predicate that basically identifies only one of the members of each α-equivalence class as legal. The choice of bound variables is made deterministically by means of a height function.

Chapter 6 recalls the theory behind the Sato encoding of binding structures, introducing the syntax of the λ-calculus formalized in the Sato style, the well-formedness predicate, and the goodness properties for heights, ensuring that the representation is adequate. This will be followed by the discussion of a formalization of Pottinger's *multivariate λ-calculus* ([53]), extending the Sato representation to a more complex setting where the same syntactical form can bind an arbitrary number of variables at once.

Chapter 7 is the result of our work at the LFCS of the University of Edinburgh under the supervision of Dr. Pollack. It presents a formalization of Plotkin's DMBEL logical framework ([49]) using the Sato encoding. DMBEL poses interesting challenges to a formalization, including the presence of dependent types and a notion of substitution, known as hereditary substitution, that simultaneously performs some "reduction" in order to keep DMBEL expressions in canonical form. The chapter presents the syntax of DMBEL and its typing rules. Afterwards, we present an extension of the Sato encoding designed to accomodate multiple binding: we give a refined notion of height and define the corresponding excellence properties ensuring that our representation is adequate. Finally we put the representation to work, providing the formal syntax for DMBEL in the Sato representation and showing that whenever a term is well typed, then it is also well formed according to the

Sato representation. To prove this property, we need some metatheoretical proofs including the fact that a well typed term is closed in its typing context.

All the formalizations discussed in the second part of the thesis are publicly available on the author's webpage `http://www.cs.unibo.it/~ricciott/`.

Common notation used in the thesis

Throughout this thesis, it will be often necessary to refer to lists of entities of the same kind. Names referring to lists will be noted with an overline, e.g. $\overline{x}$. An operation returning the length of a list is always defined, and noted $|\overline{x}|$. When referring to a list by means of the sequence of its arguments, we will generally write the elements in their order, using commas as separators (e.g.: $x_1, x_2, \ldots, x_k$). In concrete settings, for example when discussing a formalization, we will also use the ML inspired notation $[x_1; x_2; \ldots; x_k]$ (as a special case, [] will refer to the empty list). In both notations, however, lists should be intended as data structures growing on the right (i.e. the opposite of an ML list). The reverse of a given list $\overline{x}$ will be noted as $(\overline{x})^{rev}$.

For the sake of conciseness, it is sometimes particularly convenient to make the length of a list explicit, while still referring to it with a single name: when we write $\overline{x_n}$, we intend that $\overline{x_n}$ is a list of exactly n items and, in particular, that it is an abbreviation for $x_0, x_1, x_2, \ldots, x_{n-1}$ (notice the 0-based indexing). This is also beneficial in cases where a prefix of some list must be referred to explicitly: whenever $m < n$, a list $\overline{x_m}$ must always be intended as containing the first m items of list $\overline{x_n}$.

The notation for lists is extended to binders. In the case where a binder requires type annotations for its bound variables, we will use *telescopes*, in the style popularized by de Bruijn ([16]): whenever B is a binder, $\overline{x_n}$ is a list of n variable names, $\overline{T_n}$ is a list of n types, and t is admissible as the body of the binder, we will write $B\overline{x_n} : \overline{T_n}.t$ for $Bx_0 : T_0 \ldots Bx_{n-1} : T_{n-1}.t$; the meaning of the notation $B\overline{x} : \overline{T}.t$ is similar, with $\overline{x}$ and $\overline{T}$ having the same, unspecified length.

In chapters 5, 6 and 7 discussing formalizations of languages, we will employ the common notation $(X\ Y)$ to express the operation swapping two names X and Y, keeping all the other names the same, i.e.

$$(X\ Y)(Z) \triangleq \begin{cases} Y & \text{if } Z = X \\ X & \text{if } Z = Y \\ Z & \text{otherwise} \end{cases}$$

This notation will be extended to lists of names having the same length, intending iterated swap of names:

$$(\overline{X_n}\ \overline{Y_n}) = (X_0\ Y_0) \circ (X_1\ Y_1) \circ \cdots \circ (X_{n-1}\ Y_{n-1})$$

While single swaps are involutions, list swaps are not. However they are still invertible: the inverse list swap is obtained by taking the reverse lists of X_n and Y_n.

$$\forall Z.(\overline{X_n}\ \overline{Y_n})(((\overline{X_n})^{rev}\ (\overline{Y_n})^{rev})(Z)) = ((\overline{X_n})^{rev}\ (\overline{Y_n})^{rev})(\overline{X_n}\ \overline{Y_n}(Z)) = Z$$

Part I

Development of tactics for Matita

Chapter 2

Tactics in the Matita interactive theorem prover

This dissertation discusses matters related to the development of the Matita interactive theorem prover and its use as a tool for the formalization of the metatheory of programming languages. The current chapter is devoted to presenting the system, in order to lay the foundations for the following discussion. We will give a rather high-level account of the relevant parts of the architecture of Matita, followed by a brief discussion of its foundational language.

Our main interest will be in the development of tactics for Matita, that is described in the second part of the chapter. We give an account of the new architecture of the tactic subsystem we implemented with the rest of the Matita team, which is also described in [6].

The structure of the chapter is as follows: Section 2.1 is devoted to the architecture of Matita and to basic notions concerning its implementation of the Calculus of (Co)Inductive Constructions; Section 2.2 recalls the original notion of tactic in the LCF proof assistant; Section 2.3 describes the old implementation of tactics; finally, in Section 2.4 we present the type implemented in the new version of Matita.

2.1 Architecture of Matita

An interactive theorem prover is a system that allows the user to interactively prove a theorem by entering commands, called *tactics*, that allow to reduce the initial conjecture to new, simpler ones until all conjectures are trivially proved. Conjectures, which are also called *goals*, can be described as sequents, i.e. pairs formed by the list of hypotheses to be used and the local thesis to be proved. Many interactive theorem provers, including Matita, are based on type theory, by means of the Curry-Howard isomorphism: proofs are represented by terms of a foundational language (in our case, the Calculus of (Co)Inductive Constructions) and formulae by types of the same formalism.

The role of the theorem prover is to provide an interface to the user for the construction of the proofs and to guarantee their soundness (this corresponds to the typechecking of the term encoding the proof).

To partially bridge the gap between informal mathematics and the strict, pedant syntax of logical formalisms, Matita employs several representations of terms, processed by different components of the tool.

- **Completely specified terms** are an implementation of classical CIC terms, roughly as they are presented in theoretical works. At this level, terms are fully annotated with the required types, making syntax particularly verbose. The *kernel* of Matita is the trusted component of the system in charge of proof checking, essentially composed of an implementation of CIC reduction machines and of a CIC typechecker.

- **Partially specified terms** are an extension of CIC terms possibly containing holes or placeholders in place of some subterms. Such holes include linear untyped placeholders (called *implicits*) and typed metavariables (that are Curry-Howard isomorphic to conjectures, or subproofs that are yet to be filled in). Partially specified terms play a role in intermediate representations of ongoing proofs, but also in the possibility, for the human user working with the tool, to omit redundant information when introducing terms. Such missing information can later be inferred by a component called *refiner*, implementing type inference and unification.

- **Content level terms** are an abstraction of the syntactic structure of terms as they are provided by the user. Content level terms allow notational abuse, overloading of operators, and similar peculiarities that are typical of informal mathematics. A *disambiguation* engine performs a translation from this level to partially specified terms.

- For the sake of completeness, we must also cite **presentation level terms**, which capture the proper formatting of mathematical structures as they are presented to the user. They play no role in our discussion.

2.1.1 Syntax and notation of CIC terms

We give here a description of the syntax of CIC as it is implemented in Matita. We only discuss in detail some of the concepts needed for understanding the rest of the dissertation. The following presentation is a simplification of the one we gave in [4] together with Asperti, Sacerdoti Coen, and Tassi, and is also more abstract. Furthermore, we will also omit details, especially about (co)fixpoints and coinductive definitions, which are not needed for our discussion.

$t, u, v, \ldots$	$::=$	x	variables
	$\mid$	c	constants
	$\mid$	$Prop \mid Type_i$	sorts
	$\mid$	$t\ u$	application
	$\mid$	$\lambda x : t.u$	λ-abstraction
	$\mid$	let $(x : t) := u$ in v	local definitions
	$\mid$	$\Pi x : t.u$	dependent product
	$\mid$	match t in I return u [	case analysis
		$\quad k_1\ \overline{x^1} \Rightarrow v_1 \mid \ldots \mid k_n\ \overline{x^n} \Rightarrow v_n$	
		]	
	$\mid$	?	implicit arguments
	$\mid$	$?j[lc]$	metavariable occurrence (with local context)

Table 2.1: CIC terms syntax

Table 2.1.1 shows the syntax of CIC terms. While in CIC types and sorts also fall in the common notion of term, in the rest of the discussion, to increase the readability, we will allow ourselves to use the syntactic convention of indicating types (terms used as types) using letters $T, U, V, \ldots$ and sorts with $\sigma, \tau, \ldots$. I will be reserved to refer to inductive type, and k to inductive type constructors.

Variables, applications, λ-abstractions and dependent products are well-known key ingredients of typed lambda calculi, and there is little to add here. Local defini-

tions (or *let-ins*) are a fairly obvious extension inspired by functional programming languages. Some words are deserved by sorts, which include an impredicative sort of (computationally irrelevant) propositions, called Prop, and a predicative hierarchy of universes (in the style of [36]), denoted by Type_i, where the subscript references one of the user-declared universes.[1]

Constants are the other user defined notion: in general, they are names for object that include

- declarations (axioms, variables...)
- definitions (functions, theorems...)
- recursive and corecursive functions (also known as (co)fixpoints)
- (co)inductive types
- constructors of (co)inductive types

Constants are declared in an *environment* that we will consider fixed in this dissertation.

Finally, as we already said, metavariables ([21, 43]) represent currently open conjectures. Their occurrences are equipped with explicit substitutions (local contexts) that we shall ignore for our purposes.

Inductive types

The shape of the definition of an inductive type I can be approximated[2] as

$$\begin{array}{l}\textbf{inductive } I\ (x_0 : U_0) \cdots (x_{h-1} : U_{h-1}) : \Pi x_h : U_h, \ldots, x_{n-1} : U_{n-1}.\sigma \triangleq \{ \\ \quad k_0 : C_0; \\ \quad \ldots \\ \quad k_{m-1} : C_{m-1} \\ \}\end{array}$$

[1] When the actual universe being referenced is unimportant, we will just write Type

[2] Matita also allows mutual inductive type definitions, whose syntax is not relevant for this discussion.

This definition declares an inductive type I of arity $\Pi\overline{x_n} : \overline{U_n}.\sigma$, whose constructors are $k_0, \ldots, k_{m-1}$. I actually represents an inductive family, parametrized on variables $x_0, \ldots, x_{h-1}$ that must be used uniformly in I: we call them *left parameters* or simply *parameters*. Variables $x_h, \ldots, x_{n-1}$, called *indices* or *right parameters*, are allowed to be instantiated non-uniformly in I, but are also subject to stricter typechecking conditions.

The types of the constructors C_i must match the following form:

$$C_i \triangleq \Pi y_0 : V_0, \ldots, y_{k-1} : V_{k-1}.I\ x_0 \cdots x_{h-1}\ u_h \ldots u_{n-1}$$

In particular, notice that left parameters are used uniformly in the target type of the constructor, while indices range over generic terms. The same restriction holds for recursive occurrences of the type I in $V_0, \ldots, V_{k-1}$. This is better described by a concrete type definition of vectors:

inductive vector $(T : \mathrm{Type}_0) : \Pi n : nat.\mathrm{Type}_0 \triangleq \{$
vnil : vector T 0;
vcons : $\Pi x_1 : nat, x_2 : T, x_3 :$ vector $T\ x_1$.vector $T\ (S\ x_1)$
$\}$

where S constructs the successor of a given natural number. Since the type T of the values contained in the vector is fixed in the definition, it is declared as a left parameter; however, during the constructor of a vector, its length n assumes different values; therefore it must be declared as a right parameter.

Finally, to be accepted by the typechecker, an inductive definition must satisfy some additional conditions, which we state for the sake of completeness:

- the universe σ in which I lives has to be greater or equal to the sort of each of its constructors;
- for each constructor $k_i : \Pi\overline{y_k} : \overline{V_k}.I\ \overline{u}$, I must occur strictly positively in $\overline{V_k}$.

We are not interested in providing the details or the justification for these checks. For a discussion of the theoretical foundations of inductive types, the reader may consult [46].

Case analysis

Among the various categories of terms, the case analysis construct, which is similar to the match-with statement of ML-like languages, has a particularly important role in the treatment of inductive definitions. The expression

$$\begin{array}{l}\text{match } t \text{ in } I \text{ return } u \text{ [} \\ \quad k_0\ \overline{x_{p_0}} \Rightarrow v_0 \mid \ldots \mid k_{n-1}\ \overline{x_{p_{n-1}}} \Rightarrow v_{n-1} \\ \text{]}\end{array}$$

involves a term t of some inductive type I and a list of branches $k_0\ \overline{x_{p_0}} \Rightarrow v_0\ \ldots\ k_{n-1}\ \overline{x_{p_{n-1}}} \Rightarrow v_{n-1}$, one for each constructor k_i of I. Each list of variables $\overline{x_{p_i}}$ is morally an abstraction over the corresponding output term v_i and must match the type and arity of constructor k_i. If t is syntactically equal to $k_i\ \overline{t'}$, then the case analysis reduces to $v_i\{\overline{t'}/\overline{x_{p_i}}\}$.

Finally, u is the return type of the case analysis: due to the dependent type discipline, the type of a case analysis can depend on the type and value of the matching term; given this setting, the typechecker would not always be able to guess a uniform return type for all branches, which must therefore be explicitly provided. Moreover, the case analysis construct also distinguishes between left and right parameters similarly to inductive type definitions. If I has arity $\Pi x_0 : V_0, \ldots, x_{h-1} : V_{h-1}, x_h : V_h, \ldots, x_{m-1} : V_{m-1}.\sigma$, with h left parameters and $m - h$ right parameters, and t has type $I\ t'_0 \cdots t'_{h-1}\ t'_h \cdots t'_{m-1}$, then u must be a term in the form

$$\lambda x_h : V'_h, \ldots, x_{m-1} : V'_{m-1}, y : I\ t'_0 \cdots t'_{h-1}\ x_h \cdots x_{m-1}.T$$

for some type T, where each V'_i is obtained from V_i instantiating the left parameters correctly ($V'_i = V_i\{t'_0,\ldots,t'_{h-1}/x_0,\ldots,x_{h-1}\}$). The type of the whole case analysis expression will then be $u\ t'_h \cdots t'_{m-1}\ t$.

The typing rule for match is shown in Figure 2.1. While most of its hypotheses correspond to the discussion we have just made, it also makes a reference to the possibility that a pattern matching towards some sort τ may not be allowed. The reason not to allow elimination lies in the distinction between computationally relevant parts of a proof (when a term does not have sort Prop) and parts which have

$$\frac{\left(\begin{array}{l}\textbf{inductive } I\ (x_0 : T_0)\cdots(x_{h-1} : T_{h-1}) \\ \quad : \Pi x_h : T_h, \ldots, x_{m-1} : T_{m-1}.\sigma \\ \quad \triangleq \{k_0 : C_0; \ldots; k_{n-1} : C_{n-1}\}\end{array}\right) \text{ is defined} \qquad \Gamma \vdash t : I\ t_0 \cdots t_{m-1}}{\begin{array}{c}\Gamma \vdash u : \Pi x_h : T_h\{{}^{t_0,\ldots,t_{h-1}}/_{x_0,\ldots,x_{h-1}}\}, \ldots, x_{m-1} : T_{m-1}\{{}^{t_0,\ldots,t_{h-1}}/_{x_0,\ldots,x_{h-1}}\}.\tau \\ \text{elimination of } I \text{ towards sort } \tau \text{ is allowed} \\ (\Gamma \vdash \lambda\overline{x_{p_j}}.v_j : \Delta\{C_j\{{}^{t_0,\ldots,t_{h-1}}/_{x_0,\ldots,x_{h-1}}\}, u, (k_j\ t_0 \cdots t_{h-1})\})_{j=0,\ldots,n-1} \\ \hline \Gamma \vdash \text{match } t \text{ in } I \text{ return } u \text{ with } [k_0\ \overline{x_{p_0}} \Rightarrow v_0 | \ldots | k_{n-1}\ \overline{x_{p_{n-1}}} \Rightarrow v_{n-1}] : u\ t_h \cdots t_{m-1}\ t\end{array}}$$

where we define $\Delta\{\ldots\}$ as

$$\begin{aligned}\Delta\{(I\ t_0 \cdots t_{n-1}), u, t\} &= (u\ t_0 \cdots t_{n-1}\ t) \\ \Delta\{\Pi x : T.C, u, t\} &= \Pi x : T.\Delta\{C, u, (c\ x)\}\end{aligned}$$

Figure 2.1: CIC rule for case analysis

no computational content (terms whose sort is Prop). This distinction is crucial for code exportation and proof-irrelevance: the computationally irrelevant subterms are completely forgotten during the automatic exportation of code. Thus, eliminating a non-informative type to obtain an informative type must not be allowed, unless there is only one way in which the elimination can be performed.

2.2 LCF tactics

Before turning to tactic implementation in Matita, we want to recall the representation of tactics in the LCF proof assistant (see also [24], page 210):

```
type thm
type proof = thm list -> thm
type goal = form list * form
type tactic = goal -> (goal list * proof)
```

A goal is a pair formed by a context (a list of formulas that are the hypotheses) and a formula that is the thesis. A tactic can be applied to just one goal and returns both a list of new goals to be proved and a "proof". Intuitively, the tactic reduces the goal to a possibly empty list of simpler goals and asserts the existence of a "procedure" to build a proof (represented in LCF by the type `thm`) of the goal from the proofs of the subgoals. This procedure has type `proof`, i.e. it is an actual ML function from a list of proofs (`thm`s) to a proof `thm`. The `thm` data type is abstract: only functions (i.e. tactics) defined in the ML module can directly construct inhabitants of `thm`, while functions defined outside the module can only combine tactics to build proofs. Thus, if the tactics defined in the module are correct, i.e. they implement sound logical rules, all the system is guaranteed to be correct. It is this latter property that has made the LCF representation so attractive as to allow the technique to become standard.

What is actually stored in the `thm` data type is unspecified in the "LCF approach": it could range from just the goal that is proved (if we are only interested in provability) to a proof term that is a trace of the proof (if we are interested in inspecting and manipulating the proof, e.g. for proof extraction or independent checking). Nevertheless, the `thm` data type can only represent completed proofs (hence the name `thm` that stands for "theorem"). During interactive proof construction, an ongoing proof will be represented only by an ML function from a list of `thm`s to a `thm`. Such a function is obtained by composing together the second component of the return type of the tactics used so far. Being a function, it cannot be inspected or modified in any way.

The LCF data types we have presented are not sufficient alone to fully represent the state of the system between tactics application, i.e. when further input is required to the user. The system needs to store somewhere the set of goals currently open and the function that represents the on-going proof. Moreover, since a tactic can be applied by design only to a single goal, it must also single out one of the opened goals, called the *focused goal*, which will be the argument of the next tactic.

LCF also introduced the notion of *tactical*, which is a higher-order tactic. Tacticals are used to build complex, branching proofs from given tactics by applying tactics according to some strategy. Since the first tactic application can open more than one sequent, during a tactical application we also have the notion of *current goals*, which usually are the new goals recently opened by tactic application during the execution of the tactical. In particular, a tactical must decide the order in which current goals get the focus and the way goals opened by different focused goals are merged together in the set of all current goals. Since the LCF types do not allow to represent these intermediate states, the implementations of the different systems either record this information in the `thm` data type, or leave this information implicit in the control flow data structures (e.g. the stack) of the code that implements the tacticals. In his PhD thesis [33], Kirchner has described an elegant monad, called the proof monad, which allows to lift LCF tactics to tactics and tacticals working on the enriched representation.

2.3 Tactics in Matita 0.x

While the implementation of tactics in the old versions of Matita is clearly inspired by the LCF approach, the theorem prover also includes some features designed to address several of its limitations. The main difference concerns the type `proof` of incomplete proofs, which in the LCF approach is a function and cannot be inspected. In Matita, instead, a proof is a concrete type, containing the proof object in the form of a CIC term, possibly containing "holes" for currently open conjectures.

Existentially quantified metavariables (also allowed by many other current theorem provers) stand for terms that are currently unknown and that will be instantiated later on, usually by means of unification. They arise in three different situations. The first one is when they correspond to implicit, not fully constrained information in a formula, e.g. when the infix notation "$\_ + \_$" is used for the operation of an unknown semi-group in the expression $\forall x, y. x + y = y + x$, which is interpreted as $\forall x, y :?_G. x +_{?_G} y = y +_{?_G} x$ (where $?_G$ is a metavariable to be instantiated later). The

second one is when the user applies a backwards deduction rule, like ∃-introduction, but prefers to delay the choice of the witnesses as much as possible, in the spirit of constraint programming. The third situation generalizes the previous one and is obtained when a deduction rule, e.g. transitivity, is matched (or unified) against the goal, and some metavariables remain free.

Metavariables are not compatible with the LCF data type, since a metavariable can be instantiated by one tactic and the instantiation must be applied to every formula in every goal. The latter operation cannot be performed by the tactic, since it takes in input only the focused goal and not the set of all goals. The observation is not novel and can be explicitly found, for instance, in [1] where Paulson writes "the validation model above does not handle unification. Goals may not contain unknowns to be instantiated later. As a consequence, the LCF user must supply an explicit term at each ∃:right step".

On the contrary, in Matita 0.x, tactics are represented as follows:

```
type proof =
 uri option * metasenv * substitution *
   term Lazy.t * term * attribute list
type goal = int
type status = proof * goal

type tactic
val mk_tactic: (status -> proof * goal list) -> tactic
```

The type `proof` is a tuple containing, among other things, two terms: the first one is the incomplete proof object, while the second keeps its type, i.e. the statement of the theorem being proved. The other items include `metasenv` and `substitution` structures that are used for keeping track of the metavariables used in the proof: the first is an environment declaring the names of currently open (uninstantiated) metavariables, the context in which they were created and their types; the second is

a map from instantiated metavariables to their values, which will be applied lazily at the end of proof.

A tactic acts on a `status`, containing the incomplete proof and a `goal` identifying one of the open metavariables, returning a modified proof and a (possibly empty) list of `goal`s, identifying newly created metavariables resulting from the application of the tactic. When a tactic instantiates a metavariable, it can possibly instantiate other metavariables by side effect, since the `metasenv` and `substitution` structures are shared by all the subgoals.

Having a concrete datatype for incomplete proofs also provides other advantages, including the possibility of performing partial code extraction, and the ability of rendering incomplete proofs.

2.3.1 Tinycals

An additional difference between LCF and Matita 0.x is related to tacticals. Matita, like most modern interactive theorem provers, offers a user interface based on a textual script, input by the user, that is step-by-step checked by the system. The checked part is locked: no edit can be performed on that part without retracting the checked commands (i.e. an undo operation affecting the status of the ongoing proof is performed).

This interaction paradigm suffers from the big step execution semantics of LCF tacticals, which are still today the primary tool to combine together tactics and give a structure to proof scripts. The big step step semantics of tacticals is forced by their type. Being higher-order tactics, they can be executed only when all their arguments are provided. Or better, there is no semantics for the tactical if some of its arguments are unknown.

For example, when a tactic opens heterogeneous goals the user may want to use the branching tactical (`[ ... ; ... ; ... ]`) to run appropriate tactics on every branch. Since it is unlikely that he is able to fill all the blanks (i.e. `...`) in a row he is forced by the system to continuously refine its compound command, execute it to see the result, and retract to able to further refine it. This loop is not only

annoying, requiring additional key-presses or mouse clicks, but also forces the user to type the refined command in a blind way, since he cannot edit the script before he asks the system to retract the last command, and this operation also changes the displayed proof status (i.e. the user has to type the next command step looking at how the goal was many steps before).

Structuring the proof script makes it easier to fix it when it breaks, since the structure of the proof is more explicit. For example failures are detected early since new goals coming from the application of modified lemmas pop up in the right part of the proof (i.e. they are no accidentally delayed). If the user interface does not push the user into giving a proper structure to the proof script, it is unlikely that he would be happy to perform a major redesign of the axioms or basic definitions he is using, since this would break proof scripts, and fixing them would be a very expensive operation.

Another strong point against a big step evaluation semantics of operators to combine commands and structure scripts is that, unless the interaction language is declarative, just reading the proof script is not enough to re-read a proof: single commands have to be executed step-by-step to understand what is going on. In a system equipped with standard (big-step executed) tacticals, what is usually done is (in the rare case in which the proof is structured) to de-structure the proof on the fly, modifying the proof script in such a way that only a part of every compound command is executed. Re-reading a proof script is not only necessary during talks or demos, but is the main activity a team member performs when fixing a script he is not the author of (i.e. that he is not supposed to deeply understand). Given that the cost of writing a formal, mechanically checkable, proof is very high, we believe that every design choice that makes collaboration on the formalization activity harder is to be avoided.

In [55], Sacerdoti Coen, Tassi and Zacchiroli introduced a de-structured language for tacticals (called *tinycals*) that was implemented in Matita 0.x using an additional data structure (similar to the stack that is used to execute functions in a regular programming language). This allows the user, for example, to type just

the command '[' and see its result, then use a tactic, then move to the following goal typing the command '|' and after he is done with the proof branches type the command ']'.

The benefit of the approach is clear: however, since the tinycals stack is not accessible by tactics, they must be implemented as an additional layer. In other words, tactics do not know anything about tinycals and cannot take advantage from them.

2.3.2 Limitations

We now describe some limitations of the Matita 0.x approach to tactics that we tried to address.

All tactics are local A direct consequence of existential metavariables is that a "wrong" instantiation of one of them can make a different goal false, hence not provable. As an example, consider two goals generated by a transitivity law: $\Gamma \vdash a \leq ?_x$ and $\Gamma \vdash ?_x \leq b$. Here $?_x$ stands for the intermediate (still unknown) term c that makes proving $\Gamma \vdash a \leq c$ and $\Gamma \vdash c \leq b$ easier than proving $\Gamma \vdash a \leq b$. If a tactic (especially an automatic one) has a local view over the set of open conjectures (i.e. knows just the goal $\Gamma \vdash a \leq ?_x$), it is unlikely to find an instantiation for $?_x$ that makes proving $\Gamma \vdash ?_x \leq b$ simpler or even possible (think for example of the trivial but useless solution $?_x := a$ that is obtained proving the first goal with the reflexive property of $\leq$). More to the point, restricting the view of tactics to a single input sequent gives them not enough information to detect valid but pointless instantiation of metavariables. With the exception of Isabelle, all other major proof assistants that have accommodated metavariable still see a tactic as a function whose input is just one focused goal and thus does not allow the implementation of non-local tactics in the spirit of constraint programming. While Matita 0.x tactics have a view of the whole proof object, they do not know if the user has selected multiple goals, since they cannot see the tinycals stack.

A different motivation for introducing non-local tactics is given by the wish to extend the user-level $\mathcal{L}_{tac}$ language of Coq [17] with a pattern matching construct over the set of all goals, in the spirit of what is allowed over contexts. $\mathcal{L}_{tac}$ allows the lightweight definition in a script file of ad-hoc tactics that match certain configurations and proceed in the proofs exploiting the domain-knowledge. It would be useful to detect global configurations in order to look for goals that have certain shapes (e.g. can be closed using decision procedures or are more likely to be false or are all instances of a more general conjecture).

Poor implementation of declarative languages The languages of proof assistants are often classified between declarative and procedural ones.

In procedural languages, the user uses tactics that specify how the goal must be manipulated, but not what is expected from the manipulation. Intuitively, it corresponds to the information that remains in a derivation tree by erasing from the premises of each rule all (sub)-formulas that also occur in the rule conclusion. Most tactics for procedural languages are used to find proofs in a top-down way, since the amount of information that can be omitted is maximized in this way. However, tactics for bottom-up reasoning (like tactics to generate logical cuts) can also be present, but are usually more verbose.

In declarative languages, the user uses commands (that we identify with tactics) to build proofs by specifying what is proved at each step, usually omitting how it is proved. Automation supplies the missing justifications. Intuitively, it corresponds to the information that remains in a derivation tree when the name of every rule is omitted and only the tree structure and the formula are kept. Most tactics for declarative languages are used to describe proofs in a bottom-up way. Not every declarative language has tactics for top-down proof steps, but at least case analysis and induction are better captured in this way.

A long standing line of research [28, 70] has tried to implement declarative languages, i.e. declarative tactics, on top of procedural ones. However, the results so far have never been completely satisfactory for two different reasons.

The first has to do with goal selection: when the user needs to prove multiple goals (e.g. the branches of a proof by induction, or the components of a conjuction), declarative languages like Isar [68] allow to dynamically select the goal to be proved, or even to prove something, matching it against the open goal set only later, and possibly up to some easy deductions. Procedural tactics, together with the LCF limitation of just one focused goal, do not allow to implement properly this behaviour. Note that this is exactly the type of control over the proof history that tinycals allow.

The second has to do with information flow: in languages like Isar and Mizar a forward reasoning tactic can prove some fact and at the same time schedule it for usage by the tactic that ends the subproof. The latter tactic can only use the facts explicitly listed by the user in addition to those accumulated by previous tactics. The LCF data type does not allow to pass information around from one tactic to the next ones.

Unclassified goals For a formal system, every conjecture is the same: a set of hypotheses and a conclusion. This is reflected by the LCF type for tactics, where the only distinction between newly generated goals is their position in the output list. For example, when we proceed by induction, we know that some of the new conjectures will need the application of the inductive hypothesis to be solved, while other goals do not. This information is lost in the coarse LCF tactic type, but could be exploited by the system, for example, automatically running procedures like rippling on all inductive cases.

Another example where some new goals deserve a special treatment is generalized rewriting (rewriting with setoids). In that case, a rewriting step generates goals of two kinds: the rewritten conclusion, and a proof that the context under which the rewriting took place is made of morphisms. The latter class of goals can usually be solved automatically, once the user has proved that every elementary functional symbol is a morphism.

Some interactive provers, most notably PVS and ACL2, collect sets of side con-

ditions (like subtyping judgements) the user is *not* expected to immediately solve. Most of these side conditions become trivial when the user enriches the context with additional facts or assumptions and are thus temporarily set aside by these systems.

2.4 A new type for tactics

In Matita 1.0 (the new version of the system, which is currently nearing completion) we use a refined `status` type for the proof status defined as the following OCaml type.

```
(*
 * nCic.ml :
 * type obj = NUri.uri * int * metasenv * substitution * obj_kind
 *)

type tac_status = {
  pstatus : NCic.obj;
  gstatus : context_stack;
}

type tactic = tac_status -> tac_status
```

A tactic status (`tac_status`) is made of a proof status `pstatus` and a context stack `gstatus`. The proof status component carries a (partial) proof object made of a set of open goals and existentially quantified metavariables (`metasenv`), and a data type for partial proofs. In our proposal, goals and existentially quantified metavariables are handled uniformly, for instance by showing all of them to the user as goals and by allowing tactics to either instantiate a metavariable (with a term) or a goal (with a proof).

The context stack is responsible for high-level proof structuring (allowing reordering, focusing, postponing or tagging of goal sets), similarly to the old tinycals

stack. The context stack that equips the proof object also plays the same role of the indexed proof tree in [33]. We will denote tactic statuses by $\mathcal{P}$, or $(\omega; \kappa)$, where ω represents a proof object and κ a context stack.

The major difference of our type for tactics with the standard LCF one is that the input is no longer a single goal, but a global view of the ongoing proof which can be altered. Moreover, it is possible to focus simultaneously on a set of goals. For example a tactic could make no progress (in terms of closing open goals) but it could change the focus to the set of goals in which an existentially quantified metavariable occurs; then another tactic performing automatic proof search could be run on the focused goal set to find an instantiation for the metavariable that allows to solve all goals simultaneously.

```
type task =
  int * [ `Open | `Closed ] * goal * [> `No_tag ]
type context = task list * task list
type context_stack = context list
```

Figure 2.2: Sample implementation of context stacks

We now get into the detail of context stacks. Context stacks are built on the notion of *task*, a higher level abstraction of metavariables (or goals). Tasks index metavariables with numbers that are more easily understood by the user, also distinguishing between open (uninstantiated) and closed (instantiated) tasks. A task can also be associated an arbitrary tag (e.g. to mark it for automation); the unimportant tag will be denoted by $\diamond$. A goal can be present in the stack only once. We will use the notation

$$\#n \mapsto ?_k, tag$$

to refer to the open task associating index $\#n$ with metavariable $?_k$ and tag tag. Likewise, notation

$$\#n \mapsto \blacksquare ?_k, tag$$

refers to a similar, but closed task. We shall also use the symbol $\diamond$ to indicate a standard, unimportant tag.

Lists of tasks will be denoted by letters γ, δ. We also provide a notation $\gamma(\#n_1, \ldots, \#n_k)$ to indicate the sublist of γ containing exactly those tasks indexed by $\#n_1, \ldots, \#n_k$.

The contexts contained in the stack are pairs in the form $\langle \gamma, \delta \rangle$, where γ contains the *focused* tasks, and δ the *locally postponed* tasks.

For contexts stacks, we will borrow the usual sequence notation $\overline{\langle \gamma, \delta \rangle}$. The intended semantics of the stack is that a tactic should normally act on all the focused tasks at once, as designated by the topmost context in the stack, and finally update the list of focused tasks with the newly generated tasks.

This behaviour is illustrated by the following pseudo-tactic:

$$\frac{\begin{array}{c}\mathcal{P} = (\omega; \langle \gamma, \delta \rangle, \overline{\langle \gamma', \delta' \rangle}) \\ \text{act on tasks } \gamma \text{ of } \omega, \text{ obtaining } \omega' \\ \gamma'' = \text{new open tasks in } \omega'\end{array}}{puretac(\mathcal{P}) = (\omega'; \langle \gamma'', \delta \rangle, \overline{\langle \gamma', \delta' \rangle})}$$

The initial stack is composed of the single context $\langle (\#0 \mapsto ?_k, \diamond), \emptyset \rangle$ where $?_k$ refers to the original goal stated by the user.

Along with pure tactics only acting on the currently focused tasks, our type for tactics allows to express proof structuring commands that in usual systems are implemented as tacticals (higher-order tactics), including Matita 0.x tinycals. We will now see how to express tinycals in Matita 1.0.

Branching A new context is pushed onto the stack by the branching tinycal (`[`) that is used in order to be able to re-focus only on subsets of the focused tasks, for instance to apply different tactics to each goal.

$$\frac{\begin{array}{c}\gamma = (\#n_1 \mapsto ?_{k_1}, tag_1), \ldots, (\#n_m \mapsto ?_{k_m}, tag_m) \\ \mathcal{P} = (\omega; \langle \gamma, \delta \rangle, \overline{\langle \gamma', \delta' \rangle}) \\ \gamma_1 = (\#1 \mapsto ?_{k_1}, tag_1) \\ \gamma_2 = (\#2 \mapsto ?_{k_2}, tag_2), \ldots, (\#m \mapsto ?_{k_m}, tag_m)\end{array}}{branch(\mathcal{P}) = (\omega; \langle \gamma_1, \emptyset \rangle, \langle \gamma_2, \delta \rangle, \overline{\langle \gamma', \delta' \rangle})}$$

When the branching tinycal is used, the tasks in γ are re-numbered with their position in the list γ: thus the user will be able to refer to them using indices $\#1, \ldots, \#n$, where n is the number of originally focused tasks. The topmost context $\langle \gamma, \delta \rangle$ is then split in two parts, identifying together a new level of nesting in the proof structure. The new topmost context $\langle \gamma_1, \emptyset \rangle$ is made of just one focused task (indexed by #1 and no postponed tasks; the following context $\langle \gamma_2, \delta \rangle$ is similar to the previous topmost context, except for the removal of task #1: the user can switch to tasks in γ_2 using the shift and positioning tinycals.

```
let branch_tac status =
 let new_gstatus =
  match status.gstatus with
  | [] -> assert false
  | (g, t) :: s ->
     match init_pos g with (* numbers goals *)
     | [] | [ _ ] -> fail
     | task :: tl -> ([task], []) :: (tl, t) :: s
 in
  { status with gstatus = new_gstatus }
```

Figure 2.3: Sample implementation of the branching tinycal

Shift The user can stop working on a single focused task and move to the next one using the shift tinycal (|).

$$\frac{\begin{array}{c}\mathcal{P} = (\omega; \langle \gamma_1, \delta_1 \rangle, \langle \gamma_2, \delta_2 \rangle, \overline{\langle \gamma', \delta' \rangle}) \\ \gamma_2 = (\#n \mapsto ?_k, tag), \gamma_2' \\ \gamma_0 := (\#n \mapsto ?_k, tag) \\ \gamma_1' := \mathit{filter\_open}(\gamma_1)\end{array}}{\mathit{shift}(\mathcal{P}) = (\omega; \langle \gamma_0, \delta_1 \cup \gamma_1' \rangle, \langle \gamma_2', \delta_2 \rangle, \overline{\langle \gamma', \delta' \rangle})}$$

The shift tinycal operates in a context stack containing at least two contexts $\langle \gamma_1, \delta_1 \rangle$ and $\langle \gamma_2, \delta_2 \rangle$ (resulting from a previous application of the branch tinycal and

together identifying the current level of nesting). It moves what is currently focused (γ_1) on the postponed list at the top of the stack ($\delta \cup \gamma_1'$ in the conclusion of the rule); simultaneously, the first task contained in γ_2 is moved to the new focused list γ_0. The *filter_open* operation is used to remove all already closed tasks from γ_1, so that they do not get vacuously postponed.

```
let shift_tac status =
  let new_gstatus =
    match status.gstatus with
    | (g, t) :: (g', t') :: s ->
        (match g' with
          | [] -> fail
          | loc :: loc_tl ->
                (([ loc ], t ∪ filter_open g)
                :: (loc_tl, t') :: s))
     | _ -> fail
  in
    status with gstatus = new_gstatus
```

Figure 2.4: Sample implementation of the shift tinycal

Positioning Immediately after the use of branching or shift, it is possible to use the positioning tinycal ($i_1, \ldots, i_m$:) to stop working on the currently focused tasks and focus on the task numbered by $i_1, \ldots, i_m$ in the current proof nesting level.

$$\frac{\begin{array}{c}\mathcal{P} = (\omega; \langle \gamma_1, \delta_1 \rangle, \langle \gamma_2, \delta_2 \rangle, \overline{\langle \gamma', \delta' \rangle}) \\ \gamma_1 = (\#n \mapsto ?_k, tag) \\ \gamma_0 := (\gamma_1 \cup \gamma_2)(\#\overline{i_m})\end{array}}{pos_{\#\overline{i_m}}(\mathcal{P}) = (\omega; \langle \gamma_0, \delta_1 \rangle, \langle (\gamma_1 \cup \gamma_2) \setminus \gamma_0, \delta_2 \rangle, \overline{\langle \gamma', \delta' \rangle})}$$

Similarly to the shift tinycal, we operate on a context stack containing at least two contexts $\langle \gamma_1, \delta_1 \rangle$ and $\langle \gamma_2, \delta_2 \rangle$. We obtain the new focused task list γ_0 selecting

from $\gamma_1 \cup \gamma_2$ those tasks indexed by $\#\overline{i_m}$; its complement $(\gamma_1 \cup \gamma_2) \setminus \gamma_0$ is put as the focused list of the second context, so that its elements may be focused later by shift or positioning.

The *wildcard* tinycal (`*:`) is a specialized version of positioning, providing a shortcut for focusing on all the remaining non-postponed tasks of the current nesting level.

$$\frac{\mathcal{P} = (\omega; \langle \gamma_1, \delta_1 \rangle, \langle \gamma_2, \delta_2 \rangle, \overline{\langle \gamma', \delta' \rangle}) \quad \gamma_1 = (\#n \mapsto ?_k, tag)}{wildcard(\mathcal{P}) = (\omega; \langle \gamma_1 \cup \gamma_2, \delta_1 \rangle, \langle \emptyset, \delta_2 \rangle, \overline{\langle \gamma', \delta' \rangle})}$$

Merging The merge tinycal (`]`) closes the current level of nesting by merging the two contexts at the top of the stack. Its most common use when the currend focused tasks list is empty, and we need to rise to the outer nesting level to go on with the proof.

$$\frac{\mathcal{P} = (\omega; \langle \gamma_1, \delta_1 \rangle, \langle \gamma_2, \delta_2 \rangle, \overline{\langle \gamma', \delta' \rangle})}{merge(\mathcal{P}) = (\omega; \langle \gamma_2 \cup \mathit{filter\_open}(\gamma_1) \cup \delta_1, \delta_2 \rangle, \overline{\langle \gamma', \delta' \rangle})}$$

The two lists of focused tasks γ_1 and γ_2 are merged in the new topmost context, together with the postponed tasks of the previous topmost context (δ_1). The *filter_open* operation is used with similar purposes to its occurrence in the shift tinycal.

Note that the composition of '`[`', multiple '`|`'s and '`]`' is semantically equivalent to the "thens" LCF tactical.

Skipping closed tasks We have not discussed so far the use of closed tasks. Since we use goals (hence tasks) to represent also metavariables, a tactic can instantiate a metavariable which is not currently focused and thus can be anywhere in the context stack. In this case, we mark the task as closed so that, when the user will later focus on it, he will be aware that the goal has already been automatically closed by side effects. The only tactic which works on closed tasks is the `skip` tinycal that just removes the task by leaving in the script an acknowledgement (the `skip` occurrence) of the automatic choice.

```
let pos_tac i_s status =
  let new_gstatus =
    match status.gstatus with
    | [] -> assert false
    | ([ loc ], t) :: (g', t') :: s
      when is_fresh loc ->
        let l_js =
         filter (fun i,_ -> i ∈ i_s) ([loc] ∪ g')
        in
         ((l_js, t)
           :: (([ loc ] ∪ g') \ l_js, t') :: s)
    | _ -> fail
  in
    status with gstatus = new_gstatus

let wildcard_tac status =
  let new_gstatus =
    match status.gstatus with
    | [] -> assert false
    | ([ g ] , t) :: (g', t') :: s ->
            (([g] ∪ g', t) :: ([], t') :: s)
    | _ -> fail
  in
    status with gstatus = new_gstatus
```

Figure 2.5: Sample implementation of positioning tinycals

$$\frac{\mathcal{P} = (\omega; \langle\gamma, \delta\rangle, \overline{\langle\gamma', \delta'\rangle}) \quad (\gamma(i) = \blacksquare \#i \mapsto ?_{k_i}, tag_i)_{\forall i \in dom(\Gamma)}}{skip(\mathcal{P}) = (\omega; \langle\emptyset, \delta\rangle, \overline{\langle\gamma', \delta'\rangle})}$$

```
let merge_tac status =
  let new_gstatus =
    match status.gstatus with
    | [] -> assert false
    | (g, t) :: (g', t') :: s ->
        ((t ∪ filter_open g ∪ g', t') :: s)
    | _ -> fail
  in
    status with gstatus = new_gstatus
```

Figure 2.6: Sample implementation of the merge tinycal

The skip tinycal checks that all the focused tasks in the current context are closed and, in this case, it clears them out.

```
let skip_tac status =
 let new_gstatus =
  match status.gstatus with
  | [] -> assert false
  | (gl, t) :: s ->
     let gl = map (fun _,_,x,_ -> x) gl in
     if exists ((=) `Open) gl then fail
     else ([], t) :: s
 in
  { status with gstatus = new_gstatus }
```

Figure 2.7: Sample implementation of the skip tinycal

Other tinycals described in [55] require a slightly more elaborate context stack. The most interesting ones are the pair `focus`/`unfocus` that allows to focus on an arbitrary subset of the goals, whereas the focusing tinycals we have described only allows to focus on a subset of the tasks that were focused when `[` was most recently

used.

2.4.1 LCF-like tactics

Tactics as presented so far can freely manipulate the context stack. For instance, tinycals are just tactics that change the stack without changing the goals. However, the most frequent case for a tactic is still that it acts locally on a single focused goal, and does not care about focusing or postponement of the generated goals. For this reason we introduce a simplified type that corresponds to the LCF type extended to support metavariables.

```
type lcf_tactic =
  proof_status -> goal -> proof_status
```

An `lcf_tactic` takes as input a proof status and the focused goal (that must belong to the proof status) and returns a new proof status. The list of new goals can be computed by comparing the metasenv in input and in output. Passing the metasenv (and proof object) around allows the tactic to instantiate metavariables all over the proof. The justification for the tactic is recorded in the proof object. Since we put no requirements on the latter, we are free to implement it either as an ML function or as a concrete, inspectable, data structure like a λ-term if our system is based on the Curry-Howard isomorphism: this is what happens in Matita.

An `lcf_tactic` can be lifted to a `tactic` by applying it in sequence to each focused goal, collecting all the opened goals and turning all of them into the new focused goals on top of the stack. This is implemented by the *distribute* tactic. We formally express the tactic using an auxiliary definition

$$\tau \odot \langle \omega, \gamma, \gamma_o, \gamma_c \rangle \mapsto \langle \omega', \gamma'_o, \gamma'_c \rangle$$

This expression represents the evaluation of an LCF-like tactic τ on a proof object ω, with focused goals γ. ω' is the final, modified proof object, and γ'_o and γ'_c are the sets of new open and closed goals after executing τ; finally γ_o and γ_c are accumulators, used in intermediate steps of the evaluation.

Executing an LCF-like tactic on a set of focused goals is equivalent to executing it in sequence on each goal. If the set is empty, the proof object and the sets of new open and closed goals do not change

$$\frac{}{\tau \odot \langle \omega, \emptyset, \gamma_o, \gamma_c \rangle \mapsto \langle \omega, \gamma_o, \gamma_c \rangle}$$

If $\gamma = ?_k, \gamma''$ (i.e. there is at least one focused goal), there are two cases, depending on whether $?_k$ is an open goal, or it has been closed by side effect. In the latter case (i.e. $?_k \in \gamma_c$), we ignore $?_k$ and proceed to the remaining goals by recursion on γ''.

$$\frac{\begin{array}{c}\gamma = ?_k, \gamma'' \\ ?_k \in \gamma_c \\ \tau \odot \langle \omega, \gamma'', \gamma_o, \gamma_c \rangle \mapsto \langle \omega', \gamma_o', \gamma_c' \rangle\end{array}}{\tau \odot \langle \omega, \gamma, \gamma_o, \gamma_c \rangle \mapsto \langle \omega', \gamma_o', \gamma_c' \rangle}$$

If $?_k$ is still open, we first execute τ on $?_k$ obtaining a new proof object ω', update the accumulators with the new open and closed goals after this execution of τ, and finally perform recursion on γ''.

$$\frac{\begin{array}{c}\gamma = ?_k, \gamma'' \\ ?_k \notin \gamma_c \\ \tau(\omega, ?_k) = \omega' \\ \langle \gamma_o', \gamma_c' \rangle := \mathit{compare\_statuses}(\omega, \omega') \\ \tau \odot \langle \omega', \gamma'', (\gamma_o \cup \{?_k\}) \setminus \gamma_c', \gamma_c \cup \gamma_c' \rangle \mapsto \langle \omega'', \gamma_o'', \gamma_c'' \rangle\end{array}}{\tau \odot \langle \omega, \gamma, \gamma_o, \gamma_c \rangle \mapsto \langle \omega'', \gamma_o'', \gamma_c'' \rangle}$$

Here *compare_statuses* is used to obtain the pair of new open and closed goals by comparing the proof statuses before and after executing τ.

The *distribute* tactic is finally obtained from the above definition, beginning with the accumulators being empty.

$$\frac{\begin{array}{c}\mathcal{P} = (\omega; \langle \gamma, \delta \rangle, \overline{\langle \gamma', \delta' \rangle}) \\ \tau \odot \langle \omega, \gamma, \emptyset, \emptyset \rangle \mapsto \langle \omega', \gamma_o, \gamma_c \rangle\end{array}}{\mathit{distribute}_\tau(\mathcal{P}) = (\omega'; \langle \gamma_o, \delta \setminus \gamma_c \rangle, \mathit{deep\_close}(\gamma_c, \overline{\langle \gamma', \delta' \rangle}))}$$

The γ_o and γ_c resulting from iterative execution of τ are used to synthesize the new context stack: the new focused tasks correspond to new open goals, while new closed goals are removed from postponed tasks. The operation $deep\_close(\gamma_c, \overline{\langle \gamma', \delta' \rangle})$ is used to update the rest of the context stack, marking closed tasks with ■.

When implementing an `lcf_tactic`, it is sometimes useful to call a `tactic` on one goal but, because of lack of the context stack, an `lcf_tactic` can only directly call another `lcf_tactic`. Therefore, we introduce the *exec* operation to turn a tactic $\mathcal{T}$ into an `lcf_tactic` by equipping the proof status with a singleton context stack and by forgetting the returned context stack.

$$\frac{\begin{array}{c}\gamma := (\#0 \mapsto ?_k, \diamond) \\ \mathcal{P} = (\omega; \langle \gamma, \emptyset \rangle) \\ \mathcal{T}(\mathcal{P}) = (\omega'; \overline{\langle \gamma', \tau' \rangle})\end{array}}{exec_{\mathcal{T}}(\omega, ?_k) = \omega'}$$

The functions `exec` and `distribute_tac` form a retraction pair: for each proof status s and goal i,

```
exec (distribute_tac lcf_tac) s g = lcf_tac s g
```

They are inverse functions when applied to just one focused goal or alternatively when restricted to LCF-like tactics, i.e. tactics that ignore the context stack and that behave in the same way when applied at once to a set of focused goals and to each goal in turn. Thus, we can provide a semantics preserving embedding of any LCF tactic into our new data type for tactics. Moreover, as proved in [55], we can also provide a semantics preserving embedding of all LCF tacticals. In the current presentation, this is achieved by means of the block tactic that allows to execute a list of tactics in sequence:

$$\frac{}{block_{\emptyset}(\mathcal{P}) = \mathcal{P}} \qquad \frac{\begin{array}{c}\mathcal{T}(\mathcal{P}) = \mathcal{P}' \\ block_{\overline{\mathcal{T}'}}(\mathcal{P}') = \mathcal{P}''\end{array}}{block_{\mathcal{T},\overline{\mathcal{T}'}}(\mathcal{P}) = \mathcal{P}''}$$

This allows us to implement LCF tactical *thens* as:

$$thens(\mathcal{T}, \overline{\mathcal{T}'}) := block_{\mathcal{T},branch,separate(\overline{\mathcal{T}'}),merge}$$

where $separate(\mathcal{T}'_1, \ldots, \mathcal{T}'_n) = \mathcal{T}'_1, shift, \ldots, shift, \mathcal{T}'_n$.

2.4.2 Final remarks

Since our tactics are applied to the whole proof status, they can reason globally on it, e.g. by using constraint programming techniques to instantiate metavariables constrained by multiple goals. The context stack also provides a list of focused goals tactics are supposed to act on, favouring a kind of reasoning that is intermediate between the global one and the local one of LCF. As in the previous case, abstract data types can be used to prevent global reasoning in favour of the intermediate one.

As a side effect, Tinycals, introduced in [55] with the precise aim of improving the user interface of proof assistants by letting the user effectively write structured scripts, can now be implemented as a special kind of tactics. This contrasts with the previous situation, where they were additional commands to be interleaved with tactics.

Another complaint on the old type for tactics regarded the implementation of declarative commands to perform case analysis or induction. We can now deal with this issue. Suppose that we want to implement a language with the following four commands (tactics):

```
by induction on T we want to prove P
by cases on T we want to prove P
case X (arg1: T1) ... (argn: Tn):
by induction hypothesis we know P (H)
```

A proof by cases or induction is started using one of the first two tactics and continues by switching in turn to each case using the third tactic, as in the following example:

```
by induction on n we want to prove n + 0 = n
 case O:
   ....
 case S (m: nat):
   by induction hypothesis we know m+0 = m (IH)
   ...
```

The user should be free to process the cases in any order. Thus the `case` tactic should first focus on the goal that corresponds to the name of the constructor used. While LCF tactics can only work on the focused goal, and focusing must be performed outside the tactic (the `case` command cannot be implemented as a tactic and thus it cannot be easily re-used inside other tactics), in our approach, the `by cases/induction` tactics can open several new goals that are all focused at once and the `case` tactic simply works on the wanted case only by focusing on it.

Moreover, the semantics of declarative tactics are often based on a working set of justifications that are incrementally accumulated to prove the thesis. E.g. in the Isar-like declarative script

```
  n = 2 * m by H
moreover
  m * 2 = x + 1 by K
hence
  n = x + 1 by sym_times
```

the third inference is justified by the first two propositions and `sym_times`. Thus the semantics of `moreover` and `hence` is that of accumulating the justifications in some set which must be passed around in the proof status. The LCF data type for tactics does not allow to implement this set, whereas in our proposal the proof status can store any information. In particular, this kind of information is better stored in the context stack, e.g. in the form of tags.

A last complaint involved untagged goals. In the new type for tactics, goals are freely tagged in the context stack to attach information to them. A typical application consists in marking proofs of side conditions that the system should try to solve automatically, for instance by resorting to a database of ad-hoc lemmas as in the implementation of Type Classes in Coq by Sozeau [63].

```
(* distribute_tac: lcf_tactic -> tactic *)
let distribute_tac tac status =
 match status.gstatus with
 | [] -> assert false
 | (g, t) :: s ->
    (* aux [pstatus] [open goals] [close goals] *)
    let rec aux s go gc =
     function
     | [] -> s, go, gc
     | (_,switch,n,_) :: loc_tl ->
        let s, go, gc =
         (* a metavariable could have been closed
          * by side effect *)
         if n ∈ gc then s, go, gc
         else
           let sn = tac s n in
           let go',gc' = compare_statuses s sn in
           sn,((go ∪ [n]) \ gc') ∪ go',gc ∪ gc'
        in
        aux s go gc loc_tl
    in
    let s0, go0, gc0 = status.pstatus, [], [] in
    let sn, gon, gcn = aux s0 go0 gc0 g in
    (* deep_close sets all instantiated metavariables
     * to 'Close *)
    let stack = (gon, t \ gcn) :: deep_close gcn s
    in
     { gstatus = stack; pstatus = sn }
```

Figure 2.8: Sample implementation of distribute tactic

```
(* exec: tactic -> lcf_tactic *)
let exec tac pstatus g =
  let stack = [ [0, `Open, g, `No_tag ], [] ] in
  let status =
   tac { gstatus = stack ; pstatus = pstatus }
  in
   status.pstatus
```

Figure 2.9: Sample implementation of exec

Chapter 3

Unification of constructor forms in Matita

3.1 Overview

In this chapter, we report about our implementation of the `destruct` tactic, which is an automated facility to manage equations on inductive types, performing substitutions whenever possible, and ultimately simplifying the status of an on-going proof. This kind of operation, which is often called "object-level unification", because of its analogies with first-order unification ([54]), is especially valuable in logical developments dealing with data-structures or inductively-defined predicates.

While the implementation we give is clearly based on McBride's original work ([39]), we describe some aspects that are specific to our version, including direct support for the unification of Leibniz equalities.

3.2 Two notions of equality

In type theory, equality is often defined as an inductive predicate having a single "reflexivity" constructor. We describe two common definitions following this approach.

3.2.1 Leibniz's equality

Leibniz's principle of the identity of indiscernibles roughly states that two entities are equal if any property of the first also holds for the second and vice-versa. This can be translated in the Calculus of Constructions as

$$eq_L[x, y : T] \triangleq \Pi P : T \to \text{Type}.(P\ x \leftrightarrow P\ y)$$

In CIC, however, it is more convenient to use the following inductive definition:

```
inductive eq (T:Type) (x:T) : ∀y:T.Prop :=
| refl  : eq T x x.
```

If x and y have the same type T, we will write $x = y$ for `eq T x y`; we will also write $\mathcal{R}_x$ for the proof of $x = x$ obtained using the constructor `refl`. This definition, which

can only be used if x and y have the same type T, is also called "Leibniz equality", because the principle of the identity of indiscernibles is an instance of the elimination principle $\mathcal{E}_=$ of `eq`:

$$\begin{aligned}\mathcal{E}_= \quad : \quad & \Pi T : \text{Type}, x : T. \\ & \Pi P : (\Pi x_0 : T, p_0 : x = x_0.\text{Type}).P\ x\ \mathcal{R}_x \rightarrow \\ & \Pi y : T, e : x = y.P\ y\ e\end{aligned}$$

This elimination principle, read backwards, performs *rewriting* of the goal $P\ y\ e$ into $P\ x\ \mathcal{R}_x$ according to an equation $e : x = y$. Since $\mathcal{E}_=$ is an elimination principle, the following reduction rule holds:

$$\mathcal{E}_=\ T\ x\ P\ p\ x\ \mathcal{R}_x \triangleright_\iota p$$

3.2.2 John Major equality

John Major equality is similar to Leibniz equality, except that it is possible to equate terms inhabiting different types. However, as in Leibniz equality, every term is only equal to itself; thus when two terms are equal, so must be their types.

John Major equality is also defined as an inductive type:

```
inductive JMeq (T:Type) (x:T) : ∀U:Type.∀y:U.Prop :=
| refl_jmeq : JMeq T x T x.
```

If x has type T and y has type U, we will write $x \simeq y$ for `JMeq T x U y`. We also write $\mathcal{R}^J_x$ for the identity proof of $x \simeq x$ obtained using constructor `refl_jmeq`.

The standard elimination principle for John Major equality is not particularly useful. To really be able to rewrite using $\simeq$, we must introduce in the system a specific axiom for rewriting:

$$\begin{aligned}\textit{JMeqElim} \quad : \quad & \Pi T : \text{Type}, x : T. \\ & \Pi P : (\Pi x_0 : T, p_0 : x \simeq x0.\text{Type}).P\ x\ \mathcal{R}^J_x \rightarrow \\ & \Pi y : T, e : x \simeq y.P\ y\ e.\end{aligned}$$

The intended behaviour of *JMeqElim* should be that of an elimination principle: similarly to the Leibniz case, the following reduction rule should hold.

$$JMeqElim\ T\ x\ P\ p\ x\ \mathcal{R}_x^J \triangleright p$$

However, axioms are treated as black boxes by the system, therefore such a term does not reduce. In general, the use of `JMeq` makes the proofs and definitions given in Matita less computational. To alleviate this problem, we also want to provide a reasonable support for Leibniz equations on dependent types.

3.3 Rewriting dependent types with Leibniz equality

When we deal with dependent types, it is often the case that we must deal with *telescopes* of equations, rather than single equations. For example, suppose that we are given two vectors v of length m and w of length n, and that we also know that the equality between dependent pairs

$$\langle m, v\rangle = \langle n, w\rangle$$

holds. This is a perfectly legal Leibniz equality, since both terms in the equation have the same type $\Sigma x : \mathbb{N}.vec\ x$. We would like to state that v and w are equal; however, this statement is only meaningful under the condition that m and n are also equal. Because of the dependent typing discipline, if we want to rewrite v with w, we are forced to also rewrite m with n at the same time, in the goal and possibly in some hypotheses too. In case of a more complicated dependent type, it is necessary to rewrite an arbitrary number of terms at the same time.

After McBride's work [39], there is a general agreement that using John Major equality is a very natural way to deal with equations on dependent types. Since John Major equality does not impose any constraint on the types of the terms being equated, it is actually possible to state $m \simeq n$ and $v \simeq w$. Furthermore, it is possible

to state the simultaneous rewriting principle

$$\begin{aligned} EqSubst_2 \quad : \quad & \Pi T_0 : \mathrm{Type}, T_1 : (\Pi x_0 : T_0.\mathrm{Type}). \\ & \Pi P : (\Pi x_0 : T_0, x_1 : T_0\ x_0.\mathrm{Type}). \\ & \Pi a_0 : T_0, a_1 : T_1\ a_0.P\ a_0\ a_1 \rightarrow \\ & \Pi b_0 : T_0.a_0 \simeq b_0 \rightarrow \\ & \Pi b_1 : T_1\ b_0.a_1 \simeq b_1 \rightarrow P\ b_0\ b_1 \end{aligned}$$

that is also provable assuming the specific elimination principle of John Major equality. This principle is easily extended to telescopes of equations of any length.

As known in folklore, there is no actual need for John Major equality to state such principles. The reason behind this choice is mostly practical, as simultaneous rewriting for Leibniz equality is more difficult to derive and justify. In this section we give a formal account of simultaneous rewriting with Leibniz equality.

3.3.1 Leibniz telescopes

As we just said, the Leibniz equality $v = w$ is not legal, since the lengths m and n of the two vectors are not convertible. It is however possible to prove that they are propositionally equal: in fact, we may derive a hypothesis

$$e_0 : m = n$$

Using the elimination principle of the Leibniz equality, we can rewrite v using equation e_0. The rewritten vector has type $vec\ n$ and can be equated to w, which has the same type:

$$e_1 : \mathcal{E}_= \mathbb{N}\ m\ (\lambda x, p.vec\ x)\ v\ n\ e_0 = w$$

We say that e_0, e_1 is a *Leibniz telescope* of length 2. Also, the elimination principle $\mathcal{E}_=$ can be thought of as a rewriting principle for telescopes of length 1. We will therefore reference it by the notation

$$\rho_1[T_0, T_1, a_0, a_1, b_0, e_0] = \mathcal{E}_=\ T_0\ a_0\ T_1\ a_1\ b_0\ e_0$$

For telescopes of arbitrary length, the i-th equation must depend on all the previous $i-1$ equations: this amounts to rewriting its lefthand side simultaneously with a telescope. Assume we already defined the rewriting principle of order $i-1$ (notation: ρ_{i-1}); then the i-th equation in the telescope will be in the form $\rho_i[\overline{T_i}, \overline{a_i}, \overline{b_{i-1}}, \overline{e_{i-1}}] = b_i$ where we are equating a_i and b_i, under the assumption that the telescope $\overline{e_{i-1}}$ equates the two sequences of terms $\overline{a_{i-1}}$ and $\overline{b_{i-1}}$. $\overline{T_i}$ is used to reconstruct the types of all the $\overline{a_i}$ and $\overline{b_i}$.

3.3.2 Simultaneous rewriting principles

The previous commentary shows that Leibniz simultaneous rewriting principles, compared with their John Major equivalents, have a more critical status: not only are they used to rewrite telescopes of equations, but they are also needed to actually *define* them!

For a gentle introduction to Leibniz rewriting principles, let us first introduce the second order principle ρ_2, used to rewrite with telescopes of length 2, which is similar to the $EqSubst_2$ used for John Major equality.

$$
\begin{aligned}
\rho_2 \;:\; & \Pi T_0 : \mathrm{Type}, a_0 : T_0. \\
& \Pi T_1 : (\Pi x_0 : T_0, p_0 : a_0 = x_0.\mathrm{Type}), a_1 : T_1\ a_0\ \mathcal{R}_{a_0}. \\
& \Pi T_2 : (\Pi x_0 : T_0, p_0 : a_0 = x_0, x_1 : T_1\ x_0\ p_0. \\
& \qquad \rho_1[T_0, T_1, a_0, a_1, x_0, p_0] = x_1 \to \mathrm{Type}). \\
& T_2\ a_0\ \mathcal{R}_{a_0}\ a_1\ \mathcal{R}_{a_1} \to \\
& \Pi b_0 : T_0.\Pi e_0 : a_0 = b_0. \\
& \Pi b_1 : T_1\ b_0\ e_0.\Pi e_1 : \rho_1[T_0, T_1, a_0, a_1, b_0, e_0] = b_1. \\
& T_2\ b_0\ e_0\ b_1\ e_1
\end{aligned}
$$

This principle, which is provable, is not very different from $EqSubst_2$, except for the fact that it references Leibniz telescopes, made of explicitly rewritten terms. This ultimately requires the types T_1 and T_2 to be abstracted on telescopes of equations, too.

In the general case, to build an n-th order rewriting principle, we first define $\rho_0 : \lambda T_0 : \mathrm{Type}, x_0 : T_0.x_0$ (rewriting with 0 equations is the identity) and $\rho_1 \triangleq \mathcal{E}_=$

(rewriting with 1 equation is performed by the usual elimination principle for $=$). Then, to define the ρ_{n+2}, for any natural number n, we will have to assume that the lower order principles $\rho_0, \ldots, \rho_{n+1}$ have been defined, since a telescope of length $n+2$ references those rewriting principles.

The reader might be worried that, telescopes being defined in terms of rewriting principles, and rewriting principles defined in terms of telescopes, those definitions might be circular. However, if we construct rewriting principles bottom-up, beginning with ρ_2 and finally reaching any ρ_n, we can convince ourselves that the definition is well posed. We still have to give a fully formal definition of the ρ_n principles. First, in Table 3.1, we define the type of ρ_n.

The definition is built by means of parametric sub-expressions. The type of ρ_n is provided by M_0^n, which alternatively abstracts predicates $\overline{T_n}$ and terms $\overline{a_n}$ (such that the type of each a_i is obtained in terms of T_i and the previous $\overline{a_{i-1}}$); then $N_0^n[\overline{T_n}, \overline{a_n}]$ abstracts alternatively over terms $\overline{b_{n-1}}$ and the telescope of equations $\overline{e_{n-1}}$. Each e_i respects the shape of the telescope, with a_i as the lefthand side and b_i as the righthand side. The type of each b_i is returned from T_i, using the previous $\overline{b_i}$ and $\overline{e_i}$ as its arguments.

The type of each predicate T_m, depending on the previous values of $\overline{T_m}$ and $\overline{a_m}$, is built by an auxiliary definition $V_0^m[\overline{T_m}, \overline{a_m}]$, abstracting alternatively on terms $\overline{x_m}$ and equations $\overline{p_m}$; these two are typed similarly to the b_i and e_i.

We will not content ourselves with assuming the existence of the ρ_n principles: it is possible to define them concretely, in terms of the elimination principle $\mathcal{E}_=$, as shown in Table 3.2. The definition of ρ_n, where $n \geq 2$, begins with a sequence of λ-abstractions matching in number and type the Π-abstractions of M_0^n; then, to rewrite the telescope $\overline{e_n}$, we first rewrite the single equation e_{n-1}, followed by a "recursive" rewriting of order $n-1$ on the subtelescope $\overline{e_{n-1}}$. Notice that "recursion" here is purely at the meta-level: ρ_n, strictly speaking, is not defined as a recursive CIC function. There is no easy way to define ρ_n once and for all for any n, because the type of ρ_n mentions ρ_{n-1}, and the type system of Matita does not allow a recursive definition to occur inside its own type. However, we can still derive single

$$
\begin{array}{rcl}
V_n^n[\overline{T_n},\overline{a_n},\overline{x_n},\overline{p_n}] & \triangleq & \mathrm{Type} \\
V_i^n[\overline{T_n},\overline{a_n},\overline{x_i},\overline{p_i}] & \triangleq & \Pi x_i : T_i\ x_0\ p_0 \cdots x_{i-1}\ p_{i-1}. \\
 & & \Pi p_i : \rho_i[\overline{T_{i+1}},\overline{a_{i+1}},\overline{x_i},\overline{p_i}] = x_i. \\
 & & V_{i+1}^n[\overline{T_n},\overline{a_n},\overline{x_{i+1}},\overline{p_{i+1}}] \qquad (\text{when } i < n)
\end{array}
$$

$$
\begin{array}{rcl}
M_n^n[\overline{T_n},\overline{a_n}] & \triangleq & \Pi T_n : V_0^n[\overline{T_n},\overline{a_n}]. \\
 & & \Pi a_n : T_n\ a_0\ \mathcal{R}_{a_0} \cdots a_{n-1}\ \mathcal{R}_{a_{n-1}}. \\
 & & N_0^n[\overline{T_{n+1}},\overline{a_{n+1}}] \\
M_i^n[\overline{T_i},\overline{a_i}] & \triangleq & \Pi T_i : V_0^i[\overline{T_i},\overline{a_i}]. \\
 & & \Pi a_i : T_i\ a_0\ \mathcal{R}_{a_0} \cdots a_{i-1}\ \mathcal{R}_{a_{i-1}}. \\
 & & M_{i+1}^n[\overline{T_{i+1}},\overline{a_{i+1}}] \qquad (\text{when } i < n)
\end{array}
$$

$$
\begin{array}{rcl}
N_n^n[\overline{T_{n+1}},\overline{a_{n+1}},\overline{b_n},\overline{e_n}] & \triangleq & T_n\ b_0\ e_0 \cdots b_{n-1}\ e_{n-1} \\
N_i^n[\overline{T_{n+1}},\overline{a_{n+1}},\overline{b_i},\overline{e_i}] & \triangleq & \Pi b_i : T_i\ b_0\ e_0 \cdots b_{i-1}\ e_{i-1}. \\
 & & \Pi e_i : \rho_i[\overline{T_{i+1}},\overline{a_{i+1}},\overline{b_i},\overline{e_i}] = b_i. \\
 & & N_{i+1}^n[\overline{T_{n+1}},\overline{a_{n+1}},\overline{b_{i+1}},\overline{e_{i+1}}] \qquad (\text{when } i < n)
\end{array}
$$

$$
\begin{array}{rcl}
\rho_n & : & M_0^n \\
\rho_n[\overline{T_{n+1}},\overline{a_{n+1}},\overline{x_n},\overline{p_n}] & \triangleq & \rho_n\ T_0\ a_0 \cdots T_n\ a_n\ x_0\ p_0 \cdots x_{n-1}\ p_{n-1}
\end{array}
$$

Table 3.1: The type of n-th order rewriting principles

n-th order instances of ρ.

Proposition 3.1 *For all natural numbers n, the definition of ρ_n is well founded.*

Proof: For $n \leq 1$, the statement is trivial. For $n \geq 2$, we show that the definition we gave is well founded. We define the degree of the expressions involved in the

$$
\begin{array}{rcl}
\rho_0 & \triangleq & \lambda T_0, a_0.a_0 \\
\rho_1 & \triangleq & \mathcal{E}_= \\
\rho_{n+2} & \triangleq & \lambda T_0, a_0, \ldots, T_{n+2}, a_{n+2}, b_0, e_0, \ldots, b_{n+1}, e_{n+1}. \\
& & \quad \rho_1\ (T_{n+1}\ b_0\ e_0 \cdots b_n\ e_n)\ (\rho_{n+1}[\overline{T_{n+2}}, \overline{a_{n+2}}, \overline{b_{n+1}}, \overline{e_{n+1}}]) \\
& & \quad\quad (\lambda y_{n+1}, q_{n+1}.T_{n+2}\ b_0\ e_0\ b_n\ e_n\ y_{n+1}\ q_{n+1}) \\
& & \quad\quad (\rho_{n+1}\ T_0\ a_0 \cdots T_n\ a_n \\
& & \quad\quad\quad (\lambda x_0, p_0, \ldots, x_n, p_n. \\
& & \quad\quad\quad\quad T_{n+2}\ x_0\ p_0 \cdots x_n\ p_n \\
& & \quad\quad\quad\quad\quad (\rho_{n+1}[\overline{T_{n+2}}, \overline{a_{n+2}}, \overline{x_{n+1}}, \overline{p_{n+1}}]) \\
& & \quad\quad\quad\quad\quad \mathcal{R}_{\rho_{n+1}[\overline{T_{n+2}}, \overline{a_{n+2}}, \overline{x_{n+1}}, \overline{p_{n+1}}]}) \\
& & \quad\quad\quad a_{n+1}\ b_0\ e_0 \cdots b_n\ e_n) \\
& & \quad\quad b_{n+1}\ e_{n+1}
\end{array}
$$

Table 3.2: n-th order rewriting principles

definition as the following quadruples:

$$
\begin{array}{rcl}
\mathit{degree}(V_0^0) = \mathit{degree}(M_0^0) = \mathit{degree}(N_0^0) = \mathit{degree}(\rho_0) = \langle 0,0,0,0\rangle \\
\mathit{degree}(V_i^m) & = & \langle m-1, m, 0, m-i\rangle \\
\mathit{degree}(M_i^m) & = & \langle m, m-1, 1, m-i\rangle \\
\mathit{degree}(N_i^m) & = & \langle m, m-1, 0, m-i\rangle \\
\mathit{degree}(\rho_m) & = & \langle m, m, 0, 0\rangle
\end{array}
$$

This implicitly assumes $i \leq m$, which is always satisfied in the definition. Then, by direct inspection, we see that the expressions V, M, N, ρ are defined in terms of subexpression of lesser degree, under lexicographic ordering of quadruples. Since lexicographic ordering is well-founded, so is the definition of ρ_n. □

Our interest being the implementation of tactics for rewriting with dependent types, we only state that all rewriting principles are well typed.

Claim 3.2 *For all natural numbers n, ρ_n is well typed.*

A completely formal proof of this assertion would require a very involved well-founded induction. An informal though not completely satisfactory argument to justify the claim can be obtained considering the type of ρ_n compared to that of ρ_{n-1}. It is easy to convince ourselves that the abstractions of ρ_{n-1} are a subset of those of ρ_n: therefore, for ρ_n to be well typed, if ρ_{n-1} is, we can assume the types of $\overline{T_n}, \overline{a_n}, \overline{b_{n-1}}, \overline{e_{n-1}}$ to be well formed. So, the only abstracted variables whose types might not be well-formed are T_n, a_n, b_{n-1} and e_{n-1}. However, inspecting the definition, we see that their types are, essentially, composed of subexpressions of the well-formed types of $\overline{T_n}, \overline{a_n}, \overline{b_{n-1}}, \overline{e_{n-1}}$.

3.4 The `destruct` tactic

This section discusses the implementation of the `destruct` tactic, implementing unification for constructor forms, in a style which is close to McBride's style. Differences in the implementation reflect the possibility of using only Leibniz equality with its ρ_n rewriting principles. Our tactic is sufficiently generic to allow unification for both Leibniz equality and John Major equality, with a large base of shared code between the two. Currently, the implementation does not discriminate cyclic equations.

Our discussion will only describe the behaviour of the tactic in the case of Leibniz equations, since it is more general than the John Major case. Before discussing the details, we give a formal definition of constructor form:

Definition 3.1 *The set of* terms in constructor form *is the smallest set $\mathcal{S}$ such that:*

- *$x \in \mathcal{S}$ for all variables x;*
- *if k is an inductive type constructor and $t_1, \ldots, t_n \in \mathcal{S}$, then $(k\ t_1 \cdots\ t_n) \in \mathcal{S}$.*

An equation is said to be in constructor form if both sides of the equation are terms in constructor form.

Because we allow the use of Leibniz equality, in our discussion, we will also need to consider a more general form of equation:

Definition 3.2 *An equation is said to be in* rewritten constructor form *if it is in the form*

$$\rho_n[\overline{T_{n+1}}; \overline{a_{n+1}}; \overline{b_n}; \overline{e_n}] = b_n$$

and the terms a_n and b_n are in constructor form.

In practice, in rewritten constructor forms we allow that the constructor form in the left hand side be rewritten by a Leibniz telescope, so that its type matches the type of the right hand side.

The `destruct` tactic is essentially an LCF-like tactic (in the sense made clear in Section 2.4.1): it works locally on a single goal and either produces a single new goal, whose context has been rewritten as required; or it closes the selected goal when it is possible to prove a contradiction. The status $\mathcal{P}$ on which an LCF-like tactic works can be described as

$$\mathcal{P} : (\Gamma \vdash \Phi)$$

where Γ is the context of the single selected goal, and Φ its thesis. In the rest of the chapter, we will note the effect of executing a tactic α on an LCF status $\mathcal{P}$, possibly with additional parameters $\overline{x}$, as

$$\alpha(\mathcal{P}, \overline{x}) : (\Gamma' \vdash \Phi')$$

if α returns a single new goal with updated context Γ' and updated thesis Φ', and as

$$\alpha(\mathcal{P}, \overline{x}) : \emptyset$$

if α closes the current goal.

In the next section, we discuss the basic steps performed by the tactic. In Sections 3.4.2 and 3.4.3, we will see how those operation can be combined to perform the unification.

3.4.1 Unification steps

Streicher's K property

Identities are trivial tautologies and as such provide no useful information to prove the goal. Therefore, we just want to discard them, no matter what the term involved in the identity is. However, in our setting we should also consider the possibility that an identity hypothesis is explicitly referenced in the goal. Concretely, if the current goal is in the form

$$\Gamma, P : t = t \to \mathrm{Prop} \vdash \Pi e : t = t.P\ e$$

we would like to turn it in a form where e has been substituted by the canonical reflexivity proof $\mathcal{R}_t$. Unfortunately, as we know, uniqueness of identity proofs is not provable in standard CIC in the general case ([30, 31]), being equivalent to the elimination principle for John Major equality. The only thing we can do is to assume that uniqueness of identity proofs, in the form of Streicher's K property, has been proved for the type involved in the equation, either concretely or as an axiom. This allows us to construct an operation κ transforming proof problems as follows:

$$\frac{\kappa(\mathcal{P}) : \Gamma \vdash \Phi[\mathcal{R}_t]}{\mathcal{P} : \Gamma \vdash \Pi e : t = t.\Phi[e]}$$

Therefore, as expected, we cannot perform unification without assuming some form of axiom: however, using Leibniz equality, we can limit the use of axioms to the case where we really need to rewrite an identity as the reflexivity proof (compare this to the case of John Major equalities, where every rewriting step corresponds to an instance of an axiom). Furthermore, while uniqueness of identity proofs is not provable in general, it can be proved for a reasonably general class of types (for example, we can prove that identity on natural numbers has the K property): in these cases, the system can be instructed to use concrete instances of the K property, instead of the generic axiom.

Substitution

Substitution is the operation we perform when we want to get rid of an equational hypothesis that is not an identity, and one of its sides is a variable. Intuitively, if an equation e has type $x = t$, we can use the rewriting principle ρ_1 to substitute all occurrences of x with t in the goal. More formally, we provide an operation ς that behaves as follows:

$$\frac{\varsigma(\mathcal{P}, e) : \Gamma, \Gamma' \vdash \Phi[t, \mathcal{R}_t]}{\mathcal{P} : \Gamma, e : x = t, \Gamma' \vdash \Phi[x, e]}$$

A careful reader should immediately notice that the assumptions we made are not enough to ensure that using ς we will obtain a well typed goal. In fact, it is not always possible to substitute all the occurrences of some variable without breaking the dependent typing discipline. In the judgement

$$n : \mathbb{N}, f : \forall m : \mathbb{N}, v : \text{vec } m \to \text{bool}, w : vec\ n, e : n = 0 \vdash f\ n\ w = \text{true}$$

replacing n with 0 is not allowed: $f\ 0\ w$ is not well typed, because the type of w is $vec\ n$, which is not convertible with $vec\ 0$.

Clearly, this problem arises because the hypothesis w used in the goal has not been rewritten. If we generalize the goal with respect to w before performing the substitution, then the operation will be successful.

Another problem is that the substitution operation also clears hypothesis e from the context, even though it could be possibly used in subcontext Γ'. This could also be solved generalizing hypotheses from Γ' as needed. The `destruct` tactic will use operation ς carefully, after generalizing all the hypotheses depending from the variable being rewritten.

Cascade generalization

Generalization of a hypothesis is the operation, performed at the user level by the `generalize` tactic, corresponding to the proof step

$$\frac{\gamma_0(\mathcal{P}, t) : \Gamma, t : T, \Delta \vdash \Pi x : T.\Phi[x]}{\mathcal{P} : \Gamma, t : T, \Delta \vdash \Phi[t]}$$

the intended meaning of the operation being "to obtain a proof of $\Phi[t]$, we will prove the stronger statement $\Pi x : T.\Phi[x]$".

As we said in the previous paragraph, it is often necessary, in order to perform a substitution step, to generalize all the hypotheses depending on the variable x being substituted. This not only includes hypotheses directly referencing x in their types, but also those hypotheses referencing other hypotheses recursively depending on x. We call the operation used to collect all this hypotheses *cascade selection* and the resulting generalization *cascade generalization*.

Cascade selection is implemented by the following algorithm:

1. We start with a context Γ and a list `acc` that we will progressively fill with the names of hypotheses we need to generalize; `acc` will be initialized with a singleton list containing the name of the hypothesis from which we want to compute the dependencies (e.g. the name of the variable we want to substitute);

2. If Γ is empty, just return `acc`;

3. If $\Gamma = h : T, \Gamma'$, we check if any of the hypotheses in `acc` occurs in T: in this case, we add h to `acc`; otherwise, we keep `acc` the same.

4. Finally, we iterate the procedure on Γ' and the updated `acc`.

We provide two versions of the cascade generalization operation. In one case, we just want to generalize all the hypotheses depending on one variable:

$$\frac{\begin{array}{c}\text{cascade\_select}(\Gamma, x) = [t_1; \ldots; t_n] \\ t_1 : T_1, \ldots, t_n : T_n \in \Gamma \\ \gamma(\mathcal{P}, x) : \Gamma \vdash \Pi t_1 : T_1, \ldots, t_n : T_n.\Phi[t_1; \ldots; t_n]\end{array}}{\mathcal{P} : \Gamma \vdash \Phi[t_1; \ldots; t_n]}$$

In a slightly different case, there is a single hypothesis we do not want to generalize: we will remove it from the list obtained using cascade selection just after performing that operation (we write $l_1 \setminus l_2$ to mean the list obtained after removing the items in l_2 from the list l_1)

$$\frac{\begin{array}{c}\text{cascade\_select}(\Gamma, x) \setminus [e] = [t_1; \ldots; t_n] \\ t_1 : T_1, \ldots, t_n : T_n \in \Gamma \\ \gamma'(\mathcal{P}, x, e) : \Gamma \vdash \Pi t_1 : T_1, \ldots, t_n : T_n.\Phi[t_1; \ldots; t_n]\end{array}}{\mathcal{P} : \Gamma \vdash \Phi[t_1; \ldots; t_n]}$$

Injectivity and discrimination

When considering a hypothesis equating (possibly applied) constructors, there are two possible cases: if the two constructors are different, then the hypothesis is inconsistent, making the goal vacuously true

$$\frac{\begin{array}{c}\varpi(\mathcal{P}, e) : \emptyset \\ k_1 \text{ and } k_2 \text{ are different constructors}\end{array}}{\mathcal{P} : \Gamma, e : k_1\ \overline{t} = k_2\ \overline{u}, \Delta \vdash \Phi}$$

If on the other hand the terms in the equation being considered consist of the same (possibly applied) constructor, then an injectivity rule holds:

$$\frac{\varpi(\mathcal{P}, e) : \Gamma, e : k\ \overline{t_n} = k\ \overline{u_n}, \Delta \vdash \forall \overline{e_n} : \overline{t_n} = \overline{u_n}.\Phi}{\mathcal{P} : \Gamma, e : k\ \overline{t_n} = k\ \overline{u_n}, \Delta \vdash \Phi}$$

where $\overline{e_n} : \overline{t_n} = \overline{u_n}$ is a Leibniz telescope of equations. In particular, each term t_i appears as the argument of a rewriting principle involving the previous $\overline{e_i}$ equations. This marks a major difference with respect to the unification algorithm implemented in [39], where new equations introduced by injectivity are in constructor form. In our setting, instead, new equations contain applications of rewriting principles (what we have called "rewritten constructor form"). The unification procedure will make sure that these rewriting principles disappear before we actually introduce the new equational hypotheses in the context; therefore the shape of the problem is not affected and unification of constructor forms works as expected.

The two seemingly different operations of injectivity and discrimination of conflicts are actually performed by means of the same injectivity/discrimination lemma.

Given an inductive type $\mathcal{I}\ \bar{t}$, its discrimination principle has the following type:

$$
\begin{array}{rcl}
\delta_{\mathcal{I}\ \bar{t}} & \triangleq & \Pi x, y : \mathcal{I}\ \bar{t}.\text{match } x \text{ with} \\
& & [k_0\ \overline{x_{m_0}} \Rightarrow p_0 \\
& & \vdots \\
& & |k_{n-1}\ \overline{x_{m_{n-1}}} \Rightarrow p_{n-1}]
\end{array}
$$

$$
\begin{array}{rcl}
p_i & \triangleq & \text{match } y \text{ with} \\
& & [k_0\ \overline{y_{m_0}} \Rightarrow \Pi\Phi : \text{Type}.\Phi \\
& & \vdots \\
& & |k_{i-1}\ \overline{y_{m_{i-1}}} \Rightarrow \Pi\Phi : \text{Type}.\Phi \\
& & |k_i\ \overline{y_{m_i}} \Rightarrow \Pi\Phi : \text{Type}.(\Pi\overline{e_{m_i}} : \overline{x_{m_i}} = \overline{y_{m_i}}.\Phi) \to \Phi \\
& & |k_{i+1}\ \overline{y_{m_{i+1}}} \Rightarrow \Pi\Phi : \text{Type}.\Phi \\
& & \vdots \\
& & |k_{n-1}\ \overline{y_{m_{n-1}}} \Rightarrow \Pi\Phi : \text{Type}.\Phi]
\end{array}
$$

This principle is stated by means of two nested case analysis operations. Whenever two terms x and y are equal, we analyze both of them: if the number of constructors of $\mathcal{I}$ is n, there are exactly n^2 cases to consider, for each possible combination of constructors used to obtain x and y. We can imagine these cases to be disposed in a square matrix, where the columns correspond to the n possible constructors for x and the rows have the same role with respect to y. If x and y fall on the diagonal, then x is actually $k_i\ \overline{x_{m_i}}$ and y is $k_i\ \overline{y_{m_i}}$, where m_i is the number of arguments of the i-th constructor k_i; then to prove any given goal Φ, we can also assume the telescope of equalities $\overline{e_{m_i}} : \overline{x_{m_i}} = \overline{y_{m_i}}$. If on the other hand by analysing x and y we fall outside the diagonal, then we are in an inconsistent case and can prove any property Φ for free.

Proving $\delta_{\mathcal{I}\ \bar{t}}$ is easy. We just introduce in the context the terms x and y, and the hypothesis equating them:

$$\frac{\begin{array}{rcl} x & : & \mathcal{I}\ \bar{t} \\ y & : & \mathcal{I}\ \bar{t} \\ H & : & x = y \end{array}}{M_{x,y}}$$

where $M_{x,y}$ is an abbreviation for the term composed of the nested case analyses on x and y. Using hypothesis H, we can rewrite y as x. This yields the new goal

$$\frac{\begin{array}{rcl} x & : & \mathcal{I}\ \bar{t} \\ y & : & \mathcal{I}\ \bar{t} \\ H & : & x = y \end{array}}{M_{x,x}}$$

where we only need to prove the properties on the diagonal, having already discarded all the inconsistent cases for free. Then we perform case analysis on x: we get a different subgoal for each possible constructor. Each of the subgoals will be in the form

$$\frac{\begin{array}{rcl} x & : & \mathcal{I}\ \bar{t} \\ y & : & \mathcal{I}\ \bar{t} \\ H & : & x = y \\ x_1 & : & T_1 \\ & \vdots & \\ x_{m_i} & : & T_{m_i} \\ \Phi & : & \text{Type} \\ H_1 & : & \Pi \overline{e_{m_i}} : \overline{x_{m_i}} = \overline{x_{m_i}}.\Phi \end{array}}{\Phi}$$

However this is almost trivial: it is sufficient to instantiate each e_i with the identity on the corresponding term $\mathcal{R}_{x_i}$. Notice that the notation for telescopes hides the fact that the i-th equation should have the form

$$e_i : \rho_i[\ldots; \overline{x_{i+1}}; \overline{x_i}; e_i] = x_i$$

However, assuming that the previous $i - 1$ equations have been instantiated with identities, this reduces to

$$e_i : x_i = x_i$$

which can be proved by means of $\mathcal{R}_{x_i}$ too.

Introduction of hypotheses

The last operation used in the unification algorithm is introduction of new hypotheses, which at the user level is performed by the # tactic. This operation allows us to introduce in the context:

- old hypotheses that have been generalized by a cascade generalization and then rewritten;
- new equational hypotheses resulting from an injectivity step.

$$\frac{\vartheta(\mathcal{P}, y) : \Gamma, y : T \vdash \Phi[y]}{\mathcal{P} : \Gamma \vdash \Pi x : T.\Phi[x]}$$

3.4.2 Analysis of the equations

Given any equation, the `destruct` tactic must first of all understand in which of the cases for the unification algorithm it falls. Suppose that the equation being considered is in the form $t = u$; the analysis is performed as follows:

1. First, perform a convertibility check on t and u: if $t \cong u$, then the equation is really an identity and it should be cleared using the κ operation;
2. Else, if t and u are both constructor applications, check if the constructors being applied are the same; in this case, the equation is an injectivity case; otherwise, it is a conflict case; in both cases, we will use the ϖ operation;
3. Else, at least one of the two terms is a variable; perform an occur check in the other term: if the variable does not occur in the term, use operation ς; otherwise the cyclic equation should be discriminated: in this unimplemented case we will explicitly ask for user intervention.

3.4.3 The unification algorithm

The unification algorithm implemented by `destruct` searches for equations to be unified, scanning the context from the first element to the last one. The procedure considers at each step the proof problem $\mathcal{P}$, the number of hypotheses that have been added to the thesis, as a result of a cascade generalization or an injectivity step (`nprods`), and also a list of names of hypotheses that have already been considered by the unification algorithm, but not yet removed from the context (`acc`). At each step, the set of the active equations includes all the equations appearing either in the context or in the first `nprods` abstractions of the goal, minus the inactive equations referenced in `acc`.

The algorithm proceeds as follows:

1. If the context contains at least one equation, we scan the context and select the first active equation e. The lefthand side is reduced to weak head normal form (to put it in regular constructor form in case it is still in rewritten constructor form); then we analyze the shape of the equation. The following steps depend on the result of the analysis:

   (a) in the case of injectivity or conflict, we apply the discrimination principle; if it was injectivity, then we increment `nprods` by the number h of new equations introduced by that operation, we add e to `acc` and finally we iterate the procedure

   $$\texttt{destruct}(\mathcal{P}, \texttt{nprods}, \texttt{acc}) = \texttt{destruct}(\varpi(\mathcal{P}, e), \texttt{nprods} + h, e :: \texttt{acc})$$

   In the case of conflict, we terminate the procedure after closing the goal:

   $$\texttt{destruct}(\mathcal{P}, \texttt{nprods}, \texttt{acc}) = \varpi(\mathcal{P}, e)$$

   (b) in the case of a substitution, we perform a cascade generalization of all hypotheses depending on the variable x being substituted, except e, and

then use the rewriting step we defined earlier; finally we iterate the procedure, after adding to `nprods` the number h of hypotheses being generalized:

$$\texttt{destruct}(\mathcal{P}, \texttt{nprods}, \texttt{acc}) = \texttt{destruct}(\varsigma(\gamma'(\mathcal{P}, x, e), e), \texttt{nprods} + h, \texttt{acc})$$

(c) in the case of an identity, we perform a cascade generalization of all hypotheses depending on equation e, we also generalize e and finally use the operation κ and iterate the tactic

$$\texttt{destruct}(\mathcal{P}, \texttt{nprods}, \texttt{acc}) = \texttt{destruct}(\kappa(\gamma_0(\gamma(\mathcal{P}, e), e)), \texttt{nprods}, \texttt{acc})$$

2. if there are no more equations and $\texttt{nprods} > 0$, we try introducing an hypothesis from the thesis, decrement `nprods` and iterate the tactic

$$\texttt{destruct}(\mathcal{P}, \texttt{nprods}, \texttt{acc}) = \texttt{destruct}(\vartheta(\mathcal{P}, x), \texttt{nprods} - 1, \texttt{acc})$$

3. otherwise, we stop.

3.4.4 Proof of termination

We will now prove that the algorithm always terminates, by means of the classical argument using lexicographic ordering on a properly defined tuple. This proof is similar to the one given by McBride: the only complication comes from the fact that since our algorithm considers not only equations appearing in the context, but also some appearing in the goal.

Because of this peculiarity we will consider the set of all active equations in the problem (according to the definition given a few lines before) appearing in constructor form (either regular or rewritten).

We will then consider the size of the current problem as defined by a quadruple

$$\langle a, b, c, d \rangle$$

where

- a is the number of variables in the problem
- b is the number of constructor symbols appearing in the equations
- c is the number of active equations in the problem
- d is equal to the current value of `nprods`

Theorem 3.3 *For all constructor form unification problems, the sequence of transition rule applications determined at each stage by the leading equation is finite.*

Proof: We consider the various cases of the algorithm separately, showing that each of them makes the size of the problem shrink:

- in case 1(a), we apply the injection/discrimination principle, which either closes the current goal, or produces a new goal adding the current equation to `acc`, thus making it inactive; in the first subcase, the algorithm terminates immediately; in the second subcase, b shrinks, while a is not incremented;
- in case 1(b), substituting all occurrences of a variable decrements a by 1;
- in case 1(c), the number of active equations (c) is decremented by 1, while a and b do not grow;
- in case 2, we decrement d, and all other values stay the same;
- case 3 stops immediately.

□

The `nprods` parameter is not strictly needed by the algorithm, but is used to ensure that the user, after applying the tactic, is presented with a familiar context. In particular, we do not want to introduce more hypotheses than those strictly needed to unify equalities in the context: if the original thesis was a product, possibly containing more equational hypotheses, these should not be introduced in the context. The check `nprods` > 0 at point 3 of the algorithm, ensures that the introduction step cannot fire in such a situation.

3.4.5 Advantages of our implementation

We have introduced an algorithm for performing unification of constructor forms that is inspired by McBride's simplify tactic, but differs from it in two respects:

1. it allows the user to use Leibniz equality instead of John Major equality;
2. it considers the case where an equation is mentioned in another hypothesis or in the goal, and explicits a cascade generalization operation to manage the added complexity.

The possibility of using Leibniz equality does not imply that in general we are able to perform unification without assuming axioms (like Streicher's K). However, while John Major equality needs axioms for both rewriting and clearing of identities, the rewriting principles of Leibniz equality (including simultaneous rewriting principles) are definable in plain CIC. This means that we are able to prove more properties without resorting to axioms. Consider for example the following goal, containing an equality on dependent pairs:

$$\frac{\begin{array}{rcl} m & : & \mathbb{N} \\ n & : & \mathbb{N} \\ v & : & vec\ m \\ w & : & vec\ n \\ P & : & \forall x_1, x_2 : \mathbb{N}.\forall y_1 : vec\ x_1.\forall y_2 : vec\ x_2.\mathrm{Prop} \\ e & : & \langle m, v\rangle =_{\Sigma x:\mathbb{N}.vec\ x} \langle n, w\rangle \end{array}}{P\ m\ n\ v\ w}$$

The algorithm will:

1. Perform an injectivity step on equation e, producing a new goal

$$\forall e_1 : m = n.\forall e_2 : \rho_1[\mathbb{N}, vec, m, n, v, e_1] = w.P\ m\ n\ v\ w$$

2. Introduce a new hypothesis $e_1 : m = n$ in the context

3. Replace m with n (by means of hypothesis e_1) after generalizing hypothesis v, yielding the goal

$$\forall v : vec\ n.\forall e_2 : \rho_1[\mathbb{N}, vec, n, n, v, \mathcal{R}_n] = w.P\ n\ n\ v\ w$$

which reduces to

$$\forall v : vec\ n.\forall e_2 : v = w.P\ n\ n\ v\ w$$

4. Introduce the hypotheses $v : vec\ n$ and $e_2 : v = w$.

5. Rewrite with e_2 to obtain the final goal

$$P\ n\ n\ w\ w$$

Notice how we managed to avoid the use of axioms: if we used John Major equality, all the rewriting steps would resort to axioms.

Furthermore, since the only axiom we need to deal in general with Leibniz equations is Streicher's K property, the user can easily provide specialized, provable versions of K for decidable types. This is especially useful inside the definition of dependently typed algorithms, since axioms are opaque components that can block the evaluation of a function. If we used John Major equality, we woul also need to derive a rewriting principle from the specialized version of K.

The point concerning dependent occurrences of equations in other hypotheses or in the goal is more subtle. Clearly the ability to deal with this situation is more critical in the case of Leibniz equalities, because Leibniz telescopes are built precisely employing references to other equations, as opposed to John Major telescopes – in fact, McBride's algorithm never introduces this kind of dependencies. However, it is still possible that the dependencies are already present in the original goal, before performing unification. This is possible even in very simple cases: consider for example this program definition in Coq:

```
Program Definition tail (n : nat) (v : Vec (S n)) :
  { w : Vec n | exists m, JMeq v (vcons n m w) } :=
```

```
match v with
| vnil => _
| vcons m k w => w
end.
```

The definition opens, among other, a proof obligation

$$\begin{array}{rcl} n & : & \mathbb{N} \\ v & : & vec\ (S\ n) \\ m & : & \mathbb{N} \\ k & : & \mathbb{N} \\ w & : & vec\ m \\ H_1 & : & S\ m = S\ n \\ H_2 & : & vcons\ m\ k\ w \simeq v \end{array}$$
$$\hline$$
$$\exists m_0 : \mathbb{N}.v \simeq vcons\ n\ m0\ (\rho_1[\mathbb{N}, vec, m, n, w, (H^*\ n\ v\ m\ k\ w\ H_1\ H_2)])$$

where H^* is a proof of

$$\forall n : \mathbb{N}.\forall v : vec\ (S\ n).\forall m, n : \mathbb{N}.\forall w : vec\ m.S\ m = S\ n \Rightarrow vcons\ m\ k\ w = v \Rightarrow m = n$$

Clearly, H_1 and H_2 occur in the goal, needing, for the purposes of unification, an algorithm capable of performing cascade generalizations, like the one we presented in this chapter.

Chapter 4

Inversion revisited

In this chapter, we report about our development of the inversion tactic for Matita. The implementation follows the style made popular by McBride in [40], but is also more flexible, supporting a novel notion of mixed induction/inversion principles. The structure of the chapter is as follows: the first section introduces the notion of inversion of an inductive predicate; in section 2 and section 3 we describe respectively the definition of inversion principles and their extension to induction/inversion; finally section 4 discusses the implementation.

4.1 Backwards reasoning

In mathematics, and particularly in logic and computer science, it is common to define predicates by means of inductive deduction rules: when we do this, we assert that any proof of such a predicate can be obtained only by combining a finite number of copies of the deduction rules in a legal way. In CIC, and consequentially in Matita, it is possible to model this notion by means of a proper inductive type: conforming to the Curry-Howard correspondence, the inductive type encodes the predicate, and terms of that inductive type represent its possible proofs; each deduction rule corresponds to one and only one constructor of the inductive type, and vice-versa. Inductive types also match the requirement of finiteness of the proofs: each term of an inductive type must be obtained by combining its constructors a finite number of times.

When a predicate is defined by means of deduction rules, we expect that its proof may be inspected, in such a way that reasoning on the rules used to derive it should be possible. For example, let us assume that the "is even" predicate is defined using two deduction rules: the first one, which we call Even-O, has no hypothesis and states that 0 is an even number; the second one, called Even-SS, says that applying the successor function to a given natural number n, we obtain an even number $S\ (S\ n)$, provided that we already proved that n is even. While this definition might seem somewhat unnatural (mathematicians would usually define even numbers as those that can be obtained multiplying some natural number by

2), it will serve as our toy example in the rest of this section. The two rules and their representation in Matita as an inductive type are shown in Figure 4.1.

There are a small amount of proof techniques based on this principle that we call generically *backwards reasoning*: they include case analysis on the last rule used in the proof tree and, more importantly, structural induction on the proof tree.

4.1.1 Case analysis

In CIC, case analysis has a direct representation as the **match** statement, used to decompose an inductive term, analyzing the constructors used to obtain it. Matita (not unlike Coq) provides an interface to it as the cases tactic. Suppose the user is facing the goal

$$\begin{array}{rcl} n & : & \text{nat} \\ H_1 & : & n \text{ is even} \\ \hline & P\ n & \end{array}$$

where P is some unary property on natural numbers. Using case analysis on the hypothesis H_1, the user can reason on the last rule used in its derivation, thus reducing the proof to two supposedly simpler subgoals (one for each of the deduction rules for the "is even" predicate): if H_1 was obtained by means of EVEN-O, then n must be 0, and our goal becomes $P\ 0$:

$$\begin{array}{rcl} n & : & \text{nat} \\ H_1 & : & n \text{ is even} \\ \hline & P\ O & \end{array}$$

If on the other hand rule EVEN-SS was used, than we know that n is really $S\ (S\ m)$ for some m, and that m is also even, corresponding to the new goal

$$\frac{}{0 \text{ is even}}\ (\text{EVEN-O})$$

$$\frac{n \text{ is even}}{S\ (S\ n) \text{ is even}}\ (\text{EVEN-SS})$$

```
inductive even : nat → Prop :=
| even_O  : even O
| even_SS : ∀n:nat.even n → even (S (S n)).
```

Figure 4.1: Definining an inductive predicate in Matita

$$\begin{array}{lcl} n & : & \text{nat} \\ H_1 & : & n \text{ is even} \\ m & : & \text{nat} \\ H_2 & : & m \text{ is even} \\ \hline & P\ (S\ (S\ m)) & \end{array}$$

4.1.2 Structural induction

The previous is sufficient to show that very simple properties hold: for example, we could prove that n, being even, is either 0 or strictly greater than 1 (each possibility being immediate consequences of one of the deduction rules of "is even"). However, most interesting properties require, to be proved, more sophisticated backwards reasoning, like the one provided by structural induction.

Structural induction is similar to case analysis in the sense that it also opens a subgoal for each of the possible constructors of the term we are decomposing. However it also offers additional *induction hypotheses* on each of its "immediate subterms" of the same inductive type. For example, it is possible to prove that every even number is the double of some natural number by structural induction: we get two subgoals

$$\begin{array}{lcl} n & : & \text{nat} \\ H_1 & : & n \text{ is even} \\ \hline & \exists p.0 = p * 2 & \end{array}$$

$$\begin{array}{lcl} n & : & \text{nat} \\ H_1 & : & n \text{ is even} \\ m & : & \text{nat} \\ H_2 & : & m \text{ is even} \\ IH & : & \exists q.m = q * 2 \\ \hline & \exists p.S\ (S\ m) = p * 2 & \end{array}$$

In the second subgoal, since H_2 is an immediate subproof of the hypothesis H_1 on which we are performing induction, we get an associated induction hypothesis IH. By IH, we can rewrite $S\ (S\ m)$ as $S\ (S\ (q * 2))$, which is equal to $(S\ q) * 2$ up to a computation step[1]. Therefore, to prove the goal it is sufficient to take $p = S\ q$.

[1] Assuming a proper definition of multiplication as a recursive function.

CIC provides induction principles for all inductive types: depending on the version of the calculus, they can be either a primitive operation (just like the **match** expressions) or proved in terms of case analysis and terminating recursion. Even though Matita follows the latter style, induction principles are proved automatically by the system, so that we can actually consider them as primitive operations. They are applied by the **elim** tactic.

4.1.3 Inversion

At first sight, the previous two tactics seem to accomodate most of the informal proof techniques involving inductively defined predicates. However, the above approaches fail in slightly more involved scenarios. Consider for example the very simple goal

$$\begin{array}{lcl} n & : & \text{nat} \\ H_1 & : & S\ n \text{ is even} \\ \hline & n > 0 & \end{array}$$

Apparently, case analysis on H_1 is sufficient to make the proof trivial: in fact, the proof of "$S\ n$ is even" can only be obtained by means of the Even-SS rule (Even-O would never work, since 0 is not the successor of any natural number); then n is equal to $S\ m$ for some natural number m, and we only need to prove that $S\ m > 0$. However, the cases tactic yields two new subgoals

$$\begin{array}{lcl} n & : & \text{nat} \\ H_1 & : & n \text{ is even} \\ \hline & n > 0 & \end{array} \qquad \begin{array}{lcl} n & : & \text{nat} \\ H_1 & : & n \text{ is even} \\ m & : & \text{nat} \\ H_2 & : & m \text{ is even} \\ \hline & n > 0 & \end{array}$$

failing to meet logical intuition for two reasons:

- It considers the subcase for $S\ n$ equal 0, even though this is inconsistent, but does not provide any new, manifestly contradictory hypothesis, not allowing to discard this impossible case

- While case analysis identifies $S\ n$ with terms 0 and $S\ (S\ m)$ (depending on the goal), this identification is only implicit: the system provides no explicit hypothesis stating that $S\ n$ must be equal to another term: therefore we are left with the same $n > 0$ goal to prove, without knowing (explicitly) any new information concerning n.

Using induction rather than case analysis yields a similar result.

The reason behind this behaviour stands in the way case analysis is implemented. Proving a theorem under a set of hypotheses amounts to instantiating a metavariable with a term of the type corresponding to the statement of the theorem, in the typing context defined by the hypotheses. The proof problem can then be described as the typing judgment

$$n : \text{nat}, H_1 : S\ n \text{ is even} \vdash ?_1 : n > 0$$

where $?_1$ is a metavariable, representing the proof the user still has to fill in. Performing case anlysis on H_1 is equivalent to instantiating $?_1$ with the following **match** statement on H_1

match H_1 in even return $?_2$ **with**
[even_O ⇒ $?_3$
| even_SS (m:nat) (H_2:even m) ⇒ $?_4$]

where, by the typing discipline of **match** statements, $?_2\ (S\ n)\ H_1$ should be convertible with $n > 0$: this means that the unification engine of the theorem prover should synthesize a function that, when applied to $S\ n$ and H_1, returns the type $n > 0$. This is a higher order unification problem and as such it has no most general unifier. Only in particularly fortunate cases (like the ones in the previous examples) the system is able to infer a decent value for $?_2$; since however this time neither $S\ n$ nor H_1 match a subterm of $n > 0$, Matita instantiates $?_2$ with $\lambda x_1, x_2.n > 0$. This yields new subgoals for $?_3$ and $?_4$

$$n : \text{nat}, H_1 : S\ n \text{ is even} \vdash ?_3 :?_2\ 0\ \text{even\_O}$$
$$n : \text{nat}, H_1 : S\ n \text{ is even}, m : \text{nat}, H_2 : m \text{ is even} \vdash ?_4 :?_2\ (S\ (S\ m))\ (\text{even\_SS}\ m)$$

or, after substituting $?_2$

$$n : \text{nat}, H_1 : S\ n \text{ is even} \vdash ?_3 : n > 0$$
$$n : \text{nat}, H_1 : S\ n \text{ is even}, m : \text{nat}, H_2 : m \text{ is even} \vdash ?_4 : n > 0$$

It is possible to overcome this limitation. One possibility is to prove a separate *inversion lemma* for all inductive predicates. In the case of "is even", the lemma states:

$$\Pi x : \text{nat}.x \text{ is even} \rightarrow x = 0 \vee \exists y.x = S\ (S\ y) \wedge y \text{ is even}$$

Inversion lemmata can be more easily proved by case analysis, and then used to perform backwards reasoning in the more general case. If we apply this inversion lemma, instead of the cases tactic, we still have to prove $n > 0$ in two subgoals: however, in the first goal we will have an explicit hypothesis saying $S\ n = 0$, and in the second one, a similar one stating that $S\ n = S\ (S\ m)$.

Not only is this a general technique, but it can even be automatized, as shown by Cornes and Terrasse in Coq [15]. In [40], McBride suggested that the statement of inversion lemmata should be formulated in a different way, reminiscent of induction principles. The rest of the chapter discusses our implementation and extension of inversion lemmata in the style of McBride.

4.2 Proving inversion principles

Suppose we are given the following inductive type:

inductive $\mathrm{I} : \forall\, x_1 : T_1, \ldots, x_n : T_n.\sigma :=$
$\mid\ c_1 : \forall\, y_1^1 : U_1^1, \ldots, y_{k_1}^1 : U_{k_1}^1.I\ t_1^1 \cdots t_n^1$
...
$\mid\ c_m : \forall\, y_1^m : U_1^m, \ldots, y_{k_m}^m : U_{k_m}^m.I\ t_1^m \cdots t_n^m$

We want to prove an inversion lemma for I with the following statement:

$$
\begin{array}{l}
\Pi x_1 : T_1, \ldots, x_n : T_n. \\
\Pi P : \forall z_1 : T_1, \ldots, z_n : T_n.\tau. \\
\quad (\forall y_1^1 : U_1^1, \ldots, y_{k_1}^1 : U_{k_1}^1.x_1 = t_1^1 \to \ldots \to x_n = t_n^1 \to P\ t_1^1 \cdots t_n^1) \to \\
\quad \vdots \\
\quad (\forall y_1^m : U_1^m, \ldots, y_{k_m}^m : U_{k_m}^m.x_1 = t_1^m \to \ldots \to x_n = t_n^m \to P\ t_1^m \cdots t_n^m) \to \\
\qquad I\ x_1 \cdots x_n \to P\ x_1 \cdots x_n
\end{array}
$$

Basically, this provides a case analysis operation enriched with an equation for each right parameter of the inductive type I. Unsurprisingly, the lemma can be proved with a clever use of a match expression:

$$
\begin{array}{l}
\lambda x_1, \ldots, x_n.\lambda P. \\
\lambda H_1, \ldots, H_m.\lambda t. \\
\quad \text{match } t \text{ in } I \text{ return } \lambda z_1, \ldots, z_n, w.x_1 = z_1 \to \ldots x_n = z_n \to P\ z_1 \cdots z_n \\
\quad \text{with} \\
\qquad [c_1\ y_1^1 \cdots y_{k_1}^1 \Rightarrow H_1\ y_1^1 \cdots y_{k_1}^1 \\
\qquad \vdots \\
\qquad |c_m\ y_1^m \cdots y_{k_m}^m \Rightarrow H_m\ y_1^m \cdots y_{k_m}^m]
\end{array}
$$

4.3 Mixing induction and inversion

In informal mathematics, structural induction and case analysis on the final rule of a proof tree are often usend conjunctly. However as we will see in a few moments, at least in the Calculus of (Co)Inductive Constructions, structural induction does not allow to perform at once case analysis on the final rule, unless we give up on using the induction hypotheses. In this section we present the unusual technique of *induction/inversion*, which in some cases can be used to justify the informal notion of structural induction.

We start recalling the rule for induction over indexed inductive definitions. An indexed inductive definition is similar to an inductive type, but it defines at once a set of mutually-recursive inductive types differing by the values of some indices.

Syntactically, the declaration of an inductive family is isomorphic to the declaration of a judgement by giving its introduction rules. The family parameters are the arguments of the judgement. The derivation rules are the constructors of the inductive family. Positivity conditions must be satisfied by the derivation rules to ensure existence of the least fixpoint solution of the set of recursive rules. When the judgement is 0-ary (i.e. it has no parameters), we obtain a simple inductive definition. In this case, the conclusion of all derivation rules is simply the name of the inductive type being defined, providing no information.

Once an inductive family $\mathcal{I}$ is declared by giving its derivation rules (its constructors), we obtain for free recursion over the inductive family as the elimination principle corresponding to the introduction rules.[2] We briefly recall the typing judgement of induction principles for arbitrary inductive families: our syntax is similar to the one given by [69] up to some minor differences.

Assume an inductive type $\mathcal{I}$ of arity $\forall\overline{x_n} : \overline{T_n}.\sigma$, where σ is a sort. Suppose that P is a predicate of type $\forall\overline{x_n} : \overline{T_n}.\mathcal{I}\ \overline{x_n} \to \tau$, where τ is a sort, and t has type $\mathcal{I}\ \overline{u_n}$ for some (properly typed) terms $\overline{u_n}$. The application of the proper induction principle on t to prove $P\ \overline{u_n}\ t$ is written

$$\mathcal{E}^{\tau}_{\mathcal{I}}(\overline{u_n}, t, P)\{\overline{f_m}\}$$

where $\overline{f_m}$ are the proof terms for each of the m sub-cases of the induction (one for each of the constructors of $\mathcal{I}$). The expected type for the $\overline{f_m}$ is computed by the following definition:

Definition 4.1 *Let Γ be a CIC context, c, T, Q CIC terms. The operators $\Delta^Q\{\Gamma; c : T\}$ and $\Theta^Q\{\Gamma; T\}$ are defined as follows:*

$$\begin{array}{lcl}
\Delta^Q\{\Gamma; c : \mathcal{I}\ \bar{t}\} & \equiv & \Theta^Q\{\Gamma; Q\ \bar{t}\ c\} \\
\Delta^Q\{\Gamma; c : \forall x : T.U\} & \equiv & \forall x : T.\Delta^Q\{\Gamma, x : T; c\ x : U\} \\
\textit{otherwise undefined} & &
\end{array}$$

[2] Actually, in Matita elimination principles are not primitive, but are defined by means of well-founded recursion; this definition, however, is fully automated.

$$
\begin{array}{lcl}
\Theta^Q\{\emptyset; T\} & \equiv & T \\
\Theta^Q\{\Gamma, x : \forall \overline{y} : \overline{V}.\mathcal{I}\ \overline{t}; T\} & \equiv & \Theta^Q\{\Gamma; \forall \overline{y} : \overline{V}.Q\ \overline{t}\ (x\ \overline{y}) \to T\} \\
\Theta^Q\{\Gamma, x : U; T\} & \equiv & \Theta^Q\{\Gamma; T\} \textit{ if the head of } U \textit{ is not } \mathcal{I} \\
\textit{otherwise undefined} &&
\end{array}
$$

Let $k_i^{\mathcal{I}}$ of type $K_i^{\mathcal{I}}$ $(i = 1, \ldots, m)$ be the constructors of type $\mathcal{I}$. Then we can write the typing rule for the induction principle as follows:

$$
\frac{\begin{array}{c}
\Gamma \vdash \mathcal{I} : \forall \overline{x_n} : \overline{T_n}.\sigma \\
\text{for all } i = 0, \ldots, n-1\text{: } \Gamma \vdash u_i : T_i\{{}^{u_0,\ldots,u_{i-1}}/_{x_0,\ldots,x_{i-1}}\} \\
\Gamma \vdash t : \mathcal{I}\ \overline{u_n} \qquad \Gamma \vdash P : \forall \overline{x_n} : \overline{T_n}.\mathcal{I}\ \overline{x_n} \to \tau \\
\text{for all } j = 0, \ldots, m-1\text{: } \Gamma \vdash f_j : \Delta^P\{\emptyset; k_j : K_j\} \\
\text{elimination of } \mathcal{I} \text{ towards sort } \tau \text{ is allowed}^3
\end{array}}{\Gamma \vdash \mathcal{E}_{\mathcal{I}}^{\tau}(\overline{u_n}, t, P)\{\overline{f_m}\} : P\ \overline{u_n}\ t} \qquad \text{(Elim)}
$$

As we discussed earlier, induction principles for indexed inductive definitions are not well-suited for immediate applications to hypotheses in which the family parameters are instantiated with anything but variables. Applying the induction principle to some premise, we are left with a case for every constructor, disregarding the fact that only some of them could have been applied in this case. Moreover, they are exactly the same cases we would obtain by changing the indices to any other expression keeping the hypothesis well-typed. Inversion is the (derived) proof principle we need in these cases.

Inversion allows to invert derivation rules by replacing in a hypothesis a judgement with a disjunction of all the ways in which it can be obtained. Operationally, it is sufficient to perform first-order unification of the hypothesis with the conclusion of every derivation rule and, in case of success, augment the conjunction of the premises of the derivation rules with the equalities imposed by the unifier.

[3]The condition on allowed sort eliminations is not relevant to the subject of this paper; the interested reader can find more information in [69] (for a general account of elimination in CIC) and [4] (for the actual type system implemented in Matita).

$$\frac{}{0 \leq 0}\ (\textsc{Lesseq}\text{-O})$$

$$\frac{m \leq n}{m \leq S\ n}\ (\textsc{Lesseq}\text{-S})$$

$$\frac{m \leq n}{S\ m \leq S\ n}\ (\textsc{Lesseq}\text{-SS})$$

```
inductive lesseq : nat → nat → Prop :=
| lesseq_O  :
  lesseq O O
| lesseq_S  :
  ∀ m,n:nat.lesseq m n → lesseq m (S n)
| lesseq_SS :
  ∀ m,n:nat.lesseq m n → lesseq (S m) (S n).
```

Figure 4.2: Example: a definition of less-or-equal

Usually, in pen & paper proofs, it is inversion, and not induction, that is used in presence of judgements. The problem with inversion is that it does not provide inductive hypotheses over the new premises. Thus, most of the time, inversion on a judgement follows induction on the arguments of the judgement. For instance, as we will see in Chapter 5, the specification of POPLmark proves transitivity for $F_{<:}$ by induction over types followed by "induction with case analysis" (apparently similar to inversion) on the typing judgment. Note, however, that the similarity may not be correct since inversion does not provide access to an "inner inductive hypothesis".

To provide here an example of reasoning requiring mixed induction and inversion, we will use a specially crafted definition of the $\leq$ relation on natural numbers, that is given in Figure 4.2. We give an informal proof that this definition of $\leq$ is transitive.

Theorem 4.1 *If $m \leq n$ and $n \leq p$, then $m \leq p$.*

Proof: Assume $m \leq n$: we want to prove that for all p, $n \leq p$ implies $m \leq p$. By structural induction on the derivation of $m \leq n$, we have three cases:

- Case Lesseq-O: we have $m = n = 0$. The thesis becomes $\forall p.p \leq 0 \Rightarrow p \leq 0$, which is trivial.

- Case Lesseq-S: for some n', we have $n = S\ n'$ and $m \leq n'$. By induction hypothesis, we know that for all q, $n' \leq q$ implies $m \leq q$. The thesis becomes $\forall p.S\ n' \leq p \Rightarrow m \leq p$. Assume $S\ n' \leq p$: we perform inner induction on its derivation, with case analysis on the last rule used, which yields three subcases:

  - Subcase LESSEQ-O is vacuous since the conclusion of this rule does not match $S\ n' \leq p$.
  - Subcase LESSEQ-S: for some p', we have $p = S\ p'$ and $S\ n' \leq p'$. By the inner induction hypothesis, $m \leq p'$; therefore, by rule LESSEQ-S, $m \leq S\ p' = p$, as needed.
  - Subcase LESSEQ-SS: for some p', we have $p = S\ p'$ and $n' \leq p'$. By the outer induction hypothesis, after choosing $q := p'$, we get $m \leq p'$. Then by rule LESSEQ-S, $m \leq S\ p' = p$, as needed.

- Case LESSEQ-SS: for some m', n', we have $m = S\ m'$, $n = S\ n'$ and $m' \leq n'$. By induction hypothesis, we know that for all q, $n' \leq q$ implies $m' \leq q$. The thesis becomes $\forall p. S\ n' \leq p \Rightarrow S\ m' \leq p$. Assume $S\ n' \leq p$: we perform inner induction on its derivation, with case analysis on the last rule used, which yields three subcases:
  - Subcase LESSEQ-O is vacuous since the conclusion of this rule does not match $S\ n' \leq p$.
  - Subcase LESSEQ-S: for some p', we have $p = S\ p'$ and $S\ n' \leq p'$. By the inner induction hypothesis, $S\ m' \leq p'$; therefore, by rule LESSEQ-S, $S\ m \leq S\ p' = p$, as needed.
  - Subcase LESSEQ-SS: for some p', we have $p = S\ p'$ and $n' \leq p'$. By the outer induction hypothesis, after choosing $q := p'$, we get $m' \leq p'$. Then by rule LESSEQ-SS, $S\ m' \leq S\ p' = p$, as needed.

□

The proof is informal because we have not clarified what is meant by "inner induction with case analysis on the last rule", even though it seems relatively natural. It is not regular induction, because that would not allow us to discriminate inconsistent subcases (as we did in the two occurrences beginning with "Subcase LESSEQ-O"); and it is not regular inversion, because it does not provide an inner induction hypothesis.

Now suppose for a moment that we can prove the following principle:

Proposition 4.2 *Let P be a binary predicate on natural numbers. For all natural numbers x_1, x_2, $x_1 \leq x_2$ implies $P\ x_1\ x_2$ provided that the following properties hold:*

- $x_1 = 0 \Rightarrow P\ 0\ 0$;
- $\forall m, n : \mathbb{N}. m \leq n \Rightarrow \underline{(x_1 = m \Rightarrow P\ m\ n)} \Rightarrow x_1 = m \Rightarrow P\ m\ (S\ n)$;
- $\forall m, n : \mathbb{N}. m \leq n \Rightarrow (x_1 = m \Rightarrow P\ m\ n) \Rightarrow x_1 = S\ m \Rightarrow P\ (S\ m)\ (S\ n)$.

The statement resembles inversion, except that we have equations for the first argument of $\leq$, but not for the second. Apart from that, there is a notable difference: in the second branch of the principle, we have a usable induction hypothesis (which we underlined to make it more visible). The third branch still has an unusable induction hypothesis, because it requires x_1 to be equal to m, even though we know that it is equal to $S\ m$.

We call this kind of hybrid inversion rule, which also provides some induction hypotheses, an induction/inversion principle. Notice that this principle is exactly what we need to justify the inner induction with case analysis of the previous example, since the inner induction hypothesis is only used in the second branch of the induction, and the equation $(x_1 = 0)$ provided by the first branch is sufficient to discriminate the inconsistent case.

We will explain the theory of induction/inversion in the next section. For the moment, we just want to point out that the third branch of the principle still does not provide an accessible induction hypothesis, ultimately because the first argument of $\leq$ is not the same in both the premise and the conclusion of rule Lesseq-SS.

4.3.1 Defining induction/inversion principles

Consider how regular inversion could be proved if case analysis were replaced by induction.

Given a predicate $P : \forall \overline{z_n} : \overline{T_n}.\mathcal{I}\ \overline{z_n} \to \sigma$ and a vector of properly typed variables $\overline{x_n}$, we define the augmented predicate

$$\widehat{P}[\overline{x_n}] \triangleq \lambda \overline{z_n} : \overline{T_n}.\lambda z : \mathcal{I}\ \overline{z_n}.x_0 = z_0 \to \ldots \to x_{n-1} = z_{n-1} \to P\ \overline{z_n}\ z$$

Depending on the actual arity of $\mathcal{I}$, the well typedness of $\widehat{P}$ might depend on the definition of $=$. In the general discussion of inversion and induction/inversion principles, we will assume that $=$ is John Major's equality: under this assumption, $\widehat{P}$ is always well typed.

It is possible to prove the inversion principle applying the regular induction principle for $\mathcal{I}$ to the augmented predicate, as follows

$$\begin{array}{rcl} \widehat{\mathcal{E}}^{\tau}_{\mathcal{I}} & \triangleq & \lambda P, \overline{x_n}, x, \overline{H_m}.\mathcal{E}^{\tau}_{\mathcal{I}}(\overline{x_n}, x, \widehat{P}[\overline{x_n}])\{\overline{f_m}\}\ \mathcal{R}_{x_0} \cdots \mathcal{R}_{x_{n-1}} \\ & : & \forall P : (\forall \overline{z_n} : \overline{T_n[\overline{z_n}]}.\mathcal{I}\ \overline{z_n} \to \sigma). \\ & & \forall \overline{x_n} : \overline{T_n[\overline{x_n}]}.\forall x : \mathcal{I}\ \overline{x_n}.\forall \overline{f_m} : \overline{H_m[\overline{x_n}]}.P\ \overline{x_n}\ x \end{array}$$

where the actual shape of the $\overline{H_i[\overline{x_n}]}$ is the one required by the induction principle and $\mathcal{R}_t$ is the trivial reflexivity proof of $t = t$.

Now, let's take a look at the types $\overline{H_m[\overline{x_n}]}$, representing the subgoals we will have to fill in after applying such a principle. As an example, suppose that $\mathcal{I}$ and its i-th constructor k_i have the following concrete types:

$$\begin{array}{c} \mathcal{I} : \mathbb{N} \to \mathrm{Prop} \\ k_i : \mathcal{I}\ 0 \to \mathcal{I}\ 1 \end{array}$$

then we have

$$\begin{array}{rcl} H_i[x_0] & = & \Pi y : \mathcal{I}\ 0.\widehat{P}[x_0]\ 0\ y \to \widehat{P}[x_0]\ 1\ (k_i\ y) \\ & = & \Pi y : \mathcal{I}\ 0.\underbrace{(x_0 = 0 \to P\ 0\ y)}_{IH} \to x_0 = 1 \to P\ 1\ (k_i\ y) \end{array}$$

where IH represents the induction hypothesis associated to the argument of constructor k_i. Notably, IH is guarded by the condition $x_0 = 0$, which is not attainable – indeed, an equational inversion hypothesis tells us that $x_0 = 1$. This is ultimately due to the index of the inductive type not being used uniformly in the premise and

in the conclusion of the constructor. If instead the index had stayed the same, then *IH* would have been usable.

It is now clear why induction hypotheses are not provided by inversion rules: in the general case, they are dropped because inaccessible. This is what actually happens in the implementation of the Coq and Matita proof assistants, where inversion principles are automatically generated following the idea just described. However, there are situations where the induction hypothesis remains accessible.

We can generalise the previous observation to obtain the following improved induction rule: if a family parameter is *globally constant*, i.e. it remains the same in each recursive occurrence of the inductive family in its derivation rules, then the family parameter is not quantified in each premise of the induction principle, but it occurs instantiated with the value of the actual parameter in the hypothesis the principle is applied to. This is indeed the case for the induction principles of the Coq and Matita theorem provers. This observation is actually internalised in the meta-theory of the Calculus of (Co)Inductive Constructions, which allows global universal quantifications for inductive families to simplify the implementation and to have more liberal type-checking rules. However, it is possible to think of situations where an index is used uniformly in some constructors, but not in other. In such cases, it would still be possible to gain access to induction hypotheses, limited to some of the constructors. Moreover, there is the possibility that some of the indices of a constructor are used uniformly, while other indices are not. In this case, we would be able to use the induction hypothesis, provided that we agree to lose some equational hypotheses over non-uniform indices. These considerations drive us to define the following notion of induction/inversion principle we are interested in:

Theorem 4.3 (Induction/inversion principle) *Given an inductive family, we say that a formal family parameter is* locally constant *to one premise of one derivation rule if its actual value does not differ from that in the conclusion of the rule. We get a different induction/inversion principle for each subset $\mathcal{S}$ of the family parameters subject to the restriction that, if the type of a family parameter in $\mathcal{S}$ depends on another family parameter, the latter must also be in $\mathcal{S}$. The principle has the shape*

of an inversion principle with additional, accessible induction hypotheses provided for all those recursive arguments whose locally constant parameters are a superset of $\mathcal{S}$. Moreover, as in inversion principles, in each case we get a unifier as a set of additional hypotheses $F_i = A_i$ where F_i is a family parameter in $\mathcal{S}$ and A_i is the corresponding actual parameter in the conclusion of the inference rule.

Proof: Let $\mathcal{I}$ be an inductive family with arity $\forall \overline{z_n} : \overline{T_n}.\sigma$ and constructors $\overline{k_m} : \overline{K_m}$. Let P be a predicate of type $\forall \overline{z_n} : \overline{T_n}.\mathcal{I}\ \overline{z_n} \to \tau$. Assume that $\mathcal{S} = \{s_1, \ldots, s_{|\mathcal{S}|}\}$ and that $x_0, \ldots, x_{n-1}$ are properly typed variables. We define $P_{\mathcal{S}}[\overline{x_n}]$ as the predicate P partially augmented over the set $\mathcal{S}$:

$$P_{\mathcal{S}}[\overline{x_n}] \triangleq \lambda \overline{z_n} : \overline{T_n}.\lambda z : \mathcal{I}\ \overline{z_n}.x_{s_1} = z_{s_1} \to \ldots \to x_{s_{|\mathcal{S}|}} = z_{s_{|\mathcal{S}|}} \to P\ \overline{z_n}\ z$$

Then we can prove the $\mathcal{S}$-induction/inversion principle as follows

$$\begin{array}{rcl}
\mathcal{E}^{\tau}_{\mathcal{I},\mathcal{S}} & \triangleq & \lambda P : (\forall \overline{x_n} : \overline{T_n}.\mathcal{I}\ \overline{x_n} \to \tau). \\
& & \lambda \overline{x_n} : \overline{T_n}.\lambda x : \mathcal{I}\ \overline{x_n}. \\
& & \lambda f_0 : \Delta^{P_{\mathcal{S}}[\overline{x_n}]}\{\emptyset; k_0 : K_0\}. \\
& & \ldots \\
& & \lambda f_{m-1} : \Delta^{P_{\mathcal{S}}[\overline{x_n}]}\{\emptyset; k_{m-1} : K_{m-1}\}. \\
& & \mathcal{E}^{\tau}_{\mathcal{I}}(\overline{x_n}, x, P_{\mathcal{S}}[\overline{x_n}])\{\overline{f_m}\}\ \mathcal{R}_{x_{s_1}} \cdots \mathcal{R}_{x_{s_{|\mathcal{S}|}}} \\
& : & \forall P : (\forall \overline{x_n} : \overline{T_n}.\mathcal{I}\ \overline{x_n} \to \tau). \\
& & \forall \overline{x_n} : \overline{T_n}.\forall x : \mathcal{I}\ \overline{x_n}. \\
& & \Delta^{P_{\mathcal{S}}[\overline{x_n}]}\{\emptyset; k_0 : K_0\} \to \\
& & \ldots \\
& & \Delta^{P_{\mathcal{S}}[\overline{x_n}]}\{\emptyset; k_{m-1} : K_{m-1}\} \to \\
& & P\ \overline{x_n}\ x
\end{array}$$

To show that this definition satisfies the specification of an $\mathcal{S}$-induction/inversion principle, we must prove that:

1. The type of each subcase f_i is provided with an equational hypothesis for each index j in the set $\mathcal{S}$, stating that the family parameter of index j of the inverted term is equal to the corresponding paremeter in the target type of k_i.

But this is, by definition, the role of the partially augmented predicate $P_{\mathcal{S}}$. If $K_i : \forall \overline{y} : \overline{U}.V\ \overline{u}$, then the type of f_i is of the form

$$\begin{aligned}\forall \overline{y} : \overline{U}.IH_1 \to \cdots \to \cdots IH_r \\ \to x_{s_1} = u_{s_1} \to \cdots \to x_{s_{|\mathcal{S}|}} = u_{s_{|\mathcal{S}|}} \\ \to P\ \overline{u}\ (k_i\ \overline{y})\end{aligned}$$

thus satisfying the request.

2. Induction hypotheses such that all the parameters with index in $\mathcal{S}$ are locally constant are accessible. For this to happen, induction hypotheses must be in the form

$$IH \triangleq \forall \overline{a}.x_{s_1} = u_{s_1} \to \cdots \to x_{s_{|\mathcal{S}|}} = u_{s_{|\mathcal{S}|}} \to P\ \overline{u_n}\ v$$

such that $FV(u_{s_i}) \cap \overline{a} = \emptyset$ for all i, and the goal must be

$$x_{s_1} = u'_{s_1} \to \cdots \to x_{s_{|\mathcal{S}|}} = u'_{s_{|\mathcal{S}|}} \to P\ \overline{u'_n}\ v'.$$

such that for all $i \in \mathcal{S}$, $u_i = u'_i$. We can then introduce from the goal the new hypotheses $\overline{e_{|\mathcal{S}|}}$ and feed them to IH, obtaining

$$IH' \triangleq \lambda \overline{a}.IH\ \overline{a}\ \overline{e_{|\mathcal{S}|}} : \forall \overline{a}.P\ \overline{u_n}\ v$$

whose shape is the same of a regular, accessible induction hypothesis.

□

It is now clear that this induction/inversion lemma is exactly what we need to justify the informal proof, since it allows us to use induction hypotheses, but also to (partially) perform case analysis on the final rule of the derivation.

These induction/inversion rules can be automatically generated from the derivation rules of the judgement and, as well as the standard induction and inversion rules, are fully determined once the judgement is inductively defined. On the other hand, when the judgement has n family parameters, we can generate 2^n different induction/inversion principles. Indeed, standard induction and standard inversion correspond respectively to the empty and full sets of family parameters. A first

observation to reduce the number of principles to generate is that a set of induction/inversion principles makes sense only if its elements provide different induction hypotheses. In turn, this depends on the variety of locally constant parameters in the rules. Even if a large number of principles are worth generating, we can expect the proof assistant to dynamically generate them when needed.

As far as we know, the conditions for induction/inversion principles have never been characterised before. However, we have detected them in other proofs about the meta-theory of programming languages, such as the ones on LambdaDelta by F. Guidi[25]. We claim that better knowledge of them could easily result in shorter and deeper proofs.

4.4 Implementation of inversion principles in Matita

In Matita, full inversion principles for elimination towards all admissable sorts are generated automatically upon definition of a new inductive type having at least one right parameter. Clearly, it is impossible to do the same for induction/inversion principles, as the number of such principles grows very quickly with the number of right parameters. Whenever an induction/inversion principle is needed, the user must require explicitly that the system derive it, by means of the inverter statement

inverter <principle name> for <**inductive** type> [<selection>] [:
<target sort>]

where <**inductive** type> is the name of the inductive type for which an induction/inversion principle should be derived, <selection> is a pattern used to express the $\mathcal{S}$ set, and <target sort> identifies the target sort of the predicate on which the principle should act.

While it would be possible to define inversion principles generating directly a proof term for them, this approach has multiple drawbacks:

- the machinery to build such a proof term is relatively lengthy and specialized: we would like to keep the code designed specifically for performing inversion

of inductive predicates to a minimum;

- it is not easy to debug a CIC principle derived in a single step by means of an explicit proof term: this contrasts a lot with the way a user would prove such a principle, using an incremental, step by step strategy, also allowing him to understand immediately what is going wrong in case of error;

- proof terms are fully disambiguated: this means that many theory-specific notions have to be hard-coded in the procedure generating the inversion principle; specifically, we are very worried that we would have to choose once for all which definition of equality should be used by the inversion principle – Leibniz, dependent, John Major, etc.

Following our previous recommendation for tactics implementation, we build inversion principles on the existing tactic language, describing CIC terms by means of their representation as abstract syntax trees, instead of the more strict representation used by the typechecker, and also trying to keep the use of CIC terms to a minimum. In ASTs, it is possible to omit type annotations in many cases, and, furthermore, there is no need to deal with de Bruijn indices, since variables are represented by means of names. It is also possible to exploit user-level notation to identify the notion of equality that the user has currently loaded.

Concretely, the code implementing the generation of inversion principles, given an inductive type I with h left parameters, k right parameters and m constructors, will try to execute the following (pseudo-)proof script:

lemma I_inversion : $\forall x_1$:? ... $\forall x_{h+k}$:?.
$\forall$P: ($\forall y_1$:? ... $\forall y_k$:?.σ).
$\forall H_1$:? ... $\forall H_m$:?.
$\forall$Hterm: I x_1 ... x_{h+k}.
P x_{h+1} ... x_{h+k}.
(this opens one subgoal for the main theorem and a subgoal for all*
** the uninstantiated metavariables we used in the statement*
**)*

```
[ (* main proof *)
#x1 ... x_{h+k} P H1 ... Hm;
cut (x_{s1} = x_{s1} →... →x_{sk} = x_{sk} →P x_{h+1} ... x_{h+k};
[ (* main proof continued *)
#Hcut; apply Hcut;
(* opens a reflexivity proof for each selected right parameter:
* we close all of them at once
*)
apply refl;
| (* proof of the cut theorem *)
(* by induction on Hterm, matching only the rhs of each equality in the goal *)
elim Hterm in ⊢ (???% →... ???% → %)
(* a subcase for each constructor *)
[ apply H1;
| ...
| apply Hm ]
]
]
(* all metavariables have been instantiated by now *)
skip;
qed.
```

The script uses metavariables heavily in order to have most of the CIC terms derived by Matita's refiner. Only two non-trivial terms are generated by the tactic: a partial statement of the theorem, composed of a set of nested products abstracting the parameters of the inductive type $x_1, \ldots, x_{h+k}$, the goal predicate P, hypotheses for each of the subgoals generated by the inversion $H_1, \ldots, H_m$, and the term $Hterm$ to which inversion should be applied; and the statement of a "cut theorem" that introduces an equality for each right parameter belonging to the selection set $\mathcal{S} = \{s_1, \ldots, s_k\}$, which is really the heart of the inversion principle. While the types of the $H_1, \ldots, H_m$ are initially kept undefined, they are instantiated by the **apply** H_i statements to be exactly the subgoals required by the inversion principle.

This generic proof script, developed interactively with the Matita system, is easily translated to OCaml code thanks to the new design of the tactics subsystem:

```
...
let cut_theorem =
  let rs = List.map (fun x -> mk_id x) rs in
    mk_arrows rs rs selection (mk_appl (mk_id "P"::rs)) in
let cut = mk_appl [CicNotationPt.Binder
                    (‘Lambda, (mk_id "Hcut", Some cut_theorem),
                        CicNotationPt.Implicit (‘Tagged "end"));
                  CicNotationPt.Implicit (‘Tagged "cut")] in
let intros = List.map (fun x -> pp (lazy x); NTactics.intro_tac x)
               (xs@["P"]@hyplist@["Hterm"]) in
let where =
 "",0,(None,[],
  Some (
   mk_arrows ~pattern:true
    (HExtlib.mk_list (CicNotationPt.Implicit ‘JustOne) (List.length ys))
    (HExtlib.mk_list CicNotationPt.UserInput (List.length ys))
    selection CicNotationPt.UserInput)) in
let elim_tac = if is_ind then NTactics.elim_tac else NTactics.cases_tac in
let status =
 NTactics.block_tac
  (NTactics.branch_tac ::
   NTactics.case_tac "inv" ::
   (intros @
    [NTactics.apply_tac ("",0,cut);
     NTactics.branch_tac;
     NTactics.case_tac "end";
     NTactics.apply_tac ("",0,mk_id "Hcut");
     NTactics.apply_tac ("",0,mk_id "refl");
```

```
     NTactics.shift_tac;
     elim_tac ~what:("",0,mk_id "Hterm") ~where;
     NTactics.branch_tac ~force:true] @
      HExtlib.list_concat ~sep:[NTactics.shift_tac]
       (List.map (fun id-> [NTactics.apply_tac ("",0,mk_id id)]) hyplist) @
    [NTactics.merge_tac;
     NTactics.merge_tac;
     NTactics.merge_tac;
     NTactics.skip_tac])) status in
...
```

where the proof script is clearly discernible as an argument of `NTactics.block_tac`.

Part II

Formal metatheory of programming languages

Chapter 5

Some concrete encodings of binding structures

5.1 Introduction

The POPLmark challenge [7] is a set of "benchmarks" proposed by an international group of researchers in order to assess the advances of theorem proving for the verification of properties of programming languages and to promote the use and enhancement of proof assistant technology.

The set of problems has been chosen to capture many of the most critical issues in formalizing the metatheory of programming languages, comprising scope, binding, typing, and reduction. In particular, the challenge focuses on some theoretical aspects of System $F_{<:}$ [11], that is a language joining a simple and tractable syntax with a sufficiently rich and complex metatheory.

Arguably, issues related to the representation of binding structures are among the most significant choices when formalizing the metatheory of a programming language. Over the years, a number of different styles have been proposed to deal with binding, roughly divided in two different categories: first order encodings, also called *concrete* encodings, where binding is manipulated as an ordinary data structure, and higher order approaches, where binders are represented as functions, essentially reusing the binding (and instantiation) capabilities of the metalanguage. Concrete encodings include some of the best known styles, like the naïve named encoding (where bound and free variables are represented by the same sort of names, similarly to informal syntax), and the de Bruijn nameless encoding (which represents variables using indices pointing to the binder that declares them). Higher order encodings include styles particularly popular for formalizing languages in logical frameworks, like higher-order abstract syntax ([47, 26]) and weak higher-order abstract syntax ([18]).

While techniques to employ higher order approaches have also been devised for stronger type theories like those of Coq, Isabelle, and Matita (most notably *two-level* approaches, described for example in [10]), we feel that the most natural way to deal with binding in these systems is still to use concrete representations. In this chapter, we discuss the formalization of part 1A of the POPLmark challenge

using three concrete representations. While we review mainly classical techniques to formalize programming languages, our examples are a good setting to introduce the formalization of the metatheory of programming languages. Furthermore, we put to work the induction/inversion principles discussed in Chapter 4 and we analyse the issue of "formal adequacy" (in the sense of [14]).

The structure of the chapter is the following: Section 5.2 discusses the three representations of bound and free variables which we used in our solutions; in Section 5.3 we describe the proof principles and the main proofs of our solutions; Section 5.5 concludes.

5.2 Concrete encodings of variable bindings

System $F_{<:}$ is a second order lambda calculus enriched with a subtyping relation $<:$. Syntactically, it can be understood as a variant of System F, where binders involving type variables also carry annotations representing upper bounds on the concrete types that can instantiate the corresponding variable. For example, $\Lambda X_T.t$ and $\forall X_T.U$ abstract respectively a term t and a type U over types X that are subtypes of T.

Since the focus of part 1 of the POPLmark challenge is on the formalization of proofs concerning the type sublanguage of $F_{<:}$, we will not spend time discussing the full syntax of $F_{<:}$, and only report here the details needed to discuss the formalized syntax of types and typing environments here. For the full syntax of $F_{<:}$, see [11].

$S, T, \ldots$	::=		Types
	|	$X, Y, \ldots$	type variables
	|	`Top`	the supertype of any type
	|	$S \to T$	functions from S to T
	|	$\forall X_S.T$	bounded universal quantifier

Figure 5.1: Syntax of the type sublanguage of $F_{<:}$

$$\Gamma \vdash \texttt{Top} \qquad \text{(WFT-Top)}$$

$$\frac{X \in dom(\Gamma)}{\Gamma \vdash X} \qquad \text{(WFT-TVar)}$$

$$\frac{\Gamma \vdash S \qquad \Gamma \vdash T}{\Gamma \vdash S \to T} \qquad \text{(WFT-Arrow)}$$

$$\frac{X \notin dom(\Gamma) \qquad \Gamma \vdash T \qquad \Gamma, X <: T \vdash U}{\Gamma \vdash \forall X <: T.U} \qquad \text{(WFT-Forall)}$$

$$\emptyset \vdash \diamond \qquad \text{(WFE-Empty)}$$

$$\frac{x \notin dom(\Gamma) \qquad \Gamma \vdash T}{\Gamma, x : T \vdash \diamond} \qquad \text{(WFE-Cons1)}$$

$$\frac{X \notin dom(\Gamma) \qquad \Gamma \vdash T}{\Gamma, X <: T \vdash \diamond} \qquad \text{(WFE-Cons2)}$$

$$\frac{\Gamma \vdash \diamond \qquad \Gamma \vdash S}{\Gamma \vdash S <: \texttt{Top}} \qquad \text{(SA-Top)}$$

$$\frac{\Gamma \vdash \diamond \qquad X \in dom(\Gamma)}{\Gamma \vdash X <: X} \qquad \text{(SA-Refl-TVar)}$$

$$\frac{X <: U \in \Gamma \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T} \qquad \text{(SA-Trans-TVar)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(SA-Arrow)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X_{S_1}.S_2 <: \forall X_{T_1}.T_2} \qquad \text{(SA-All)}$$

Figure 5.2: Well-formedness and subtyping rules of $F_{<:}$

The type sublanguage of $F_{<:}$ (Fig. 5.1) consists of type variables, the type `Top` (which is supertype of any type), arrow types (functions from one type to another) and universal types (polymorphic expressions); environments may carry both typing constraints (on term variables) and subtyping constraints (on type variables). Figure 5.2 shows the subset of the $F_{<:}$ typesystem we formalized. Notice that in

this formulation of the type system, the subtyping judgment is formalized by means of an algorithmic rules, which are directed by the syntax. Instead, the original presentation of $F_{<:}$ replaced rules SA-Refl-TVar and SA-Trans-TVar with the following *declarative* rules:

$$\frac{X <: U \in \Gamma \qquad \Gamma \vdash \diamond}{\Gamma \vdash X <: U} \tag{SA-TVar}$$

$$\frac{\Gamma \vdash T}{\Gamma \vdash T <: T} \tag{SA-Refl-TVar}$$

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \tag{SA-Trans-TVar}$$

Part 1a of the POPLmark challenge asks to prove that the two formulations of algorithmic subtyping are equivalent and, in particular, that algorithmic subtyping is reflexive and transitive.

Since $F_{<:}$ makes use of binders not only in terms, but also in types, we must deal with the well-known problems of α-equivalence and avoidance of variable capture. The most common approaches to these difficulties require to rewrite the syntax in such a way that α-equivalent terms are syntactically equal. One way to do this is to drop names altogether: variables can be expressed by means of indices, whose value uniquely identifies the level at which the variable is bound; this is how de Bruijn's representation works (Fig. 5.3).

$S, T, \ldots$	::=		Types
	|	$\#0, \#1, \ldots$	type indices
	|	`Top`	the supertype of any type
	|	$S \to T$	functions from S to T
	|	$\forall_S.T$	bounded universal quantifier

Figure 5.3: Syntax of $F_{<:}$ (de Bruijn): types

A de Bruijn index expresses the number of binders, on the path between the index we are considering and the root of the AST, to reach the binder we want

$S, T, \ldots$	::=		Types
	|	$X, Y, \ldots$	free type variables
	|	$\#0, \#1, \ldots$	type indices
	|	`Top`	the supertype of any type
	|	$S \to T$	functions from S to T
	|	$\forall_S.T$	bounded universal quantifier

Figure 5.4: Syntax of $F_{<:}$ (locally nameless): types

to reference. Dangling indices will then represent free variables. This poses one inconvenience with de Bruijn's representation: when performing a substitution, indices representing free variables in the substituted term might need to be updated (*lifted*) in order to stay coherent; this can complicate both the statements and the proofs of many lemmata. The *locally nameless* representation [52] is a variation on de Bruijn's representation, where bound variables are represented by indices (so that α-equivalence and equality are the same) and free variables are represented by names (eliminating the need to lift free indices in substituted terms). While the locally nameless style does not forbid dangling indices, well formed expressions will belong to the subset of *locally closed* expressions, i.e. expressions that may contain free variable names, but not dangling indices. The syntax locally nameless syntax is the same as the de Bruijn representation, except for the addition of free type variables (Fig. 5.4).

Typing environments in the locally nameless approach are similar to their informal counterparts. They are defined as lists of bounds, which are pairs (variable name, type), together with a boolean value to discriminate typing bounds on term variables from subtyping bounds on type variables.

In the de Bruijn approach, we do not have names and bounds are identified by their position inside the environment. The dangling indices inside a bound must be resolved in the part of the environment which precedes that bound (essentially, this means that entries in a context are treated as a specific kind of binder). We will use the notation $\bullet <: T$ to refer to a subtyping bound in a de Bruijn typing

environment.

The last concrete approach to binding we take into account is the named variables approach, in which names are used for both free and bound variables. Its syntax is the closest possible to the informal presentation of Fig. 5.1: however we will see that the formalization of its type system requires some additional care.

5.3 Formalization

We now discuss our formalizations of Part 1A of the POPLmark challenge. The first part of this section deals with the formalization of the type system using the encodings mentioned in Section 5.2. In the second part, we present some of the proof principles used in the solutions. Finally, we describe the main proofs of each formalization.

5.3.1 The type system

To restate the well-formedness and subtyping judgments in the de Bruijn encoding, it is sufficient to remember the key differences of this encoding with respect to the informal syntax:

- named variables are replaced by indices, with an explicit management of binding: the dangling index $\#n$ refers to the n-th entry of its environment (from right to left, 0 based);
- each environment entry lives in a different environment: in order to use the content of an environment entry in a judgment, we must relocate it to the environment of that judgment.

The first change happens to be more of an advantage than an issue: it allows us not to worry at all about names, at the same time keeping the statement of rules concerning binding very natural, similar to informal practice. The second change, however, needs a more careful handling, since relocation must be treated

explicitly. Fig. 5.5 shows the de Bruijn formalization of the less trivial rules of $F_{<:}$: the notations $|\Gamma|$ and $\Gamma(n)$ refer respectively to the length of environment Γ and to the n-th entry of Γ; $T \uparrow n$ is the variable lifting operation, defined as follows:

$$T \uparrow_k n \;=\; \begin{cases} \#m \uparrow_k n \;=\; m+n & \text{if } k \le m \\ \#m \uparrow_k n \;=\; m & \text{if } k > m \\ \mathtt{Top} \uparrow_k n \;=\; \mathtt{Top} & \\ (U \to V) \uparrow_k n \;=\; (U \uparrow_k n) \to (V \uparrow_k n) & \\ (\forall_U.V) \uparrow_k n \;=\; \forall_{U\uparrow_k n}.(V \uparrow_{k+1} n) & \end{cases}$$

$$T \uparrow n \;=\; T \uparrow_0 n$$

Lifting provides the notion of relocation we needed, since each environment entry lives in an initial segment of the full environment.

$$\frac{n < |\Gamma|}{\Gamma \vdash \#n} \qquad \text{(WFT-TFree)}$$

$$\frac{\Gamma \vdash T \qquad \Gamma, \bullet <: T \vdash U}{\Gamma \vdash \forall_T.U} \qquad \text{(WFT-All)}$$

$$\frac{\Gamma \vdash \diamond \qquad n < |\Gamma|}{\Gamma \vdash \#n <: \#n} \qquad \text{(SA-Refl-TVar)}$$

$$\frac{\Gamma(n) = \bullet <: U \qquad \Gamma \vdash (U \uparrow n+1) <: T}{\Gamma \vdash \#n <: T} \qquad \text{(SA-Trans-TVar)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, \bullet <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall_{S_1}.S_2 <: \forall_{T_1}.T_2} \qquad \text{(SA-All)}$$

Figure 5.5: Some rules of the de Bruijn-style formalization of $F_{<:}$.

In the locally nameless encoding, we get a more immediate treatment of environments, since relocation of environment entries is not needed. On the contrary, binding needs a more complex treatment, because of the use of explicit names for free variables. In particular, the rules whose conclusions involve binders cannot be

fully structural on types: on one side, we want the type system to only deal with locally closed types (since locally-closedness is a necessary condition for a type to be well formed); on the other side, in a well formed type $\forall_U.V$, V is in general not locally closed.

Of course the solution is to replace the dangling index #0 of V with a proper free variable X. However this kind of reasoning hides more complexity than meets the eye. For example, we might translate the ALL rule to the locally nameless encoding, obtaining easily:

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X <: T_1 \vdash S_2\{^X/_{\#0}\} <: T_2\{^X/_{\#0}\}}{\Gamma \vdash \forall_{S_1}.S_2 <: \forall_{T_1}.T_2}$$

where $S_2\{^X/_{\#0}\}$ means "the type S_2 where every occurrence of the dangling index #0 has been replaced with a free type name X". Please notice the use of X: nowhere do we state if the right premise should hold for a specific X or for any X. Indeed, both alternatives are partially incorrect because, for reasons of well-formedness, we must require that X be fresh; assuming this condition of well-formedness is met, alternative solutions for quantification have been proposed in literature. Universal and existential quantification lead to formulations of the type system which we respectively call *strong* and *weak* (after Urban and Pollack [65]). However, these names are somewhat misleading since it can be proved that the two formulations are logically equivalent: this comes from the fact that the subtyping judgment is an *equivariant* predicate, i.e. one whose validity is invariant under finite permutations of variable names.

The concept of equivariance, which is a key point of nominal logics [48], was exploited in the solution proposed by Leroy [34], as well as in a previous version of our locally nameless solution. However, upon discovering that the proofs related to equivariance accounted for about one third of our code, we decided to go for a more direct approach.

It can be noted that, in informal logical practice, it is convenient to use the weak (existential) variant when we want to construct a proof of $\Gamma \vdash \forall_{S_1}.S_2 <: \forall_{T_1}.T_2$

(we only need to show that the premises hold for one suitable X); on the other hand, when reasoning backwards, the strong (universal) variant is more useful, as it provides stronger induction principles. A more complex co-finite quantification [8] has been used by Charguéraud for his locally nameless solution in Coq. In our locally nameless solution, we chose to use the strong formulation of the type system, which is sufficient to obtain very compact proofs (for a total of 350 lines). Still, the statement that strong typing rules capture the informal intuition of a type system is controversial: this issue will be further discussed in section **??**.

Figure 5.6 describes the rules for well-formedness and subtyping of universal types, as formalized in the locally nameless encoding.

$$\frac{\Gamma \vdash T \qquad \text{for all } X : \left(\begin{array}{l} X \notin dom(\Gamma) \wedge X \notin \mathrm{FV}(U) \Rightarrow \\ \Gamma, X <: T \vdash U\{^X/_{\#0}\} \end{array} \right)}{\Gamma \vdash \forall_T.U} \qquad \text{(WFT-All)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \text{for all } X : \left(\begin{array}{l} X \notin dom(\Gamma) \Rightarrow \\ \Gamma, X <: T_1 \vdash S_2\{^X/_{\#0}\} <: T_2\{^X/_{\#0}\} \end{array} \right)}{\Gamma \vdash \forall_{S_1}.S_2 <: \forall_{T_1}.T_2} \qquad \text{(SA-All)}$$

Figure 5.6: Some rules of the locally nameless formalization of $F_{<:}$.

Our last formalization uses the named variables approach. Ideally, the formalization of the type system should be very close to the informal presentation of Fig. 5.2. However, at some point, α-conversion must be taken into account, otherwise we will never be able to prove a subtyping relation between two universal types binding different variables.

There are basically two ways to deal with α-conversion:

- α-conversion can be formalized separately from the subtyping judgment (either

algorithmically or as an inductive predicate); then an additional rule for the subtyping judgment will be provided:

$$\frac{\Gamma \vdash S <: T \qquad S =_\alpha S' \qquad T =_\alpha T'}{\Gamma \vdash S' <: T'} \quad \text{(SA-Alpha)}$$

- the rules WFT-All and SA-All can be rephrased in such a way that the subtyping judgment is directly derivable even if their bound variables are different:

$$\frac{\text{for all } Y \notin dom(\Gamma): \begin{pmatrix} \Gamma \vdash T \\ (Y \in \mathrm{FV}(U) \Rightarrow Y = X) \Rightarrow \\ \Gamma, Y <: T \vdash (Y\ X) \cdot U \end{pmatrix}}{\Gamma \vdash \forall X_T.U} \text{(WFT-Forall)}$$

$$\frac{\text{for all } Z \notin dom(\Gamma): \begin{pmatrix} \Gamma \vdash T_1 <: S_1 \\ (Z \in \mathrm{FV}(S_2) \Rightarrow Z = X) \Rightarrow \\ (Z \in \mathrm{FV}(T_2) \Rightarrow Z = Y) \Rightarrow \\ \Gamma, Z <: T_1 \vdash (Z\ X) \cdot S_2 <: (Z\ Y) \cdot T_2 \end{pmatrix}}{\Gamma \vdash \forall X_{S_1}.S_2 <: \forall Y_{T_1}.T_2} \text{(SA-All)}$$

where $(X\ Y) \cdot -$ is the name swapping operator, replacing every occurrence of X with Y and vice-versa, not caring for binders.

We will avoid the first solution, since rules like SA-Alpha make the subtyping judgment less algorithmic, which would contrast with the spirit of the POPLmark challenge. However the second solution can seem a little puzzling at first. The swap-based statement of α-conversion was originally due to Gabbay and Pitts [20] and is very well-suited to formalization, since it simplifies the handling of name-capture. For what concerns quantification over free variables, again we follow the schema of universal quantification over all acceptable names Z. Z is acceptable if:

- it's not in the domain of Γ;

- it does not cause variable capture inside S_2 or T_2: for this condition to hold, one must verify that if $Z \in \mathrm{FV}(S_2)$, then $Z = X$, and that if $Z \in \mathrm{FV}(T_2)$, then $Z = Y$.

5.3.2 Proof principles

Most proofs given in the specification of the POPLmark challenge are by structural induction on some type. However it is often the case, particularly in the locally-nameless representation, that structural induction on types does not yield a strong enough induction hypothesis to reason on sub-typing in the case of bounded quantification: for example, to prove $\forall_S.T$, we obtain an induction hypothesis on T, whereas we now need an induction hypothesis on $T\{^X/_{\#0}\}$ for all X.

Instead of using induction on types, a very natural proof technique consists in doing structural induction on (proof trees for) the well-formedness type judgment. For instance, induction over a proof of $\Gamma \vdash T$ yields exactly the four cases of a proof by induction over T (i.e. $T = \mathtt{Top}$, $T = X$, $T = T_1 \to T_2$ and $T = \forall_{T_1}.T_2$); the second induction hypothesis in the last case is the strong one we usually need, i.e. that the binary property P (on pairs typing context-type) we are proving holds for $\Gamma, X <: T_1$ and $T_2\{^X/_{\#0}\}$ for any type variable X free in both Γ and T_2.

In our opinion, and as already noticed by others (like [52]) proofs by structural induction on the well-formedness judgment are more than a technical trick due to an unnatural representation: they are the natural way to reason on types (and terms) of a language. Indeed, note that structural induction on types and structural induction on well-formedness type judgments yield exactly the same hypotheses when types are considered up to α-equivalence. Thus we may think of the proofs in the specification of the POPLmark challenge as proofs by structural induction on well-formedness judgments.

Once decided that informal proofs by structural induction on types are to be formalized with structural induction on the well-formedness judgment for types, the informal proof still presents a suspicious proof step. In [7], Lemma A.3 (transitivity and narrowing), the proof is done "*by induction on the structure of* Q. *. . . We proceed*

by an inner induction on the structure of $\Gamma \vdash S <: Q$*, with a case analysis on the final rule of this derivation and of* $\Gamma \vdash Q <: T$*. . . . by the inner induction hypothesis . . .*". The question is how to formalize an "inner induction on the structure, with a case analysis on the final rule". As we argued in the previous chapter, in the Calculus of (Co)Inductive Constructions structural induction does not allow to perform at once case analysis on the final rule, unless we give up on using the "inner induction hypotheses". The proof may probably still be understood as a proof by induction on the size of the derivation, followed by case analysis on the last rule used. However, such a proof is more involved and more difficult to carry out in systems that favour structural induction (such as Coq and Matita). Although this proof principle has been "implicitly" exploited in several solutions in Coq to the POPLmark challenge, none of them make it explicit, resulting in an obfuscated proof whose key point is unclear and which is difficult to port to variations of the calculus. Instead, we have employed Matita's induction/inversion facility to automatically derive a principle informal notion of "inner induction on the structure, with a case analysis on the final rule".

Induction/inversion principles

We show the induction/inversion principle for the sub-typing judgment of Fig. 5.6 where we choose $\mathcal{S} = \{\Gamma, T\}$ (i.e. the typing context and the second type). The choice is driven by the Trans rule where Γ and T are locally constant parameters, whereas the second argument is not (being U in the premise, and X in the conclusion). Indeed, note that we get an almost-standard inversion principle were we have traded hypotheses on L with the induction hypothesis in the **Trans** case.

Theorem 5.1 ($\{\Gamma, T\}$-Induction/inversion for $\Gamma \vdash S <: T$) *Let P be a ternary predicate over triples (Δ, L, R). For all Δ, L, R we have $\Delta \vdash L <: R$ implies $P(\Delta, L, R)$ provided that*

Top $\forall \Gamma, S.\ \Gamma \vdash \diamond \Rightarrow \Gamma \vdash S \Rightarrow (\Delta = \Gamma) \Rightarrow (R = \mathtt{Top}) \Rightarrow P(\Gamma, S, \mathtt{Top})$

Refl $\forall \Gamma, X.\ \Gamma \vdash \diamond \Rightarrow X \in dom\Gamma \Rightarrow (\Delta = \Gamma) \Rightarrow (R = X) \Rightarrow P(\Gamma, X, X)$

Trans $\forall \Gamma, X, U, T,\ X <: U \in \Gamma \Rightarrow \Gamma \vdash U <: T \Rightarrow P(\Gamma, U, T) \Rightarrow (\Delta = \Gamma) \Rightarrow (R = T) \Rightarrow P(\Gamma, X, T)$

Arrow $\forall \Gamma, S_1, S_2, T_1, T_2.\ \Gamma \vdash T_1 <: S_1 \Rightarrow \Gamma \vdash S_2 <: T_2 \Rightarrow (\Delta = \Gamma) \Rightarrow (R = T_1 \to T_2) \Rightarrow P(\Gamma, S_1 \to S_2, T_1 \to T_2)$

All $\forall \Gamma, S_1, S_2, T_1, T_2.\ \Gamma \vdash T_1 <: S_1 \Rightarrow (\forall X, X \notin dom(\Gamma) \Rightarrow \Gamma, X <: T_1 \vdash S_2\{^X/_{\#0}\} <: T_2\{^X/_{\#0}\}) \Rightarrow (\Delta = \Gamma) \Rightarrow (R = \forall_{T_1}.T_2) \Rightarrow P(\Gamma, \forall_{S_1}.S_2, \forall_{T_1}.T_2)$

It is clear that this induction/inversion lemma is exactly what we need to justify the informal proof, since it allows to use the "inner hypothesis" (in the **Trans** case), but also to (partially) perform "case analysis on the final rule of the derivation". What is surprising at first is that such a proof principle, that seems quite ad-hoc in the informal proof, is actually a general proof principle. Indeed, we want to note that these induction/inversion rules can be automatically generated from the derivation rules of the judgment and, as well as the standard induction and inversion rules, are fully determined once the judgment is inductively defined.

5.3.3 Proofs

In this section we will discuss briefly the main proofs of the solutions to POPLmark part 1a that we have formalized in Matita, based on the proof techniques of Section 5.3.2. We will begin with the locally nameless solution, which we believe has a more basic presentation than the other two.

Locally nameless

The first property we must show is the reflexivity of subtyping.

Theorem 5.2 (reflexivity (locally nameless encoding)) *Let Γ be a typing environment, and T a type; if Γ is well-formed and T is well-formed in Γ, then $\Gamma \vdash T <: T$.*

Proof: Once it has been proved that, for all Γ and T, $\Gamma \vdash T$ implies $\mathrm{FV}(T) \subseteq dom(\Gamma)$, the proof is trivial by induction on the derivation of $\Gamma \vdash T$. Matita is able to prove almost every case of the induction by means of standard automation. □

The following theorem asserts a stronger weakening property than the one described in the specifications: here weakening on well formed environments is defined as set inclusion, instead of concatenation of two disjoint environments. In this way we are exempted from proving the less tractable lemma on permutations.

Theorem 5.3 (weakening (locally nameless encoding))

1. *Let Γ be a typing environment, T a type. If $\Gamma \vdash T$, then for all environments Δ such that $\Gamma \subseteq \Delta$, we get $\Delta \vdash T$.*

2. *Let Γ be a typing environment, T, U types. If $\Gamma \vdash T <: U$, for all well-formed environments Δ such that $\Gamma \subseteq \Delta$, we get $\Delta \vdash T <: U$.*

Proof: The first point follows easily from a straightforward induction on the derivation of $\Gamma \vdash T$. The second point follows from an induction on the derivation of $\Gamma \vdash T <: U$ (the proposition proved in part (i) is used in the Top case). Once again standard automation turns out to be very useful. □

Unlike the specifications, we decided to prove narrowing and transitivity separately. Our statements are also slightly stronger than the ones provided in the specifications. This is ultimately due to the locally nameless encoding: in fact, since the encoding of the All rule is not fully structural with respect to the types mentioned in it, the induction on the structure of a type, used in the informal proof, is not sufficient to prove narrowing and transitivity in our setting. Instead, we will use an induction on the derivation of some judgments.

Theorem 5.4 (narrowing (locally nameless encoding)) *For all typing environments Γ, Γ', for all types U, P, M, N and for all variables X, if*

1. $\Gamma' \vdash P <: U$

2. $\Gamma \vdash M <: N$

3. for all Γ'', T *if* $\Gamma', \Gamma'' \vdash U <: T$ *then* $\Gamma', \Gamma'' \vdash P <: T$

then forall Δ *s.t.* $\Gamma = \Gamma', X <: U, \Delta$*, the judgment* $\Gamma', X <: P, \Delta \vdash M <: N$ *holds.*

Proof: We proceed by induction on the derivation of $\Gamma \vdash M <: N$. The interesting case is SA-Trans-TVar: in this case, $M = Y$, where Y is a type variable. If $X = Y$, we prove the statement by means of rule SA-Trans-TVar. Since $X <: P \in (\Gamma', X <: P, \Delta)$ (trivially), we only need to prove $\Gamma', X <: P, \Delta \vdash P <: N$: this is obtained by means of hypothesis 3 ($\Gamma', X <: P, \Delta \vdash U <: N$ holds by induction hypothesis). If $X \neq Y$, the goal is obtained trivially by induction hypothesis. □

Last, we turn to proving the main property, i.e. transitivity of subtyping. Again, we use a slightly different statement from the specifications, but the proof follows closely the suggested structure.

Theorem 5.5 (transitivity (locally nameless encoding)) *Let* T *a type,* Γ' *a typing environment such that* $\Gamma' \vdash T$*. For any typing environment* Γ *such that* $dom(\Gamma') \subseteq dom(\Gamma)$*, and for all types* R, U*, if* $\Gamma \vdash R <: T$ *and* $\Gamma \vdash T <: U$ *then* $\Gamma \vdash R <: U$*.*

Proof: We proceed by induction on the derivation of $\Gamma' \vdash T$, followed by $\{\Gamma, T\}$-induction/inversion on $\Gamma \vdash R <: T$. The interesting case is WFT-All: in this case, $T = \forall_{T'}.T''$ and, by applying the unifier provided by the principle, only two cases are possible:

- $R = X$ (where X is a type variable) and $X <: V \in \Gamma$ (for some type V). The thesis follows from the induction/inversion hypothesis, by means of rule Trans.

- $R = \forall_{R'}.R''$. In this case, by inversion on $\Gamma \vdash \forall_{T'}.T'' <: U$ we show U is either `Top` or $\forall_{U'}.U''$. In the first case, showing that $\Gamma \vdash \forall_{R'}.R'' <: \texttt{Top}$ is trivial. In the second case, the difficult part is to show that, for all $X \notin dom(\Gamma)$, $\Gamma, X <: U' \vdash R''\{^X/_{\#0}\} <: U''\{^X/_{\#0}\}$. By the induction hypothesis, we only need to prove $\Gamma, X <: U' \vdash R''\{^X/_{\#0}\} <: T''\{^X/_{\#0}\}$ and $\Gamma, X <: U' \vdash T''\{^X/_{\#0}\} <:$

$U''\{^X/_{\#0}\}$: these follow from the induction/inversion hypothesis, together with narrowing. Please notice that the hypothesis $dom(\Gamma') \subseteq dom(\Gamma)$, here, is essential, since otherwise the typing environments in the induction hypothesis and in the goal would not match.

□

When proving reflexivity and transitivity, our formalization of the All rule requires to prove that some judgment holds for any fresh variable X. As we pointed out in section 5.3, since the subtyping judgment is equivariant, it is sufficient that it hold for one fresh X: following this intuition, Leroy, in his solution to the challenge, decided to prove this alternate "for one" rule. Apparently, this should have simplified the proofs of reflexivity and transitivity, thus in a previous version of our solution we decided to follow closely his approach; however, proving the "for one" rule required a great effort (approximately 500 lines of code out of 1500). Moreover, proofs can be completed quite easily even without the "for one" rule. The most difficult case is probably in the reflexivity: we must prove that $\Gamma \vdash \forall_T.U <: \forall_T.U$, knowing (by hypothesis) that $\forall_T.U$ is well-formed in Γ, and (by induction hypothesis) that for any $X \notin dom(\Gamma) \cup \mathrm{FV}(U)$, $\Gamma, X <: T \vdash U\{^X/_{\#0}\} <: U\{^X/_{\#0}\}$ holds; now if we apply the "for one" version of the rule, it's sufficient to prove that for some $Y \notin dom(\Gamma)$, the judgment $\Gamma, Y <: T \vdash U\{^Y/_{\#0}\} <: U\{^Y/_{\#0}\}$ holds – then we choose Y to be fresh both in Γ and T, and the thesis follows trivially from the induction hypothesis; using the original All rule is only apparently more difficult: we need to prove the same judgment for any $Y \notin dom(\Gamma)$ but, since $\forall T.U$ is well-formed in Γ, one can easily prove that no variables outside Γ can be free in U, thus the induction hypothesis is sufficient even in this case.

De Bruijn nameless encoding

While the concern about readability of terms containing nameless dummies, which is also brought against locally nameless solutions, is debatable, the fact that de Bruijn open terms must be explicitly lifted when the environment is changed, is a serious matter. The statement of theorems must be carefully tuned and while we do not

feel that the readability of the proofs is compromised, the ease of formalization is impaired to some extent. Still, formalization of the properties of dangling dummies has some interest *per se*, and theorem provers provide all the tools to carry out their proofs.

The proof of reflexivity in the de Bruijn encoding is even easier than in the locally nameless encoding (exactly 3 proof steps in Matita); weakening, however, is the typical example of a theorem whose statement must be somewhat reworked in the de Bruijn encoding. The relation $\Gamma \subseteq \Delta$ we had used in the locally nameless version, denoting that Γ is extended by Δ, possibly with some entries permuted, is not meaningful for de Bruijn environments: while the entries of a locally nameless environment can be permuted consistently without updating the free names, in a de Bruijn environment the dangling dummies must be also permuted explicitly.

We are tempted to state weakening by means of environment concatenation and lifting:

> *For all environments* Γ, Γ' *and types* S, T*, if* $\Gamma \vdash S <: T$ *and* $\Gamma, \Gamma' \vdash \diamond$*, then* $\Gamma, \Gamma' \vdash S \uparrow |\Gamma'| <: T \uparrow |\Gamma'|$.

The statement is correct but its proof (as noted in the POPLmark specifications [7]) requires a permutation lemma which is precisely what we wanted to avoid in the first place.

The best we can do is to prove a strong version of weakening, implying the permutation lemma, just like we did in the locally nameless formalization. However, the notion of environment inclusion needs to be significantly refined. On the other hand, lifting is not sufficient to deal with permutations, and a generalization is needed.

Definition 5.1 *The* map application *of a function* $f : \mathbb{N} \to \mathbb{N}$ *on* $F_{<:}$ *types is*

defined as follows:

$$f \cdot T = \begin{cases} f \cdot \#n & = \#f(n) \\ f \cdot \mathtt{Top} & = \mathtt{Top} \\ f \cdot (T \to U) & = f \cdot T \to f \cdot U \\ f \cdot (\forall_T.U) & = \forall_{f \cdot T}.(\widehat{f} \cdot U) \end{cases}$$

where $\widehat{f}$ *is defined as:*

$$\widehat{f}(x) = \begin{cases} 0 & \text{if } x = 0 \\ f(y) + 1 & \text{if } x = y + 1 \end{cases}$$

Definition 5.2 *A function* $f : \mathbb{N} \to \mathbb{N}$ *is an* environment extension map *from* Γ *to* Δ *(notation:* $\Gamma \subseteq_f \Delta$*) if and only if it is injective and for all* $n < |\Gamma|$*,* $f(n) < |\Delta|$ *and* $f \cdot (\Gamma(n) \uparrow n + 1) = \Delta(f(n)) \uparrow f(n) + 1$*.*

In simple terms, an environment extension map is a more explicit version of the $\subseteq$ relation used in the locally nameless formalization. Its definition can be paraphrased by saying that for every n, the n-th entry of Γ, relocated at the top level (by lifting) and then mapped to the environment Δ (by means of f) must be equal to the $f(n)$-th entry of Δ, relocated to the top level (again by lifting).

We can then prove weakening in the following form.

Theorem 5.6 (weakening (de Bruijn encoding)) For all environments Γ, Δ, if for some f, $\Gamma \subseteq_f \Delta$, $\Gamma \vdash S <: T$ and $\Delta \vdash \diamond$, then $\Delta \vdash f \cdot S <: f \cdot T$.

While the proof of the above statement is similar to its locally nameless counterpart, automation turns out to be a bit less effective.

It can be noted that lifting and environment extension maps have an interesting relation. Let $\Uparrow_k^m \colon \mathbb{N} \to \mathbb{N}$ be the family functions defined as follows:

$$\Uparrow_k^m (n) = \begin{cases} m + n & \text{if } k \leq n \\ n & \text{else} \end{cases}$$

We show that for all types T, $T \uparrow_k m = \Uparrow_k^m \cdot T$. As a corollary, for all environments Γ, Δ, $\Gamma \subseteq_{\Uparrow_0^{|\Delta|}} \Gamma, \Delta$: therefore the version of weakening involving environment extension maps also implies the previous statement with concatenation of environments and lifting.

Narrowing and transitivity are then proved separately, following the same strategy, if not the letter, of the locally nameless proofs.

Theorem 5.7 (narrowing (de Bruijn encoding)) *For all typing environments* Γ,Γ' *and for all types* U, P, M, N*, if*

1. $\Gamma' \vdash P <: U$
2. $\Gamma \vdash M <: N$
3. *for all* Γ'', S, T*, if* $\Gamma', \Gamma'' \vdash S <: (U \uparrow |\Gamma''|)$ *and* $\Gamma', \Gamma'' \vdash (U \uparrow |\Gamma''|) <: T$ *imply* $\Gamma', \Gamma'' \vdash S <: T$

then for all Δ *s.t.* $\Gamma = \Gamma', \bullet <: U, \Delta$*, the judgment* $\Gamma', \bullet <: P, \Delta \vdash M <: N$ *holds.*

Theorem 5.8 (transitivity (de Bruijn encoding)) Let S, T, U be types, Γ a typing environment, f a function from naturals to naturals. If $\Gamma \vdash S <: f \cdot T$ and $\Gamma \vdash f \cdot T <: U$, then $\Gamma \vdash S <: U$.

Somewhat surprisingly, the above statement of transitivity does not require f to be an environment extension map: it is sufficient for f to be a function from naturals to naturals. The particular statement of the theorem is needed in order to get an induction hypothesis which is sufficiently strong to imply the weak transitivity requirement of the previous narrowing theorem. The proof also exploits the $\{\Gamma, T\}$-induction/inversion principle, similarly to the corresponding proof in the locally nameless encoding.

The usual statements of transitivity and narrowing are then obtained as corollaries.

Named representation

Our solution using the named representation follows a radically different approach from the other two: instead of proving the transitivity of subtyping directly on types with named variables, we decided to provide a translation of types with named variables to locally nameless types. This induces a translation of environments and, consequently, a translation of whole subtyping judgments: if we can prove that the subtyping judgment on types with named variables is adequate and faithful with respect to the corresponding judgment on locally nameless types, we can obtain the transitivity on types with named variables as a corollary, from the transitivity on locally nameless types.

This kind of formalization, similar to transformations performed by actual compilers, has an interest in itself and hides some difficulties: therefore it seemed to be a good companion to the problems of the POPLmark challenge.

First, we need to define an algorithm providing the intended encoding of types with named variables into locally nameless types.

$$\llbracket T \rrbracket_\ell = \begin{cases} \llbracket \mathtt{Top} \rrbracket_\ell & = \mathtt{Top} & \\ \llbracket X \rrbracket_\ell & = \#n & \text{if } n = posn(X, \ell) \\ \llbracket X \rrbracket_\ell & = X & \text{if } X \notin \ell \\ \llbracket T' \to T'' \rrbracket_\ell & = \llbracket T' \rrbracket_\ell \to \llbracket T'' \rrbracket_\ell & \\ \llbracket \forall X_{T'}.T'' \rrbracket_\ell & = \forall_{\llbracket T' \rrbracket_\ell}.\llbracket T'' \rrbracket_{X,\ell} & \end{cases}$$

$$\llbracket \Gamma \rrbracket = \begin{cases} \llbracket \emptyset \rrbracket & = \emptyset \\ \llbracket \Gamma', x : T \rrbracket & = \llbracket \Gamma' \rrbracket, x : \llbracket T \rrbracket \\ \llbracket \Gamma', X <: T \rrbracket & = \llbracket \Gamma' \rrbracket, X <: \llbracket T \rrbracket \end{cases}$$

where ℓ is a list of names used to trace non-locally bound variables. We will use the notation $|\ell|$ to indicate the length of list ℓ.

The encoding of a type is obtained beginning with ℓ being empty and it is denoted by $\llbracket \cdot \rrbracket$; the list is updated with a new variable whenever we enter the scope of a quantifier; the encoding $\llbracket X \rrbracket_\ell$ is X when $X \notin \ell$ (meaning X is a free variable); if $X \in \ell$, the encoding $\llbracket X \rrbracket_\ell$ is $\#n$, where n is the position of X in ℓ (meaning X

is a bound variable, and the "distance" of its binder is n); the encoding of a type commutes with all other constructs. The encoding of an environment is the same environment where every (sub)typing bound has been replaced by its encoding.

We can also show that this translation is surjective: for every locally closed type T in the locally nameless representation, there exists a type T' in the named variables representation, such that $\llbracket T' \rrbracket = T$.

The key lemmata we need to prove adequacy are the following:

Theorem 5.9 *For all types T and lists of variables ℓ_1,ℓ_2, if for all variables X, $X \in \ell_1 \iff X \in \ell_2$ and $X \in \ell_1 \Longrightarrow \mathtt{posn}(X, \ell_1) = \mathtt{posn}(X, \ell_2)$, then $\llbracket T \rrbracket_{\ell_1} = \llbracket T \rrbracket_{\ell_2}$.*

Proof: By structural induction on T. □

Theorem 5.10 *For all types T and variables X,Y, if $X \in \ell$ and if $Y \in \mathrm{FV}(T)$ implies $Y \in \ell$, then $\llbracket T \rrbracket_\ell = \llbracket (X\ Y) \cdot T \rrbracket_{(X\ Y)\cdot\ell}$.*

Proof: By structural induction on T. □

Theorem 5.11 *For all types T, lists of variables ℓ and natural numbers n, if $|\ell| \leq n$ then $\llbracket T \rrbracket_\ell \{^U/_{\#n}\} = \llbracket T \rrbracket_\ell$.*

Proof: By structural induction on T. □

Theorem 5.12 *For all types T, variables X and lists of variables ℓ, $\llbracket T \rrbracket_\ell = \llbracket T \rrbracket_{\ell,X} \{^X/_{\#|\ell|}\}$.*

Proof: By structural induction on T. □

Theorem 5.13 *For all variables X,Y and types T, if $X \in FV(T) \Rightarrow X = Y$, then $\llbracket (X\ Y) \cdot T \rrbracket = \llbracket T \rrbracket_Y \{^X/_{\#0}\}$.*

Proof: Actually, the theorem is obtained as a corollary from a stronger statement:

Given a list of variables ℓ, if X and Y are not in ℓ and $X \in \mathrm{FV}(T) \Rightarrow X = Y$, then $[\![(X\ Y) \cdot T]\!]_\ell = [\![T]\!]_{\ell,Y}\{^X/_{\#|\ell|}\}$.

Our proof is by induction on the *weight* of T, then by case analysis again on T. The weight of T is defined as follows:

$$\|T\| = \begin{cases} \|X\| &= 0 \\ \|\mathtt{Top}\| &= 0 \\ \|U \to V\| &= \max(\|U\|, \|V\|) + 1 \\ \|\forall X <: U.V\| &= \max(\|U\|, \|V\|) + 1 \end{cases}$$

If $T = Z$ for some variable Z, consider the cases

- $X = Z$: by hypothesis we also know $X = Y$. Then we must prove:

  $$[\![X]\!]_\ell = [\![X]\!]_{\ell,X}\{^X/_{\#|\ell|}\}$$

  Since $X \notin \ell$, this is equivalent to

  $$X = \#|\ell|\{^X/_{\#|\ell|}\}$$

  which is trivial.

- $X \neq Z$, $Y = Z$. We must prove:

  $$[\![(X\ Y) \cdot Y]\!]_\ell = [\![Y]\!]_{\ell,Y}\{^X/_{\#|\ell|}\}$$

  Since $X \notin \ell$ and $Y \notin \ell$, this is equivalent to

  $$X = \#|\ell|\{^X/_{\#|\ell|}\}$$

  which is trivial.

- $X \neq Z$ and $Y \neq Z$. Then we must prove:

  $$[\![Z]\!]_\ell = [\![Z]\!]_{\ell,Y}\{^X/_{\#|\ell|}\}$$

  Considering the cases $Z \in \ell$ or $Z \notin \ell$, we can conclude that the two sides are identical.

If $T = \forall Z_U.V$, we must prove

$$\forall_{\llbracket (X\ Y)\cdot U \rrbracket_\ell}.\llbracket (X\ Y)\cdot V\rrbracket_{(X\ Y)\cdot Z,\ell} = \forall_{\llbracket U\rrbracket_{\ell,Y}\{X/\#|\ell|\}}.\llbracket V\rrbracket_{Z,\ell,Y}\{X/\#|\ell|+1\}$$

Since the sources are equal by induction hypothesis, its sufficient to prove that the targets are the same:

$$\llbracket (X\ Y)\cdot V\rrbracket_{(X\ Y)\cdot Z,\ell} = \llbracket V\rrbracket_{Z,\ell,Y}\{X/\#|\ell|+1\}$$

We consider the following cases:

- $Y = Z$: we must prove

  $$\llbracket (X\ Y)\cdot V\rrbracket_{(X\ Y)\cdot Y,\ell} = \llbracket V\rrbracket_{Y,\ell,Y}\{X/\#|\ell|+1\}$$

  or equivalently

  $$\llbracket (X\ Y)\cdot V\rrbracket_{(X\ Y)\cdot (Y,\ell)} = \llbracket V\rrbracket_{Y,\ell,Y}\{X/\#|\ell|+1\}$$

  By theorem 5.9, we rewrite the right-hand side as $\llbracket V\rrbracket_{Y,\ell}\{\#|\ell|+1/X\}$, and by theorem 5.11 as $\llbracket V\rrbracket_{Y,\ell}$. Therefore, we only need to show that

  $$\llbracket (X\ Y)\cdot V\rrbracket_{(X\ Y)\cdot (Y,\ell)} = \llbracket V\rrbracket_{Y,\ell,Y}$$

  which is obtained by theorem 5.10.

- $Y \neq Z$ and $X = Z$: we must prove

  $$\llbracket (X\ Y)\cdot V\rrbracket_{(X\ Y)\cdot X,\ell} = \llbracket V\rrbracket_{X,\ell,Y}\{X/\#|\ell|+1\}$$

  or equivalently

  $$\llbracket (X\ Y)\cdot V\rrbracket_{(X\ Y)\cdot (X,\ell)} = \llbracket V\rrbracket_{X,\ell,Y}\{X/\#|\ell|+1\}$$

  We rewrite the right-hand side by theorem 5.10, yielding the equation

  $$\llbracket (X\ Y)\cdot V\rrbracket_{(X\ Y)\cdot (X,\ell)} = \llbracket (X\ Y)\cdot V\rrbracket_{(X\ Y)\cdot (X,\ell),X}\{X/\#|\ell|+1\}$$

  This is obtained by lemma 5.12.

- $Y \neq Z$ and $X \neq Z$: we must prove

$$[\![(X\ Y) \cdot V]\!]_{Z,\ell} = [\![V]\!]_{Z,\ell,Y}\{^{X}/_{\#|\ell|+1}\}$$

This is proved by induction hypothesis.

□

We are now able to prove the adequacy theorem.

Theorem 5.14 (adequacy and faithfulness)

1. *Let* Γ *be a typing environment,* T, U *types in the named presentation. If* $\Gamma \vdash T <: U$*, then* $[\![\Gamma]\!] \vdash [\![T]\!] <: [\![U]\!]$*.*

2. *Let* Γ *be a typing environment,* T, U *types in the locally nameless encoding. Let* Γ', T', U' *such that* $\Gamma = [\![\Gamma']\!]$*,* $T = [\![T']\!]$ *and* $U = [\![U']\!]$*. If* $\Gamma \vdash T <: U$*, then* $\Gamma' \vdash T' <: U'$*.*

Proof: Adequacy is proved by a straightforward induction on the derivation of $\Gamma \vdash T <: U$. Almost all the cases are easy (and are proved automatically by Matita), except for the "for all" case, requiring us to prove that

$$[\![\Gamma]\!] \vdash \forall_{[\![S_1]\!]}.[\![S_2]\!]_X <: \forall_{[\![T_1]\!]}.[\![T_2]\!]_Y$$

under the following induction hypotheses

$$\begin{aligned}
IH_1 :&\ [\![\Gamma]\!] \vdash [\![T_1]\!] <: [\![S_1]\!] \\
IH_2 :&\ \text{for all } Z \notin \mathrm{FV}(\Gamma): \\
&\ (Z \in \mathrm{FV}(S_2) \Rightarrow Z = X) \Rightarrow \\
&\ (Z \in \mathrm{FV}(T_2) \Rightarrow Z = Y) \Rightarrow \\
&\ [\![\Gamma]\!], Z <: [\![T_1]\!] \vdash [\![(Z\ X) \cdot S_2]\!] <: [\![(Z\ Y) \cdot T_2]\!]
\end{aligned}$$

By SA-All and IH_1, we reduce to the problem of proving

$$\text{for all } W \notin \mathrm{FV}(\Gamma):\ [\![\Gamma]\!], W <: [\![T_1]\!] \vdash [\![S_2]\!]_X\{^{W}/_{\#0}\} <: [\![T_2]\!]_Y\{^{W}/_{\#0}\}$$

This follows easily from IH_2 by means of theorem 5.13.

The proof of faithfulness basically mirrors that of adequacy and is performed by providing an algorithm to compute the backwards encoding of a locally closed type. □

5.4 Adequacy of strong typing rules

While the strong formulation of typing rules we used in the locally nameless and in the named formalization is intuitively sound, we should be aware that such rules construct, basically, infinitely wide proof trees (where an occurrence of a rule containing a universally quantified premise can be intended as having a different premise for each possible choice of names). This kind of infinitary structure cannot be regarded as *real* syntax, so we still want to provide evidence of its adequacy.

In this section we prove (by the standard means of equivariance lemmata) that a weak formulation of the named typing rules (using existentially quantified premises) is equivalent to the strong one we used in the formalization. By Theorem 5.14 we subsume the property that the strong typing rules used in the locally nameless formalization are also adequate.

The weak type system is obtained replacing the WFT-Forall and SA-All rules by the following ones:

$$\frac{Y \notin dom(\Gamma) \qquad Y \in \mathrm{FV}(U) \Rightarrow Y = X \qquad \Gamma \vdash T \qquad \Gamma, Y <: T \vdash (Y\ X) \cdot U}{\Gamma \vdash \forall X_T.U} \text{ (WFT-Forall-W)}$$

$$\frac{Z \in \mathrm{FV}(S_2) \Rightarrow Z = X \qquad Z \in \mathrm{FV}(T_2) \Rightarrow Z = Y \qquad \Gamma \vdash T_1 <: S_1 \qquad \Gamma, Z <: T_1 \vdash (Z\ X) \cdot S_2 <: (Z\ Y) \cdot T_2}{\Gamma \vdash \forall X_{S_1}.S_2 <: \forall Y_{T_1}.T_2} \text{ (SA-All-W)}$$

In the rest of the section we will use the symbols $\vdash_w$ and $\vdash_s$ to distinguish weak and strong judgments. We will also abuse notation a little, using $(X\ Y) \cdot \Gamma$ to denote the context resulting from the application of the $(X\ Y)$ swap to both the domain and the codomain of Γ.

Lemma 5.15

1. $\Gamma \vdash \diamond \Rightarrow (X\ Y) \cdot \Gamma \vdash \diamond$
2. $\Gamma \vdash_s T \Rightarrow (X\ Y) \cdot \Gamma \vdash_s (X\ Y) \cdot T$

3. $\Gamma \vdash_s T <: U \Rightarrow (X\ Y) \cdot \Gamma \vdash_s (X\ Y) \cdot T <: (X\ Y) \cdot U$

Proof: All the proofs are by structural induction on the given judgment. While we are mostly interested in proving equivariance for the subtyping judgment, this proof exploits equivariance of well-formedness of contexts and types in cases SA-Top and SA-Refl-TVar. Here we only show the proof of most complicate case of part 3, involving rule SA-All.

We want to prove

$$(X\ Y) \cdot \Gamma \vdash_s (X\ Y) \cdot (\forall X'_{S_1}.S_2) <: \forall (X\ Y) \cdot (\forall Y'_{T_1}.T_2)$$

or equivalently, by the definition of swaps

$$(X\ Y) \cdot \Gamma \vdash_s \forall (X\ Y) \cdot X'_{(X\ Y)\cdot S_1}.(X\ Y) \cdot S_2 <: \forall (X\ Y) \cdot Y'_{(X\ Y)\cdot T_1}.(X\ Y) \cdot T_2$$

By induction hypotheses we know that

$$\begin{array}{ll} IH_1: & (X\ Y) \cdot \Gamma \vdash_s (X\ Y) \cdot T_1 <: (X\ Y) \cdot S_1 \\ IH_2: & \text{for all } Z' \notin dom(\Gamma): \\ & \quad (Z' \in \mathrm{FV}(S_2) \Rightarrow Z' = X) \Rightarrow \\ & \quad (Z' \in \mathrm{FV}(T_2) \Rightarrow Z' = Y) \Rightarrow \\ & \quad (X\ Y) \cdot (\Gamma, Z' <: T_1) \vdash_s (X\ Y) \cdot (Z'\ X') \cdot S_2 <: (X\ Y) \cdot (Z'\ Y') \cdot T_2 \end{array}$$

After applying rule SA-All to the thesis and using hypothesis IH_1, we are left with the goal

$$\begin{array}{l} (X\ Y) \cdot \Gamma, Z <: (X\ Y) \cdot T_1 \vdash_s \\ \quad (Z\ (X\ Y) \cdot X') \cdot (X\ Y) \cdot S_2 <: (Z\ (X\ Y) \cdot Y') \cdot (X\ Y) \cdot T_2 \end{array}$$

for some Z such that $Z \notin dom((X\ Y) \cdot \Gamma)$, $Z \in \mathrm{FV}((X\ Y) \cdot S_2) \Rightarrow Z = (X\ Y) \cdot X'$ and $Z \in \mathrm{FV}((X\ Y) \cdot T_2) \Rightarrow Z = (X\ Y) \cdot Y'$. Then, by the properties of permutations, we rewrite IH_2 as:

$$\begin{array}{l} \text{for all } Z' \notin dom(\Gamma): \\ \quad (Z' \in \mathrm{FV}(S_2) \Rightarrow Z' = X') \Rightarrow \\ \quad (Z' \in \mathrm{FV}(T_2) \Rightarrow Z' = Y') \Rightarrow \\ \quad (X\ Y) \cdot \Gamma, (X\ Y) \cdot Z' <: (X\ Y) \cdot T_1 \vdash_s \\ \qquad ((X\ Y) \cdot Z'\ (X\ Y) \cdot X') \cdot (X\ Y) \cdot S_2 <: ((X\ Y) \cdot Z'\ (X\ Y) \cdot Y') \cdot (X\ Y) \cdot T_2 \end{array}$$

By taking $Z' = (X\ Y)\cdot Z$ and remembering that $(X\ Y)\cdot(X\ Y)\cdot Z = Z$, we prove the goal simply applying IH_2: in fact, we can prove easily that the freshness conditions on Z also imply the side conditions of IH_2 (instantiated on $(XY)\cdot Z$):

$$\begin{aligned}&(X\ Y)\cdot Z \notin dom(\Gamma)\\ &(X\ Y)\cdot Z \in \mathrm{FV}(S_2) \Rightarrow (X\ Y)\cdot Z = X'\\ &(X\ Y)\cdot Z \in \mathrm{FV}(T_2) \Rightarrow (X\ Y)\cdot Z = Y'\end{aligned}$$

□

Theorem 5.16

1. $\Gamma \vdash_w T \iff \Gamma \vdash_s T$
2. $\Gamma \vdash_w T <: U \iff \Gamma \vdash_s T <: U$

Proof: The proofs are by structural induction on the derivation of the given judgment. Here we only discuss the "for all" case of subtyping judgments.

$\Longleftarrow$: The premises of the weak typing judgment are a particular case of those of the strong typing judgment, making this direction trivial.

$\Longrightarrow$: Given Z such that $Z \notin dom(\Gamma)$, $Z \in \mathrm{FV}(S_2) \Longrightarrow Z = X$ and $Z \in FV(T_2) \Longrightarrow Z = Y$, we know by induction hypothesis that

$$\begin{aligned}IH_1 :&\ \ \Gamma \vdash_s T_1 <: S_1\\ IH_2 :&\ \ \Gamma, Z <: T_1 \vdash_s (Z\ X)\cdot S_2 <: (Z\ Y)\cdot T_2\end{aligned}$$

We want to prove that

$$\Gamma \vdash_s \forall X_{S_1}.S_2 <: \forall Y_{T_1}.T_2$$

By rule SA-All, also using hypothesis IH_1, we only have to prove that

$$\text{for all } Z' \notin dom(\Gamma): \left(\begin{array}{l}(Z' \in \mathrm{FV}(S_2) \Rightarrow Z' = X) \Rightarrow\\ (Z' \in \mathrm{FV}(T_2) \Rightarrow Z' = Y) \Rightarrow\\ \Gamma, Z' <: T_1 \vdash (Z'\ X)\cdot S_2 <: (Z'\ Y)\cdot T_2\end{array}\right)$$

We assume such a Z' and then choose a variable $W \notin dom(\Gamma)$: by equivariance, IH_2 implies

$$IH_2' : \quad (W\ Z) \cdot (\Gamma, Z <: T_1) \vdash_s (W\ Z) \cdot (Z\ X) \cdot S_2 <: (W\ Z) \cdot (Z\ Y) \cdot T_2$$

Furthermore, also by equivariance, the thesis is equivalent to:

$$(W\ Z') \cdot (\Gamma, Z' <: T_1) \vdash_s (W\ Z') \cdot (Z'\ X) \cdot S_2 <: (W\ Z) \cdot (Z'\ Y) \cdot T_2$$

Knowing Z, Z', W are all disjoint from $dom(\Gamma)$ (therefore $Z, Z', W \notin \mathrm{FV}(T_2)$, too), we prove

$$(W\ Z) \cdot (\Gamma, Z <: T_1) = (W\ Z') \cdot (\Gamma, Z' <: T_1) = \Gamma, W <: T_1$$

By the freshness properties of Z and Z', we can also prove that

$$(W\ Z) \cdot (Z\ X) \cdot S_2 = (W\ Z') \cdot (Z'\ X) \cdot S_2 = (W\ X) \cdot S_2$$
$$(W\ Z) \cdot (Z\ Y) \cdot T_2 = (W\ Z') \cdot (Z'\ Y) \cdot T_2 = (W\ Y) \cdot S_2$$

Thus, rewriting appropriately, we prove the thesis by IH_2'. □

5.5 Conclusions

The POPLmark challenge proved to be a valuable test-bench for the Matita theorem prover. Remarkably, it allowed us to identify a new proof principle that we called *induction/inversion* and that we implemented in Matita. The principle seems to have been implicitly adopted in several solutions (see for example [12, 29]) but never made explicit before. We believe that our proof where it is explicit is not only easier to understand, but also more faithful to the informal proof of the POPLmark specification.

Our de Bruijn solution bears some similarities with Maggesi and Hirschowitz's solution based on nested datatypes [29]: in particular our notion of environment extension map is comparable to their "relative well-formedness" predicate. Still, the structures used by the two solutions are very different (nested abstract syntax uses

dependently typed structures and formalizes contexts as functions, while we went for a more standard approach). As concerns size, our de Bruijn solution has not been tuned for compactness and is still relatively lengthy (with a size comparable to that of the second locally nameless solution): in general, de Bruijn formalizations have proved to be more synthetic than locally nameless ones, and we believe that automation might help reducing its size significantly.

While the solutions using the de Bruijn or the locally nameless encoding are compact (576 and 350 lines respectively), the solution using named variable requires 1270 lines. However, it is not really comparable to the other ones, since it is based on a completely different proof strategy.

As for the encoding issue, as already pointed out by other authors, we agree that the locally nameless approach leads to proofs which are more readable (in comparison with de Bruijn's representation): this is due to the fact that we do not have to deal with free indices. We also believe that while it may be possible to obtain a smaller solution using a pure de Bruijn approach, the proofs tend to become much less linear, making the locally nameless approach still preferable.

The biggest drawback with the locally nameless approach is that typing rules which deal with binders are required (reasoning backwards) to make indices disappear, in such a way that they are not fully structural on the types. This means that where the paper proof would use a straight induction on a type, we are required to use an induction on its well-formedness derivation. For the same reasons, the cases with binders are also the most difficult to deal with in the adequacy proof of the named variables encoding with respect to the locally nameless encoding.

Chapter 6

Canonical locally named encoding

In the previous chapter we have seen three representations of binding: among them, the locally nameless approach has the notable feature of making a syntactical distinction between global and local variables, which is a simple but effective strategy to avoid variable capture when substituting in a term. Because of this feature, we say that the locally nameless approach is a local representation of binding.

Local representations of binding can be traced back to Frege: in [19], he showed how to formulate the syntax of a logical language with binders using two distinct sets of names for global variables (represented by Latin letters) and local variables (represented by German letters). This approach, which we could call *locally named*, was studied in the context of machine formalization of type theory by McKinna and Pollack ([41, 42]). Needless to say, using names instead of indices allows one to build infinitely many α-equivalent (but not syntactically equal) formulations of any term containing binders, unless we can make some stronger assumption on the way terms are built: we say that this representation is not *canonical*.

The present chapter describes a refinement of the locally named approach, introduced in [61, 62] by Sato and Pollack, where α-equivalent terms must also be syntactically equal, while retaining good algebraic properties. This approach was motivated by earlier works by Sato studying the semantics of binding structures ([57, 58, 59, 56, 60]).

The contents of this chapter are based on a paper we wrote with Pollack and Sato ([51]), with added emphasis on our contribution (the extension of the Sato representation to languages allowing multiple binding and simultaneous substitutions and its use in a concrete formalization of the multivariate λ-calculus). This is preceded by an introduction to the Sato representation described in the simpler context of pure λ-calculus.

6.1 Symbolic expressions

In this section we recall the basis for defining the syntax of the λ-calculus in the Sato encoding. We start with two distinct, denumerably infinite sets of atoms: the set

$\mathbb{X}$ of global variables (or parameters), whose elements will be denoted $X, Y, Z, \ldots$, and the set $\mathbb{V}$ of local variables, denoted $x, y, z, \ldots$. Even though this is not strictly needed for the Sato encoding, we will identify local variables with natural numbers ($\mathbb{V} = \mathbb{N}$); for global variables, instead, it will be sufficient to assume that they have a decidable equality. $\overline{\mathbb{X}}$ and $\overline{\mathbb{V}}$ will represent the sets of lists of global and local variables, respectively. When we are not interested in the order of items in a list, or in the multiplicity of the items, but only in membership, we will refer to them as (finite) sets and abuse the notation, employing set operators to manipulate them.

The set of symbolic expressions for the λ-calculus, denoted $\mathbb{S}$, is the set of expressions generated by the following rules:

$$\frac{}{X : \mathbb{S}} \qquad \frac{}{x : \mathbb{S}}$$

$$\frac{M : \mathbb{S} \qquad N : \mathbb{S}}{(M\ N) : \mathbb{S}} \qquad \frac{M : \mathbb{S}}{[x]\ M : \mathbb{S}}$$

where we regard $(M\ N)$ as the application of M to N , and $[x]\ M$ as the abstraction binding the local variable x in the symbolic expression M. Informally, we consider the occurrences of some local variable to be bound by the nearest abstraction binding the same name; however, notice that there is no actual binding explicit in this free construction: in particular, we still regard expressions like $[x]\ x$ and $[y]\ y$ as syntactically different, if $x \neq y$, even though they are, in a sense, α-convertible.

We define two operations collecting the sets of local and global variables used in a symbolic expression

$$\begin{aligned}
\mathsf{LV}(X) &\triangleq \emptyset \\
\mathsf{LV}(x) &\triangleq \{x\} \\
\mathsf{LV}(M\ N) &\triangleq \mathsf{LV}(M) \cup \mathsf{LV}(N) \\
\mathsf{LV}([x]\ M) &\triangleq \mathsf{LV}(M) \setminus \{x\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{GV}(X) &\triangleq \{X\} \\
\mathsf{GV}(x) &\triangleq \emptyset \\
\mathsf{GV}(M\ N) &\triangleq \mathsf{GV}(M) \cup \mathsf{GV}(N) \\
\mathsf{GV}([x]\ M) &\triangleq \mathsf{GV}(M)
\end{aligned}$$

We will borrow the notation $X \# M_1, \ldots, M_n$ from nominal logic to mean that $X \notin \mathsf{GV}(M_1) \cup \ldots \cup \mathsf{GV}(M_n)$.

The two distinct notions of variables prompt us to also define two substitution operations, one for substituting parameters, the other for substituting local variables.

Definition 6.1 *The substitution of a symbolic expression N for a global variable X in a symbolic expression M, denoted $M\{^N/_X\}$ is defined as follows*

$$\begin{array}{rcl} Y\{^N/_X\} & \triangleq & \begin{cases} N & \text{if } X = Y \\ Y & \text{if } X \neq Y \end{cases} \\ x\{^N/_X\} & \triangleq & x \\ (M\ P)\{^N/_X\} & \triangleq & (M\{^N/_X\}\ P\{^N/_X\}) \\ (\texttt{[}x\texttt{]}\ M)\{^N/_X\} & \triangleq & \texttt{[}x\texttt{]}\ (M\{^N/_X\}) \end{array}$$

Definition 6.2 *The substitution of a symbolic expression N for a local variable x in a symbolic expression M, denoted $M\{^N/_x\}$ is defined as follows*

$$\begin{array}{rcl} X\{^N/_x\} & \triangleq & X \\ y\{^N/_x\} & \triangleq & \begin{cases} N & \text{if } x = y \\ y & \text{if } x \neq y \end{cases} \\ (M\ P)\{^N/_x\} & \triangleq & (M\{^N/_x\}\ P\{^N/_x\}) \\ (\texttt{[}y\texttt{]}\ M)\{^N/_x\} & \triangleq & \begin{cases} \texttt{[}y\texttt{]}\ M & \text{if } x = y \\ \texttt{[}y\texttt{]}\ (M\{^N/_x\}) & \text{if } x \neq y \end{cases} \end{array}$$

Finally, given a symbolic expression M and a global variable X, we define an operation collecting all the local variables that are bound on the path between the root of M and any occurrence of X in M. We will use this operation to state some properties of the theory of the Sato encoding of binding.

$$\frac{}{\texttt{vclosed}\ X} \qquad \frac{\texttt{vclosed}\ M \qquad \texttt{vclosed}\ N}{\texttt{vclosed}\ (M\ N)} \qquad \frac{\texttt{vclosed}\ M}{\texttt{vclosed}\ (\texttt{[}x\texttt{]}\ (M\{x/x\}))}$$

Table 6.1: `vclosed` predicate for the λ-calculus (Sato)

Definition 6.3 *The function* $\mathsf{E} : \mathbb{X} \to \mathbb{S} \to \overline{\mathbb{V}}$ *is defined as follows:*

$$\begin{aligned}
\mathsf{E}_X(Y) &\triangleq \emptyset \\
\mathsf{E}_X(x) &\triangleq \emptyset \\
\mathsf{E}_X(M\ N) &\triangleq \mathsf{E}_X(M) \cup \mathsf{E}_X(N) \\
\mathsf{E}_X(\texttt{[}x\texttt{]}\ M) &\triangleq \begin{cases} \emptyset & \textit{if } X \# M \\ \{x\} \cup \mathsf{E}_X(M) & \textit{else} \end{cases}
\end{aligned}$$

6.2 Well-formed lambda terms

As the reader should have already noticed, symbolic expressions are not a good representation for λ-terms: in fact, it is clear that there are more symbolic expressions than λ-terms. For example, expressions like $\texttt{[}x\texttt{]}\ (x\ y)$, where $x \neq y$, do not represent any λ-term: this is because that expression contains a *dangling local variable*, i.e. a local variable that is not bound by any abstraction. This phenomenon is not different from what happens in other local representations of binding like the locally nameless encoding: in both the locally nameless encoding and the Sato approach, it is necessary to identify a subset of the raw expressions that is *variable closed* ([41, 42]), where by "variable closed" we mean that its elements do not contain dangling local variables. In the Sato representation, this subset can be identified by the inductive predicate whose rules we summarize in Table 6.1.

Notice that substitution for global variables does not prevent variable capture in the case of raw symbolic expressions; but it is, as expected, capture avoiding when restricted to variable closed expressions – quite unsurprisingly, since there is no dangling local variable to possibly capture. Variable closed expressions are, in fact, an adequate representation for the λ-calculus. However they are still not completely satisfying, since they are not canonical. What we would like to have is a

$$\frac{}{X : \mathbb{L}}\ (\text{L-VAR}) \quad \frac{M : \mathbb{L} \qquad N : \mathbb{L}}{(M\ N) : \mathbb{L}}\ (\text{L-APP}) \quad \frac{M : \mathbb{L} \qquad x = \mathsf{F}_X(M)}{([x]\ (M\{x/X\}))}\ (\text{L-ABS})$$

Table 6.2: $\mathbb{L}$ predicate for the λ-calculus

representation of binding ensuring that α-convertible terms are really syntactically equal. This is usually obtained, as in the locally nameless representation, by throwing away local names altogether, replacing them by nameless dummies referencing an abstraction by index rather than by name. Instead, in the Sato approach, we choose to keep the local names; canonicity will be ensured by giving a rule for choosing names to be bound: in other words, we further refine the set of variable closed symbolic expressions by keeping one and only one representative for each class of α-equivalent terms.

Concretely, this rule is expressed by a *height* function $\mathsf{F} : \mathbb{X} \to \mathbb{S} \to \mathbb{V}$: each time it is necessary to replace a parameter X in a symbolic expression M with some local variable (as it is the case when we must construct an abstraction in a bottom-up fashion), that local variable will be equal to $\mathsf{F}_X(M)$. We will call the subset of symbolic expressions built according to this rule the set of *well-formed terms*, or simply λ-terms. This set is defined by an inductive predicate $- : \mathbb{L}$ whose rules are summarized in Table 6.2.

We stress the fact that rule L-ABS, intended as a constructor, takes two arguments X and M and builds an abstraction whose abstracted local variable is uniquely determined. In fact, we think that this operation, closely mimicking the informal syntax of the λ-calculus, is particularly meaningful in itself: we will thus provide the notation

$$\mathsf{abs}_X M \triangleq [\mathsf{F}_X(M)]\ (M\{^{\mathsf{F}_X(M)}/_X\})$$

that allows us to rewrite rule L-ABS in a particularly readable form:

$$\frac{M : \mathbb{L}}{\mathsf{abs}_X M : \mathbb{L}}\ (\text{L-ABS})$$

The dual operation of *instantiation* of an abstraction (denoted ▼) is defined as

$$[x]\ M \blacktriangledown N \triangleq M\{^{N}/_{x}\}$$

Of course not any function of the proper type will yield a good height, resulting in an adequate representation of the λ-calculus. In the next section, we will concretely define the function F and state the three key properties that guarantee the adequacy of the Sato representation.

6.2.1 An excellent height function

While it is trivial to see that λ-expressions as we defined them are a subset of the variable closed expressions, we still have not attained our goal of defining a set of canonical expressions. Indeed, if we take our height function to be the following (rather pathological) F'

$$\mathsf{F}'_X(M) \triangleq \begin{cases} 0 & \text{if } X = Y \\ 1 & \text{else} \end{cases}$$

where Y is a fixed global variable, we can easily build α-equivalent λ-terms that are syntactically different: for example, supposing $Y \neq Z$, we have

$$[0]\ 0 = \mathsf{abs}_Y Y \neq \mathsf{abs}_Z Z = [1]\ 1$$

Furthermore, F' does not prevent variable capture during the construction of an abstraction: if Z_1 and Z_2 are not equal to Y, then

$$\mathsf{abs}_{Z_1}(\mathsf{abs}_{Z_2}(Z_1\ Z_2)) = [1]\,([1]\,(1\ 1))$$

with the occurrence of Z_1 captured by the inner abstraction. This implies that the expected equation

$$\mathsf{abs}_X\ M \blacktriangledown N = M\{^{N}/_{X}\}$$

does not hold for all X, M, N.

Also importantly, while we know that variable closed expressions are closed with respect to substitution for global variables, it is not clear that the same property holds for λ-terms. Indeed, we can prove the opposite using the following height:

$$\begin{array}{rcll} \mathsf{F''}_X(M\ N) & = & \mathsf{F''}_X(M) + \mathsf{F''}_X(N) + 1 & \\ \mathsf{F''}_X(M) & = & 0 & \text{if } M \text{ is not an application} \end{array}$$

For distinct X, Y, Z, take

$$M \triangleq \mathsf{abs}_Y(X\ Y) = \texttt{[1]}\ (X\ 1) : \mathbb{L}$$

and then consider $M\{^{(Z\ Z)}/_X\} = \texttt{[1]}\ ((Z\ Z)\ 1)$. For this term to be well formed, we should be able to express it as $\mathsf{abs}_{Y'}((Z\ Z)\ Y')$: however the result of this operation is $\texttt{[2]}\,((Z\ Z)\ 2)$ for all Y'. This means that if we choose $\mathsf{F''}$ as our height, well-formed terms are not closed with respect to substitution for global parameters.

In the following definition of *good* height functions, we will summarize a set of properties that, together, guarantee that the Sato encoding is well behaved, excluding the bad behaviours we just mentioned. In the rest of the chapter, we will often reference *finite permutations*, denoted $\pi, \pi', \ldots$. A finite permutation on a set S is a bijection $\pi : S \to S$ such that $\pi(s) \neq s$ only for finitely many $s \in S$. We will also write $\tilde{\pi}$ for the inverse permutation of π.

Given a finite permutation π on $\mathbb{X}$, we can map it to symbolic expressions as follows:

$$\begin{array}{rcl} \pi \cdot X & \triangleq & \pi(X) \\ \pi \cdot x & \triangleq & x \\ \pi \cdot (M\ N) & \triangleq & (\pi \cdot M\ \pi \cdot N) \\ \pi \cdot \texttt{[}x\texttt{]}\ M & \triangleq & \texttt{[}x\texttt{]}\ (\pi \cdot M) \end{array}$$

Definition 6.4 *A function* $\mathsf{H} : \mathbb{X} \to \mathbb{S} \to \mathbb{V}$ *is a* good *height for the* λ*-calculus if the following three properties hold:*

(HE) H *is* equivariant*: for all finite permutations* π*, global variables* X*,* λ*-terms* M*,* $\mathsf{H}_X(M) = \mathsf{H}_{\pi(X)}(\pi \cdot M)$*;*

(HF) H *is* fresh*: for all global variables* X *and* λ*-terms* M*,* $\mathsf{H}_X(M) \notin \mathsf{E}_X(M)$*;*

(HP) H *preserves substitution: for all global variables* X, Y*,* λ*-terms* M, Q*, if* $X \neq Y$ *and* $X \# Q$*, then* $\mathsf{H}_X M = \mathsf{H}_X(M\{^Q/_Y\})$

While good heights are sufficient to ensure adequacy, the notion of good height function can be further relaxed so that its properties are not limited to λ-terms, but extended to symbolic expressions: we call such height functions *excellent*.

Definition 6.5 *A function* $\mathsf{H} : \mathbb{X} \to \mathbb{S} \to \mathbb{V}$ *is an* excellent *height for the* λ*-calculus if the following three properties hold:*

(XHE) H *is* equivariant*: for all finite permutations* π*, global variables* X*, symbolic expressions* M*,* $\mathsf{H}_X(M) = \mathsf{H}_{\pi(X)}(\pi \cdot M)$*;*

(XHF) H *is* fresh*: for all global variables* X *and symbolic expressions* M*,* $\mathsf{H}_X(M) \notin \mathsf{E}_X(M)$*;*

(XHP) H *preserves substitution: for all global variables* X, Y*, symbolic expressions* M, Q*, if* $X \neq Y$ *and* $X \# Q$*, then* $\mathsf{H}_X(M) = \mathsf{H}_X(M\{^Q/_Y\})$

We are now ready to show that excellent height functions exist by concretely defining the F function we intend to use.

Definition 6.6 *The height function* $\mathsf{F} : \mathbb{X} \to \mathbb{S} \to \mathbb{V}$ *is defined as follows:*

$$
\begin{aligned}
\mathsf{F}_X(Y) &\triangleq \begin{cases} 1 & \text{if } X = Y \\ 0 & \text{else} \end{cases} \\
\mathsf{F}_X(x) &\triangleq 0 \\
\mathsf{F}_X(M\ N) &\triangleq \max\{\mathsf{F}_X(M), \mathsf{F}_X(N)\} \\
\mathsf{F}_X([x]\ M) &\triangleq \begin{cases} \mathsf{F}_X(M) & \text{if } \mathsf{F}_X(M) = 0 \text{ or } \mathsf{F}_X(M) > x \\ x + 1 & \text{else} \end{cases}
\end{aligned}
$$

Theorem 6.1 *The* F *function of definition 6.6 is excellent.*

Proof: We will not go into the details of the proof thoroughly (similar, though more involved, proof will be given in the next pages), but will briefly sketch the most important issues.

(XHE) The proof that for all finite permutations π, $\mathsf{F}_X(M) = \mathsf{F}_{\pi\cdot X}(\pi \cdot M)$ follows easily by structural induction on M. In the case $M = [x]\, N$, where we know by induction hypothesis that $\mathsf{F}_X(N) = \mathsf{F}_{\pi\cdot X}(\pi \cdot N)$, consider two subcases, depending on whether $0 < \mathsf{F}_X(N) = \mathsf{F}_{\pi\cdot X}(\pi \cdot N) \leq x$ or not.

(XHF) We actually prove a stronger property: for all local variables $x \geq \mathsf{F}_X(M)$, $x \notin \mathsf{E}_X(M)$. This is obtained by structural induction on M. In the case $M = [y](N)$, we only consider the case in which $X \in GV(N)$ (since, otherwise, $\mathsf{E}_X([y]\, N) = \emptyset$). This implies $\mathsf{F}_X(N) > 0$. Then, proceed by cases, depending on whether $\mathsf{F}_X(N) \leq y$ or $\mathsf{F}_X(N) > y$: the thesis follows easily by the induction hypothesis.

(XHP) The proof that $X \neq Y$ and $X \# Q$ imply $\mathsf{F}_X(M) = \mathsf{F}_X(M\{^Q/_Y\})$ is easy by structural induction on M. If $M = Z$, consider two subcases for $Z = Y$ and $Z \neq Y$ (also remember that $X \# Q$ implies $\mathsf{F}_X(Q) = 0$).

□

As we were saying, good or excellent heights guarantee that the Sato representation is an adequate encoding of the λ-calculus. **(XHE)** implies that heights do not depend on the specific global variables used to construct a term (thus $\mathsf{abs}_Y\, Y = \mathsf{abs}_Z\, Z$ for all Y, Z); **(XHF)** prevents variable capture; **(XHP)** allows us to prove that substitution for global variables preserves well-formedness.

We will now mention some more general properties that, assuming that we use a good height function, we can prove for the Sato representation. The following property mimics α-conversion.

Lemma 6.2 *If $M : \mathbb{L}$ and $Y \# \mathsf{abs}_X\, M$, there exists a term N such that $N : \mathbb{L}$ and $\mathsf{abs}_X\, M = \mathsf{abs}_Y\, N$.*

Furthermore, as we already said, substitution for global variables is well behaved:

Lemma 6.3 *If $M : \mathbb{L}$ and $N : \mathbb{L}$, then $M\{^N/_X\} : \mathbb{L}$.*

Finally, instantiation also preserves well-formedness:

Lemma 6.4 *If* $\mathsf{abs}_X\ M : \mathbb{L}$ *and* $N : \mathbb{L}$*, then* $\mathsf{abs}_X\ M \blacktriangledown N : \mathbb{L}$*.*

As a final remark, let us mention that good heights are sufficient to prove that the Sato representation is adequate with respect to the nominal representation commonly used in Nominal Isabelle [64].

6.3 Multiple binding

The style we have just seen is suitable for representing more complicated languages than the pure λ-calculus with minor modifications: for example, we can easily think of rather obvious extensions of the Sato representation to typed λ-calculi with binding both at the term and at the type level; in languages involving different sorts of variables, it is possible to use a different height for each sort. As long as the binding structures are limited to the case of a single bound variable, we believe that the Sato representation can be extended with very little effort.

In many languages (including most real world programming languages), however, *multiple* binding – i.e. the use of single syntactic constructs to abstract an arbitrary number of variables at the same time – is the norm. Such constructs lift the complexity of formalization one level up, therefore it is a good idea to see how difficult is to adapt the Sato representation to this more involved case. In the rest of the chapter, we present a formalization of one of the simplest languages involving multi-binders.

6.3.1 The multivariate λ-calculus

If we have to think of a toy language allowing multiple variables to be bound at the same time, probably we will come up with a variation of the pure λ-calculus, where λ's abstract a list of variables instead of a single variable, as shown by the following informal grammar:

$$M, N ::= x \mid (M\ N) \mid \lambda \overline{x}.M$$

As always, we write $(M\ N_1 \cdots N_k)$ as syntactic sugar for $(\cdots (M\ N_1) \cdots N_k)$. Notice that, according to this grammar, multiple abstraction and iterated abstraction are not the same thing: the abstract syntax tree of $\lambda x, y.x$ is different from the one of $\lambda x.\lambda y.x$. The two terms are not different by accident: we really mean that they behave differently, and in fact this is the whole point of having multi-binders as a syntactic construct. A multiple abstraction forms a redex only if it is fully applied: for example, if M is a term and y is not free in it, iterated abstractions reduce as follow

$$(\lambda x.\lambda y.x)\ M \longrightarrow_\beta \lambda y.M$$

but the corresponding term built using multiple binders should be already in β-normal form

$$(\lambda x, y.x)\ M \not\longrightarrow_\beta$$

Redexes of multi-binders should have the form

$$(\lambda x_1, \ldots, x_n.M)\ N_1 \cdots N_n \longrightarrow_\beta M\{^{N_1,\ldots,N_n}/_{x_1,\ldots,x_n}\}$$

where the substitution is intended as a simultaneous substitution rather than iterated substitution.

This language, called the *multivariate λ-calculus*, was originally introduced by Pottinger in [53], and was motivated by the study of combinatory logic: it is possible to translate combinators to terms in the multivariate λ-calculus in such a way that reductions of combinatory logic have a one-one correspondence with reductions in the multivariate λ-calculus (this is not possible with ordinary λ-calculus).

We will now see how the multivariate λ-calculus can be formalized using the Sato representation.

6.3.2 Representing multi-binders

An iterative approach

The most obvious view of multiple binders is that they are not different from single binders at all. There is absolutely nothing preventing us from interpreting (infor-

mally) a multi-λ-abstraction as a special case of iterated single abstractions: the symbolic expressions could therefore use a list of local variables to represent the list of bound variables in the informal term with a one-one correspondence.

If we follow this style, it will be possible to make the Sato representation work with only minor modifications with respect to the regular λ-calculus: we just need to compute the height of each bound name in a bottom-up fashion. Without any pretension of being formal, the encoding of $\lambda X_1, X_2, \ldots, X_n.M$ would be represented as the concrete symbolic expression M_1, computed as follows:

$$\begin{array}{rcl}
x_n & \triangleq & \mathsf{F}_{X_n}(M) \\
M_n & \triangleq & \lambda x_n.(M\{^{x_n}/_{X_n}\}) \\
x_{n-1} & \triangleq & \mathsf{F}_{X_{n-1}}(M_n) \\
M_{n-1} & \triangleq & \lambda x_{n-1}, x_n.(M'_n\{^{x_{n-1}}/_{X_{n-1}}\}) \quad \text{if } M_n = \lambda x_n.M'_n \\
 & \ldots & \\
x_2 & \triangleq & \mathsf{F}_{X_2}(M_3) \\
M_2 & \triangleq & \lambda x_2, \ldots, x_n.(M'_3\{^{x_2}/_{X_2}\}) \quad \text{if } M_3 = \lambda x_3, \ldots, x_n.M'_3 \\
x_1 & \triangleq & \mathsf{F}_{X_1}(M_2) \\
M_1 & \triangleq & \lambda x_1, \ldots, x_n.(M'_2\{^{x_1}/_{X_1}\}) \quad \text{if } M_2 = \lambda x_2, \ldots, x_n.M'_2
\end{array}$$

where F is a proper height function defined on multivariate symbolic expressions.

We believe that this approach, mapping each bound variable of an informal abstraction to a separate concrete local variable, is not advisable. In languages like the multivariate λ-calculus, multiple binders should always be considered as atomic operations, since they are always built in one step, and always fully instantiated in one step. The iterated operation we just described is thus a very poor representation of multivariate abstractions. This is not just a stylistic criticism: the pragmatics of multiple binders is so different that this representation would also have a negative impact on the formalization effort and on the clarity of the final result. These considerations extend to other encodings of binding, as pointed out by other authors, particularly in the case of the locally nameless encoding (see [13, 8]).

A simultaneous approach

Since we have major objections against the iterative approach, we will now consider if it is possible to encode a whole multi-binder by means of a single local variable. To identify a specific formal parameter of a multiple abstraction encoded by means of a single local variable x, we can use a pair $\langle x, i\rangle$, where i is a natural number identifying the i-th element in the list of formal parameters.

Our intuition tells us that the original height function for the pure λ-calculus can be extended to a *multi-height*, capable of computing the height of a list of parameters in a lambda term. Such a function should simply identify all the parameters belonging to the given list.

A possible encoding of the multivariate λ-calculus in this simultaneous approach could be

$$M, N ::= X | \langle x, i\rangle \,|\, (M\ N) \,|\, [x, n]\ M$$

where $[x, n]\ M$ encodes an abstraction of length n binding the local variable x in the subterm M, and $\langle x, i\rangle$ refers to the i-th argument of the nearest abstraction identified by x.

6.3.3 Multivariate symbolic expressions

The above representation is still not completely satisfying for a reason: simultaneous multi-binders are matched by iterated binary applications rather than single n-ary applications. We can expect such a representation to have a bad impact in the formalization of β-reduction, essentially because the shape of a redex would be a function of the number of the bound variables. In this case, we think it is a good idea to force the nested applications of informal syntax to be collapsed in a variable arity application: in practice, we just make the syntactic sugar for n-ary applications formal – and mandatory.

Concretely, we must distinguish a smaller class of *value* terms (non-applications, denoted $V, W, \ldots$), which can be used as the head of an application. All terms (denoted $M, N, \ldots$) are (n-ary) applications, with values being represented by means

of the "empty application" injection $V \mapsto (V\ [])$. This is reflected in two different, mutually defined types of symbolic expressions: $\mathbb{S}^{tm}$(for applications) and $\mathbb{S}^{val}$(for values); we will also denote the type of lists of (term) symbolic expressions as $\overline{\mathbb{S}^{tm}}$.

$$\frac{}{X : \mathbb{S}^{val}} \qquad \frac{}{\langle x, i\rangle : \mathbb{S}^{val}}$$

$$\frac{V : \mathbb{S}^{val} \qquad \overline{N} : \overline{\mathbb{S}^{tm}}}{(V\ \overline{N}) : \mathbb{S}^{tm}} \qquad \frac{M : \mathbb{S}^{tm}}{[x, n]\ M : \mathbb{S}^{val}}$$

In abstractions $[x, n]\ M$, we consider all occurrences of x inside M to be bound, regardless of the associated index. Should the index of x exceed the arity n of the local variable, it cannot not be captured by an outer abstraction. In a symbolic expression

$$[x, m]\ ([x, n]\ \langle x, i\rangle)$$

where $n \leq i < m$, $\langle x, i\rangle$ does not refer to the outer abstraction: instead we regard it as meaningless, and the whole expression as ill-formed. We say that such occurrences of local variables are *locally dangling*, because they identify a certain multi-binder, but no specific position inside it; this is opposed to the more common notion of (globally) dangling local variables, whose name is not bound anywhere.

Occasionally, we will use values where a term is expected, keeping the empty application coercion implicit, writing V instead of $(V\ [])$. Sometimes it is not necessary to distinguish between term and value symbolic expressions: we will denote generic symbolic expressions as $S, T, \ldots$ and their type as $\mathbb{S}$. We will also speak of structural induction on a symbolic expression, meaning mutual induction on the types $\mathbb{S}^{tm}$ and $\mathbb{S}^{val}$.

The operations computing the sets of global and unbound local variables of a given symbolic expression are defined in the obvious way. The abstraction case of LV^{val} is justified by what we said earlier about locally dangling local variable, which

are meaningless but still bound.

$$\begin{aligned}
\mathsf{LV}^{val}(X) &\triangleq \emptyset \\
\mathsf{LV}^{val}(\langle x, i\rangle) &\triangleq \{x\} \\
\mathsf{LV}^{val}(\,[x, n]\; M) &\triangleq \mathsf{LV}^{tm}(M) \setminus \{x\} \\
\mathsf{LV}^{tm}(V\ \overline{N}) &\triangleq \mathsf{LV}^{val}(V) \cup \mathsf{LV}^{\overline{tm}}(\overline{N}) \\
\mathsf{LV}^{\overline{tm}}([N_1; \ldots; N_k]) &\triangleq \mathsf{LV}^{tm}(N_1) \cup \ldots \cup \mathsf{LV}^{tm}(N_k)
\end{aligned}$$

$$\begin{aligned}
\mathsf{GV}^{val}(X) &\triangleq \{X\} \\
\mathsf{GV}^{val}(\langle x, i\rangle) &\triangleq \emptyset \\
\mathsf{GV}^{val}(\,[x, n]\; M) &\triangleq \mathsf{GV}^{tm}(M) \\
\mathsf{GV}^{tm}(V\ \overline{N}) &\triangleq \mathsf{GV}^{val}(V) \cup \mathsf{GV}^{\overline{tm}}(\overline{N}) \\
\mathsf{GV}^{\overline{tm}}([N_1; \ldots; N_k]) &\triangleq \mathsf{GV}^{tm}(N_1) \cup \ldots \cup \mathsf{GV}^{tm}(N_k)
\end{aligned}$$

When there is no ambiguity, we will omit the superscripts and just write LV or GV. As always, we will use the notation $X \# S$ to say that $X \notin \mathsf{GV}(S)$. With a small abuse of notation, we will also write $S_1, \ldots, S_m \# T_1, \ldots, T_n$ to mean $(\mathsf{GV}(S_1) \cup \ldots \cup \mathsf{GV}(S_m)) \cap (\mathsf{GV}(T_1) \cup \ldots \cup \mathsf{GV}(T_n)) = \emptyset$.

We now define the two simultaneous substitution operations, for local and global variables. Notice that we must choose whether variables should be replaced by terms or values: while the variables being substituted are values, to implement operations like reduction we must be able to replace variables with generic terms. This also implies that substitution in a value will return a term (e.g.: $X\{^M/_X\} = M$, where M is a term).

Some small complication arises when substituting in an application: intuitively, $(V\ N)\{^M/_X\} = (V\{^M/_X\}\ N\{^M/_X\})$, but $V\{^M/_X\}$ in general is not a value, therefore it cannot appear as the head of an application. This corresponds to the case where we must merge two nested applications, using the following `mkappl` operation:

$$\texttt{mkappl}((V\ \overline{M}), \overline{N}) \triangleq (V\ (\overline{M} @ \overline{N}))$$

Substitution for global variables must replace a list of global variables with a list

of an equal number of terms. The operation is defined as follows:

$$
\begin{array}{rcl}
Y\{\overline{M_k}/\overline{X_k}\} & \triangleq & \begin{cases} M_i & \text{if } Y \in \overline{X_k} \text{ and } i = \mathtt{posn}(Y, \overline{X_k}) \\ Y & \text{if } Y \notin \overline{X_k} \end{cases} \\
\langle x, i\rangle\{\overline{M_k}/\overline{X_k}\} & \triangleq & \langle x, i\rangle \\
(\mathtt{[}x, n\mathtt{]}\ N)\{\overline{M_k}/\overline{X_k}\} & \triangleq & \mathtt{[}x, n\mathtt{]}\ (N\{\overline{M_k}/\overline{X_k}\}) \\
(V\ \overline{N})\{\overline{M_k}/\overline{X_k}\} & \triangleq & \mathtt{mkappl}(V\{\overline{M_k}/\overline{X_k}\}, \overline{N}\{\overline{M_k}/\overline{X_k}\}) \\
[N_1; \ldots; N_m]\{M_k/X_k\} & \triangleq & [N_1\{M_k/X_k\}; \ldots; N_m\{M_k/X_k\}]
\end{array}
$$

$\mathtt{posn}(Y, \overline{X_k})$ is the position of Y in the list X_k and is meaningful only if $Y \in \overline{X_k}$. If $\overline{X_k}$ contains multiple occurrences of Y, `posn` returns the **largest** index i such that $X_i = Y$. Also notice that while in this definition we assume that both $\overline{X_k}$ and $\overline{M_k}$ have the same length, in the formalization the substitution is also defined for lists of different length (in this case, the longer list is trimmed to the size of the shorter one by dropping the last elements). In the formalization, however, the substitution is only used when the lengths of the two lists are coherent.

Substitution for local variables is defined similarly; however in this case we will substitute all free occurrences of a single local variable with a term from a given list, depending on the associated index (i.e. $\langle x, i\rangle\{\overline{M_k}/x\} = M_i$):

$$
\begin{array}{rcl}
X\{\overline{M_k}/x\} & \triangleq & X \\
\langle y, i\rangle\{\overline{M_k}/x\} & \triangleq & \begin{cases} M_i & \text{if } x = y \text{ and } i < k \\ \langle y, i\rangle & \text{else} \end{cases} \\
(\mathtt{[}y, n\mathtt{]}\ N)\{\overline{M_k}/x\} & \triangleq & \begin{cases} \mathtt{[}y, n\mathtt{]}\ N & \text{if } x = y \\ \mathtt{[}y, n\mathtt{]}\ (N\{\overline{M_k}/x\}) & \text{else} \end{cases} \\
(V\ \overline{N})\{\overline{M_k}/x\} & \triangleq & \mathtt{mkappl}(V\{\overline{M_k}/x\}, \overline{N}\{\overline{M_k}/x\}) \\
[N_1; \ldots; N_m]\{M_k/x\} & \triangleq & [N_1\{M_k/x\}; \ldots; N_m\{M_k/x\}]
\end{array}
$$

Again, this operation is defined also for unsound arguments, corresponding to the case $\langle x, i\rangle\{\overline{M_k}/x\}$ when $i > k$. Ideally, if we are replacing local variables with a list of k terms, it means that the local variable has been abstracted by a binder of arity k, therefore it only appears together with indices smaller than k: this is always true in our development. Furthermore, since locally dangling local variables are still

considered bound, the definition of abstraction cases is greatly simplified. If we were to allow substitution of locally dangling local variables, we should define it as

$$([y,n]\ N)\{\overline{M_k}/x\} \triangleq \begin{cases} [y,n]\ N & \text{if } x = y \text{ and } n \geq k \\ [y,n]\ (N\{\langle x,0\rangle,\ldots\langle x,n-1\rangle,M_n,\ldots,M_k/x\}) & \text{if } x = y \text{ and } n < k \\ [y,n]\ (N\{\overline{M_k}/x\}) & \text{else} \end{cases}$$

with a less uniform treatment of abstractions, which is not only imore difficult to understand, but also to carry on in the formalization.

Substitution for local variables is particularly used in the special case of *variable opening*, denoted as follows:

$$S[x \mapsto \overline{X_k}] \triangleq S\{\overline{X_k}/x\}$$

Its inverse, *variable closing*, is defined as

$$S[\overline{X_k} \mapsto x] \triangleq S\{\langle x,0\rangle,\ldots,\langle x,k-1\rangle/\overline{X_k}\}$$

We just state the following standard properties of substitutions (the proofs are obtained by mutual induction on term and value symbolic expressions).

Lemma 6.5

1. *If* $\overline{X} \# S$, *then* $S\{\overline{M}/\overline{X}\} = S$.
2. *If* $x \notin \mathsf{LV}(S)$, *then* $S\{\overline{M}/x\} = S$.

Lemma 6.6 *If* $\overline{X} \# \overline{Y}, \overline{N}$, *then for all symbolic expressions* S *we have*

$$S\{\overline{M}/\overline{X}\}\{\overline{N}/\overline{Y}\} = S\{\overline{N}/\overline{Y}\}\{\overline{M\{\overline{N}/\overline{Y}\}}/\overline{X}\}$$

Lemma 6.7 *Substitution for both global and local variables is equivariant: if* π *is a finite permutation of global variables, then*

$$\begin{aligned} \pi \cdot (S\{\overline{M}/\overline{X}\}) &= (\pi \cdot S)\{\pi\cdot\overline{M}/\pi\cdot\overline{X}\} \\ \pi \cdot (S\{\overline{M}/x\}) &= (\pi \cdot S)\{\pi\cdot\overline{M}/x\} \end{aligned}$$

6.3.4 Well-formed multivariate terms

Given the above definition of symbolic expressions, it is just natural to consider an extended notion of height $\mathsf{F} : \overline{\mathbb{X}} \to \mathbb{S}^{tm} \to \mathbb{V}$. As usual, we will first define well-formed terms parametrically on F, and later argue for the existence of well-behaved heights.

The basic definition we need to express the set of well-formed terms is the abstraction operation

$$\mathsf{abs}_{\overline{X}} M \triangleq [\mathsf{F}_{\overline{X}}(M), |\overline{X}|]\ (M[\overline{X} \mapsto \mathsf{F}_{\overline{X}}(M)])$$

Similarly to the case of the λ-calculus, this operation corresponds to the informal multivariate λ-abstraction. The predicates $\mathbb{L}^{tm}$ and $\mathbb{L}^{val}$, are then defined as shown in Table 6.3. In rule Lval-Abs, we require that the list of global variables $\overline{X}$ being abstracted is composed of distinct names: this is a natural assumption, because all well-formed λ-abstractions can be expressed using distinct names, while the opposite is not true (if a name Y occurs in $\overline{X}$ more than once, the innermost occurrence shadows the other ones, meaning that a term can be expressed in this shape only if the shadowed variables are not actually used in the body of the abstraction).

We will often write $\mathbb{L}$ for $\mathbb{L}^{tm}$ or $\mathbb{L}^{val}$ if the distinction is unimportant or is clear from the context. We will also speak of induction on the derivation of a well-formedness judgment, meaning mutual structural induction on the types $\mathbb{L}^{tm}$ and $\mathbb{L}^{val}$.

We also lift the instantiation operation to the multivariate case as follows:

$$[x, n]\ M \blacktriangledown \overline{N} \triangleq M\{{}^{\overline{N}}/_{x}\}$$

This operation will be used only when $|\overline{N}| = n$.

6.3.5 Excellent multivariate heights

We now extend the excellence properties of Section 6.2.1 so as to make them meaningful for the multivariate λ-calculus. This requires lifting the definition of E to

$$\frac{}{X : \mathbb{L}^{val}}\ (\textsc{Lval-Var}) \qquad \frac{M : \mathbb{L}^{tm} \qquad \overline{X} \text{ are distinct}}{\mathsf{abs}_{\overline{X}} M : \mathbb{L}^{val}}\ (\textsc{Lval-Abs})$$

$$\frac{V : \mathbb{L}^{val} \qquad \text{for all } M \in \overline{N},\ M : \mathbb{L}^{tm}}{(V\ \overline{N}) : \mathbb{L}^{tm}}\ (\textsc{Ltm-App})$$

Table 6.3: Well-formedness predicate for the multivariate λ-calculus

multivariate symbolic expressions as follows:

$$\begin{aligned}
\mathsf{E}_{\overline{X}}(Y) &\triangleq \emptyset \\
\mathsf{E}_{\overline{X}}(\langle x, i\rangle) &\triangleq \emptyset \\
\mathsf{E}_{\overline{X}}([x, n]\ M) &\triangleq \begin{cases} \emptyset & \text{if } \overline{X} \# M \\ \{x\} \cup \mathsf{E}_{\overline{X}}(M) & \text{else} \end{cases} \\
\mathsf{E}_{\overline{X}}(V\ \overline{N}) &\triangleq \mathsf{E}_{\overline{X}}(V) \cup (\textstyle\bigcup_{M \in \overline{N}} \mathsf{E}_{\overline{X}}(M))
\end{aligned}$$

Definition 6.7 *A function* $\mathsf{H} : \overline{\mathbb{X}} \to \mathbb{S} \to \mathbb{V}$ *is an* excellent *height for the multivariate λ-calculus if the following three properties hold:*

(MHE) H *is* equivariant*: for all finite permutations π, lists of parameters $\overline{X}$ and symbolic expressions S, $\mathsf{H}_{\overline{X}}(S) = \mathsf{H}_{\pi \cdot \overline{X}}(\pi \cdot S)$;*

(MHF) H *is* fresh*: for all lists of parameters $\overline{X}$ and symbolic expressions S, $\mathsf{H}_{\overline{X}}(S) \notin \mathsf{E}_{\overline{X}}(S)$;*

(MHP) H *preserves term substitution: for all lists of parameters $\overline{X}$ and $\overline{Y_k}$, for all symbolic expressions S and for all lists of term symbolic expressions $\overline{N_k}$, if $\overline{X} \# \overline{Y_k}, \overline{N_k}$, then $\mathsf{H}_{\overline{X}}(S) = \mathsf{H}_{\overline{X}}(S\{^{\overline{N_k}}/_{\overline{Y_k}}\})$.*

Definition 6.8 *The function* F *is defined as follows*

$$\begin{aligned}
\mathsf{F}_{\overline{X}}(Y) &\triangleq \begin{cases} 1 & \textit{if } Y \in \overline{X} \\ 0 & \textit{else} \end{cases} \\
\mathsf{F}_{\overline{X}}(\langle x, i\rangle) &\triangleq 0 \\
\mathsf{F}_{\overline{X}}([x, n]\ M) &\triangleq \begin{cases} \mathsf{F}_{\overline{X}}(M) & \textit{if } \mathsf{F}_{\overline{X}}(M) = 0 \textit{ or } \mathsf{F}_{\overline{X}}(M) > x \\ x + 1 & \textit{otherwise} \end{cases} \\
\mathsf{F}_{\overline{X}}(V\ [N_1; \ldots; N_k]) &\triangleq \max(\mathsf{F}_{\overline{X}}(V), \mathsf{F}_{\overline{X}}(N_1), \ldots, \mathsf{F}_{\overline{X}}(N_k))
\end{aligned}$$

Lemma 6.8 *For all S and $\overline{X}$, $\mathsf{F}_{\overline{X}}(S) = 0$ if and only if $\overline{X} \# S$.*

Proof: Easy induction on the structure of S. □

Theorem 6.9 *The function F of Definition 6.8 is excellent.*

Proof: We prove the three properties separately.

(MHE) By structural induction on S:

- case $S = Y$: Y is either in $\overline{X}$ or not; if it is, then $\pi \cdot Y \in \pi \cdot \overline{X}$ too, which implies $\mathsf{F}_{\overline{X}}(Y) = \mathsf{F}_{\pi \cdot \overline{X}}(\pi \cdot Y) = 1$; if on the contrary $Y \notin \overline{X}$, then $\pi \cdot Y \notin \pi \cdot \overline{X}$ (π is a permutation, therefore it is injective), implying $\mathsf{F}_{\overline{X}}(Y) = \mathsf{F}_{\pi \cdot \overline{X}}(\pi \cdot Y) = 0$;
- case $S = x$: $\mathsf{F}_{\overline{X}}(x) = \mathsf{F}_{\pi \cdot \overline{X}}(\pi \cdot x) = \mathsf{F}_{\pi \cdot \overline{X}}(x) = 0$ holds trivially;
- case $S = [\![x, n]\!]\ M$: define
$$y \triangleq \mathsf{F}_{\overline{X}}(M)$$
by induction hypothesis, we know that $y = \mathsf{F}_{\pi \cdot \overline{X}}(\pi \cdot M)$; there are two cases: if $0 < y \leq x$, then $\mathsf{F}_{\overline{X}}([\![x; n]\!]\ M) = \mathsf{F}_{\pi \cdot \overline{X}}(\pi \cdot ([\![x, n]\!]\ M)) = x + 1$; if $y = 0$ or $y > x$, then $\mathsf{F}_{\overline{X}}([\![x, n]\!]\ M) = \mathsf{F}_{\pi \cdot \overline{X}}(\pi \cdot ([\![x, n]\!]\ M)) = y$;
- case $S = (V\ \overline{N_n})$: we must prove that
$$\max\{\mathsf{F}_{\overline{X}}(V), \mathsf{F}_{\overline{X}}(N_0), \ldots, \mathsf{F}_{\overline{X}}(N_{n-1})\} = \max\{\mathsf{F}_{\pi \cdot \overline{X}}(\pi \cdot V), \mathsf{F}_{\pi \cdot \overline{X}}(N_0), \ldots, \mathsf{F}'_{\pi \cdot \overline{X}}(\pi \cdot N_{n-1})\}$$
the equation is satisfied, since the arguments of the two max functions are pairwise equal by induction hypothesis;

(MHF) We obtain the property as a corollary of the stronger statement

$$\text{for all } x,\ x \geq \mathsf{F}_{\overline{X}}(S) \Longrightarrow x \notin \mathsf{E}_{\overline{X}}(S)$$

By structural induction on S:

- cases $S = Y$ or $S = y$: in both cases, $\mathsf{E}_{\overline{X}}(S) = \emptyset$, thus the thesis follows trivially;
- case $S = [y, n]\ M$: define

$$z \triangleq \mathsf{F}_{\overline{X}}(M)$$
$$z' \triangleq \mathsf{F}_{\overline{X}}([y, n]\ M)$$

by induction hypothesis, we know that

$$\text{for all } x',\ x' \geq \mathsf{F}_{\overline{X}}(M) \Longrightarrow x' \notin \mathsf{E}_{\overline{X}}(M)$$

notice in particular that the property holds for $x' = z$ or greater, since z satisfies the required inequality; then consider two subcases:

  - if $\overline{X} \# [y, n]\ M$ then $\mathsf{E}_{\overline{X}}([y, n]\ M) = \emptyset$, thus the thesis follows trivially;
  - if $\overline{X} \cap GV([y, n]\ M) \neq \emptyset$, we must prove that

$$z' \notin \{y\} \cup \mathsf{E}_{\overline{X}}(M)$$

we know by lemma 6.8 that $z \neq 0$, therefore either $0 < z \leq y$, implying $z' = y + 1$ (thus also $z' > z$), or $z > y$, implying $z' = z$: in both cases, the property follows by induction hypotheses.

- case $S = (V\ N_1 \cdots N_n)$: we must prove that

$$\forall x \geq \max\{\mathsf{F}_{\overline{X}}(V), \mathsf{F}_{\overline{X}}(N_1), \ldots, \mathsf{F}_{\overline{X}}(N_n)\} : x \notin \mathsf{E}_{\overline{X}}(V) \cup \mathsf{E}_{\overline{X}}(N_1) \cup \ldots \cup \mathsf{E}_{\overline{X}}(N_n)$$

the property is satisfied, since each we can prove by induction hypothesis that x is not in any of the sublists $\mathsf{E}_{\overline{X}}(V), \mathsf{E}_{\overline{X}}(N_1), \ldots, \mathsf{E}_{\overline{X}}(N_n)$;

(MHP) By structural induction on S:

- case $S = Z$: Z is either in $\overline{Y_k}$ or not; if it is, then $Z \notin \overline{X}$ and $Z\{\overline{N_k}/\overline{Y_k}\} = N_i$ for some $i = 1, \ldots, k$: this implies $\mathsf{F}_{\overline{X}}(Z) = 0 = \mathsf{F}_{\overline{X}}(N_i)$ by lemma 6.8, since $X \# \overline{N_k}$ by hypothesis; if $Z \notin \overline{Y_k}$, then $Z\{\overline{N_k}/\overline{Y_k}\} = Z$ and the thesis follows trivially;

- case $S = x$: $x\{\overline{N_k}/\overline{Y_k}\} = x$, thus the thesis follows trivially;
- case $S = [x, n]\ M$: define
$$y \triangleq \mathsf{F}_{\overline{X}}(M)$$
by induction hypothesis, we know that $\mathsf{F}_{\overline{X}}(M) = \mathsf{F}'_{\overline{X}}(M\{\overline{N_k}/\overline{Y_k}\})$; this also implies that
$$y = \mathsf{F}_{\overline{X}}(M\{\overline{N_k}/Y_k\})$$
there are two cases: if $0 < y \leq x$, then $\mathsf{F}_{\overline{X}}([x, n]\ M) = \mathsf{F}_{\overline{X}}([x, n]\ (M\{\overline{N_k}/\overline{Y_k}\})) = x{+}1$; if $y = 0$ or $y > x$, then $\mathsf{F}_{\overline{X}}([x, n]\ M) = \mathsf{F}_{\overline{X}}([x; \sigma_1, \ldots, \sigma_n]\ (M\{\overline{N_k}/\overline{Y_k}\})) = y$;
- case $S = (V\ M_1 \cdots M_n)$: we must prove that
$$\begin{aligned}&\max\{\mathsf{F}_{\overline{X}}(V), \mathsf{F}_{\overline{X}}(M_1), \ldots, \mathsf{F}_{\overline{X}}(M_n)\} = \\ &\max\{\mathsf{F}_{\overline{X}}(V\{\overline{N_k}/\overline{Y_k}\}), \mathsf{F}_{\overline{X}}(M_1\{\overline{N_k}/\overline{Y_k}\}), \ldots, \mathsf{F}'_{\overline{X}}(M_n\{\overline{N_k}/\overline{Y_k}\})\}\end{aligned}$$
the equation is satisfied, since the arguments of the two max functions are pairwise equal by induction hypothesis;

□

To be precise, the above definitions are actually formalized using specialized mutually recursive functions E^{tm}, E^{val}, F^{tm} and F^{val}: however, in this discussion, we prefer to be slightly less formal in order to avoid the notational burden required by mutual definitions.

In general, we are able to carry on formalizations using the Sato representation without relying on the concrete definition of a height, as long as we know that it is excellent. This is what we do: in the rest of the discussion, we will assume that the F function referenced by $\mathbb{L}^{tm}$, $\mathbb{L}^{val}$ and abs is excellent, but will otherwise treat it as an opaque definition.

Lemma 6.10 *If π is a finite permutation, then $\pi \cdot \mathsf{abs}_{\overline{X}} M = \mathsf{abs}_{\pi \cdot \overline{X}}(\pi \cdot M)$.*

Proof: Using property **(MHE)** of excellent heights. □

Lemma 6.11 *For all symbolic expressions S, $S : \mathbb{L} \Longrightarrow \pi \cdot S : \mathbb{L}$*

Proof: By structural induction on the derivation of $S : \mathbb{L}$. □

Lemma 6.12 *If $\overline{X} \# \overline{Y} @ \overline{N}$, then $(\mathsf{abs}_{\overline{X}} M)\{{}^{\overline{N}}/_{\overline{Y}}\} = \mathsf{abs}_{\overline{X}}(M\{{}^{\overline{N}}/_{\overline{Y}}\})$.*

Proof: Unfolding the definition of abs, we see that we must prove

$$(\llbracket x, n \rrbracket\ (M[\overline{X} \mapsto x]))\{{}^{\overline{N}}/_{\overline{Y}}\} = \llbracket y, n \rrbracket\ (M\{{}^{\overline{N}}/_{\overline{Y}}\}[\overline{X} \mapsto y])$$

where $x = \mathsf{F}_{\overline{X}}(M)$, $y = \mathsf{F}_{\overline{X}}(M\{{}^{\overline{N}}/_{\overline{Y}}\})$ and $n = |\overline{X}|$. Then by **(MHP)** $x = y$, and we only need to prove

$$M[\overline{X} \mapsto x]\{{}^{\overline{N}}/_{\overline{Y}}\} = M\{{}^{\overline{N}}/_{\overline{Y}}\}[\overline{X} \mapsto x]$$

To prove this equality, it is sufficient to unfold the definition of variable closing and then use Lemma 6.6. □

Lemma 6.13 *If $\overline{X}$ and $\overline{Y}$ are lists of distinct global variables having equal length, such that $\overline{X} \# \overline{Y}$ and $\overline{Y} \# \mathsf{abs}_{\overline{X}} M$, then $\mathsf{abs}_{\overline{X}} M = \mathsf{abs}_{\overline{Y}}(\pi \cdot M)$, where $\pi = (\overline{X}\ \overline{Y})$.*

Proof: Under the given hypotheses, $\overline{Y} = \pi \cdot \overline{X}$. Then $\mathsf{abs}_Y(\pi \cdot M) = \mathsf{abs}_{\pi \cdot X}(\pi \cdot M) = \pi \cdot \mathsf{abs}_X M$ (using Lemma 6.10). However, both $\overline{X}$ and $\overline{Y}$ are fresh in $\mathsf{abs}_{\overline{X}} M$, therefore the permutation is ineffective, yielding the thesis. □

Corollary 6.14 ("α-conversion" for abs) *For all terms M, if $M : \mathbb{L}$, $\overline{X}$ and $\overline{Y}$ are lists of distinct global variables having equal length and $\overline{Y} \# \mathsf{abs}_{\overline{X}} M$, there exists a term N such that $N : \mathbb{L}$ and $\mathsf{abs}_{\overline{X}} M = \mathsf{abs}_{\overline{Y}} N$.*

Proof: Let $\overline{Z}$ be a list of global variables such that $\overline{Z} \# \overline{X}, \overline{Y}, \mathsf{abs}_{\overline{X}} M$ and $|\overline{Z}| = |\overline{X}|$. Let $\pi = (\overline{Z}\ \overline{Y}) \circ (\overline{X}\ \overline{Y})$ and choose $N = \pi \cdot M$. By Lemma 6.11, $N : \mathbb{L}$. Also notice that by a double application of Lemma 6.13, $\mathsf{abs}_{\overline{X}} M = \mathsf{abs}_{\overline{Y}} N$, as needed. □

Lemma 6.15 *Given $S : \mathbb{L}$, for all lists of global variables $\overline{X}$ and lists of terms $\overline{N}$, if for all $M \in \overline{N}$, $M : \mathbb{L}$, then $S\{{}^{\overline{N}}/_{\overline{X}}\} : \mathbb{L}$.*

Proof: By induction on the derivation of $S : \mathbb{L}$. In the case LVAL-ABS, where $S = \mathsf{abs}_{\overline{Y}} M'$, we must prove that

$$(\mathsf{abs}_{\overline{Y}} M')\{^{\overline{N}}/_{\overline{X}}\} : \mathbb{L}$$

knowing by induction hypothesis that for all $\overline{X'}$ and $\overline{N'}$, $M'\{^{\overline{N'}}/_{\overline{X'}}\} : \mathbb{L}$. Choose a list of distinct global variables $\overline{Z}$ such that $\overline{Z} \# \overline{X}, \overline{Y}, \overline{N}, \mathsf{abs}_{\overline{X}} M$ and let $\pi = (\overline{X}\ \overline{Z})$. By Lemma 6.13

$$\mathsf{abs}_{\overline{X}} M' = \mathsf{abs}_{\overline{Z}}(\pi \cdot M')$$

then, by Lemma 6.12, we must prove

$$\mathsf{abs}_{\overline{Z}}((\pi \cdot M')\{^{\overline{N}}/_{\overline{X}}\}) : \mathbb{L}$$

or equivalently

$$(\pi \cdot M')\{^{\overline{N}}/_{\overline{X}}\} : \mathbb{L}$$

By properties of permutations, this is equivalent to

$$(\pi \cdot M)\{^{\pi \cdot \tilde{\pi} \cdot \overline{N}}/_{\pi \cdot \tilde{\pi} \cdot \overline{X}}\} : \mathbb{L}$$

By Lemma 6.7 and 6.11, we must prove

$$M\{^{\tilde{\pi} \cdot \overline{N}}/_{\tilde{\pi} \cdot \overline{X}}\} : \mathbb{L}$$

that is obtained by the induction hypothesis with $\overline{N'} = \tilde{\pi} \cdot \overline{N}$ and $\overline{X'} = \tilde{\pi} \cdot \overline{X}$. □

We now show two important adequacy properties: first, well-formed expressions do not contain unbound local variables; second, well-formed abstractions do not contain locally dangling local variables.

Lemma 6.16 *If $S : \mathbb{L}$, then $\mathsf{LV}(S) = \emptyset$.*

Proof: Standard proof by induction on the derivation of $S : \mathbb{L}$. □

To show well-formed abstractions do not contain locally dangling local variables, we define the "next index" operator, taking a symbolic expression S and a local

variable x and returning the successor of the maximum index i such that $\langle x, i\rangle$ occurs unbound in S:

$$\begin{array}{rcll}
\mathrm{ni}(X, x) & \triangleq & 0 & \\
\mathrm{ni}(\langle x, i\rangle, x) & \triangleq & i+1 & \\
\mathrm{ni}(\langle y, i\rangle, x) & \triangleq & 0 & \text{when } x \neq y \\
\mathrm{ni}(\,[x, n]\ M, x) & \triangleq & 0 & \\
\mathrm{ni}(\,[y, n]\ M, x) & \triangleq & \mathrm{ni}(M, x) & \text{when } x \neq y \\
\mathrm{ni}(V\ \overline{N}, x) & \triangleq & \max\{\mathrm{ni}(V), \mathrm{ni}(\overline{N}, x)\} & \\
\mathrm{ni}([N_1; \ldots; N_k], x) & \triangleq & \max\{\mathrm{ni}(N_1, x), \ldots, \mathrm{ni}(N_k, x)\} &
\end{array}$$

Lemma 6.17 *If* $[x, n]\ M : \mathbb{L}$*, then* $\mathrm{ni}(M, x) \leq n$ *(hence* x *is not locally dangling in* $[x, n]\ M$*).*

Proof: Since $[x, n]\ M : \mathbb{L}$, it must be equal to $\mathsf{abs}_{\overline{X_n}}\ M'$ for some $\overline{X_n}$ and M' such that $M' : \mathbb{L}$. Therefore we have

$$M = M'[\overline{X_n} \mapsto x] = M'\{^{\langle x,0\rangle,\ldots,\langle x,n-1\rangle}/_{\overline{X_n}}\}$$

Since $M' : \mathbb{L}$, $x \notin \mathsf{LV}(M')$, therefore $\mathrm{ni}(M', x) = 0$, thus $\mathrm{ni}(M'\{^{\langle x,0\rangle,\ldots,\langle x,n-1\rangle}/_{\overline{X_n}}\}$ cannot be greater than n (since the only unbound occurrences of x will be the ones that are being substituted, whose maximum index is $n-1$). □

Lemma 6.18 *If* $|\overline{X}| = |\overline{M}|$*,* $x \notin \mathsf{LV}(S)$ *and* $x \notin \mathsf{E}_{\overline{X}}(S)$*, then* $S[\overline{X} \mapsto x]\{^{\overline{M}}/_x\} = S\{^{\overline{M}}/_{\overline{X}}\}$.

Proof: By structural induction on S. In the case $S = [y, n]\ N$, we have $x \notin \mathsf{E}_{\overline{X}}([y, n]\ N)$, implying that either $\overline{X} \# N$ and $\mathsf{E}_{\overline{X}}([y, n]\ N) = \emptyset$, or $\mathsf{E}_{\overline{X}}([y, n]\ N) = \{y\} \cup \mathsf{E}_{\overline{X}}(N)$ and there exists some $Y \in \overline{X}$ such that $Y \in \mathsf{GV}(N)$.

- In the first case, $([y, n]\ N)[\overline{X} \mapsto x]\{^{\overline{M}}/_x\} = ([y, n]\ N)\{^{\overline{M}}/_{\overline{X}}\}$ can be rewritten to $([y, n]\ N)\{^{\overline{X}}/_x\} = [y, n]\ N$: the two sides of the equation are equal since by hypothesis $x \notin \mathsf{LV}([y, n]\ N)$.

- In the second case, we can prove $x \neq y$: then the goal becomes

$$\texttt{[}y, n\texttt{]}\ (N[\overline{X} \mapsto x]\{{}^{\overline{M}}/_{x}\}) = \texttt{[}y, n\texttt{]}\ (N\{{}^{\overline{M}}/_{\overline{X}}\})$$

  This follows by the induction hypothesis on N, since $x \notin \mathsf{E}_{\overline{X}}(N)$ and $x \notin \mathsf{LV}(N)$.

□

Lemma 6.19 *Suppose that* $\mathsf{abs}_{\overline{X}}M : \mathbb{L}$, $|\overline{X}| = |\overline{N}|$ *and for all* $N' \in \overline{N}$, $N' : \mathbb{L}$. *Then*

$$\mathsf{abs}_{\overline{X}}M \blacktriangledown \overline{N} : \mathbb{L}.$$

Proof: Let $x = \mathsf{F}_{\overline{X}}(M)$ and $n = |\overline{X}|$: then we must prove

$$M[\overline{X} \mapsto x]\{{}^{\overline{N}}/_{x}\} : \mathbb{L}$$

Using Lemma 6.16 and property **(MHF)**, we prove that $x \notin \mathsf{LV}(M)$ and $x \notin \mathsf{E}_{\overline{X}}(M)$. Then by Lemma 6.18, the goal becomes

$$M\{{}^{\overline{N}}/_{\overline{X}}\} : \mathbb{L}$$

that is a trivial consequence of Lemma 6.15. □

6.3.6 β-reduction

We formalize β-reduction as an inductive judgment: the definition employs three mutually defined judgment forms for values, terms and lists of terms (in the case of lists of terms, the intended meaning is that reduction happens in exactly one term in the list, leaving the other ones untouched). The rules we formalized are shown in Table 6.4.

The reduction step is defined by rule BRed. Recall that application of an abstraction to a list of terms that is too short does not contract; if the list of terms, instead, is longer than needed, the application does contract, with some arguments left over. BRed expresses this behaviour saying that in order for the an application

$$\frac{\overline{X} \text{ is a duplicate-free list} \qquad |\overline{X}| = |\overline{N}| \qquad M : \mathbb{L} \qquad \text{for all } N'' \in \overline{N}@\overline{N'},\ N'' : \mathbb{L}}{((\mathsf{abs}_{\overline{X}} M)\ (\overline{N}@\overline{N'})) \longrightarrow_\beta \mathtt{mkappl}(\mathsf{abs}_{\overline{X}} M \blacktriangledown \overline{N}, \overline{N'})} \quad (\textsc{BRed})$$

$$\frac{V \longrightarrow_\beta W \qquad \text{for all } M' \in \overline{M},\ M' : \mathbb{L}}{(V\ \overline{M}) \longrightarrow_\beta (W\ \overline{M})}\ (\textsc{BApp1}) \qquad \frac{\overline{M} \longrightarrow_\beta \overline{N} \qquad V : \mathbb{L}}{(V\ \overline{M}) \longrightarrow_\beta (V\ \overline{N})}\ (\textsc{BApp2})$$

$$\frac{M \longrightarrow_\beta N \qquad \overline{X} \text{ is a duplicate-free list}}{\mathsf{abs}_{\overline{X}} M \longrightarrow_\beta \mathsf{abs}_{\overline{X}} N} \quad (\textsc{BXi})$$

$$\frac{M \longrightarrow_\beta M' \qquad \text{for all } N \in \overline{N'},\ \mathrm{N} : \mathbb{L}}{\overline{N'}, M \longrightarrow_\beta \overline{N'}, M'}\ (\textsc{BTml1}) \qquad \frac{M : \mathbb{L} \qquad \overline{N} \longrightarrow_\beta \overline{N'}}{\overline{N}, M \longrightarrow_\beta \overline{N'}, M}\ (\textsc{BTml2})$$

Table 6.4: β-reduction rules for the multivariate λ-calculus

to contract, the head must be an abstraction of some length n, and it must be possible to split the list of arguments in two sublists $\overline{N}$ and $\overline{N'}$, with $\overline{N}$ of length n. The result of the contraction is obtained by instantiating the abstraction with $\overline{N}$ and applying the result of the instantiation to the leftover arguments $\overline{N'}$: since the result of an instantiation is a term (and not a value), this uses the `mkappl` operation.

Notice that the rules (in particular rules BRed and BXi) use a *forward* presentation, which as we argued is closer to the informal syntax. The use of abs in BXi hides the fact that the concrete representation of $\mathsf{abs}_{\overline{X}} M$ and $\mathsf{abs}_{\overline{X}} N$ might use different local names for the bound variables, because $\mathsf{F}_{\overline{X}}(M)$ is not necessarily equal to $\mathsf{F}_{\overline{X}}(N)$.

The theory of the Sato representation we developed in the previous sections is sufficient to prove that this definition of β-reduction is well-behaved.

Theorem 6.20 *If $S \longrightarrow_\beta T$, then*

1. *$S : \mathbb{L}$ and $T : \mathbb{L}$*
2. *for all finite permutations π, $\pi \cdot S \longrightarrow_\beta \pi \cdot T$*

3. $\mathsf{GV}(T) \subseteq \mathsf{GV}(S)$.

Proof: By induction on the derivation of $S \longrightarrow_\beta T$. When proving property 1, for the subcase BRED, the proof uses Lemma 6.19. □

Chapter 7

A formalization of an algebraic logical framework

Simple formalisms, including sufficiently compact variants of the λ-calculus, similar to the examples of $F_{<:}$ and the multivariate λ-calculus we presented in Chapters 5 and 6, are particularly popular case studies for all representations of binding, since they allow people experimenting with encodings of syntax to focus on the key issues of formal proofs, without the need to deal with overly intricate structures. There is howevere a possibility that important matters only arising in some larger languages might be overlooked.

This is more of a worry in the case of a recently developed representation, like the Sato representation we presented in the previous chapter. Out of this concern, we decided to test it against a more serious language. Our attention fell on some recent logical frameworks where only *canonical forms* are well typed expressions. Such systems employ a particularly stratified syntax, together with *hereditary substitution*, to keep the terms in normal form.

The reason for the study of such systems lies in the correspondence between objects in the framework and entities in the object theory, that in non-canonical systems is typically a bijection only up to $\beta\eta$-conversion. Canonical logical frameworks include a subsystem of ELF known simply as the Canonical Logical Framework ([27, 35]), the Type Framework ([2, 3]), the Concurrent Logical Framework ([66]), Dependent Contextual Modal Type Theory ([45]), and Gordon Plotkin's DMBEL ([49, 50]). This last system, albeit limited to a (canonical) second order fragment of ELF, shows many of the issues of the other canonical formalisms we cited (most notably, dependent types and a form of hereditary substitution).

This chapter discusses a formalization of DMBEL in the Matita interactive theorem prover. In the first section we recall the definition of DMBEL. Section 2 begins the discussion of the formalized syntax, introducing the notion of symbolic expressions. Section 3 is about well formed expressions and the related notion of excellent height. Section 4 describes the formalization of hereditary substitution and related issues. In Section 5 we formalize the type system. In the last section, we draw final conclusions on the Sato representation.

7.1 Informal syntax

We now present the syntax of the DMBEL logical framework, whose grammar is summarized in Table 7.1. Expressions in DMBEL fall in one of three syntactic categories:

- *terms* (notation: $t, t_1, t_2, \ldots, u, \ldots$) used to express the entities of the object theory;
- *types* (notation: $\sigma, \tau, \ldots$) that the type system assigns to terms;
- *abstraction terms* (notation: $a, a_1, a_2, \ldots, b, \ldots$) taking as input a list of terms and returning a term.

These three categories are mutually defined, yet clearly distinct: this stratification allows DMBEL to enforce the *canonicity* of its expressions, roughly meaning that the only well formed expressions are those in β-normal, η-long[1] form. Since all the expressions must be in canonical form, the system does not have a computation rule.

Terms can contain two kinds of variables: *term variables* (notation: $x, x_1, x_2, \ldots, y, \ldots$) serving as a placeholder for other terms, and *abstraction variables* (notation: $\varphi, \varphi_1, \varphi_2, \ldots, \psi, \ldots$ which can be instantiated with abstraction terms. Since the syntax enforces terms to be in canonical form, abstraction variables are always fully applied to a number of terms matching their ariety.

Terms can also contain functional constants (notation: $f, f_1, \ldots, g, \ldots$). Functional constants, much like abstraction variables, are always fully applied but, contrarily to abstraction variables, they are applied to abstraction terms rather than regular terms.

DMBEL has dependent types: a type constant (notation: $S, T, \ldots$) must always be applied to a number of abstraction terms matching its ariety. Finally, abstraction terms are used to bind an arbitrary number of term variables of a chosen type inside a term, and are thus similar to multiple λ-abstractions. We write $(x_1 : \sigma, x_2 : \tau)\, t$

[1]Here we are borrowing Robin Adams's terminology from [2, p. 63]: roughly speaking, an expression is said to be in η-long form if η-expanding any of its sub-expressions yields a redex

to mean the abstraction term binding the variables x_1 of type σ and x_2 of type τ in the term t. Since we have dependent types, in this example x_1 is bound not only in t, but also in τ. Unsurprisingly, we assume that abstraction terms are identified up to α-conversion.

There is no binder for abstraction variables, which always appear free in DMBEL expressions.

Abstraction types, i.e. the types of abstraction terms, do not appear in terms, types or abstraction terms, but are used for typechecking purposes. Similarly to abstraction terms, we write $(\Gamma)\,\tau$ to mean the abstraction type binding the context Γ in the type τ.

Three more structures appear in typing judgments: *contexts* $\Gamma, \Delta, \ldots$ associate to every free term variable its type; *abstraction contexts* $\Phi, \Psi, \ldots$ do the same, mapping abstraction variables to abstraction types; the operations *dom* and *cod* return respectively the list of the names declared in a context (or abstraction context) and the list of the associated types (or abstraction types). A *signature* Σ declares functional and type constants, parametrized on an abstraction context. A declaration $S(\Phi)$ means that the type constant S must be applied to a list of abstraction terms whose (dependent) type is expressed by the abstraction context Φ; similarly, a declaration $f(\Phi) : \sigma$ means that the functional constant f must be applied to a list of abstraction terms whose type is expressed by Φ, and will return a term of type σ, where the variables declared in Φ can appear in σ.

As an example, in the following signature[2]

$$\begin{aligned}\Sigma \triangleq\ & \mathtt{prop},\ \mathtt{proof}(\varphi : \mathtt{prop}),\ \mathtt{Imp}(\varphi : \mathtt{prop}, \psi : \mathtt{prop}) : \mathtt{prop},\\ & \mathtt{ImpI}(\varphi_1 : \mathtt{prop}, \varphi_2 : \mathtt{prop}, \varphi_3 : (\mathtt{proof}(\varphi_1))\,\mathtt{proof}(\varphi_2)) : \mathtt{proof}(\mathtt{Imp}(\varphi_1, \varphi_2))\end{aligned}$$

we declare

- the 0-ary type `prop` of propositions
- the type $\mathtt{proof}(P)$ of the proofs of P, where P is a proposition

[2]Syntactic sugar has been used to hide empty abstractions and applications where they are needed by the syntax of DMBEL.

- the operation `Imp` constructing a proposition $\texttt{Imp}(P, Q)$ from propositions P and Q

- the operation `ImpI` that turns a procedure taking a proof of P and returning a proof of Q into a proof of $\texttt{Imp}(P, Q)$ (think of the implication introduction rule in natural deduction).

Following the presence of two kinds of variables in our syntax, we also define two different kinds of substitution, whose definition is shown in Table 7.2. *Substitution of term variables* is the usual notion of simultaneous substitution, replacing a vector of different variables with a vector of terms of equal length. It is defined by structural induction on terms, types, abstraction terms and abstraction types, and commutes with all the syntactic forms (in the case of abstractions, a condition on the bound variables prevents variable capture). This operation preserves the syntactical well-formedness of DMBEL expressions and therefore does not create any redex.

Substitution of abstraction variables, instead, is a form of *hereditary* substitution. Its aim is, morally, to replace a vector of different abstraction variables with a vector of abstraction terms of equal length. However we really know that this cannot be the case: abstraction variables only appear fully applied to their arguments, while abstraction terms are never applied to anything, in order to guarantee that all expressions be in canonical form.

Informally, a naïve substitution of an abstraction term for an abstraction variable would create a β-redex, i.e. a term that is not in normal form. The result of the operation, instead, should be the canonical term corresponding to the naïve notion of substitution. This means that, when substituting an abstraction variable, we should also perform some "reduction" steps (i.e. more substitutions, this time on term variables) in order to finally compute a canonical expression.

Hereditary substitution of abstraction variables is defined by structural induction and commutes with most syntactic constructions, with the notable exception of applied abstraction variables: if we want to replace the variable φ with $(\Gamma)\, u$ in $\varphi(t_1, \ldots, t_n)$, we first compute a recursive call on the arguments of the ap-

σ, τ	::=	$S(\overline{a})$	types
α, β	::=	$(\Gamma)\sigma$	abstraction types
terms:			
t, u	::=	x	term variables
	|	$f(\overline{a})$	applied functional constants
	|	$\varphi(\overline{t})$	applied functional variables
a, b	::=	$(\Gamma)t$	abstraction terms
contexts:			
Γ, Δ	::=	$\emptyset$	empty context
	|	$\Gamma, x : \sigma$	context entry
abstraction contexts:			
Φ, Ψ	::=	$\emptyset$	empty abs. context
	|	$\Phi, \varphi : \alpha$	abs. context entry
signatures:			
Σ	::=	$\emptyset$	empty signature
	|	$\Sigma, S(\Phi)$	type constant declaration
	|	$\Sigma, f(\Phi) : \sigma$	functional constant declaration

Table 7.1: Syntax of DMBEL

Substitution of term variables:

$$
\begin{array}{rcl}
y\{\bar{t}/\bar{x}\} & \triangleq & \begin{cases} t_i & \text{if } y = x_i \\ y & \text{if } y \# x_1, \ldots, x_n \end{cases} \\
f(\overline{a_n})\{\bar{t}/\bar{x}\} & \triangleq & f(a_0\{\bar{t}/\bar{x}\}, \ldots, a_{n-1}\{\bar{t}/\bar{x}\}) \\
\varphi(\overline{u_n})\{\bar{t}/\bar{x}\} & \triangleq & \varphi(u_0\{\bar{t}/\bar{x}\}, \ldots, u_{n-1}\{\bar{t}/\bar{x}\}) \\
S(\overline{a_n})\{\bar{t}/\bar{x}\} & \triangleq & S(a_0\{\bar{t}/\bar{x}\}, \ldots, a_{n-1}\{\bar{t}/\bar{x}\}) \\
((\overline{y_n} : \overline{\sigma_n})\, u)\{\bar{t}/\bar{x}\} & \triangleq & (y_0 : \sigma_0\{\bar{t}/\bar{x}\}, \ldots, y_{n-1} : \sigma_{n-1}\{\bar{t}/\bar{x}\})(u\{\bar{t}/\bar{x}\}) \quad \text{if } \overline{y_n} \# \bar{x}, \bar{t} \\
((\overline{y_n} : \overline{\sigma_n})\, \tau)\{\bar{t}/\bar{x}\} & \triangleq & (y_0 : \sigma_0\{\bar{t}/\bar{x}\}, \ldots, y_{n-1} : \sigma_{n-1}\{\bar{t}/\bar{x}\})(\tau\{\bar{t}/\bar{x}\}) \quad \text{if } \overline{y_n} \# \bar{x}, \bar{t}
\end{array}
$$

Hereditary substitution of functional variables:

$$
\begin{array}{rcl}
x\{\overline{a_m}/\overline{\varphi_m}\} & \triangleq & x \\
f(\overline{b_n})\{\overline{a_m}/\overline{\varphi_m}\} & \triangleq & f(b_0\{\overline{a_m}/\overline{\varphi_m}\}, \ldots, b_{n-1}\{\overline{a_m}/\overline{\varphi_m}\}) \\
\psi(\overline{t_n})\{\overline{a_m}/\overline{\varphi_m}\} & \triangleq & \begin{cases} u\{t_0\{\overline{a_m}/\overline{\varphi_m}\}, \ldots, t_{n-1}\{\overline{a_m}/\overline{\varphi_m}\}/dom(\Gamma)\} & \text{if } \psi = \varphi_i \text{ and } a_i = (\Gamma)\, u \\ \psi(t_0\{\overline{a_m}/\overline{\varphi_m}\}, \ldots, t_{n-1}\{\overline{a_m}/\overline{\varphi_m}\}) & \text{if } \psi \# \overline{\varphi_m} \end{cases} \\
S(\overline{b_n})\{\overline{a_m}/\overline{\varphi_m}\} & \triangleq & S(b_0\{\overline{a_m}/\overline{\varphi_m}\}, \ldots, b_{n-1}\{\overline{a_m}/\overline{\varphi_m}\}) \\
((\overline{y_n} : \overline{\sigma_n})\, u)\{\overline{a_m}/\overline{\varphi_m}\} & \triangleq & (y_0 : \sigma_0\{\overline{a_m}/\overline{\varphi_m}\}, \ldots, y_{n-1} : \sigma_{n-1}\{\overline{a_m}/\overline{\varphi_m}\})(u\{\overline{a_m}/\overline{\varphi_m}\}) \quad \text{if } \overline{y_n} \# \overline{a_m} \\
((\overline{y_n} : \overline{\sigma_n})\, \tau)\{\overline{a_m}/\overline{\varphi_m}\} & \triangleq & (y_0 : \sigma_0\{\overline{a_m}/\overline{\varphi_m}\}, \ldots, y_{n-1} : \sigma_{n-1}\{\overline{a_m}/\overline{\varphi_m}\})(\tau\{\overline{a_m}/\overline{\varphi_m}\}) \quad \text{if } \overline{y_n} \# \overline{a_m}
\end{array}
$$

Table 7.2: Substitutions in DMBEL

plication $t_1\{{}^{(\Gamma)\,u}/_{\varphi}\}, \ldots, t_n\{{}^{(\Gamma)\,u}/_{\varphi}\} = t'_1, \ldots, t'_n$, then return the term substitution $u\{{}^{t'_1,\ldots,t'_n}/_{dom(\Gamma)}\}$. Hereditary substitution is always terminating, thus well defined, since every case is obtained by means of recursive calls on smaller terms, possibly combining them with other terminating operations.

7.1.1 Type system

The type system of DMBEL comprises nine different judgment forms, whose deduction rules are mutually defined. In a sense, the most basic judgment form is the *signature well-formedness*, denoted $\vdash \Sigma$, ensuring that the types of the constants declared in Σ are well defined. Two more judgment forms are used to state the

well-formedness of abstraction contexts (notation: $\vdash_\Sigma \Phi$) and contexts (notation: $\Phi \vdash_\Sigma \Gamma$).

There are two formation judgments for types (notation: $\Phi;\Gamma \vdash_\Sigma \sigma$) and abstraction types ($\Phi;\Gamma \vdash_\Sigma \alpha$), essentially stating that all the free variables occurring in σ or α are contained in Γ or Φ, and that the type constant involved in the judgment is applied to well typed arguments, whose type matches that of the type constant.

The remaining four judgments are for the typing of terms (and abstraction terms) or *records* of terms (or of abstraction terms):

- $\Phi;\Gamma \vdash_\Sigma t \Longrightarrow \sigma$ states that t is a well typed term and has type σ
- $\Phi;\Gamma \vdash_\Sigma a \Longrightarrow \alpha$, states that a is a well typed abstraction term, whose type is α
- $\Phi;\Gamma \vdash_\Sigma \overline{t} \Longleftarrow \Delta$ means that the record $\overline{t}$ is well typed against the context Δ
- $\Phi;\Gamma \vdash_\Sigma \overline{a} \Longleftarrow \Psi$ means that the abstraction record $\overline{a}$ is well typed against the abstraction context Ψ.

In the last two judgments, Δ and Ψ are contexts used to express the dependent type of a record.

Table 7.3 shows the type system of DMBEL. Even though we are not interested in discussing all the rules, we think it is advisable to make a remark about the typing rules. The distinction between $\Longleftarrow$ and $\Longrightarrow$ is intentional and concerns the algorithmic interpretation of the type system: the dependent type of records is not unique, therefore, when typing a compound expression – e.g. a functional constant f applied to an abstraction record $\overline{a}$ – we cannot expect an algorithm to infer for $\overline{a}$ a type exactly matching the expected type for the arguments of f. In the style of bidirectional typechecking, we distinguish type inference judgments $\Phi;\Gamma \vdash_\Sigma e \Longrightarrow E$ for single expressions, where the typechecker takes in input a signature Σ, the contexts Φ and Γ, and an expression (term or abstraction term) e, and finally returns the type E of e (if e is well typed); and type checking judgments $\Phi;\Gamma \vdash_\Sigma \overline{e} \Longleftarrow \overline{E}$ for records, whose input also includes the expected type $\overline{E}$ of

Signatures:

$$\frac{}{\vdash []}\ (\text{SO-Empty}) \qquad \frac{\vdash_\Sigma \Phi}{\vdash \Sigma, S(\Phi)}\ (\text{SO-Tp}) \qquad \frac{\Phi; [] \vdash_\Sigma \sigma}{\vdash \Sigma, f(\Phi) : \sigma}\ (\text{SO-Tm})$$

Abstraction contexts:

$$\frac{\vdash \Sigma}{\vdash_\Sigma []}\ (\text{ACO-Empty}) \qquad \frac{\Phi; [] \vdash_\Sigma \alpha}{\vdash_\Sigma \Phi, \varphi : \alpha}\ (\text{ACO-Cons})$$

Contexts:

$$\frac{\vdash_\Sigma \Phi}{\Phi \vdash_\Sigma []}\ (\text{CO-Empty}) \qquad \frac{\Phi; \Gamma \vdash_\Sigma \sigma}{\Phi \vdash_\Sigma \Gamma, x : \sigma}\ (\text{CO-Cons})$$

Type and abstraction type formation:

$$\frac{S(\Psi) \in \Sigma \qquad \Phi; \Gamma \vdash_\Sigma \overline{a} \Leftarrow \Psi}{\Phi; \Gamma \vdash_\Sigma S(\overline{a})}\ (\text{TO-Intro}) \qquad \frac{\Phi; \Gamma, \Delta \vdash_\Sigma \sigma}{\Phi; \Gamma \vdash_\Sigma (\Delta)\,\sigma}\ (\text{ATO-Intro})$$

Term and abstraction term typing:

$$\frac{x : \sigma \in \Gamma \qquad \Phi \vdash_\Sigma \Gamma}{\Phi; \Gamma \vdash_\Sigma x \Longrightarrow \sigma}\ (\text{TI-Var}) \qquad \frac{f(\Psi) : \sigma \in \Sigma \qquad \Phi; \Gamma \vdash_\Sigma \overline{a} \Leftarrow \Psi}{\Phi; \Gamma \vdash_\Sigma f(\overline{a}) \Longrightarrow \sigma\{{}^{\overline{a}}/_{dom(\Psi)}\}}\ (\text{TI-Appcon})$$

$$\frac{\varphi : (\Delta)\,\tau \in \Phi \qquad \Phi; \Gamma \vdash_\Sigma \overline{t} \Leftarrow \Delta}{\Phi; \Gamma \vdash_\Sigma \varphi(\overline{t}) \Longrightarrow \tau\{{}^{\overline{t}}/_{dom(\Delta)}\}}\ (\text{TI-Appvar}) \qquad \frac{\Phi; \Gamma, \Delta \vdash_\Sigma t \Longrightarrow \sigma}{\Phi; \Gamma \vdash_\Sigma (\Delta)\,t \Longrightarrow (\Delta)\,\sigma}\ (\text{ATI-Intro})$$

Record typing:

$$\frac{\Phi \vdash_\Sigma \Gamma}{\Phi; \Gamma \vdash_\Sigma [] \Leftarrow []}\ (\text{TC-Empty}) \qquad \frac{\Phi; \Gamma \vdash_\Sigma t \Longrightarrow \sigma \qquad \Phi; \Gamma \vdash_\Sigma \overline{u} \Leftarrow \Delta\{{}^{t}/_{x}\}}{\Phi; \Gamma \vdash_\Sigma t, \overline{u} \Leftarrow x : \sigma, \Delta}\ (\text{TC-Cons})$$

Abstraction record typing:

$$\frac{\Phi \vdash_\Sigma \Gamma}{\Phi; \Gamma \vdash_\Sigma [] \Leftarrow []}\ (\text{ATC-Empty}) \qquad \frac{\Phi; \Gamma \vdash_\Sigma a \Longrightarrow \alpha \qquad \Phi; \Gamma \vdash_\Sigma \overline{b} \Leftarrow \Psi\{{}^{a}/_{\varphi}\}}{\Phi; \Gamma \vdash_\Sigma a, \overline{b} \Leftarrow \varphi : \alpha, \Psi}\ (\text{ATC-Cons})$$

Table 7.3: Typing rules of DMBEL

the record; the output of the typechecker would therefore be a boolean asserting whether the record $\overline{e}$ can be typed with type $\overline{E}$ or not.

7.1.2 Summary of DMBEL

The syntax of DMBEL poses an interesting case for a formalization in an interactive theorem prover. We here review some of its most challenging features:

- the syntax and the typing rules rely heavily on mutually inductive definitions, leading to verbose and complicated induction principles;
- multiple binding, which we addressed in the previous chapter, is here combined with dependent types: this detail can be expected to have a major impact on the formalization;
- hereditary substitution combines substitution and controlled reduction of terms; even though recently this operation has been investigated in the small setting of a formalization concerning the simply typed λ-calculus ([32]), its treatment in more involved languages is still challenging, as we will see in the following pages.

In the next section, we will propose a formal syntax of DMBEL based on the Sato representation.

7.2 Symbolic expressions in DMBEL

We now present the concrete syntax used in our formalization of DMBEL, following the simultaneous approach we have discussed in the previous chapter. We define the terms, types, abstraction terms and abstraction types as four CIC inductive types, here denoted as $\mathbb{S}^{tm}$, $\mathbb{S}^{tp}$, $\mathbb{S}^{atm}$ and $\mathbb{S}^{atp}$. It is also natural to consider symbolic expressions (noted as $\mathbb{S}$) as the disjoint union of these four syntactic categories. We will also use the notation $\overline{\mathbb{S}^{tm}}$, $\overline{\mathbb{S}^{tp}}$, $\overline{\mathbb{S}^{atm}}$ to refer to lists of symbolic expressions of

the proper class. Sometimes we will use letters $E, F, \ldots$ to refer to generic symbolic expressions. The rules defining DMBEL symbolic expressions are the following:

$$\frac{}{X : \mathbb{S}^{tm}} \qquad \frac{}{\langle x, i\rangle : \mathbb{S}^{tm}}$$

$$\frac{\overline{a} : \overline{\mathbb{S}^{atm}}}{f(\overline{a}) : \mathbb{S}^{tm}} \qquad \frac{\overline{t} : \overline{\mathbb{S}^{tm}}}{\varphi(\overline{t}) : \mathbb{S}^{tm}}$$

$$\frac{\overline{\sigma} : \overline{\mathbb{S}^{tp}} \qquad t : \mathbb{S}^{tm}}{[x; \overline{\sigma}]\ t : \mathbb{S}^{atm}} \qquad \frac{\overline{\sigma} : \overline{\mathbb{S}^{tp}} \qquad \tau : \mathbb{S}^{tp}}{[x; \overline{\sigma}]\ \tau : \mathbb{S}^{atp}}$$

$$\frac{\overline{a} : \overline{\mathbb{S}^{atm}}}{S(\overline{a}) : \mathbb{S}^{tp}}$$

The most important changes in the formalized syntax are in terms and abstraction terms. In terms, according to the simultaneous approach to multiple binding, it is necessary to split the informal term variables into (global) *term parameters* (taken from the infinite set $\mathbb{X}$ and denoted, as always, with capital $X, Y, \ldots$) and *term local variables*: the latter will be represented with pairs $\langle x, i\rangle$, where x is a local name (belonging to a distinguished set of local names $\mathbb{V}$) and i a natural number. There is no need to do the same for abstraction variables, since DMBEL does not have binders for abstraction variables: all abstraction variables are parameters. In the formalization, abstraction variables will be taken from the same $\mathbb{X}$ set used for term parameters, since the syntax is sufficient to discriminate the two cases. In this presentation of the formalization, however, we will note abstraction variables by the greek letters $\varphi, \psi, \ldots$ for clarity purposes.

Concrete abstraction terms bind a single local name x in a list of types $\overline{\sigma_n}$ and in a term t and are denoted $[x; \overline{\sigma_n}]\ t$. An occurrence of $\langle x, i\rangle$ inside t refers to the $(i+1)$-th entry of the multibinder, reading the list of abstracted types leftwards (i.e., it refers to the type σ_{n-i-1}). x is also bound in all types σ_j: if $\langle x, i\rangle$ occurs in σ_j, then it must be associated to the $(i+1)$-th entry leftwards from σ_j (i.e, to the type σ_{j-i-1}).

As we did in the formalization of the multivariate λ-calculus, we regard the case where the index of a local variable is too big for the associated binder as an ill-

formedness condition: more precisely, we say that if $\langle x, i\rangle$ occurs unbound in t (resp. σ_j) and $n \leq i$ (resp. $j \leq i$), then that occurrence is locally dangling in $[x, \overline{\sigma_n}]\ t$. As we explained in the previous chapter, our treatment of locally dangling variables allows a more elegant definition of substitution for local variables and thus much easier reasoning and formalization.

Similar considerations hold for abstraction types, binding a single local name x in a list of types $\overline{\sigma}$ and in a type τ, which are denoted $[x; \overline{\sigma}]\ \tau$. The other syntactic constructions are very close to the informal syntax.

We will now define the preliminary operations we need in order to formalize DMBEL.

Definition 7.1 *If π is a finite permutation on $\mathbb{X}$, the operation $\pi \cdot E$ permuting all term global variables in a symbolic expression E is defined as follows:*

$$\begin{aligned}
\pi \cdot X &\triangleq \pi(X) \\
\pi \cdot \langle x, i\rangle) &\triangleq \langle x, i\rangle \\
\pi \cdot f(a_1, \ldots, a_k) &\triangleq f(\pi \cdot a_1, \ldots, \pi \cdot a_k) \\
\pi \cdot \varphi(t_1, \ldots, t_k) &\triangleq \varphi(\pi \cdot t_1, \ldots, \pi \cdot t_k) \\
\pi \cdot S(a_1, \ldots, a_k) &\triangleq S(\pi \cdot a_1, \ldots, \pi \cdot a_k) \\
\pi \cdot [x; \sigma_1, \ldots, \sigma_k]\ t &\triangleq [x; \pi \cdot \sigma_1, \ldots, \pi \cdot \sigma_k]\ (\pi \cdot t) \\
\pi \cdot [x; \sigma_1, \ldots, \sigma_k]\ \tau &\triangleq [x; \pi \cdot \sigma_1, \ldots, \pi \cdot \sigma_k]\ (\pi \cdot \tau)
\end{aligned}$$

We also define the permutation of a context Γ as follows:

$$\pi \cdot \Gamma \triangleq \begin{cases} [] & \text{if } \Gamma = [] \\ \pi \cdot \Gamma', \pi(X) : \pi \cdot \sigma & \text{if } \Gamma = \Gamma', X : \sigma \end{cases}$$

Definition 7.2 *The list of the* global term variables *of a DMBEL symbolic expres-*

sion, noted $\mathsf{GV}(-)$*, is defined as follows:*

$$\begin{array}{rcl}
\mathsf{GV}(X) & \triangleq & \{X\} \\
\mathsf{GV}(\langle x, i\rangle) & \triangleq & \emptyset \\
\mathsf{GV}(f(a_1, \ldots, a_k)) & \triangleq & \bigcup_{1\leq i\leq k} \mathsf{GV}(a_i) \\
\mathsf{GV}(\varphi(t_1, \ldots, t_k)) & \triangleq & \bigcup_{1\leq i\leq k} \mathsf{GV}(t_i) \\
\mathsf{GV}(S(a_1, \ldots, a_k)) & \triangleq & \bigcup_{1\leq i\leq k} \mathsf{GV}(a_i) \\
\mathsf{GV}(\,[x; \sigma_1, \ldots, \sigma_k]\; t) & \triangleq & (\bigcup_{1\leq i\leq k} \mathsf{GV}(\sigma_i)) \cup \mathsf{GV}(t) \\
\mathsf{GV}(\,[x; \sigma_1, \ldots, \sigma_k]\; \tau) & \triangleq & (\bigcup_{1\leq i\leq k} \mathsf{GV}(\sigma_i)) \cup \mathsf{GV}(\tau)
\end{array}$$

Notice that this operation does not collect abstraction variables, because they are not needed our formalization.

Definition 7.3 *The list of the* local names *of a DMBEL symbolic expression, noted* $\mathsf{LV}(-)$*, is defined as follows:*

$$\begin{array}{rcl}
\mathsf{LV}(X) & \triangleq & \emptyset \\
\mathsf{LV}(\langle x, i\rangle) & \triangleq & \{x\} \\
\mathsf{LV}(f(a_1, \ldots, a_k)) & \triangleq & \bigcup_{1\leq i\leq k} \mathsf{LV}(a_i) \\
\mathsf{LV}(\varphi(t_1, \ldots, t_k)) & \triangleq & \bigcup_{1\leq i\leq k} \mathsf{LV}(t_i) \\
\mathsf{LV}(S(a_1, \ldots, a_k)) & \triangleq & \bigcup_{1\leq i\leq k} \mathsf{LV}(a_i) \\
\mathsf{LV}(\,[x; \sigma_1, \ldots, \sigma_k]\; t) & \triangleq & ((\bigcup_{1\leq i\leq k} \mathsf{LV}(\sigma_i)) \cup \mathsf{LV}(t)) \setminus \{x\} \\
\mathsf{LV}(\,[x; \sigma_1, \ldots, \sigma_k]\; \tau) & \triangleq & ((\bigcup_{1\leq i\leq k} \mathsf{LV}(\sigma_i)) \cup \mathsf{LV}(\tau)) \setminus \{x\}
\end{array}$$

Definition 7.4 *The simultaneous substitution replacing a list of parameters* $\overline{X_n}$ *with a list of terms* $\overline{u_n}$ *in a DMBEL symbolic expression, denoted* $-\{\overline{u_n}/\overline{X_n}\}$ *is defined*

as follows

$$
\begin{array}{rcl}
Y\{\overline{u_n}/\overline{X_n}\} & \triangleq & \begin{cases} u_i & \text{if } Y \in \overline{X_n} \text{ and } i = \mathtt{posn}(Y, \overline{X_n}) \\ Y & \text{if } Y \notin \overline{X_n} \end{cases} \\
\langle x, i\rangle\{\overline{u_n}/\overline{X_n}\} & \triangleq & \langle x, i\rangle \\
f(a_1, \ldots, a_k)\{\overline{u_n}/\overline{X_n}\} & \triangleq & f(a_1\{\overline{u_n}/\overline{X_n}\}, \ldots, a_k\{\overline{u_n}/\overline{X_n}\}) \\
\varphi(t_1, \ldots, t_k)\{\overline{u_n}/\overline{X_n}\} & \triangleq & \varphi(t_1\{\overline{u_n}/\overline{X_n}\}, \ldots, t_k\{\overline{u_n}/\overline{X_m}\}) \\
S(a_1, \ldots, a_k)\{\overline{u_n}/\overline{X_n}\} & \triangleq & S(a_1\{\overline{u_n}/\overline{X_n}\}, \ldots, a_k\{\overline{u_n}/\overline{X_n}\}) \\
([x; \sigma_1, \ldots, \sigma_k]\ t)\{\overline{u_n}/\overline{X_n}\} & \triangleq & [x; \sigma_1\{\overline{u_n}/\overline{X_n}\}, \ldots \sigma_k\{\overline{u_n}/\overline{X_n}\}]\ (t\{\overline{u_n}/\overline{X_n}\}) \\
([x; \sigma_1, \ldots, \sigma_k]\ \tau)\{\overline{u_n}/\overline{X_n}\} & \triangleq & [x; \sigma_1\{\overline{u_n}/\overline{X_n}\}, \ldots \sigma_k\{\overline{u_n}/\overline{X_n}\}]\ (\tau\{\overline{u_n}/\overline{X_n}\})
\end{array}
$$

We also define a notation for mapping this notion of substitution to a context:

$$
\Gamma\{\overline{u_n}/\overline{X_n}\} \triangleq \begin{cases} [] & \text{if } \Gamma = [] \\ \Gamma'\{\overline{u_n}/\overline{X_n}\}, Y : \sigma\{\overline{u_n}/\overline{X_n}\} & \text{if } \Gamma = \Gamma', Y : \sigma \end{cases}
$$

Notice that in this algorithmic definition, we do not require the two input lists to have the same length: however, in the formalization, this property will always be satisfied. With a small notational abuse, in the rest of the chapter we will write $\Gamma\{\overline{t}/\overline{X}\}$ for the operation replacing $\overline{X}$ with $\overline{t}$ in the codomain of Γ. Also notice that $\mathtt{posn}(Y, \overline{X_n})$, as always, returns the largest index i such that $X_i = Y$.

We also give a substitution replacing local variables with terms. Since many local variables can be associated to the same local name (differing only by their index), this operation replaces all the local variables sharing a certain local name, with a list of terms; the index of the local variable is used to choose one of the terms inside the list.

Definition 7.5 *The substitution replacing all the occurrences of a local name x with a list of terms $\overline{u_n}$ in a DMBEL symbolic expression, denoted $-\{\overline{u_n}/x\}$ is defined as*

follows

$$
\begin{array}{rcl}
X\{\overline{u_n}/x\} & \triangleq & X \\
\langle y, i\rangle\{\overline{u_n}/x\} & \triangleq & \begin{cases} u_i & \text{if } x = y \text{ and } i < n \\ \langle y, i\rangle & \text{else} \end{cases} \\
f(\overline{a_k})\{\overline{u_n}/x\} & \triangleq & f(a_0\{\overline{u_n}/x\}, \ldots, a_{k-1}\{\overline{u_n}/x\}) \\
\varphi(\overline{t_k})\{\overline{u_n}/x\} & \triangleq & \varphi(t_0\{\overline{u_n}/x\}, \ldots, t_{k-1}\{\overline{u_n}/x\}) \\
S(\overline{a_k})\{\overline{u_n}/x\} & \triangleq & S(a_0\{\overline{u_n}/x\}, \ldots, a_{k-1}\{\overline{u_n}/x\}) \\
(\,[y; \overline{\sigma_k}]\ t)\{\overline{u_n}/x\} & \triangleq & \begin{cases} [y; \overline{\sigma_k}]\ t & \text{if } x = y \\ [y; \sigma_0\{\overline{u_n}/x\}, \ldots \sigma_{k-1}\{\overline{u_n}/x\}]\ (t\{\overline{u_n}/x\}) & \text{else} \end{cases} \\
(\,[y; \overline{\sigma_k}]\ \tau)\{\overline{u_n}/x\} & \triangleq & \begin{cases} [y; \overline{\sigma_k}]\ \tau & \text{if } x = y \\ [y; \sigma_0\{\overline{u_n}/x\}, \ldots \sigma_{k-1}\{\overline{u_n}/x\}]\ (\tau\{\overline{u_n}/x\}) & \text{else} \end{cases}
\end{array}
$$

Definition 7.6 *The* variable opening *operation, substituting a list of parameters* $\overline{X_n}$ *for a local name* x *in a symbolic expression* E*, denoted* $E[x \mapsto \overline{X_n}]$ *is defined as*

$$E[x \mapsto \overline{X_n}] \triangleq E\{\overline{X_n}/x\}$$

Analogously, we define the variable opening operation turning a list of types into a context as follows

$$(\overline{\sigma_n})[x \mapsto \overline{X_n}] \triangleq \begin{cases} [] & \text{if } \overline{\sigma_n} = [] \\ (\overline{\sigma_{n-1}})[x \mapsto \overline{X_{n-1}}], X_n : (\sigma_n[x \mapsto \overline{X_{n-1}}]) & \text{else} \end{cases}$$

Variable opening is used when opening an abstraction, to replace dangling local variables with fresh parameters. The converse operation to variable opening is variable closure, used when constructing an abstraction.

Definition 7.7 *The* variable closure *operation, substituting local variables* $\langle x, 0\rangle, \ldots, \langle x, n-1\rangle$ *for a list of parameters* $\overline{X_n}$ *in a symbolic expression* E*, denoted* $E[\overline{X_n} \mapsto x]$ *is defined as follows:*

$$E[\overline{X_n} \mapsto x] \triangleq E\{^{\langle x,0\rangle,\ldots,\langle x,n-1\rangle}/_{\overline{X_n}}\}$$

Analogously, we define the variable closure operation turning a context Γ *into a list of types as follows*

$$[\Gamma \mapsto x] \triangleq \begin{cases} [] & \text{if } \Gamma = [] \\ [\Gamma' \mapsto x], \sigma[dom(\Gamma') \mapsto x] & \text{if } \Gamma = \Gamma', X : \sigma \end{cases}$$

Following the example of the multivariate λ-calculus formalization, we formulate the following extended properties (proved in the formalization by induction on symbolic expressions).

Lemma 7.1

1. *If* $\overline{X} \# E$*, then* $E\{\overline{t}/\overline{X}\} = E$*.*
2. *If* $x \notin \mathsf{LV}(E)$*, then* $E\{\overline{M}/x\} = E$*.*

Lemma 7.2 *If* $\overline{X} \# \overline{Y} @ \overline{u}$*, then for all symbolic expressions* E *we have*

$$E\{\overline{t}/\overline{X}\}\{\overline{u}/\overline{Y}\} = E\{\overline{u}/\overline{Y}\}\{\overline{t\{\overline{u}/\overline{Y}\}}/\overline{X}\}$$

Lemma 7.3 *Substitution for global variables is equivariant: if* π *is a finite permutation of global variables, then*

$$\begin{aligned} \pi \cdot (E\{\overline{t}/\overline{X}\}) &= (\pi \cdot E)\{\overline{\pi \cdot t}/\overline{\pi \cdot X}\} \\ \pi \cdot (E\{\overline{t}/x\}) &= (\pi \cdot E)\{\overline{\pi \cdot t}/x\} \end{aligned}$$

Corollary 7.4

1. *Closure of symbolic expressions is equivariant:*

$$\pi \cdot (E[\overline{X} \mapsto x]) = (\pi \cdot E)[\pi \cdot \overline{X} \mapsto x]$$

2. *Closure of contexts is equivariant:*

$$\pi \cdot [\Gamma \mapsto x] = [\pi \cdot \Gamma \mapsto x]$$

Proof: Follows from Lemma 7.3 after unfolding the definition of closure. Part 2 requires induction on Γ. $\square$

7.3 Well formed expressions

Similarly to the Abs operation for the λ-calculus, we also want to provide a defined notation for building abstraction terms and types in a way that is closer to the informal notion of abstraction. As always, these operation will be parametric of a height function F, which in the case of DMBEL must have type $\mathtt{list}\ (\mathbb{X} \times \mathtt{tp}) \to \mathbb{S} \to \mathbb{V}$, where $\mathtt{list}\ (\mathbb{X} \times \mathtt{tp})$ is the type of contexts. We will use contexts to compute heights, because they encode all the nested scopes defined by a dependent multi-binder.

We now define the operations to build canonical abstractions, which we will call "build operations", and for which we will use the same notation of informal abstractions.

Definition 7.8 ("build" operations) *The function* $\mathtt{build\_atm} : \mathtt{list}\ (\mathbb{X} \times \mathtt{tp}) \to \mathtt{tm} \to \mathtt{atm}$*, is defined, parametrically on a height* F*, as follows*

$$\mathtt{build\_atm}\ \Gamma\ t \triangleq [\mathsf{F}_\Gamma(t); [\Gamma \mapsto \mathsf{F}_\Gamma(t)]]\ t[dom(\Gamma) \mapsto \mathsf{F}_\Gamma(t)]$$

Similarly, the function $\mathtt{build\_atp} : \mathtt{list}\ (\mathbb{X} \times \mathtt{tp}) \to \mathtt{tp} \to \mathtt{atp}$*, is defined, parametrically on a height* F*, as follows*

$$\mathtt{build\_atp}\ \Gamma\ \tau \triangleq [\mathsf{F}_\Gamma(\tau); [\Gamma \mapsto \mathsf{F}_\Gamma(\tau)]]\ \tau[dom(\Gamma) \mapsto \mathsf{F}_\Gamma(\tau)]$$

We will write $(\Gamma)\,t$ *and* $(\Gamma)\,\tau$ *as a compact notation respectively for* $\mathtt{build\_atm}\ \Gamma\ \mathtt{t}$ *and for* $\mathtt{build\_atm}\ \Gamma\ \tau$*.*

Definition 7.9 (instantiation) *The instantiation operation is defined on abstraction terms and abstraction types as follows:*

$$[x; \overline{\sigma_n}]\ t \blacktriangledown \overline{u_n} \triangleq t\{\overline{u_n}/x\}$$

$$[x; \overline{\sigma_n}]\ \tau \blacktriangledown \overline{u_n} \triangleq \tau\{\overline{u_n}/x\}$$

Before giving the rules defining well formed terms, we still need a notion that identifies a more tractable form of contexts.

Definition 7.10 (regular context) *A context* Γ *is said to be* regular *if and only if*

- *all the names of its domain are distinct*
- *if* $\Gamma = \Gamma', X : \sigma, \Gamma''$*, then* $X \# cod(\Gamma')$*; in other words, a parameter can occur in the codomain of the context only after its declaration.*

Regular contexts are a natural definition, because they are the contexts involved in typing judgments. They are also much nicer to work with than generic contexts, since many natural properties only hold for regular contexts.

Lemma 7.5 *If* Γ *is a regular context,* t *is a term and* σ *is a type, then* $dom(\Gamma) \# (\Gamma)\, t$ *and* $dom(\Gamma) \# (\Gamma)\, \sigma$*.*

Table 7.4 shows the rules defining the well-formedness judgment for DMBEL expressions formalized in our encoding. These judgments and their rules are formalized in Matita as the four mutual inductive predicates `Ltm`, `Ltp`, `Latm` and `Latp`, one for each kind of symbolic expression. Also notice that the rules, being defined on top of the build operations, are parametric on some height function F.

Parameters are canonical, and so are applications whose components are all canonical. Local variables are only created in the rules involving abstractions using the build operations, ensuring, as a consequence, that they correspond to the correct heights. Rules involving abstractions also require that the contexts involved in the proof of canonicity be regular. We believe that reasoning on regular contexts (essentially enforcing the property known as "Barendregt's convention" [9, page 26]) is considerably easier.

We now argue that well-formed symbolic expressions are an adequate representation of informal DMBEL expressions: in fact they are variable closed and do not contain locally dangling local variables.

Lemma 7.6 *If* $E : \mathbb{L}$*, then* $\mathsf{LV}(E) = \emptyset$*.*

$$\frac{}{X : \mathbb{L}}\ (\textsc{Ltm-Var}) \qquad \frac{\text{for all } b \in \overline{a},\ b : \mathbb{L}}{f(\overline{a}) : \mathbb{L}}\ (\textsc{Ltm-Appcon})$$

$$\frac{\text{for all } u \in \overline{t},\ u : \mathbb{L}}{\varphi(\overline{t}) : \mathbb{L}}\ (\textsc{Ltm-Appvar}) \qquad \frac{\begin{array}{c} t : \mathbb{L} \\ \Gamma \text{ is regular} \\ \text{for all } \sigma \in \mathit{cod}(\Gamma),\ \sigma : \mathbb{L} \end{array}}{(\Gamma)\, t : \mathbb{L}}\ (\textsc{Latm-Mk-Atm})$$

$$\frac{\text{for all } b \in \overline{a},\ b : \mathbb{L}}{S(\overline{a}) : \mathbb{L}}\ (\textsc{Ltm-Mk-Tp}) \qquad \frac{\begin{array}{c} \tau : \mathbb{L} \\ \Gamma \text{ is regular} \\ \text{for all } \sigma \in \mathit{cod}(\Gamma),\ \sigma : \mathbb{L} \end{array}}{(\Gamma)\, \tau : \mathbb{L}}\ (\textsc{Latp-Mk-Atp})$$

Table 7.4: Canonical DMBEL expressions

Proof: Standard proof by induction on the derivation of $E : \mathbb{L}$. □

To show that abstractions do not contain locally dangling local variables, we proceed similarly to the corresponding proof in the multivariate λ-calculus, using a "next index" operator on symbolic expressions:

$$\begin{array}{rcll}
\mathrm{ni}(X, x) & \triangleq & 0 & \\
\mathrm{ni}(\langle x, i\rangle, x) & \triangleq & i + 1 & \\
\mathrm{ni}(\langle y, i\rangle, x) & \triangleq & 0 & \text{when } x \neq y \\
\mathrm{ni}(f(\overline{a}), x) & \triangleq & \mathrm{ni}(\overline{a}, x) & \\
\mathrm{ni}(S(\overline{a}), x) & \triangleq & \mathrm{ni}(\overline{a}, x) & \\
\mathrm{ni}(\varphi(\overline{t}), x) & \triangleq & \mathrm{ni}(\overline{t}, x) & \\
\mathrm{ni}(\,[x, \overline{\sigma_n}]\ t, x) & \triangleq & 0 & \\
\mathrm{ni}(\,[y, \overline{\sigma_n}]\ t, x) & \triangleq & \max\{\mathrm{ni}(\overline{\sigma_n}, x), \mathrm{ni}(t, x)\} & \text{when } x \neq y \\
\mathrm{ni}(\,[x, \overline{\sigma_n}]\ \tau, x) & \triangleq & 0 & \\
\mathrm{ni}(\,[y, \overline{\sigma_n}]\ \tau, x) & \triangleq & \max\{\mathrm{ni}(\overline{\sigma_n}, x), \mathrm{ni}(\tau, x)\} & \text{when } x \neq y \\
\mathrm{ni}([E_1; \ldots; E_k], x) & \triangleq & \max\{\mathrm{ni}(E_1, x), \ldots, \mathrm{ni}(E_k, x)\} &
\end{array}$$

Lemma 7.7

1. *If* $[x, \overline{\sigma_n}]\ t : \mathbb{L}$*, then* $\mathrm{ni}(t, x) \leq n$ *and for all* $i = 0, \ldots, n-1$*,* $\mathrm{ni}(\sigma_i, x) \leq i$*: hence* x *is not locally dangling in* $[x, \overline{\sigma_n}]\ t$*.*

2. *If* $[x, \overline{\sigma_n}]\ \tau : \mathbb{L}$, *then* $\mathrm{ni}(\tau, x) \leq n$ *and for all* $i = 0, \ldots, n-1$, $\mathrm{ni}(\sigma_i, x) \leq i$: *hence* x *is not locally dangling in* $[x, \overline{\sigma_n}]\ \tau$.

Proof: The proof is similar to the one presented in Lemma 6.17, with an added induction on the list of abstracted types. □

7.3.1 An excellent height for DMBEL

As always, to state the properties of height functions, we define an auxiliary function E. Its definition, however, is complicated by the fact that we must deal with multi-binders defining nested scopes, rather than regular binders with single scopes.

$$
\begin{array}{rcl}
\mathsf{E}'_{\overline{X}}(Y) & \triangleq & \emptyset \\
\mathsf{E}'_{\overline{X}}(\langle x, i\rangle) & \triangleq & \emptyset \\
\mathsf{E}'_{\overline{X}}(f(\overline{a_n})) & \triangleq & \mathsf{E}'_{\overline{X}}(a_0) \cup \ldots \cup \mathsf{E}'_{\overline{X}}(a_{n-1}) \\
\mathsf{E}'_{\overline{X}}(\varphi(\overline{t_n})) & \triangleq & \mathsf{E}'_{\overline{X}}(t_0) \cup \ldots \cup \mathsf{E}'_{\overline{X}}(t_{n-1}) \\
\mathsf{E}'_{\overline{X}}(S(\overline{a_n})) & \triangleq & \mathsf{E}'_{\overline{X}}(a_0) \cup \ldots \cup \mathsf{E}'_{\overline{X}}(a_{n-1}) \\
\mathsf{E}'_{\overline{X}}([x; \overline{\sigma_n}]\ t) & \triangleq & \begin{cases} \emptyset & \text{if } \overline{X} \# [x; \overline{\sigma_n}]\ t \\ \{x\} \cup \mathsf{E}'_{\overline{X}}(t) \cup \left(\bigcup_{0 \leq i < n} \mathsf{E}'_{\overline{X}}(\sigma_i)\right) & \text{else} \end{cases} \\
\mathsf{E}'_{\overline{X}}([x; \overline{\sigma_n}]\ \tau) & \triangleq & \begin{cases} \emptyset & \text{if } \overline{X} \# [x; \overline{\sigma_n}]\ \tau \\ \{x\} \cup \mathsf{E}'_{\overline{X}}(\tau) \cup \left(\bigcup_{0 \leq i < n} \mathsf{E}'_{\overline{X}}(\sigma_i)\right) & \text{else} \end{cases} \\
\mathsf{E}_{\Gamma}(E) & \triangleq & \begin{cases} \emptyset & \text{if } \Gamma = \emptyset \\ \mathsf{E}'_{dom(\Gamma)}(E) \cup \mathsf{E}_{\Gamma'}(\sigma) & \text{if } \Gamma = \Gamma', X : \sigma \end{cases}
\end{array}
$$

The definition actually uses an auxiliary function $\mathsf{E}'_{\overline{X}}(E)$, to compute the list of those local variables occurring in binding position on a path between the root of a symbolic expression E and any occurrence of a parameter belonging to the list $\overline{X}$. The function E_{Γ}(E) iterates E' taking into account the multiple scopes defined by the context Γ. For example, in the case of a context $\Gamma = X_1 : \sigma_1, X_2 : \sigma_2, X_3 : \sigma_3$, $\mathsf{E}_{\Gamma}(E)$ will produce the set $\mathsf{E}'_{X_1,X_2,X_3}(E) \cup \mathsf{E}'_{X_1,X_2}(\sigma_3) \cup \mathsf{E}'_{X_1}(\sigma_2)$.

Lemma 7.8 *If* $|\overline{X}| = |\overline{t}|$, $x \notin \mathsf{LV}(E)$ *and* $x \notin \mathsf{E}'_{\overline{X}}(E)$, *then* $E[\overline{X} \mapsto x]\{{}^{\overline{t}}/_{x}\} = E\{{}^{\overline{t}}/_{\overline{X}}\}$.

Proof: The proof, by structural induction, follows the spirit of Lemma 6.18. □

We are now ready to look for a well-behaved – we will say, like in chapter 6, *excellent* – height function in the context of DMBEL. First, we will define what we mean by "well-behaved", then we will propose a candidate height function; finally, we will prove that the proposed function is really well-behaved.

Definition 7.11 *A function* $\mathsf{H} : \mathtt{list}\ (\mathbb{X} \times \mathtt{tp}) \to \mathbb{S} \to \mathbb{V}$ *is an* excellent *height for DMBEL if the following three properties hold:*

(DHE) H *is* equivariant*: for all finite permutations* π*, contexts* Γ*, symbolic expressions* E*,* $\mathsf{H}_\Gamma(E) = \mathsf{H}_{\pi\cdot\Gamma}(\pi \cdot E)$*;*

(DHF) H *is* fresh*: for all contexts* Γ *and symbolic expressions* E*,* $\mathsf{H}_\Gamma(E) \notin \mathsf{E}_\Gamma(E)$*;*

(DHP) H *preserves term substitution: for all contexts* Γ*, symbolic expressions* E*, lists of parameters* $\overline{X}$ *and lists of terms* $\bar{t}$*, if* $\overline{X}$ *and* $\bar{t}$ *have the same length, and* $dom(\Gamma) \# \overline{X}, \bar{t}$*, then* $\mathsf{H}_\Gamma(E) = \mathsf{H}_{\Gamma\{\bar{t}/\overline{X}\}}(E\{\bar{t}/\overline{X}\})$*.*

Definition 7.12 *The height function* $\mathsf{F} : \mathtt{list}\ (\mathbb{X} \times \mathtt{tp}) \to \mathbb{S} \to \mathbb{V}$ *is defined as follows:*

$$
\begin{array}{rcl}
\mathsf{F}'_{\overline{X}}(Y) & \triangleq & \begin{cases} 1 & \text{if } Y \in \overline{X} \\ 0 & \text{else} \end{cases} \\
\mathsf{F}'_{\overline{X}}(f(\overline{a_n})) & \triangleq & \max\{\mathsf{F}'_{\overline{X}}(a_0), \ldots, \mathsf{F}'_{\overline{X}}(a_{n-1})\} \\
\mathsf{F}'_{\overline{X}}(\varphi(\overline{t_n})) & \triangleq & \max\{\mathsf{F}'_{\overline{X}}(t_0), \ldots, \mathsf{F}'_{\overline{X}}(t_{n-1})\} \\
\mathsf{F}'_{\overline{X}}(S(\overline{a_n})) & \triangleq & \max\{\mathsf{F}'_{\overline{X}}(a_0), \ldots, \mathsf{F}'_{\overline{X}}(a_{n-1})\} \\
\mathsf{F}'_{\overline{X}}(\,[x;\overline{\sigma_n}]\ t) & \triangleq & \begin{cases} \max\{\mathsf{F}'_{\overline{X}}(\sigma_0), \ldots, \mathsf{F}'_{\overline{X}}(\sigma_{n-1}), \mathsf{F}'_{\overline{X}}(t)\} \\ \qquad \text{if } \max\{\mathsf{F}'_{\overline{X}}(\sigma_0), \ldots, \mathsf{F}'_{\overline{X}}(\sigma_{n-1}), \mathsf{F}'_{\overline{X}}(t)\} = 0 \text{ or} > x \\ x+1 \quad \text{else} \end{cases} \\
\mathsf{F}'_{\overline{X}}(\,[x;\overline{\sigma_n}]\ \tau) & \triangleq & \begin{cases} \max\{\mathsf{F}'_{\overline{X}}(\sigma_0), \ldots, \mathsf{F}'_{\overline{X}}(\sigma_{n-1}), \mathsf{F}'_{\overline{X}}(\tau)\} \\ \qquad \text{if } \max\{\mathsf{F}'_{\overline{X}}(\sigma_0), \ldots, \mathsf{F}'_{\overline{X}}(\sigma_{n-1}), \mathsf{F}'_{\overline{X}}(\tau)\} = 0 \text{ or} > x \\ x+1 \quad \text{else} \end{cases} \\
\mathsf{F}_\emptyset(E) & \triangleq & 0 \\
\mathsf{F}_{\Gamma,X:\sigma}(E) & \triangleq & \max\{\mathsf{F}_\Gamma(\sigma), \mathsf{F}'_{dom(\Gamma),X}(E)\}
\end{array}
$$

Lemma 7.9 *For all symbolic expressions E and lists of global variables $\overline{X}$, $\mathsf{F}'_{\overline{X}}(E) = 0$ if and only if $\overline{X} \# E$.*

Sketch of Proof: Follows trivially by structural induction on E. □

To prove the excellence property for F, it is convenient to state similar properties for the auxiliary function F'; the proof that F is excellent will then be subsumed as an easy corollary. Thus, we define the following auxiliary properties:

(DHE') for all symbolic expressions E, lists of global variables $\overline{X}$ and finite permutations π, $\mathsf{F}'_{\overline{X}}(E) = \mathsf{F}'_{\pi\cdot\overline{X}}(\pi \cdot E)$;

(DHF') for all symbolic expressions E, lists of global variables $\overline{X}$, local variables x, if $x \geq \mathsf{F}'_{\overline{X}}(E)$, then $x \notin \mathsf{E}'_{\overline{X}}(E)$;

(DHP') for all lists of global variables $\overline{X}$ and $\overline{Y}$, lists of terms $\bar{t}$ and symbolic expressions E, if $\overline{Y}$ and $\bar{t}$ have the same length, and if $\overline{X} \# \overline{Y}, \bar{t}$, then $\mathsf{F}'_{\overline{X}}(E) = \mathsf{F}'_{\overline{X}}(E\{{}^{\bar{t}}/_{\overline{Y}}\})$.

Theorem 7.10 *The three properties **(DHE')**, **(DHF')** and **(DHP')** hold.*

Proof: We prove the three properties separately.

(DHE') By structural induction on E:

- case $E = Y$: Y is either in $\overline{X}$ or not; if it is, then $\pi\cdot Y \in \pi\cdot\overline{X}$ too, which implies $\mathsf{F}'_{\overline{X}}(Y) = \mathsf{F}'_{\pi\cdot\overline{X}}(\pi \cdot Y) = 1$; if on the contrary $Y \notin \overline{X}$, then $\pi \cdot Y \notin \pi \cdot \overline{X}$ (π is a permutation, therefore it is injective), implying $\mathsf{F}'_{\overline{X}}(Y) = \mathsf{F}'_{\pi\cdot\overline{X}}(\pi \cdot Y) = 0$;
- case $E = \langle x, i\rangle$: $\mathsf{F}'_{\overline{X}}(\langle x, i\rangle) = \mathsf{F}'_{\pi\cdot\overline{X}}(\pi\cdot\langle x, i\rangle) = \mathsf{F}'_{\pi\cdot\overline{X}}(\langle x, i\rangle) = 0$ holds trivially;
- case $E = [x; \sigma_1, \ldots, \sigma_n]\ t$: define

$$y \triangleq \max\{\mathsf{F}'_{\overline{X}}(\sigma_1), \ldots, \mathsf{F}'_{\overline{X}}(\sigma_n), \mathsf{F}'_{\overline{X}}(t)\}$$

by induction hypothesis, we know that $\mathsf{F}'_{\overline{X}}(t) = \mathsf{F}'_{\pi\cdot\overline{X}}(\pi \cdot t)$ and that, for all $\sigma \in \sigma_1, \ldots, \sigma_n$, $\mathsf{F}'_{\overline{X}}(\sigma) = \mathsf{F}'_{\pi\cdot\overline{X}}(\pi \cdot \sigma)$; this also implies that

$$y = \max\{\mathsf{F}'_{\pi\cdot\overline{X}}(\pi \cdot \sigma_1), \ldots, \mathsf{F}'_{\pi\cdot\overline{X}}(\pi \cdot \sigma_n), \mathsf{F}'_{\pi\cdot\overline{X}}(\pi \cdot t)\}$$

there are two cases: if $0 < y \leq x$, then $\mathsf{F}'_{\overline{X}}(\,[x; \sigma_1, \ldots, \sigma_n]\; t) = \mathsf{F}'_{\pi\cdot\overline{X}}(\pi \cdot (\,[x; \sigma_1, \ldots, \sigma_n]\; t)) = x + 1$; if $y = 0$ or $y > x$, then $\mathsf{F}'_{\overline{X}}(\,[x; \sigma_1, \ldots, \sigma_n]\; t) = \mathsf{F}'_{\pi\cdot\overline{X}}(\pi \cdot (\,[x; \sigma_1, \ldots, \sigma_n]\; t)) = y$;

- case $E = [x; \overline{\sigma}]\; \tau$ is proved similarly to the previous case;
- case $E = f(a_1, \ldots, a_n)$: we must prove that

  $$\max\{\mathsf{F}'_{\overline{X}}(a_1), \ldots, \mathsf{F}'_{\overline{X}}(a_n)\} = \max\{\mathsf{F}'_{\pi\cdot\overline{X}}(\pi \cdot a_1), \ldots, \mathsf{F}'_{\pi\cdot\overline{X}}(\pi \cdot a_n)\}$$

  the equation is satisfied, since the arguments of the two max functions are pairwise equal by induction hypothesis;
- cases $E = \varphi(\overline{t})$ and $E = S(\overline{a})$ are proved similarly to the previous case.

(DHF') By structural induction on E:

- cases $E = Y$ or $E = y$: in both cases, $\mathsf{E}'_{\overline{X}}(E) = \emptyset$, thus the thesis follows trivially;
- case $E = [y; \sigma_1, \ldots, \sigma_n]\; t$: define

  $$z \triangleq \max\{\mathsf{F}'_{\overline{X}}(\sigma_1), \ldots, \mathsf{F}'_{\overline{X}}(\sigma_n), \mathsf{F}'_{\overline{X}}(t)\}$$
  $$z' \triangleq \mathsf{F}'_{\overline{X}}(\,[y; \sigma_1, \ldots, \sigma_n]\; t)$$

  by induction hypothesis, we know that

  $$\forall x' \geq \mathsf{F}'_{\overline{X}}(t) : x' \notin \mathsf{E}'_{\overline{X}}(t)$$
  $$\forall \sigma \in \sigma_1, \ldots, \sigma_n; \forall x' \geq \mathsf{F}'_{\overline{X}}(\sigma) : x' \notin \mathsf{E}'_{\overline{X}}(\sigma)$$

  notice in particular that both properties hold for $x' = z$ or greater, since z satisfies the required inequation; then consider two subcases:

  - if $\overline{X} \# [y; \sigma_1, \ldots, \sigma_n]\, t$, then $\mathsf{E'}_{\overline{X}}([y; \sigma_1, \ldots, \sigma_n]\, t) = \emptyset$, thus the thesis follows trivially;
  - if $\overline{X} \cap GV([y; \sigma_1, \ldots, \sigma_n]\, t) \neq \emptyset$, we must prove that

$$z' \notin y, \mathsf{E'}_{\overline{X}}(\sigma_1) \cup \ldots \cup \mathsf{E'}_{\overline{X}}(\sigma_n) \cup \mathsf{E'}_{\overline{X}}(t)$$

    we know by lemma 7.9 that $z \neq 0$, therefore either $0 < z \leq y$, implying $z' = y + 1$ (thus also $z' > z$), or $z > y$ (implying $z' = z$): in both cases, the property follows by induction hypotheses.

- case $E = [x; \overline{\sigma}]\, \tau$ is proved similarly to the previous case;
- case $E = f(a_1, \ldots, a_n)$: we must prove that

$$\forall x \geq \max\{\mathsf{F'}_{\overline{X}}(a_1), \ldots, \mathsf{F'}_{\overline{X}}(a_n)\} : x \notin \mathsf{E'}_{\overline{X}}(a_1) \cup \ldots \cup \mathsf{E'}_{\overline{X}}(a_n)$$

  the property is satisfied, since we can prove by induction hypothesis that x is not in any of the subsets $\mathsf{E'}_{\overline{X}}(a_1), \ldots, \mathsf{E'}_{\overline{X}}(a_n)$;
- cases $E = \varphi(\overline{t})$ and $E = S(\overline{a})$ are proved similarly to the previous case.

(DHP') By structural induction on E:

- case $E = Z$: Y is either in $\overline{Y}$ or not; if it is, then $Z \notin \overline{X}$ and $Z\{\overline{t}/\overline{Y}\} = t_i$ for some $t_i \in \overline{t}$: this implies $\mathsf{F'}_{\overline{X}}(Z) = 0 = \mathsf{F'}_{\overline{X}}(t_i)$ by lemma 7.9, since $X \# \overline{t}$ by hypothesis; if $Z \notin \overline{Y}$, then $Z\{\overline{t}/\overline{Y}\} = Z$ and the thesis follows trivially;
- case $E = x$: $x\{\overline{t}/\overline{Y}\} = x$, thus the thesis follows trivially;
- case $E = [x; \sigma_1, \ldots, \sigma_n]\, u$: define

$$y \triangleq \max\{\mathsf{F'}_{\overline{X}}(\sigma_1), \ldots, \mathsf{F'}_{\overline{X}}(\sigma_n), \mathsf{F'}_{\overline{X}}(u)\}$$

  by induction hypothesis, we know that $\mathsf{F'}_{\overline{X}}(u) = \mathsf{F'}_{\overline{X}}(u\{\overline{t}/\overline{Y}\})$ and that, for all $\sigma \in \sigma_1, \ldots, \sigma_n$, $\mathsf{F'}_{\overline{X}}(\sigma) = \mathsf{F'}_{\overline{X}}(\sigma\{\overline{t}/\overline{Y}\})$; this also implies that

$$y = \max\{\mathsf{F'}_{\pi \cdot \overline{X}}(\pi \cdot \sigma_1), \ldots, \mathsf{F'}_{\pi \cdot \overline{X}}(\pi \cdot \sigma_n), \mathsf{F'}_{\pi \cdot \overline{X}}(\pi \cdot t)\}$$

there are two cases: if $0 < y \leq x$, then $\mathsf{F}'_{\overline{X}}([x;\sigma_1,\ldots,\sigma_n]\ t) = \mathsf{F}'_{\overline{X}}(([x;\sigma_1,\ldots,\sigma_n]\ t)\{\overline{t}/\overline{Y}\}) = x{+}1$; if $y = 0$ or $y > x$, then $\mathsf{F}'_{\overline{X}}([x;\sigma_1,\ldots,\sigma_n]\ t) = \mathsf{F}'_{\overline{X}}(([x;\sigma_1,\ldots,\sigma_n]\ t)\{\overline{t}/\overline{Y}\}) = y$;

- case $E = [x;\overline{\sigma_1}]\ \tau$ is proved similarly to the previous case;
- case $E = f(a_1,\ldots,a_n)$: we must prove that

$$\max\{\mathsf{F}'_{\overline{X}}(a_1),\ldots,\mathsf{F}'_{\overline{X}}(a_n)\} = \max\{\mathsf{F}'_{\overline{X}}(a_1\{\overline{t}/\overline{Y}\}),\ldots,\mathsf{F}'_{\overline{X}}(a_n\{\overline{t}/\overline{Y}\})\}$$

the equation is satisfied, since the arguments of the two max functions are pairwise equal by induction hypothesis;

- cases $E = \varphi(\overline{t})$ and $E = S(\overline{a})$ are proved similarly to the previous case.

□

Corollary 7.11 *The function* F *of definition 7.12 is excellent*

Proof: Let $\Gamma = X_1 : \sigma_1,\ldots,X_n : \sigma_n$. Then

$$y = \mathsf{F}_\Gamma(E) = \max\{\mathsf{F}'_{X_1}(\sigma_2),\ldots,\mathsf{F}'_{X_1,\ldots,X_{n-1}}(\sigma_n),\mathsf{F}'_{X_1,\ldots,X_n}(E)\}$$

We prove separately the three properties:

(DHE) by **(DHE')** we prove

$$y = \max\{\mathsf{F}'_{\pi\cdot X_1}(\pi\cdot\sigma_2),\ldots,\mathsf{F}'_{\pi\cdot X_1,\ldots,\pi\cdot X_{n-1}}(\pi\cdot\sigma_n),\mathsf{F}'_{\pi\cdot X_1,\ldots,\pi\cdot X_n}(\pi\cdot E)\}$$

thus, by the definition of F, we get

$$\mathsf{F}_\Gamma(E) = y = \mathsf{F}_{\pi\cdot\Gamma}(\pi\cdot E)$$

as expected;

(DHF) we easily prove that

$$\begin{aligned} y &\geq \mathsf{F}'_{X_1}(\sigma_2) \\ &\vdots \\ y &\geq \mathsf{F}'_{X_1,\ldots,X_{n-1}}(\sigma_n) \\ y &\geq \mathsf{F}'_{X_1,\ldots,X_n}(E) \end{aligned}$$

therefore, by **(DHF')**, we get:

$$\begin{aligned} y &\notin \mathsf{E}'_{X_1}(\sigma_2) \\ &\vdots \\ y &\notin \mathsf{E}'_{X_1,\ldots,X_{n-1}}(\sigma_n) \\ y &\notin \mathsf{E}'_{X_1,\ldots,X_n}(E) \end{aligned}$$

since by definition we have

$$\mathsf{E}_\Gamma(E) = \mathsf{E}'_{X_1}(\sigma_2) \cup \ldots \cup \mathsf{E}'_{X_1,\ldots,X_{n-1}}(\sigma_n) \cup \mathsf{E}'_{X_1,\ldots,X_n}(E)$$

we get

$$y \notin \mathsf{E}_\Gamma(E)$$

as expected;

(DHP) By hypothesis, the domain of Γ (i.e. $X_1, \ldots, X_n$) is apart from $\overline{Y}, \bar{t}$. This also means that every subset of $X_1, \ldots, X_n$ also enjoys the same property: we can therefore use property **(DHP')** and prove that

$$y = \max\{\mathsf{F}'_{X_1}(\sigma_2\{\bar{t}/\overline{Y}\}), \ldots, \mathsf{F}'_{X_1,\ldots,X_{n-1}}(\sigma_n\{\bar{t}/\overline{Y}\}), \mathsf{F}'_{X_1,\ldots,X_n}(E\{\bar{t}/\overline{Y}\})\}$$

that by the definition of F is equivalent to

$$y = \mathsf{F}_{\Gamma\{\bar{t}/\overline{Y}\}}(E\{\bar{t}/\overline{Y}\})$$

as we wished to prove. □

Knowing that F is excellent, we can prove all the expected adequacy properties for our encoding of DMBEL. We state the most important ones.

Lemma 7.12 `build_atm` *is equivariant, i.e. for all finite permutations* π *the following equations hold:*

$$\pi \cdot ((\Gamma)\, t) = (\pi \cdot \Gamma)\, \pi \cdot t$$
$$\pi \cdot ((\Gamma)\, \tau) = (\pi \cdot \Gamma)\, \pi \cdot \tau$$

Sketch of Proof: After unfolding the definitions of `build_atm` and `build_atp`, use property **(DHE)** and Corollary 7.4. □

Lemma 7.13 *If* Γ *is a regular context, then:*

1. *for all finite permutations* π, $\pi \cdot \Gamma$ *is regular;*
2. *if* $\overline{X}$ *is a list of global variables and* $\overline{u}$ *is a list of terms such that* $|\overline{X}| = |\overline{u}|$ *and* $dom(\Gamma) \# \overline{u}$, $\Gamma\{^{\overline{u}}/_{\overline{X}}\}$ *is a regular context.*

Sketch of Proof: Both properties are easy by induction on Γ, only needing basic properties of permutations and substitutions. □

Lemma 7.14 *For all finite permutations* π *and symbolic expressions* E, *if* $E : \mathbb{L}$, *then* $\pi \cdot E : \mathbb{L}$.

Sketch of Proof: Easy induction on the derivation of $E : \mathbb{L}$. In the case of abstractions, use Lemma 7.12 and part 1 of Lemma 7.13. □

Lemma 7.15 *If* $(\Gamma)\, t : \mathbb{L}$, *for all lists of global variables* $\overline{Y}$ *such that* $\overline{Y} \# (\Gamma)\, t$, *where the length of* $\overline{Y}$ *and the length of* Γ *are equal, we have*

$$(\Gamma)\, t = ((dom(\Gamma)\ \overline{Y}) \cdot \Gamma)\, ((dom(\Gamma)\ \overline{Y}) \cdot t)$$

Similarly, if $(\Gamma)\, \tau : \mathbb{L}$, *then for all lists of global variables* $\overline{Y}$ *such that* $\overline{Y} \# (\Gamma)\, \tau$, *where the length of* $\overline{Y}$ *and the length of* Γ *are equal, then*

$$(\Gamma)\, \tau = ((dom(\Gamma)\ \overline{Y}) \cdot \Gamma)\, ((dom(\Gamma)\ \overline{Y}) \cdot \tau)$$

Proof: By Lemma 7.12, choosing $\pi = (dom(\Gamma)\ \overline{Y})$, we know that:

$$((dom(\Gamma)\ \overline{Y}) \cdot \Gamma)\,((dom(\Gamma)\ \overline{Y}) \cdot t) = (dom(\Gamma)\ \overline{Y}) \cdot (\Gamma)\,t$$

but since $(\Gamma)\,t : \mathbb{L}$, we also know that Γ is regular, therefore $dom(\Gamma) \# (\Gamma)\,t$; furthermore, $\overline{Y} \# (\Gamma)\,t$ by hypothesis. Since permutations of fresh names are ineffective, we also have

$$(dom(\Gamma)\ \overline{Y}) \cdot (\Gamma)\,t = (\Gamma)\,t$$

If we assume that $\overline{Y}$ is a distinct list disjoint from the domain of Γ, we easily prove that the domain of the permuted term is equal to $\overline{Y}$.

The proof in the case of abstraction types follows the same technique. □

The above property is useful to get the equivalent of the informal notion of α-conversion, since it allows us to replace the global names used to build an abstraction with arbitrary sufficiently fresh names.

Lemma 7.16 *If* $dom(\Gamma) \# \overline{X}, \mathsf{GV}(\overline{u})$ *then*

$$((\Gamma)\,t)\{\overline{u}/\overline{X}\} = (\Gamma\{\overline{u}/\overline{X}\})\,(t\{\overline{u}/\overline{X}\})$$

Proof: After unfolding the definition of `build_atm`, we must prove

$$(\texttt{[}x; [\Gamma \mapsto x]\texttt{]}\ (t[dom(\Gamma) \mapsto x]))\{\overline{u}/\overline{X}\} = \texttt{[}y; [\Gamma\{\overline{u}/\overline{X}\} \mapsto y]\texttt{]}\ (t[dom(\Gamma\{\overline{u}/\overline{X}\}) \mapsto y])$$

where $x = \mathsf{F}_\Gamma(t)$ and $y = \mathsf{F}_{\Gamma\{\overline{u}/\overline{X}\}}(t\{\overline{u}/\overline{X}\})$. Clearly **(DHP)** implies $x = y$; furthermore, $dom(\Gamma) = dom(\Gamma\{\overline{u}/\overline{X}\})$ (easily provable by induction on Γ). Rewriting accordingly, after a computation step, we must prove

1. $[\Gamma \mapsto x]\{\overline{u}/\overline{X}\} = [\Gamma\{\overline{u}/\overline{X}\} \mapsto x]$
2. $t[dom(\Gamma) \mapsto x]\{\overline{u}/\overline{X}\} = (t\{\overline{u}/\overline{X}\})[dom(\Gamma) \mapsto x]$

The second point is proved using Lemma 7.2. The first point is proved by induction on Γ, also employing Lemma 7.2 when needed. □

Lemma 7.17 *For all symbolic expressions E, for all lists of parameters $\overline{X}$ and lists of terms $\overline{u}$ having the same length, if $E : \mathbb{L}$ and for all $t \in \overline{u}$, $t : \mathbb{L}$, then*

$$E\{\overline{t}/\overline{x}\} : \mathbb{L}$$

Proof: The proof is by induction on the derivation of $E : \mathbb{L}$. The interesting case is $E = (\Gamma)\, u$, where we know that Γ is regular, $\sigma : \mathbb{L}$ for all $\sigma \in \mathit{cod}(\Gamma)$ and $t : \mathbb{L}$. By lemma 7.15, we rewrite the goal as

$$((\mathit{dom}(\Gamma)\ \overline{Y}) \cdot \Gamma)\,((\mathit{dom}(\Gamma)\ \overline{Y}) \cdot u)\{\overline{t}/\overline{x}\} = ((\Delta)\, v)\{\overline{t}/\overline{x}\}$$

where $\overline{Y} \# \mathit{dom}(\Gamma), \overline{X}, (\Gamma)\, u, \overline{t}$ has length equal to that of Γ. We prove easily that $\mathit{dom}(\Delta) = \overline{Y}$. Under these assumptions, Lemma 7.16 implies

$$((\Delta)\, v)\{\overline{t}/\overline{x}\} = (\Delta\{\overline{t}/\overline{x}\})\,(v\{\overline{t}/\overline{x}\})$$

Then we must prove

$$(\Delta\{\overline{t}/\overline{x}\})\,(v\{\overline{t}/\overline{x}\}) : \mathbb{L}$$

Lemma 7.13 (both parts) implies that $\Delta\{\overline{t}/\overline{x}\}$ is a regular context. Furthermore, we can show that $v\{\overline{t}/\overline{x}\} : \mathbb{L}$ and that for all $\sigma \in \mathit{cod}(\Delta\{\overline{t}/\overline{x}\})$, $\sigma : \mathbb{L}$ (this is obtained recalling the definition of Δ and v, combining the induction hypotheses and exploiting the equivariance of the $\mathbb{L}$ judgment (Lemma 7.14)). Thus we can prove the thesis using rule LATM-MK-ATM. □

Lemma 7.18 *1. If $[x;\overline{\sigma}]\ t : \mathbb{L}$, for all $u' \in \overline{u}$ $u' : \mathbb{L}$, and $|\overline{\sigma}| = |\overline{u}|$, then*

$$[x;\overline{\sigma}]\ t \blacktriangledown \overline{u} : \mathbb{L}$$

2. If $[x;\overline{\sigma}]\ \tau : \mathbb{L}$, for all $u' \in \overline{u}$ $u' : \mathbb{L}$, and $|\overline{\sigma}| = |\overline{u}|$, then

$$[x;\overline{\sigma}]\ \tau \blacktriangledown \overline{u} : \mathbb{L}$$

Proof: We prove part 1: by inversion on $[x;\overline{\sigma}]\ t : \mathbb{L}$, there exist Γ, t' such that

$$[x;\overline{\sigma}]\ t = (\Gamma)\, t'$$
$$\Gamma \text{ is regular}$$
$$\forall \tau \in \mathit{cod}(\Gamma) : \tau : \mathbb{L}$$
$$t' : \mathbb{L}$$

By Lemma 7.15 we can find π, Δ, t'' such that

$$\Delta = \pi \cdot \Gamma$$
$$t'' = \pi \cdot t'$$
$$dom(\Delta) \# t, \overline{\sigma}, dom(\Gamma)$$
$$[x; \overline{\sigma}]\ t = (\Delta)\, t''$$

Using Lemmata 7.13 and 7.14 we prove that

$$\Delta \text{ is regular}$$
$$\forall \tau \in cod(\Delta) : \tau : \mathbb{L}$$
$$t'' : \mathbb{L}$$

Since $[x; \overline{\sigma}]\ t = (\Delta)\, t''$, unfolding the definition of `build_atm` we can prove

$$x = \mathsf{F}_\Delta(t'')$$
$$\overline{\sigma} = [\Delta \mapsto x]$$
$$t = t''[dom(\Delta) \mapsto x]$$

After substitution and a computation step, the thesis becomes

$$t''[dom(\Delta) \mapsto \mathsf{F}_\Delta(t'')]\{^{\overline{u}}/_{\mathsf{F}_\Delta(t'')}\} : \mathbb{L}$$

Since $\mathsf{LV}(t'') = \emptyset$ (Lemma 7.6) and $\mathsf{F}_\Delta(t'') \notin \mathsf{E}_\Delta(t'')$ by **(DHF)**, by Lemma 7.8 the thesis becomes

$$t''\{^{\overline{u}}/_{dom(\Delta)}\} : \mathbb{L}$$

that follows from Lemma 7.17.

Proof of part 2 is similar. □

7.4 Hereditary substitution

It is worth noting that the definition of excellent height does not mention hereditary substitution. Since the **(DHP)** property of F proved essential in showing that regular substitution preserves the $\mathbb{L}$ predicate (Lemma 7.17), we ask ourselves if an

excellent height is sufficient for the same property to hold also in the case of hereditary substitution or we should assume something more. The answer is negative. Worse than that, we can show that all excellent DMBEL heights are **not** invariant under hereditary substitution. This issue is better described using a small example.

Let H be an excellent DMBEL height, and define $x \triangleq \mathsf{H}_{X:\sigma}(X)$, for some type σ. Then, construct the term

$$t \triangleq \varphi(X, f(\,[x;\sigma]\; X))$$

where f is a functional constant and φ an abstraction variable (clearly we can assume $f : (X : \sigma)\,\sigma$ and $\varphi : (X_1 : \sigma, X_2 : \sigma)\,\sigma$. Now, let fst be the DMBEL abstraction term encoding the first projection of a pair on σ:

$$fst \triangleq (X_1 : \sigma, X_2 : \sigma)\, X_1$$

clearly, we must define hereditary substitution so that the equation $t\{^{fst}/_{\varphi}\} = X$ holds, therefore

$$\mathsf{H}_{X:\sigma}(t\{^{fst}/_{\varphi}\}) = x$$

If property **(DHP)** could be extended to hereditary substitution, we should also have

$$\mathsf{H}_{X:\sigma}(t) = x$$

however, we have constructed t so that

$$\mathsf{E}_{X:\sigma}(t) = \{x\}$$

implying $x = \mathsf{H}_{X:\sigma}(t) \in \mathsf{E}_{X:\sigma}(t)$. This is impossible, since we have supposed H is excellent.

Since the above counterexample uses ill formed terms, we could wonder whether restricting us to good heights (heights that are well behaved only with respect to well formed terms) can solve the problem. Again, the answer is negative: this time we will also use the second projection of a pair:

$$snd \triangleq (X_1 : \sigma, X_2 : \sigma)\, X_2$$

Let t, u be arbitrary well formed terms, and φ an abstraction variable: if good heights were stable by hereditary substitution, for all X we would have

$$\mathsf{H}_X(\varphi(t,u)) = \mathsf{H}_X((\varphi(t,u))\{^{fst}/_{\varphi}\} = \mathsf{H}_X(t)$$
$$\mathsf{H}_X(\varphi(t,u)) = \mathsf{H}_X((\varphi(t,u))\{^{snd}/_{\varphi}\} = \mathsf{H}_X(u)$$

Notice that we have not made any assumption about t or u except for their well-formedness, meaning that heights of all global variables in all well formed terms must be equal to a fixed local variable x. Clearly we cannot express binding using a single local variable. In fact this is contradictory with the **(HF)** property: to show this, just choose two parameters $Y \neq Z$; then $\mathsf{abs}_{Y:\sigma}(Z) = [x;\sigma]\ Z$ for all well formed types σ; $\mathsf{H}_Z([x;\sigma]\ Z$ must also be equal to the same x, but unfortunately $x \in \mathsf{E}_Z([x;\sigma])$, yielding the contradiction.

What is happening here? Roughly speaking, **(DHP)** means that excellent heights have a very nice property allowing us to define first order substitution directly on symbolic expressions, in the most obvious way, preserving well-formedness of expressions with no added effort. Hereditary substitution, however, puts in much more freedom in the shape a substituted term can assume. In short, there is no hope of finding a height with similar properties lifted to hereditary substitution.

Luckily, property **(DHP)** has a somewhat different (less critical) status compared to **(DHE)** and **(DHF)**: the latter two are necessary for avoiding variable capture and ensuring that the representation is canonical; the former one is sufficient to prove that well-formedness is stable with respect to substitution, but we can still hope it is not necessary, if substitution is defined in a clever way. In fact, if we *recompute* the correct height any time hereditary substitution crosses an abstraction, then well-formedness of terms will be preserved, essentially, by definition.

The idea is to define substitution only in terms of well-behaved operations. Recalling the informal definition of hereditary substitution, the difficult step is to make it commute with the informal notion of abstraction, as in

$$((\Gamma)\,t)\{^{\overline{a}}/_{\overline{\varphi}}\} = (\Gamma\{^{\overline{a}}/_{\overline{\varphi}}\})\,(t\{^{\overline{a}}/_{\overline{\varphi}}\}) \quad \text{if } dom(\Gamma)\,\#\,\overline{a}$$

In our case, $(\Gamma)\, t$ represents a defined operation, not a concrete constructor: as such, pattern matching is not allowed on it, but only against concrete abstractions in the form $[x;\overline{\sigma}]\ u$. What we can do in this case is to open $\overline{\sigma}$ and u with respect to an arbitrary vector of distinct, sufficiently fresh global variables $\overline{X}$:

$$\Delta \triangleq \overline{\sigma}[x \mapsto \overline{X}]$$
$$u' \triangleq u[x \mapsto \overline{X}]$$

It is then possible to prove that Δ is regular and its codomain is well formed, and that u' is well-formed. Furthermore, $[x;\sigma]\ u = (\Delta)\, u'$. Assuming that $\overline{X}$ is also fresh with respect to $\mathsf{GV}(\overline{a})$, we can then define

$$([x;\overline{\sigma}]\ u)\{{}^{\overline{a}}/_{\overline{\varphi}}\} = (\Delta\{{}^{\overline{a}}/_{\overline{\varphi}}\})\,(u'\{{}^{\overline{a}}/_{\overline{\varphi}}\})$$

Albeit somewhat unnatural, this strategy is fit to give an algoritmic definition of hereditary substitution in the Sato representation. An annoying problem with this definition, though, is that we want to perform recursion on arguments (Δ and u') that are not *structurally* smaller than $[x;\overline{\sigma}]\ u$. Being unable to automatically prove that the algorithm is total, Matita will not accept this definition. A common way to solve this kind of problem is to add to the substitution function a parameter (in the form of a natural number), encoding an upper bound on the number of recursive calls needed by the substitution operation to complete.

If we used this upper bound idiom, Matita would accept the algorithmic definition of hereditary substitution. Both the upper bound and the use of variable opening, however, make the complexity of reasoning on hereditary substitution somewhat worse.

A different approach is to define hereditary substitution as an inductive predicate: we lose the possibility of computing the result of a substitution directly, but we do not have to deal with the restrictions imposed on recursive functions. Since the definition of hereditary substitution as an inductive predicate can be much closer to its informal counterpart than an algorithmic definition, reasoning is considerably simplified, and so are adequacy concerns. Should an algorithmic definition

be needed, we could still give it later and prove its equivalence with the inductive predicate.

Figure 7.1 shows the definition of hereditary substitution we give in the formalization. The two rules S2X-Red and S2X-Tm-Appvar concern abstraction variable applications, depending on whether the applied variable ψ is in the domain of the substitution or not: in the former case, an auxiliary definition lookup is used to express which abstraction term a belonging to the list $\overline{b}$ must be substituted; an arity check is also needed to ensure that the substitution only takes place for suitable arguments. There are also judgments for substituting in every item of a list (including the case of the codomain of a context): S2X-Nil is used as the base case for all such judgments.

The key rule of this definition, however, is S2X-Mk-Atm, stated in a form that closely reminds of its informal definition (including the side condition requiring a sufficiently fresh domain for the abstracted context). The use of a build operation on properly substituted contexts and expressions is what actually prompts the definition to recompute the height of the abstraction under consideration.

We can prove that hereditary substitution preserves the regularity of contexts: however the proof is mutual with other properties about the global variables of the expressions involved in the substitution.

Lemma 7.19 *For all lists of abstraction terms $\overline{b}$ and lists of abstraction variables $\overline{\varphi}$, the following properties hold:*

1. *if $t\{\overline{b}/\overline{\varphi}\} \downarrow t'$ then $\mathsf{GV}(t') \subseteq \mathsf{GV}(t) \cup \mathsf{GV}(\overline{b})$;*
2. *if $\sigma\{\overline{b}/\overline{\varphi}\} \downarrow \sigma'$ then $\mathsf{GV}(\sigma') \subseteq \mathsf{GV}(\sigma) \cup \mathsf{GV}(\overline{b})$;*
3. *if $a\{\overline{b}/\overline{\varphi}\} \downarrow a'$ then $\mathsf{GV}(a') \subseteq \mathsf{GV}(a) \cup \mathsf{GV}(\overline{b})$;*
4. *if $\overline{t}\{\overline{b}/\overline{\varphi}\} \downarrow \overline{t'}$ then $\mathsf{GV}(\overline{t'}) \subseteq \mathsf{GV}(\overline{t}) \cup \mathsf{GV}(\overline{b})$;*
5. *if $\overline{a}\{\overline{b}/\overline{\varphi}\} \downarrow \overline{a'}$ then $\mathsf{GV}(\overline{a'}) \subseteq \mathsf{GV}(\overline{a}) \cup \mathsf{GV}(\overline{b})$;*

$$\text{lookup}(\overline{\varphi}, \overline{b}, \psi, a) \iff \left(\begin{array}{l} \exists \overline{\varphi'}, \overline{\varphi''}, \overline{b'}, \overline{b''}. \\ \quad \overline{\varphi} = \varphi', \psi, \varphi'' \wedge \overline{b} = \overline{b'}, a, \overline{b''} \wedge \\ \quad |\overline{\varphi'}| = |\overline{b'}| \wedge |\overline{\varphi''}| = |\overline{b''}| \wedge \\ \quad \psi \notin \overline{\varphi''} \end{array} \right)$$

$$\frac{}{X\{\overline{b}/\overline{\varphi}\} \downarrow X}\ (\text{S2X-Tm-Var}) \qquad \frac{\overline{a}\{\overline{b}/\overline{\varphi}\} \downarrow \overline{a'}}{f(\overline{a})\{\overline{b}/\overline{\varphi}\} \downarrow f(\overline{a'})}\ (\text{S2X-Tm-Appcon})$$

$$\frac{\overline{t}\{\overline{b}/\overline{\varphi}\} \downarrow \overline{t'} \qquad \psi \notin \overline{\varphi}}{\psi(\overline{t})\{\overline{b}/\overline{\varphi}\} \downarrow \psi(\overline{t'})}\ (\text{S2X-Tm-Appvar}) \qquad \frac{\begin{array}{c}\text{lookup}(\overline{\varphi}, \overline{b}, \psi, a) \\ \text{arity}(a) = |\overline{t}| \\ \overline{t}\{\overline{b}/\overline{\varphi}\} \downarrow \overline{t'}\end{array}}{\psi(\overline{t})\{\overline{b}/\overline{\varphi}\} \downarrow a \blacktriangledown \overline{t'}}\ (\text{S2X-Red})$$

$$\frac{\overline{a}\{\overline{b}/\overline{\varphi}\} \downarrow \overline{a'}}{S(\overline{a})\{\overline{b}/\overline{\varphi}\} \downarrow S(\overline{a'})}\ (\text{S2X-Mk-Tp})$$

$$\frac{\begin{array}{cc}\Gamma\{\overline{b}/\overline{\varphi}\} \downarrow \Gamma' & t\{\overline{b}/\overline{\varphi}\} \downarrow t' \\ dom(\Gamma) \# \overline{b} & \Gamma \text{ is regular}\end{array}}{((\Gamma)\, t)\{\overline{b}/\overline{\varphi}\} \downarrow (\Gamma')\, t'}\ (\text{S2X-Mk-Atm})$$

$$\frac{}{[]\{\overline{b}/\overline{\varphi}\} \downarrow []}\ (\text{S2X-Nil})$$

$$\frac{\overline{t}\{\overline{b}/\overline{\varphi}\} \downarrow \overline{t'} \qquad t_1\{\overline{b}/\overline{\varphi}\} \downarrow t_1'}{\overline{t}, t_1\{\overline{b}/\overline{\varphi}\} \downarrow \overline{t'}, t_1'}\ (\text{S2X-Tm-Cons})$$

$$\frac{\overline{a}\{\overline{b}/\overline{\varphi}\} \downarrow \overline{a'} \qquad a_1\{\overline{b}/\overline{\varphi}\} \downarrow a_1'}{\overline{a}, a_1\{\overline{b}/\overline{\varphi}\} \downarrow \overline{a'}, a_1'}\ (\text{S2X-Atm-Cons})$$

$$\frac{\Gamma\{\overline{b}/\overline{\varphi}\} \downarrow \Gamma' \qquad \sigma\{\overline{b}/\overline{\varphi}\} \downarrow \sigma'}{\Gamma, X : \sigma\{\overline{b}/\overline{\varphi}\} \downarrow \Gamma', X : \sigma'}\ (\text{S2X-Ctx-Cons})$$

Figure 7.1: DMBEL hereditary substitution (as formalized)

6. *if* $\Gamma\{\overline{b}/\overline{\varphi}\} \downarrow \Gamma'$ *then* $\mathsf{GV}(cod(\Gamma')) \subseteq \mathsf{GV}(cod(\Gamma)) \cup \mathsf{GV}(\overline{b})$ *and* $(dom(\Gamma) \# \mathsf{GV}(\overline{b}) \wedge \Gamma$ is regular $\Longrightarrow \Gamma'$ is regular).

Proof: The proof is by mutual induction on the six different forms of substitution judgment. It is not surprising that preservation of the regularity of contexts depends on other properties of global variables; however the opposite is also true.

We only consider the cases for abstraction terms and for contexts, since the other ones are easy by induction hypothesis.

In the case of abstraction terms, we must prove, under the hypotheses of judgment S2X-Mk-Atm

$$\begin{array}{ll} H_1 & \Gamma\{\overline{b}/\overline{\varphi}\} \downarrow \Gamma' \\ H_2 & t\{\overline{b}/\overline{\varphi}\} \downarrow t' \\ H_3 & \Gamma \text{ is regular} \\ H_4 & dom(\Gamma) \# \mathsf{GV}(\overline{b}) \end{array}$$

that the following inclusion holds:

$$\mathsf{GV}((\Gamma')\, t) \subseteq \mathsf{GV}((\Gamma)\, t) \cup \mathsf{GV}(\overline{b})$$

We know by induction hypothesis that

$$\begin{array}{ll} IH_1 & \mathsf{GV}(t') \subseteq \mathsf{GV}(t) \cup \mathsf{GV}(\overline{b}) \\ IH_2 & \mathsf{GV}(cod(\Gamma')) \subseteq \mathsf{GV}(cod(\Gamma)) \cup \mathsf{GV}(\overline{b}) \\ IH_3 & dom(\Gamma) \# \overline{b} \wedge \Gamma \text{ is regular } \Longrightarrow \Gamma' \text{ is regular} \end{array}$$

To prove the inclusion, consider any $X \in \mathsf{GV}((\Gamma')\, t')$: then either $X \in \mathsf{GV}([\Gamma' \mapsto x])$ or $X \in \mathsf{GV}(t'[dom(\Gamma') \mapsto x])$, where x is a proper local variable.

- in the first case, we can show that $X \in \mathsf{GV}(cod(\Gamma'))$ and $X \# dom(\Gamma')$ (this only holds since Γ' is regular, therefore we are using the extended induction hypothesis IH_3); we can also prove that $dom(\Gamma) = dom(\Gamma')$; these properties, together with IH_2, are sufficient to show that $X \in \mathsf{GV}((\Gamma)\, t) \cup \mathsf{GV}(\overline{b})$
- in the second case, we prove $X \in \mathsf{GV}(t)$ and $X \# dom(\Gamma')$; since, again, $dom(\Gamma) = dom(\Gamma')$, the thesis follows quite easily from IH_1.

For the case of contexts, only consider rule S2X-Ctx-Cons (S2X-Nil is trivial). Under the hypotheses

$$H_1 \quad \Gamma\{\overline{b}/\overline{\varphi}\} \downarrow \Gamma'$$
$$H_2 \quad \sigma\{\overline{b}/\overline{\varphi}\} \downarrow \sigma'$$

and induction hypotheses

$$IH_1 \quad \mathsf{GV}(\sigma') \subseteq \mathsf{GV}(\sigma) \cup \mathsf{GV}(\overline{b})$$
$$IH_2 \quad \mathsf{GV}(\mathit{cod}(\Gamma')) \subseteq \mathsf{GV}(\mathit{cod}(\Gamma)) \cup \mathsf{GV}(\overline{b})$$
$$IH_3 \quad \mathit{dom}(\Gamma) \# \overline{b} \wedge \Gamma \text{ is regular } \Longrightarrow \Gamma' \text{ is regular}$$

we must prove the following two propositions:

$$\mathsf{GV}(\mathit{cod}(\Gamma', Y : \sigma') \subseteq \mathsf{GV}(\mathit{cod}(\Gamma, Y : \sigma)) \cup \mathsf{GV}(\overline{b})$$
$$\Gamma, Y : \sigma \text{ is regular } \wedge \mathit{dom}(\Gamma, Y : \sigma) \# \overline{b} \Longrightarrow \Gamma', Y : \sigma' \text{ is regular}$$

- In the first case, just consider any X such that $X \in \mathsf{GV}(\mathit{cod}(\Gamma', Y : \sigma'))$: then either $X \in \mathsf{GV}(\mathit{cod}(\Gamma'))$ or $X \in \mathsf{GV}(\sigma')$; the thesis follows using IH_1 and IH_2.

- In the second case, we can assume

  $$H_3 : \Gamma, Y : \sigma \text{ is regular}$$
  $$H_4 : \mathit{dom}(\Gamma, Y : \sigma) \# \mathsf{GV}\overline{b}$$

  then Γ is also regular (since it is a subcontext of a regular context) and, by IH_3, we can prove that Γ' is also regular. To prove thesis, we still need to show that

  $$\Gamma', Y : \sigma' \text{ is regular}$$

  this holds if and only if $Y \notin \mathsf{GV}(\sigma') \cup \mathsf{GV}(\mathit{cod}(\Gamma'))$. If this were not true, we could use hypotheses H_4,IH_1 and IH_2 to show that $\Gamma, Y : \sigma$ is not regular, reaching a contradiction.

□

The last lemma we prove about hereditary substitution states that it preserves well-formedness. Actually, since the input of hereditary substitution is always required to be well formed (because of the way we defined the judgment), to prove this property we only need to require that the codomain of the substitution is well-formed.

Lemma 7.20 *For all lists of well formed abstraction terms $\overline{b}$ and lists of abstraction variables $\overline{\varphi}$, the following properties hold:*

1. $t\{\overline{b}/\overline{\varphi}\} \downarrow t' \Longrightarrow t' : \mathbb{L}$
2. $\sigma\{\overline{b}/\overline{\varphi}\} \downarrow \sigma' \Longrightarrow \sigma' : \mathbb{L}$
3. $a\{\overline{b}/\overline{\varphi}\} \downarrow a' \Longrightarrow a' : \mathbb{L}$
4. $\overline{t}\{\overline{b}/\overline{\varphi}\} \downarrow \overline{t'} \Longrightarrow \forall t' \in \overline{t'} : t' : \mathbb{L}$
5. $\overline{a}\{\overline{b}/\overline{\varphi}\} \downarrow \overline{a'} \Longrightarrow \forall a' \in \overline{a'} : a' : \mathbb{L}$
6. $\Gamma\{\overline{b}/\overline{\varphi}\} \downarrow \Gamma' \Longrightarrow \forall \sigma \in cod(\Gamma') : \sigma : \mathbb{L}$

Sketch of Proof: The proof is by mutual induction on the six judgment forms of hereditary substitution. All cases are straightforward: notice that in case S2X-Red we must use Lemma 7.18; in case S2X-Mk-Atm, instead, we use Lemma 7.19. □

7.5 Formalization of the type system

Table 7.5 presents the type system of DMBEL as we formalized it. We now discuss the most important differences with respect to the informal presentation we gave earlier.

As expected, the context well-formedness rules enforce the global variables they declare to be distinct: in fact we will see that a well-formed context, in the formalization, is always regular. The other major change, compared to the informal rules, concerns hereditary substitution. Having defined hereditary substitution as a predicate rather than a computable function slightly modifies rules TI-Appcon and ATC-Cons: for example, in TI-Appcon, instead of saying that the type of $f(\overline{a})$ is $\sigma\{\overline{a}/_{dom(\Psi)}\}$, we state that its type must be some type σ' for which we can prove $\sigma\{\overline{a}/_{dom(\Psi)}\} \downarrow \sigma'$. The meaning of the rule is still very close to the usual intuition and indeed, in our opinion, its adequacy is much clearer than what it would be if we chose to define hereditary substitution as a computable function, using a mix

Signatures:

$$\frac{}{\vdash \emptyset}\ (\text{SO-Empty}) \qquad \frac{\vdash_\Sigma \Phi \quad S \notin dom(\Sigma)}{\vdash \Sigma, S(\Phi)}\ (\text{SO-Tp}) \qquad \frac{\Phi; \emptyset \vdash_\Sigma \sigma \quad f \notin dom(\Sigma)}{\vdash \Sigma, f(\Phi) : \sigma}\ (\text{SO-Tm})$$

Abstraction contexts:

$$\frac{\vdash \Sigma}{\vdash_\Sigma \emptyset}\ (\text{ACO-Empty}) \qquad \frac{\Phi; \emptyset \vdash_\Sigma \alpha \quad \varphi \notin dom(\Phi)}{\vdash_\Sigma \Phi, \varphi : \alpha}\ (\text{ACO-Cons})$$

Contexts:

$$\frac{\vdash_\Sigma \Phi}{\Phi \vdash_\Sigma \emptyset}\ (\text{CO-Empty}) \qquad \frac{\Phi; \Gamma \vdash_\Sigma \sigma \quad x \notin dom(\Gamma)}{\Phi \vdash_\Sigma \Gamma, x : \sigma}\ (\text{CO-Cons})$$

Type and abstraction type formation:

$$\frac{S(\Psi) \in \Sigma \quad \Phi; \Gamma \vdash_\Sigma (\overline{a})^{rev} \Longleftarrow \Psi}{\Phi; \Gamma \vdash_\Sigma S(\overline{a})}\ (\text{TO-Intro}) \qquad \frac{\Phi; \Gamma, \Delta \vdash_\Sigma \sigma}{\Phi; \Gamma \vdash_\Sigma (\Delta)\,\sigma}\ (\text{ATO-Intro})$$

Term and abstraction term typing:

$$\frac{X : \sigma \in \Gamma \quad \Phi \vdash_\Sigma \Gamma}{\Phi; \Gamma \vdash_\Sigma X \Longrightarrow \sigma}\ (\text{TI-Var}) \qquad \frac{f(\Psi) : \sigma \in \Sigma \quad \Phi; \Gamma \vdash_\Sigma (\overline{a})^{rev} \Longleftarrow \Psi \quad \sigma\{\overline{a}/_{dom(\Psi)}\} \downarrow \sigma'}{\Phi; \Gamma \vdash_\Sigma f(\overline{a}) \Longrightarrow \sigma'}\ (\text{TI-Appcon})$$

$$\frac{\varphi : [x; \overline{\sigma}]\ \tau \in \Phi \quad \overline{X} \# dom(\Gamma) \qquad |\overline{X}| = |\overline{\sigma}| \quad \Phi; \Gamma \vdash_\Sigma (\overline{t})^{rev} \Longleftarrow \overline{\sigma}[x \mapsto \overline{X}]}{\Phi; \Gamma \vdash_\Sigma \varphi(\overline{t}) \Longrightarrow [x; \overline{\sigma}]\ \tau \blacktriangledown \overline{t}}\ (\text{TI-Appvar}) \qquad \frac{\Phi; \Gamma, \Delta \vdash_\Sigma t \Longrightarrow \sigma}{\Phi; \Gamma \vdash_\Sigma (\Delta)\, t \Longrightarrow (\Delta)\,\sigma}\ (\text{ATI-Intro})$$

Record typing:

$$\frac{\Phi \vdash_\Sigma \Gamma}{\Phi; \Gamma \vdash_\Sigma \emptyset \Longleftarrow \emptyset}\ (\text{TC-Empty}) \qquad \frac{\Phi; \Gamma \vdash_\Sigma t \Longrightarrow \sigma \quad \Phi; \Gamma \vdash_\Sigma \overline{u} \Longleftarrow \Delta\{t/_X\}}{\Phi; \Gamma \vdash_\Sigma \overline{u}, t \Longleftarrow \Delta, X : \sigma}\ (\text{TC-Cons})$$

Abstraction record typing:

$$\frac{\Phi \vdash_\Sigma \Gamma}{\Phi; \Gamma \vdash_\Sigma \emptyset \Longleftarrow \emptyset}\ (\text{ATC-Empty}) \qquad \frac{\Psi\{a/_\varphi\} \downarrow \Psi' \quad \Phi; \Gamma \vdash_\Sigma a \Longrightarrow \alpha \quad \Phi; \Gamma \vdash_\Sigma \overline{b} \Longleftarrow \Psi'}{\Phi; \Gamma \vdash_\Sigma \overline{b}, a \Longleftarrow \Psi, \varphi : \alpha}\ (\text{ATC-Cons})$$

Table 7.5: Typing rules of DMBEL (as formalized)

of variable opening and build operations to update the heights in the result of the substitution: our definition of hereditary substitution is close to the informal syntax and only depends on the adequacy of the build operations, as does the rest of the formalization.

Our goal is to prove that the formalized type system only involves well formed expressions in its judgments. Before that, however, we need to prove another important property of this system: well-typed expressions are closed in their typing context.

First we introduce an auxiliary definition:

Definition 7.13 *A context* Γ *is* strongly regular *if and only if it is regular and*

$$\forall \Gamma', \Gamma'', X, \sigma : \Gamma = \Gamma', X : \sigma, \Gamma'', \mathsf{GV}(\sigma) \subseteq dom(\Gamma')$$

The actual theorem is obtained by mutual induction. Since the statement is quite verbose, we introduce the following abbreviations:

- $P_{sig}(\Sigma) \triangleq \left(\begin{array}{c} (\forall S(\Phi) \in \Sigma.\mathsf{GV}(cod(\Phi)) = \emptyset) \wedge \\ (\forall (f(\Phi) : \sigma) \in \Sigma.\mathsf{GV}(cod(\Phi)) \cup \mathsf{GV}(\sigma) = \emptyset \end{array} \right)$
- $P_{actx}(\Phi) \triangleq \mathsf{GV}(cod(\Phi)) = \emptyset$
- $P_{ctx}(\Gamma) \triangleq \Gamma$ is strongly regular
- $P_{tp}(\Gamma, \sigma) \triangleq \mathsf{GV}(\sigma) \subseteq dom(\Gamma)$
- $P_{atp}(\Gamma, \alpha) \triangleq \mathsf{GV}(\alpha) \subseteq dom(\Gamma)$
- $P_{tm}(\Gamma, t) \triangleq \mathsf{GV}(t) \subseteq dom(\Gamma)$
- $P_{atm}(\Gamma, a) \triangleq \mathsf{GV}(a) \subseteq dom(\Gamma)$
- $P_{tml}(\Gamma, \overline{t}) \triangleq \mathsf{GV}(\overline{t}) \subseteq dom(\Gamma)$
- $P_{atml}(\Gamma, \overline{a}) \triangleq \mathsf{GV}(\overline{a}) \subseteq dom(\Gamma)$

Theorem 7.21 *The following properties hold:*

1. $\vdash \Sigma$ implies $P_{sig}(\Sigma)$
2. $\vdash_\Sigma \Phi$ implies $P_{sig}(\Sigma) \wedge P_{actx}(\Phi)$
3. $\Phi \vdash_\Sigma \Gamma$ implies $P_{sig}(\Sigma) \wedge P_{actx}(\Phi) \wedge P_{ctx}(\Gamma)$
4. $\Phi; \Gamma \vdash_\Sigma \sigma$ implies $P_{sig}(\Sigma) \wedge P_{actx}(\Phi) \wedge P_{ctx}(\Gamma) \wedge P_{tp}(\Gamma, \sigma)$
5. $\Phi; \Gamma \vdash_\Sigma \alpha$ implies $P_{sig}(\Sigma) \wedge P_{actx}(\Phi) \wedge P_{ctx}(\Gamma) \wedge P_{atp}(\Gamma, \alpha)$
6. $\Phi; \Gamma \vdash_\Sigma t \Longrightarrow \sigma$ implies $P_{sig}(\Sigma) \wedge P_{actx}(\Phi) \wedge P_{ctx}(\Gamma) \wedge P_{tp}(\Gamma, \sigma) \wedge P_{tm}(\Gamma, t)$
7. $\Phi; \Gamma \vdash_\Sigma a \Longrightarrow \alpha$ implies $P_{sig}(\Sigma) \wedge P_{actx}(\Phi) \wedge P_{ctx}(\Gamma) \wedge P_{atp}(\Gamma, \alpha) \wedge P_{atm}(\Gamma, a)$
8. $\Phi; \Gamma \vdash_\Sigma \overline{t} \Longleftarrow \Delta$ implies $P_{sig}(\Sigma) \wedge P_{actx}(\Phi) \wedge P_{ctx}(\Gamma) \wedge P_{tml}(\Gamma, \overline{t})$
9. $\Phi; \Gamma \vdash_\Sigma \overline{a} \Longleftarrow \Psi$ implies $P_{sig}(\Sigma) \wedge P_{actx}(\Phi) \wedge P_{ctx}(\Gamma) \wedge P_{atml}(\Gamma, \overline{a})$

Sketch of Proof: The full proof, by mutual induction on 9 different judgment forms, is long and tedious, but not particularly challenging: essentially, the proof uses only basic properties of substitutions, including, in case TI-Tm-Appcon, part 2 of Lemma 7.19. □

Finally, we prove that all well typed expressions are well-formed. Again for the sake of readability, we introduce some abbreviations:

- $Q_{sig}(\Sigma) \triangleq \left(\begin{array}{c} (\forall S(\Phi) \in \Sigma. \forall \alpha \in cod(\Phi). \alpha : \mathbb{L}) \wedge \\ (\forall (f(\Phi) : \sigma) \in \Sigma. \sigma : \mathbb{L} \wedge \forall \alpha \in cod(\Phi). \alpha : \mathbb{L} \end{array} \right)$
- $Q_{actx}(\Phi) \triangleq \forall \alpha \in cod(\Phi). \alpha : \mathbb{L}$
- $Q_{ctx}(\Gamma) \triangleq \forall \sigma \in cod(\Gamma). \sigma : \mathbb{L}$
- $Q_{tp}(\sigma) \triangleq \sigma : \mathbb{L}$
- $Q_{atp}(\alpha) \triangleq \alpha : \mathbb{L}$
- $Q_{tm}(t) \triangleq t : \mathbb{L}$
- $Q_{atm}(a) \triangleq a : \mathbb{L}$

- $Q_{tml}(\overline{t}) \triangleq \forall u \in \overline{t}.u : \mathbb{L}$
- $Q_{atml}(\overline{a}) \triangleq \forall b \in \overline{a}.b : \mathbb{L}$

Theorem 7.22 *The following properties hold:*

1. $\vdash \Sigma$ implies $Q_{sig}(\Sigma)$;
2. $\vdash_\Sigma \Phi$ implies $Q_{sig}(\Sigma) \wedge Q_{actx}(\Phi)$
3. $\Phi \vdash_\Sigma \Gamma$ implies $Q_{sig}(\Sigma) \wedge Q_{actx}(\Phi) \wedge Q_{ctx}(\Gamma)$
4. $\Phi;\Gamma \vdash_\Sigma \sigma$ implies $Q_{sig}(\Sigma) \wedge Q_{actx}(\Phi) \wedge Q_{ctx}(\Gamma) \wedge Q_{tp}(\sigma)$
5. $\Phi;\Gamma \vdash_\Sigma \alpha$ implies $Q_{sig}(\Sigma) \wedge Q_{actx}(\Phi) \wedge Q_{ctx}(\Gamma) \wedge Q_{atp}(\alpha)$
6. $\Phi;\Gamma \vdash_\Sigma t \Longrightarrow \sigma$ implies $Q_{sig}(\Sigma) \wedge Q_{actx}(\Phi) \wedge Q_{ctx}(\Gamma) \wedge Q_{tp}(\sigma) \wedge Q_{tm}(t)$
7. $\Phi;\Gamma \vdash_\Sigma a \Longrightarrow \alpha$ implies $Q_{sig}(\Sigma) \wedge Q_{actx}(\Phi) \wedge Q_{ctx}(\Gamma) \wedge Q_{atp}(\alpha) \wedge Q_{atm}(a)$
8. $\Phi;\Gamma \vdash_\Sigma \overline{t} \Longleftarrow \Delta$ implies $Q_{sig}(\Sigma) \wedge Q_{actx}(\Phi) \wedge Q_{ctx}(\Gamma) \wedge Q_{tml}(\overline{t})$
9. $\Phi;\Gamma \vdash_\Sigma \overline{a} \Longleftarrow \Psi$ implies $Q_{sig}(\Sigma) \wedge Q_{actx}(\Phi) \wedge Q_{ctx}(\Gamma) \wedge Q_{atml}(\overline{a})$

Proof: By mutual induction on the nine judgment forms. Most cases of the proof are straightforward: we will here discuss three cases needing a more careful treatment, exploiting previous results.

In the case of rule TI-Appcon

$$\frac{\begin{array}{c} f(\Psi) : \sigma \in \Sigma \\ \Phi;\Gamma \vdash_\Sigma \overline{a} \Longleftarrow \Psi \\ \sigma\{^{\overline{a}}/_{dom(\Psi)}\} \downarrow \sigma' \end{array}}{\Phi;\Gamma \vdash_\Sigma f(\overline{a}) \Longrightarrow \sigma'} \text{(TI-Appcon)}$$

we must prove, among other things, that $\sigma' : \mathbb{L}$ holds. We know by induction hypothesis that

$$\sigma : \mathbb{L} \text{ and } \forall b \in \overline{a}.b : \mathbb{L}$$

Then the thesis holds by part 2 of Lemma 7.20.

In the case of rule TI-Appvar

$$\frac{\begin{array}{c}\varphi : [x;\overline{\sigma}]\ \tau \in \Phi \\ \overline{X} \# dom(\Gamma) \qquad |\overline{X}| = |\overline{\sigma}| \\ \Phi;\Gamma \vdash_\Sigma \bar{t} \Longleftarrow \overline{\sigma}[x \mapsto \overline{X}]\end{array}}{\Phi;\Gamma \vdash_\Sigma \varphi(\bar{t}) \Longrightarrow [x;\overline{\sigma}]\ \tau \blacktriangledown \bar{t}}\ (\text{TI-Appvar})$$

to prove that $[x;\overline{\sigma}]\ \tau \blacktriangledown \bar{t} : \mathbb{L}$, we consider the induction hypotheses

$$[x;\overline{\sigma}]\ \tau : \mathbb{L} \text{ and } \forall u \in \bar{t}.u : \mathbb{L}$$

Then the thesis holds by Lemma 7.18.

In the case of rule ATI-Intro

$$\frac{\Phi;\Gamma,\Delta \vdash_\Sigma t \Longrightarrow \sigma}{\Phi;\Gamma \vdash_\Sigma (\Delta)\, t \Longrightarrow (\Delta)\, \sigma}\ (\text{ATI-Intro})$$

to prove that $(\Delta)\, t : \mathbb{L}$ and $(\Delta)\, \sigma : \mathbb{L}$ we can use the induction hypotheses

$$t : \mathbb{L} \quad \sigma : \mathbb{L} \quad \forall \tau \in cod(\Gamma,\Delta).\tau : \mathbb{L}$$

For the thesis to hold, we must still show that Δ is a regular context: by part 6 of Theorem 7.21, applied to the hypothesis $\Phi;\Gamma,\Delta \vdash_\Sigma t \Longrightarrow \sigma$ we know $P_{ctx}(\Gamma,\Delta)$, or equivalently, that (Γ,Δ) is a strongly regular context. Therefore Δ is regular, since it is a subcontext of a (strongly) regular context. □

7.6 Strengthened induction

Even though we did not feel the need to use a strengthened induction principle in this formalization[3], we think we should spend a couple of words on this topic. To get a strengthened induction principle for well-formed expressions, we can redefine the well-formedness judgment replacing the rule for abstraction terms with the following one

$$\frac{\text{for all } \pi.\ \left(\begin{array}{c}\pi \cdot \Gamma \text{ is regular} \\ \pi \cdot t : \mathbb{L} \\ \text{for all } \sigma \in cod(\pi \cdot \Gamma), \sigma : \mathbb{L}\end{array}\right)}{(\Gamma)\, t : \mathbb{L}}$$

[3]The proof of Lemma 7.17 might use a strengthened induction principle, but the ad hoc permutation argument we used is still reasonably simple.

Clearly the rule concerning abstraction types can be changed in a similar way. The induction principle associated to this formulation is strong because it provides induction hypotheses for all possible permutations of the names occurring in an abstraction $(\Gamma)\, t$, and Lemma 7.15 allows us to use permutations to get an α-equivalent representation of $(\Gamma)\, t$, using a context with a sufficiently fresh domain.

We can prove that both formulations of well formedness are equivalent by the usual means, since by Lemma 7.14 and Lemma 7.13 well-formedness and regularity are equivariant.

7.7 Conclusions

After formalizing two considerably involved languages by means of the Sato representation, we feel obliged to spend some words on this experience.

First of all, it is worth noting that the effort necessary to formalize the basic infrastructure concerning symbolic expressions and heights is considerable: nevertheless, the results needed to carry on the formalizations are very similar, as it is possible to see comparing the results of this chapter to those in Chapter 6. In general, we believe that our formalization can serve as a good reference for other people interested in the task of formalizing a language using the Sato representation, thus keeping the aforementioned overhead to a minimum.

In Section 7.4, we argued that some natural operations defined on well formed expressions (like hereditary substitution) break the algebraic beauty of the Sato approach. It is possible to deal with this problem, even though the solution is not completely satisfying.

Finally, we note that even in large formalizations like ours, it is not necessary to exploit the concrete definition of a specific height function to carry on the proofs: any excellent (or possibly good) height will do the job. We believe that this abstract approach is not only more generic, but also contributes to a clearer, "less polluted" result.

Chapter 8

Conclusions

In this dissertation, we have presented our work on the Matita interactive theorem prover, both as an implementor and as a user interested in the formalization of the metatheory of programming languages.

The first part of the thesis discussed the part of our implementation work that was devoted to the development of tactics for Matita, with a particular attention to those tactics that are crucial in developments involving data structures and inductively defined predicate.

Chapter 2 was mainly concerned with the presentation of Matita; in it, we discussed the new style for the implementation of tactics that we developed with the rest of the Matita team. We believe that the new style is more flexible, and suited to the implementation of smarter tactics. This style was implemented in the new version of Matita, that is currently nearing its completion.

In Chapter 3, we described an implementation of first order unification for constructor forms (excluding the discrimination of cyclic equations). The peculiarity of our implementation, that is otherwise similar to the one given in [39], is that it is designed to work indifferently with equations using Leibniz equality or John Major equality (the first case requiring deeper reasoning on dependent types). Coincidentally, we gave, to our knowledge, the first formal account of telescopic rewriting for Leibniz equality.

Chapter 4 discussed the theory and implementation of inversion principles. We focus on the reasons that make the the intuitive combination of induction and inversion in type theory somewhat difficult to formalize. Here our main contribution was the identification of a generalization of inversion principles that allows, in some cases, to mix induction and inversion more naturally.

While in our implementation of tactics we tried to incorporate some mildly novel features, the operation was a moderate success also from a software engineering perspective. The new implementation of the destruct tactic, albeit more complex than the previous one (based on the simpler technique of "predecessors" and limited to Leibniz equality), only increased the code size by 9%; in the case of inversion principles, that were also made more flexible, the code shrinked dramatically by

72%, also becoming much more readable.

The second part of the dissertation presented several formalizations related to the metatheory of programming languages. In Chapter 5, we presented three different solutions to part 1A of the Poplmark challenge. The purpose of this presentation was twofold: on one hand, it gave us the possibilikty of showing the specificities of some mainstream approaches to the representation of binding structures; on the other hand, in these formalizations, we also focused on the application of the generalized inversion principle we discussed in Chapter 4.

The last two chapters were dedicated to a novel locally named representation of binding by Sato and Pollack, which we adapted to languages involving multiple binding combined with dependent types.

Chapter 6 recalled the bases of the representation in the setting of pure λ-calculus, then discussed our formalization of Pottinger's multivariate λ-calculus.

Chapter 7 formalized, using the same representation, the DMBEL logical framework. This last work highlighted an important difficulty in the use of our representation in languages employing a notion of hereditary substitution.

As for all works involving implementation, part of our future efforts will be devoted to the continued development of Matita. Currently, we are planning a major redesign of the user interface including a different idiom for writing scripts, and possibly the integration of hyperlinks in proof scripts.

Another direction we are interested in is related more specifically to the formalization of programming languages. Currently the only theorem prover offering a specific infrastructure for the formalization of binding structures using a concrete representation is Isabelle, by means of the nominal package. However, even in this case, we feel that there is still a lot to work, especially for what concerns the automation of strengthening of induction principles.

References

[1] S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors. *Handbook of logic in computer science (vol. 2): background: computational structures*. Oxford University Press, Inc., New York, NY, USA, 1992.

[2] Robin Adams. *A modular hierarchy of logical frameworks*. PhD thesis, The University of Manchester, 2004.

[3] Robin Adams. Lambda free logical frameworks. *Annals of Pure and Applied Logic*, 2009. Submitted. Available on-line at `http://arxiv.org/abs/0804.1879`.

[4] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A compact kernel for the Calculus of Inductive Constructions. *Sadhana*, 34(1):71–144, 2009.

[5] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *TPHOLs 2009*, volume 5674/2009 of *LNCS*, pages 84–98. Springer-Verlag, 2009.

[6] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A new type for tactics. To appear in the proceedings of ACM SIGSAM PLMMS 2009, ISBN 978-1-60558-735-6. Published as Technical Report UBLCS-2009-14, 2009.

[7] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory

for the masses: The POPLmark challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, 2005.

[8] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In George C. Necula and Philip Wadler, editors, *POPL*, pages 3–15. ACM, 2008.

[9] Hendrik P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, NY, 1984. Revised Edition.

[10] Venanzio Capretta and Amy P. Felty. Combining de bruijn indices and higher-order abstract syntax in coq. In *Proceedings of TYPES 2006, volume 4502 of Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2006.

[11] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping (TACS). In Takayasu Ito and Albert R. Meyer, editors, *Proc. of 1st Int. Symp. on Theor. Aspects of Computer Software, TACS'91, Sendai, Japan, 24–27 Sept 1991*, volume 526 of *?*, pages 750–770. Springer-Verlag, Berlin, 1991.

[12] Arthur Charguéraud. Submissions to the PoplMark challenge. `http://www.chargueraud.org/arthur/research/2006/poplmark/`, 2006.

[13] Arthur Charguéraud. The locally nameless representation. Unpublished. http://arthur.chargueraud.org/research/2009/ln/, July 2009.

[14] James Cheney, Michael Norrish, and René Vestergaard. Formalizing adequacy for higher-order abstract syntax: a case study ". *Journal of Automated Reasoning*, 2010. To appear.

[15] Cristina Cornes and Delphine Terrasse. Automating inversion of inductive predicates in coq. In *TYPES '95: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 85–104, London, UK, 1996. Springer-Verlag.

[16] N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Inf. Comput.*, 91:189–204, April 1991.

[17] David Delahaye and Roberto di Cosmo. Information retrieval in a Coq proof library using type isomorphisms. In *Proceedings of TYPES 99*, volume Lecture Notes in Computer Science. Springer-Verlag, 1999.

[18] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in coq. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer Berlin / Heidelberg, 1995. 10.1007/BFb0014049.

[19] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Halle, 1879.

[20] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224, Washington, DC, USA, 1999. IEEE Computer Society Press.

[21] Herman Geuvers and Gueorgui I. Jojgov. Open proofs and open terms: A basis for interactive logic. In J. Bradfield, editor, *Computer Science Logic: 16th International Workshop, CSL 2002*, volume 2471 of *Lecture Notes in Computer Science*, pages 537–552. Springer-Verlag, January 2002.

[22] Eduardo Giménez. Structural recursive definitions in type theory. In *ICALP*, pages 397–408, 1998.

[23] Andrew D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *HUG '93: Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and it Applications*, pages 413–425, London, UK, 1994. Springer-Verlag.

[24] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. Edinburgh LCF: a mechanised logic of computation. volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[25] Ferruccio Guidi. Lambda types on the lambda calculus with abbreviations. Research report UBLCS-2006-25, Department of Computer Science, University of Bologna, November 2006.

[26] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40:143–184, January 1993.

[27] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17:613–673, 2007.

[28] John Harrison. A Mizar Mode for HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *LNCS*, pages 203–220. Springer-Verlag, 1996.

[29] André Hirschowitz and Marco Maggesi. Submission to the PoplMark challenge. `http://web.math.unifi.it/~maggesi/poplmark/Part1a.v`, 2007.

[30] Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 208–212. IEEE Computer Society Press, July 1994.

[31] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *In Venice Festschrift*, pages 83–111. Oxford University Press, 1996.

[32] Chantal Keller and Thorsten Altenkirch. Normalization by hereditary substitutions. To appear in the proceedings of Mathematical Structured Functional Programming, 2010.

[33] Florent Kirchner. *Interoperable proof systems*. PhD thesis, École Polytechnique, 2007.

[34] Xavier Leroy. A locally nameless solution to the POPLmark challenge. Research report 6098, INRIA, January 2007.

[35] William Lovas and Frank Pfenning. A bidirectional refinement type system for lf. In *Electronic Notes in Theoretical Computer Science, 196:113–128*, January 2008.

[36] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[37] The Matita interactive theorem prover. `http://matita.cs.unibo.it`.

[38] Conor McBride. Inverting inductively defined relations in lego. In *TYPES FOR PROOFS AND PROGRAMS, '96, VOLUME 1512 OF LNCS*, pages 236–253. Springer-Verlag, 1998.

[39] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[40] Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, volume 2277 of *LNCS*. Springer-Verlag, 2002.

[41] James McKinna and Robert Pollack. Pure type systems formalized. In *TLCA '93: Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 289–305, London, UK, 1993. Springer-Verlag.

[42] James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23:373–409, 1999.

[43] César Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. PhD thesis, INRIA, November 1997.

[44] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[45] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9:23:1–23:49, June 2008.

[46] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supŕieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[47] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM.

[48] Andrew M. Pitts. Nominal logic: A first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

[49] Gordon Plotkin. An algebraic framework for logics and type theories. Talk given at LFMTP'06 (August 2006).

[50] Randy Pollack. Some recent logical frameworks. Talk given at ProgLog, slides available at `homepages.inf.ed.ac.uk/rpollack/export/canonicalLF_talk.pdf` (February 2007).

[51] Randy Pollack, Masahiko Sato, and Wilmer Ricciotti. A canonical locally named representation of binding. *To appear in a special issue of Journal of Automated Reasoning*, 2010.

[52] Robert Pollack. Closure under alpha-conversion. In Henk Barendregt and Tobias Nipkow, editors, *Proceedings of the Workshop on Types for Proofs and*

Programs, pages 313–332, Nijmegen, The Netherlands, 1993. Springer-Verlag LNCS 806.

[53] Garrel Pottinger. A tour of the multivariate lambda calculus. In J. Michael Dunn and Anil Gupta, editors, *Truth or Consequences: Essays in Honor of Nuel Belnap*, pages 209–229. Kluwer Academic Publishers, Dordrecht, Boston, and London, 1990.

[54] Robinson J. A. A machine-oriented logic based on the resolution principle. In *Journal of the ACM*, volume 2, pages 23–41, 1965.

[55] Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Tinycals: step by step tacticals. In *Proceedings of User Interface for Theorem Provers 2006*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 125–142. Elsevier Science, 2006.

[56] M Sato. Theory of judgments and derivations. In *In: Arikawa, Shinohara (Eds.), Progress in Discovery Science. Vol. 2281 of LNAI*, pages 78–122. Springer-Verlag, 2002.

[57] Masahiko Sato. Theory of symbolic expressions, i. In *Theoretical Computer Science 22*, pages 19–55. 1983.

[58] Masahiko Sato. Theory of symbolic expressions, ii. *Publ. RIMS, Kyoto University*, pages 455–540, 1985.

[59] Masahiko Sato. *An abstraction mechanism for symbolic expressions*, pages 381–391. Academic Press Professional, Inc., San Diego, CA, USA, 1991.

[60] Masahiko Sato. A simple theory of expressions, judgments and derivations. In *ASIAN 2004, Lecture Notes in Computer Science 3321*, page 437. Springer, 2004.

[61] Masahiko Sato. External and internal syntax of the lambda-calculus. In Kutsia Buchberger, Ida, editor, *Proceedings of the Austrian-Japanese workshop on Symbolic Computation in Software Science, SCSS 2008*, pages 176–195, 2008.

[62] Masahiko Sato and Randy Pollack. External and internal syntax of the lambda-calculus. *J. Symb. Comput.*, 45(5):598–616, 2010.

[63] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Proceedings of TPHOLs*, pages 278–293, 2008.

[64] C. Urban. Nominal Techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

[65] Christian Urban and Robert Pollack. Strong induction principles in the locally nameless representation of binders. In *Workshop on Mechanized Metatheory*, 2007.

[66] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework i: Judgments and properties. Technical report, 2003.

[67] Markus Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs*, pages 307–322, 1997.

[68] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics*, pages 167–184, 1999.

[69] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, May 1994.

[70] Freek Wiedijk. Mmode, a mizar mode for the proof assistant coq. Technical Report NIII-R0333, University of Nijmegen, 2003.