Computing with relations, functions, and bindings

Ulysse Gerard

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2019IPPAX005

Thèse de doctorat

ÉCOLE POLYTECHNIQUE

IP PARIS

Computing with relations, functions, and bindings

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à l'École Polytechnique

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Palaiseau, le 18 octobre 2019, par

M. ULYSSE GÉRARD

Composition du Jury :

Gilles Dowek Directeur de Recherche, ENS Cachan (LSV)	Président
Catherine Dubois Directrice de Recherche, Ensiee	Rapporteur
Hugo Herbelin Directeur de Recherche, Inria/Université de Paris, (IRIF)	Rapporteur
Chantal Keller Maître de Conférences, Université Paris-Saclay (LRI)	Examinatrice
Enrico Tassi Chargé de Recherche, Inria Sophia Antipolis	Examinateur
Dale Miller Directeur de recherche, Inria Saclay	Directeur de thèse

Résumé

Cette thèse s'inscrit dans la longue tradition de l'étude des relations entre logique mathématique et calcul et plus spécifiquement de la programmation déclarative. Nous nous intéresserons notamment à la programmation logique pour laquelle l'exécution d'un programme correspond à la recherche d'une preuve ainsi qu'à la programmation fonctionnelle dans laquelle les programmes sont des preuves dont la simplification constitue l'exécution. Ce travail est divisé en deux contributions principales. Chacune d'entre-elles utilise des résultats récents de la théorie de la démonstration pour développer de techniques novatrices utilisant déduction logique et fonctions pour effectuer des calculs.

La première contribution de ce travail consiste en la description et la formalisation d'une nouvelle technique utilisant le mécanisme de la focalisation (un moyen de guider la recherche de preuve) pour distinguer les calculs fonctionnels qui se dissimulent dans les preuves déductives. À cet effet nous formulons un calcul des séquents focalisé pour l'arithmétique de Heyting où points-fixes et égalité sont définis comme des connecteurs logiques. Partant du fait que les prédicats encodant des calculs fonctionnels peuvent êtres vus comme des singletons (un de leurs paramètres est uniquement déterminé par la valeur des autres), il est toujours possible de positionner ces prédicats singletons dans les phases négatives de la preuve, les identifiant ainsi avec un calcul fonctionnel. Ce positionnement est possible en raison de l'ambiguïté de la polarité des singleton dans un système focalisé. Ce processus peut se généraliser aux relations d'équivalences. Cette technique est intéressante car elle elle n'étend en aucune façon la logique sous-jacente: ni opérateur de choix, ni règles de réécritures ne sont nécessaires. Notre logique reste donc purement relationnelle même lorsqu'elle calcule des fonctions. Afin d'éviter que les phases négatives se poursuivent indéfiniment dans le cas de calculs plus complexes, nous introduisons un mécanisme de suspensions, à mi-chemin entre focalisation forte et faible. Nous proposons également, en nous basant sur ce travail théorique, un certain nombre d'extensions simples de l'assistant de preuve Abella ayant pour but d'automatiser certaines parties de la recherche de preuves.

La seconde contribution de cette thèse est la conception d'un nouveau langage de programmation fonctionnelle: MLTS. De nouveau, nous utilisons des travaux théoriques récents en logique: la sémantique de MLTS est ainsi une théorie au sein de la logique $\mathcal{G}$, la logique de raisonnement de l'assistant de preuve Abella. La logique $\mathcal{G}$ utilise un opérateur spécifique: ∇, qui est un quantificateur sur des noms "frais" et autorise un traitement naturel des preuves manipulant des objets pouvant contenir des lieurs. Ce traitement s'appuie sur la gestion naturelle des lieurs fournie par le calcul des séquents. La syntaxe de MLTS est basée sur celle du langage de programmation OCaml mais fournit des constructions additionnelles permettant aux lieurs présents dans les termes de se déplacer au niveau du programme. Il

est ainsi possible de définir des types de données comprenant des lieurs et de déconstruire ces types via une opération de pattern-matching adaptée: un nouveau quantificateur *nab* permet de déclarer les nominaux apparaissant dans un pattern. D'autres opérations sont fournies, comme la création d'un nouveau nominal frais (opérateur *new*) et la construction d'abstractions (opérateur *backslash*). De plus, toutes les opérations sur la syntaxe respectent l'alpha et la béta conversion. Ces deux aspects forment l'approche syntaxique des lieurs appelée λ-tree syntax. Cette partie présente longuement le language à l'aide de nombreux exemples puis décrit son système de typage et sa sémantique naturelle. Le language obtenu apparaît syntaxiquement comme un ajout très succinct au language OCaml, mais est doté d'une grande expressivité vis-à-vis des structures de données contenant des lieurs. Un prototype d'implémentation du langage est fourni, permettant à chacun d'expérimenter facilement en ligne (`https://trymlts.github.io`).

Ce travail a été effectué au sein de l'équipe Parsifal de l'Inria Saclay.

Acknowledgments

First of all I would like to thank Dale who supervised this work and guided me during these years. He provided the setting but also the freedom necessary to my progress, and was always available to discuss and answer my interrogations.

I also want to thank Catherine Dubois and Hugo Herbelin who kindly accepted to review my manuscript and whose precise remarks contributed to its improvement. I also thank Gilles Dowek, Chantal Keller and Enrico Tassi who agreed to take part in the jury of this thesis.

Finally I want to thank all the people who directly or indirectly contributed to the realization of this work: Gabriel Scherer who showed me a fresh point of view over my work, present and former members of the Parsifal team, my friends, my parents and of course Milena.

Contents

A programming language is low level when its programs require attention to the irrelevant.

Alan J. Perlis, *Epigrams in Programming*

Introduction

Mathematical logic and computer science are tightly intertwined. From the hardware design of microprocessors to the creation of new programming languages and analysis of programs, logic is an inevitable discipline to study and design digital systems. However, the most widely used programming languages have often been created incrementally, sometimes without clear semantics, and can have complex and undefined behaviors. An important part of the effort of computer scientists is to be able to understand and verify the correctness of programs written in these languages. But computer science tools can also be used to create new languages directly inspired (and validated) by known mathematical methods. This document focuses on this second aspect and especially on what is called *Declarative Programming*.

Declarative Programming is an umbrella term with multiple interpretations. We consider here one of its commonly admitted meanings: a programming language is said to be declarative if it is based on mathematical logic. By opposition to languages created "from scratch" and usually inspired by low-level considerations on what a microprocessor understands and how a computer is architectured, declarative programming languages leverage well-known mathematical and logical frameworks to build their syntax and semantics. This has several decisive consequences and one of them is that it is easier to reason about programs written in such languages because of their direct relation with familiar formal mathematical systems. It is thus much easier to prove formal properties over programs written in akin languages.

Such "declarative" languages are numerous, span multiple paradigms and make different uses of mathematical logic in their conception. Two common examples of such uses are "proof search as program execution" and "programs as proofs simplification": On the one hand logic is used to inspire syntax as in logic programming languages. This is probably the most straightforward mapping: programs are directly written as sets of formulas and inference rules using variables to express facts and relations about the problem to solve. Computation in logic programming is all about the search for a proof with these given rules, using backtracking to find a path and unification to instantiate variables. If no proof can be found, the answer is a failure, if one or several proofs are found the correct substitutions of variables needed for these successful proofs are returned. As a trivial example, consider the following program made of two clauses terminated with a period:

```
mortal A :- cat A.     % A is mortal if A is a cat
cat socrate.           % socrate is a cat
```

When asked the query "`mortal A.`", the program will search for matching rules or facts and here it will discover that one way to be mortal is to be a cat. Therefore it will try to prove "`cat A`" and doing that will find that the only possible proof requires the substitution "`A = socrate`".

On the other hand, in functional programming languages, a program is seen as a proof and its execution corresponds to the simplification of that proof. This correspondence is embodied by the Curry-Howard isomorphism which establishes a bridge between proof systems and models of computation: A typed program corresponds to a proof, and the formula it proves is the type of that program. This duality between proof and programs, formulas and types, is very fertile and led to broad research topics and to the creation of tools like the Coq system in which proofs are seen as programs that can be formalized, checked and run.

The present thesis pursues the studies of interactions between mathematical logic and computation, starting with modern proof theory results to design new ways to compute with relations and with functions. The first part of this thesis uses focused proof search to reveal functional computations from relations. Indeed, in the arithmetic setting, most proofs involve some computations and those can lead to tedious unfoldings in a proof assistant such as the Abella interactive theorem prover. The focusing discipline (which was originally designed to control and reduce the non-determinism of proof search in Gentzen-style sequent calculi) will allow us to organize proofs in a way that highlights the computational parts and to treat them as functional computation. This will lead us to propose partial automation for proofs in Abella. The second part of this thesis describe the design of a new functional programming language whose semantic is a theory inside $\mathcal{G}$-logic, the reasoning logic of Abella. $\mathcal{G}$-logic uses a specific quantifier: ∇ ("nabla"), a fresh name quantifier, which allows for a natural treatment of proofs over structures with bindings based on the natural handling of bindings of sequent calculus (using eigenvariables). We leverage this feature to give first class citizenship to binders in datastructures in our new language. To syntactically encode theses structures we use λ-tree syntax where terms are encoded as simply typed λ-terms with equality modulo $\alpha\beta\eta$-conversion and bound variables never become free: instead their binding scope is allowed to move.

Outline of the thesis

- Chapter 1 serves as a technical introduction to the rest of the document. We present in it the various notions and tools that are required for a good understanding of that thesis, namely the simply typed λ-calculus (lambda-calculus) and the sequent calculus for intuitionistic logic and Heyting arithmetic. Only common material for both chapters 2 and 3 will be introduced in this part, specific notions will be introduced at the beginning of their respective chapter.

- Chapter 2 proposes a novel approach to isolate computational parts of arithmetic proofs over relations as functions that does not extend the underlying logic but uses the two-phase construction of focused proofs to capture functional computation entirely within some of these phases. We also sketch a few proposals aiming at adding some automation to the Abella proof assistant based on this approach of computation which fits entirely within the logical setting.

- Chapter 3 presents the design and prototyping of MLTS, a new functional programming language that uses the λ-tree syntax approach to encode bindings that appear within data structures. In this setting, bindings can never become free nor escape their scope: instead, binders in data structures are permitted to *move* into binders within programs. By adding additional sites within programs that directly support this movement of bindings we hope to have achieved an elegant and natural approach to the age-old issue of bindings. The natural semantic for this language can be seen as a logical theory with a rich logic leveraging $\mathcal{G}$-logic, the powerful and well-studied system behind the Abella proof assistant.

The present document is based on the following publications:

- **Functional programming with λ-tree syntax** by Ulysse Gérard, Dale Miller, and Gabriel Scherer. Draft dated 14 May 2019. Submitted.
- **Functional programming with λ-tree syntax: a progress report** by Ulysse Gérard and Dale Miller. LFMTP 2018: Logical Frameworks and Meta-Languages: Theory and Practice, Oxford, 7 July 2018. `https://hal.inria.fr/hal-01806129`.
- **Computation-as-deduction in Abella: work in progress** by Kaustuv Chaudhuri, Ulysse Gérard, and Dale Miller. LFMTP 2018: Logical Frameworks and Meta-Languages: Theory and Practice, Oxford, 7 July 2018. `https://hal.inria.fr/hal-01806154`.
- **Separating Functional Computation from Relations** by Ulysse Gérard and Dale Miller. CSL 2017, page 23:1-23:17. `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/csl2017.pdf`.

Chapter 1

The $\mathcal{H}$ proof system for Heyting arithmetic

This chapter serves as an introduction to the common material necessary to the reading and comprehension of the rest of this document. This includes the syntax of terms and formulas we will use and the basic intuitionistic $\mathcal{H}$ calculus for Heyting arithmetic which adopts the axioms of Peano arithmetic, but uses intuitionistic logic as its rules of inference. This base system will be augmented with more advanced features in chapters 2 and 3.

Therefore, in this thesis we only manipulate proof systems for intuitionistic logic, that is, systems that do not include the law of excluded middle (which states that for any proposition A, either A is true or its negation is) nor double negation elimination (which states that if A is true then it is not the case that A is not true).

The term language we are going to use is the simply typed λ-calculus originally conceived by Church. We will only give a short introduction to this system in the first part of this chapter, and the interested reader may find much more details in textbooks like [Bar84] and [Bar92].

This chapter is organized in three sections. Section 1 presents the simply typed λ-calculus, Section 2 presents the syntax of terms and formulas of $\mathcal{H}$. Section 3 presents the sequent calculus rules for the purely logical fragment of $\mathcal{H}$ and then for the arithmetic fragment.

1 The typed λ-calculus

The lambda-calculus is a formal system first introduced by Alonzo Church that allows one to reason about syntax and computation. It is widely used as a mean to model programming languages but can also serve to work with formulas using first order quantifiers in mathematical logic — it was in fact first designed with this intent. We do not aim to give a full course about λ-calculus here, but instead will present some of its core constructs and syntax that are necessary to the understanding of the rest of this work.

1.1 Syntax of terms

The λ-calculus can be seen as a very basic functional programming language whose terms are described by the following grammar:

u, v	::=		terms
	|	x, y, z	(typed) variables
	|	c	(typed) constants
	|	$\lambda x_\tau.\, t$	abstraction
	|	$u\ v$	application

Such a grammar should be read inductively as follows: "A term is either a variable or a constant or an abstraction of a variable over a term or an application of two terms". We indistinctly designate terms as "expressions".

The abstraction $\lambda x_\tau.\, u$ represents the *binding* of a variable x of type τ over the expression u, which is called the *body* of the abstraction. We say that x is *bound* in u. One can see u as a term with holes labelled by x that may be filled later by some other term. Using the programming language analogy, it can be understood as "the function that given a x of type τ returns u". If u is of type τ_2, we say that the term $\lambda x_\tau.\, u$ is of type $\tau \to \tau_2$. We will generally omit the type annotation in abstractions, but it can often easily be inferred from the context.

Because of the abstraction mechanism, variables can be either *free* or *bound* in a term. Here is the precise definition of these notions:

Definition 1.1. Given a term u, the set of the *free variables* of u is denoted $\mathsf{FV}(u)$ and defined by induction:

1. $\mathsf{FV}(x) = \{x\}$, for x a variable
2. $\mathsf{FV}(\lambda x.\, u) = \mathsf{FV}(u) \setminus \{x\}$
3. $\mathsf{FV}(u\ v) = \mathsf{FV}(u) \cup \mathsf{FV}(v)$

Dually, $\mathsf{BD}(u)$, the set of *bound* variables of u, is defined as:

1. $\mathsf{BD}(x) = \emptyset$, for x a variable
2. $\mathsf{BD}(\lambda x.\, u) = \mathsf{BD}(u) \cup \{x\}$
3. $\mathsf{BD}(u\ v) = \mathsf{BD}(u) \cup \mathsf{BD}(v)$

For example, in $(\lambda x.\, x\ y)$, x is a bound variable and y a free variable. But in the subterm $x\ y$ both x and y are free variables.

1.2 Typing using sequent calculus

Figure 1.1 presents the typing rules for our λ-terms in the form of a Gentzen's style *sequent calculus* [Gen35]. Sequent calculi are deductive systems composed of *inference rules* manipulating *sequents*. The sequents represent the *states* of the proof and are usually constituted of at least two zones separated by a metalogical connective called turnstile ($\vdash$). These zones can be either lists, sets, multisets or singletons of terms and formulas. The *inference rules*, which describe steps transforming the sequents —hence going from one state to another—, are of the form:

$$\frac{\text{<premise sequent(s)>}}{\text{<conclusion sequent>}}$$

$$\frac{}{\Sigma, x : \tau \vdash x : \tau}\ \mathsf{Var} \qquad \frac{c : \tau \in \mathcal{C}}{\Sigma \vdash c : \tau}\ \mathsf{Cons}$$

$$\frac{\Sigma, x : \tau_1 \vdash u : \tau_2}{\Sigma \vdash \lambda x_{\tau_1}.\ u : \tau_1 \to \tau_2}\ \mathsf{Lam} \qquad \frac{\Sigma \vdash u : \tau_1 \to \tau_2 \quad \Sigma \vdash v : \tau_2}{\Sigma \vdash u\ v : \tau_2}\ \mathsf{App}$$

Figure 1.1: The typing system for simply-typed λ-terms

In the proof-search setting, such rules should be read in a bottom-up fashion, that is from their conclusion to their premises. We call a *derivation* a tree structure of occurrences of inference rules: a derivation has one conclusion, its end sequent, and possibly several premises. A derivation with no premises is called a *proof*.

The typing system of Fig. 1.1 makes use of sequents of the form $\Sigma \vdash u : \tau$, called *typing sequents* which are formed of 3 zones: a set Σ of assignments of types to variables on the left of the turnstile, a term on the right, and the type of this term separated from it by a colon. An expression u is said to be a *well-formed* λ-term over the set of constants $\mathcal{C}$ when there is a context Σ such that the judgement $\Sigma \vdash u : \tau$ is derivable in that system. As we stated before, in the proof-search setting we usually read such rules in a bottom-up fashion as we are looking for a proof of a given sequent. For example, the rule "Lam" should be read "The term $\lambda x_{\tau_1}.\ u$ has type $\tau_1 \to \tau_2$ if u has type τ_2 knowing that x is of type τ_1".

Example 1.2. The typing derivation for the term $\lambda x_{\tau_1 \to \tau_2}.\ x\ c$ where c is a constant of type τ_1 is:

$$\dfrac{\dfrac{\dfrac{}{\{x : \tau_1 \to \tau_2\} \vdash x : \tau_1 \to \tau_2}\ \mathsf{Var} \qquad \dfrac{c : \tau_1 \in \mathcal{C}}{\{x : \tau_1 \to \tau_2\} \vdash c : \tau_1}\ \mathsf{Cons}}{\{x : \tau_1 \to \tau_2\} \vdash x\ c : \tau_2}\ \mathsf{App}}{\varnothing \vdash \lambda x_{\tau_1 \to \tau_2}.\ x\ c : (\tau_1 \to \tau_2) \to \tau_2}\ \mathsf{Lam}$$

1.3 Computing with the λ-calculus

In this section we recall the definition of λ-conversion given in [MN12]. First of all we say that u is *free for* a variable x in s if the free occurrences of x in s are not in the scope of any abstraction that binds free variables of u. For example the term $f\ y$ is not free for u in $\lambda y.\ (g\ u\ y)$, and we cannot substitute u by $f\ y$ in $\lambda y.\ (g\ u\ y)$ without *capturing* the free occurrence of y in $f\ y$: $\lambda \mathbf{y}.\ (g\ (f\ \mathbf{y})\ y)$.

We write $t[u/x]$ to denote the term t in which all free occurrences of x have been replaced by u. This operation is called *substitution*.

- We call α-*rewriting* the replacement of a subterm $\lambda x.\ u$ by $\lambda y.\ u[y/x]$ when y is free for x in u and y is not free in u.
- α-*conversion* is the reflexive, symmetric and transitive closure of α-rewriting.
- We call β-contraction the replacement of a subterm $(\lambda x.\ u)\ v$ by $u[v/x]$. β-expansion is the converse operation.
- β-*reduction* is the reflexive and transitive closure of the union of α-conversion and β-contraction. The symmetric and transitive closure of β-reduction is called β-*conversion*.

- η-contraction is the replacement of a subterm $\lambda x.\ (u\ x)$ by u, provided that x is not free in u. The converse operation is called η-expansion.
- η-*reduction* is the reflexive and transitive closure of η-contraction and finally η-*conversion* is the symmetric and transitive closure of η-reduction.

Example 1.3.

- The two terms $\lambda x.\ \lambda y.\ f\ (g\ x)\ y$ and $\lambda z.\ \lambda t.\ f\ (g\ z)\ t$ are related by α-conversion.
- η-contraction of $\lambda x.\ (\lambda y.\ f\ (g\ x)\ y)$ yields the term $\lambda x.\ f\ (g\ x)$.
- $\lambda x.\ \lambda y.\ f\ ((\lambda u.\ \lambda v.\ v)\ (g\ y)\ (g\ x))\ y$ β-contracts to $\lambda x.\ \lambda y.\ f\ ((\lambda v.\ v)\ (g\ x))\ y$ which β-contracts to $\lambda x.\ \lambda y.\ f\ (g\ x)\ y$.

The transitive closure of α, β and η conversions is called λ-*conversion*. These rules define a computational behavior for the λ-calculus. A term of the form $(\lambda x.\ u)\ v$ is called a β-redex and in the programming language analogy represents the application of a function to an argument. Such a term can be reduced by substituting x by v in u, an operation denoted as: $u[v/x]$. This substitution yields the result of the function application. A term with no β-redex is said to be in β-*normal form*.

The simple mechanism of λ-calculus can express many computational behaviors using suited encodings. One of the most well known is Church's encoding of natural numbers. Natural numbers can be built using two things: the number zero (z) and the successor operation (s) which adds 1 to a natural number. In this encoding the number 3 is represented by $\lambda s.\ \lambda z.\ s\ (s\ (s\ (z)))$.

One way to understand this representation is that the number n is encoded by n successive applications of the function s. The addition operation can thus be defined as: $\lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z)$.

Example 1.4. Successive β-reductions computing the addition of 1 and 2:

$$\begin{aligned}
&(\lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z))\ (\lambda s.\ \lambda z.\ s\ z)\ (\lambda s.\ \lambda z.\ s\ (s\ z))\\
\to_\beta^{\times 2}\ &\lambda s.\ \lambda z.\ (\lambda s'.\ \lambda z'.\ s'\ z')\ \ s\ ((\lambda s''.\ \lambda z''.\ s''\ (s''\ z''))\ \ s\ z)\\
\to_\beta^{\times 2}\ &\lambda s.\ \lambda z.\ (\lambda s'.\ \lambda z'.\ s'\ z')\ \ s\ (s\ (s\ z))\\
\to_\beta^{\times 2}\ &\lambda s.\ \lambda z.\ s\ (s\ (s\ z))
\end{aligned}$$

(where $\to_\beta^{\times n}$ denotes n successive β-reductions)

Multiplication, division, exponentiation, boolean values and many more can be defined using such encodings.

2 Terms and formulas for $\mathcal{H}$

Following Church's simple theory of types [Chu40; And09], we will use simply typed λ-terms to represent our terms and formulas. We call o the type of formulas. Unlike Church we do not use the single sort ι for terms which are not a formula. Instead we consider any type in which o does not appear to be a valid term type. We denote such types by τ.

In more precise terms, the grammar of types and terms is the following:

Definition 2.1.

$$\begin{array}{rr} \text{Types} & \alpha, \beta ::= \ o \mid \tau \mid \alpha \to \beta \\ \text{Term types} & \tau ::= \ \tau \in \mathcal{T} \mid \tau_1 \to \tau_2 \\ \text{Terms} & u, v ::= \ x \mid c_\tau \in \mathcal{C} \mid \lambda x_\tau.\ u \mid u\ v \end{array}$$

where $\mathcal{T}$ is a set of basic term types also called *kinds* and $\mathcal{C}$ is a set of constants.

Propositional intuitionistic logic formulas are given by the logical connectives $\wedge$, $\vee$, $\supset$, $\forall_\tau$, $\exists_\tau$, the logical constants t and f, and a set of atomic formulas $\mathcal{A}$:

$$\begin{array}{r} t, f, a \in \mathcal{A} : o \\ \wedge, \vee, \supset : o \to o \to o \\ \forall_\tau, \exists_\tau : (\tau \to o) \to o \end{array}$$

That is, the complete grammar of formulas is the following:

Definition 2.2.

$$\begin{array}{r} \mathrm{A}, \mathrm{B} ::= \ t \mid f \mid a \in \mathcal{A} \mid \mathrm{A} \wedge \mathrm{B} \mid \mathrm{A} \vee \mathrm{B} \\ \mid \mathrm{A} \supset \mathrm{B} \mid \forall_\tau(\lambda x_\tau.\ \mathrm{A}) \mid \exists_\tau(\lambda x_\tau.\ \mathrm{A}) \end{array}$$

This style of encoding quantifiers, also called λ-*tree syntax* [MP99] comes from Church [Chu40] and should be familiar to the users of λProlog and Abella. This syntax leverages the terms of the λ-calculus as a representation mechanism for formulas, thus allowing a uniform treatment of bindings in terms and formulas. For instance, in the formula $\exists_\tau(\lambda x_\tau.\ \mathrm{P}(x_\tau))$ the λ-abstraction is used to describe the usual scoping effect of the existential quantifier.

To ease the writing and reading of this document we will continue to use the more lightweight concrete syntax $\exists_\tau x_\tau.t$ to denote such formulas in most cases. We will also get rid of the subscripts indicating the types of terms when they can be easily inferred from the context.

It is important to notice that in the arithmetic setting we will not make use of the atomic formulas ($\mathcal{A}$) and their associated initial rule because all "atoms" will be defined using the fixed point predicate. However having such undefined atomic formulas is convenient for pedagogical purposes when describing the following system $\mathcal{H}$ and the proof-search and focusing mechanisms.

3 The $\mathcal{H}$ sequent calculus

$\mathcal{H}$ (the "h" stands for Heyting) is a sequent calculus proof system for arithmetic where fixed points and term equality are *logical connectives*: that is, they are defined via their left- and right-introduction rules. This work builds on earlier work by McDowell & Miller [MM00] and Momigliano & Tiu [TM12]. It is important to notice that we work with cut-free sequent calculi, that is, we are not interested in the computational meaning provided by the cut-elimination procedure but by the one carried by the search for a proof. The cut rule is still a part of $\mathcal{H}$, but we will not make use of it. We will show this rule nonetheless in this introductory part, but

it won't be reminded systematically in the following systems. Our presentation of $\mathcal{H}$ makes use of the sequent calculus formalism presented in Section 1.2 with one more kind of sequent: proof sequents. The next definition provides a number of notations that we are going to use.

Definition 3.1. Notations

- A, B, C denote formulas
- u, v denote terms in $\beta\eta$-long normal form
- τ denotes a term type
- Γ is a multiset of formulas called *hypothesis*
- Σ is a list of typed term variables, called the *signature*, which are considered bound over a sequent

We call $\beta\eta$-long normal form the variant of β-normal form that also takes into account η-conversion. Typed λ-terms of this form have the following structure.

$$\lambda x_1 \ldots \lambda x_n.\ (h\ t_1 \ldots t_p) \text{ for } (n, p \geq 0)$$

where h is either a constant or a variable, the terms t_i are also in $\beta\eta$-long normal form and the body $(h\ t_1 \ldots t_p)$ has a type without arrows (a non-functional type).

Furthermore, we are going to assume that *bindings only occur at the formula level and not the term level*, that is only in quantification and, later, fixpoints. Enforcing this restriction allows us to stay in the realm of first-order unification and focus on the core concepts of our work: the computational meaning of relations. (However we could relax these restrictions as other papers have explored the full higher-order situation like in [MM00].)

Definition 3.2. The $\mathcal{H}$ calculus makes use of two kinds of sequents, both composed of three zones:

proof SEQUENTS	$\Sigma : \Gamma \vdash A$
typing SEQUENTS	$\Sigma \vdash u : \tau$

In Fig. 1.2, when we write $y : \tau, \Sigma$ we imply that y does not already appear as one of the variables in Σ.

3.1 The logical core of $\mathcal{H}$

Fig. 1.2 shows the rules for a fragment of $\mathcal{H}$, including connectives $\supset$, $\wedge$ and $\vee$ and the identity rules. This fragment corresponds to Gentzen's LJ sequent calculus.

Identity While it will always remain a part of our systems, the usual axiom rule that deals with "undefined" atoms will not be used when we manipulate arithmetic terms because atoms will be defined as fixed-points. Moreover, as stated earlier, this work will focus on cut-free formulas and thus we will not make use of the cut rule, but it is still a part of the complete system.

Rules for $\supset$, $\wedge$ and $\vee$ are the standard rules for intuitionistic propositional logic. As an example of how these rules should be understood in the proof search setting, the rule $\supset_r$ is to be read: "Knowing Γ, A implies B is true if, knowing Γ and A, B is

IDENTITY RULES

$$\frac{}{\Gamma, A \vdash A}\ \mathsf{ax} \qquad \frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B}\ \mathsf{cut}$$

PROPOSITIONAL INTUITIONISTIC LOGIC RULES

$$\frac{}{\Gamma, f \vdash A}\ f \qquad \frac{}{\Gamma \vdash t}\ t$$

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}\ \vee_l \qquad \frac{\Gamma \vdash B_i}{\Gamma \vdash B_1 \vee B_2}\ \vee_{r, i\in\{1,2\}}$$

$$\frac{\Gamma, B_1, B_2 \vdash C}{\Gamma, B_1 \wedge B_2 \vdash C}\ \wedge_l \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge_r$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ \supset_r \qquad \frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \supset B \vdash C}\ \supset_l$$

Figure 1.2: $\mathcal{H}$ rules for $\supset$, $\wedge$ and $\vee$

true". Notice that in this fragment of $\mathcal{H}$ the Σ environment remains unused and that is why it is not shown in these rules. Its usefulness will appear later with the addition of rules for quantification and equality.

Example 3.3. The formula $P \supset Q \supset (\neg Q \supset \neg P)$, where $\neg P$ denotes $P \supset f$ has the following proof derivation (that should be read bottom-up) in $\mathcal{H}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Q \supset f, P \vdash P}\ \mathsf{ax} \quad \dfrac{\dfrac{}{P, Q \vdash Q}\ \mathsf{ax}}{Q \supset f, P, Q \vdash f}\ \mathsf{f}}{P \supset Q, Q \supset f, P \vdash f}\ \supset_l}{P \supset Q, Q \supset f \vdash P \supset f}\ \supset_r}{P \supset Q \vdash (Q \supset f) \supset (P \supset f)}\ \supset_r}{\varnothing \vdash P \supset Q \supset ((Q \supset f) \supset (P \supset f))}\ \supset_r$$

Quantification In order to support first-order quantification we extend sequents to permit the proof-level binding mechanism of *eigenvariables* [Gen35]. To that end, we prefix all sequents with Σ :, where Σ is a list of variables that are considered bound over the sequent. When we write a prefix as $y : \tau, \Sigma$, we imply that y does not already appear as one of the variables in Σ. We use the expression $B[t/x]$ to denote the $\beta\eta$-long normal form of $(\lambda x.\, B)\ t$.

The inference rules for quantification that should be added to the system shown in Fig. 1.2 are as follows.

$$\frac{y : \tau, \Sigma : \Gamma, B[y/x] \vdash C}{\Sigma : \Gamma, \exists x_\tau.\, B \vdash C}\ \exists_l \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma : \Gamma \vdash B[t/x]}{\Sigma : \Gamma \vdash \exists x_\tau.\, B}\ \exists_r$$

$$\frac{\Sigma \vdash t : \tau \quad \Sigma : \Gamma, B[t/x] \vdash C}{\Sigma : \Gamma, \forall x_\tau.\, B \vdash C}\ \forall_l \qquad \frac{y : \tau, \Sigma : \Gamma \vdash B[y/x]}{\Sigma : \Gamma \vdash \forall x_\tau.\, B}\ \forall_r$$

The $\exists_r$ and $\forall_l$ rules make use of a typing assumption (highlighted in green): $\Sigma \vdash t : \tau$ denotes the fact that t is a term in $\beta\eta$-long form of type τ. The associated proof system is the standard typing system for simply-typed λ-terms shown in Fig. 1.1.

3.2 The arithmetic part of $\mathcal{H}$

Finally, we extend the previously presented system for intuitionistic logic with equality and fixed-points treated as logical connectives.

Term equality The inference rules for term equality are adapted from early papers by Schroeder-Heister [Sch93] and Girard [Gir92] (see also [MM00]).

$$\frac{}{\Sigma : \Gamma \vdash t = t} =_r \qquad \frac{}{\Sigma : \Gamma, s = t \vdash \mathrm{B}} =_l, \; s \text{ and } t \text{ not unifiable}$$

$$\frac{\Sigma\theta : \Gamma\theta \vdash \mathrm{B}\theta}{\Sigma : \Gamma, s = t \vdash \mathrm{B}} =_l, \; \theta \text{ the mgu of } s \text{ and } t$$

Where *mgu* stands for "most general unifier".

Example 3.4. Proof of the transitivity of equality:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y : \tau : \varnothing \vdash y = y} =_r}{z : \tau, y : \tau : y = z \vdash y = z} =_l \text{ with } \theta = \{z \mapsto y\}}{z : \tau, y : \tau, x : \tau : x = y, y = z \vdash x = z} =_l \text{ with } \theta = \{x \mapsto y\}}{z : \tau, y : \tau, x : \tau : x = y \wedge y = z \vdash x = z} \wedge_l}{z : \tau, y : \tau, x : \tau : \varnothing \vdash (x = y \wedge y = z) \supset (x = z)} \supset_r}{y : \tau, x : \tau : \varnothing \vdash \forall z_\tau.\ (x = y \wedge y = z) \supset (x = z)} \forall_r}{x : \tau : \varnothing \vdash \forall y_\tau.\ \forall z_\tau.\ (x = y \wedge y = z) \supset (x = z)} \forall_r}{\varnothing : \varnothing \vdash \forall x_\tau.\ \forall y_\tau.\ \forall z_\tau.\ (x = y \wedge y = z) \supset (x = z)} \forall_r$$

Notice that while provability in the propositional fragment is known to be decidable [Gen35], it has been shown in [VM10] that adding these rules for term equality and quantification results in an undecidable logic even if we restrict to just first-order terms and quantifiers and even without any predicate symbols (and, hence, without atomic formulas).

Fixed-points There have been many treatments of fixed points and induction within proof systems such as those involving Peano's axioms and induction schemes or those using a specially designed proof system such as Scott induction [GMW79]. Here, we restrict our attention to the rather minimalist setting where the fixed point operator μ is treated as a logical connective in the sense that it has left- and right-introduction rules: these rules simply unfold μ-expressions. While the resulting fixed point operator is self-dual and rather weak, it can still play a useful role in proving some weak theorems of arithmetic [Gir92; Sch93; MM00] and it can provide an interesting proof theory for aspects of model checking [Bae08; HM17; TNM05]. It is possible to describe a more powerful proof system for fixed points that uses induction and co-induction to describe the introduction rules for the *least* and *greatest* fixed points [MM00; TM12].

The logical constant μ is parameterized by a list of typed constants as follows:

$$\mu^n_{\tau_1,\ldots,\tau_n} : ((\tau_1 \to \ldots \to \tau_n \to o) \to \tau_1 \to \ldots \to \tau_n \to o) \to \tau_1 \to \ldots \to \tau_n \to o$$

where $n \geq 0$ and $\tau_1, \ldots, \tau_n$ are simple types and o is the type of formulas.

Expressions of the form $\mu^n_{\tau_1,\ldots,\tau_n} B t_1 \ldots t_n$ will be abbreviated as simply $\mu B \bar{t}$ (where $\bar{t}$ denotes the list of terms $t_1 \ldots t_n$). We shall also restrict fixed point expressions to use only *monotonic* higher-order abstraction: that is, in the expression $\mu^n_{\tau_1,\ldots,\tau_n} B t_1 \ldots t_n$ the expression B is equivalent (via $\beta\eta$-conversion) to $\lambda P_{\tau_1 \to \ldots \to \tau_n \to o}\ \lambda x^1_{\tau_1} \ldots \lambda x^n_{\tau_n}\ B'$ where all occurrences of the variable P in B′ occur to the left of an implication an even number of times.

The unfolding of the fixed point expression $\mu B\, \bar{t}$ yields $B(\mu B)\, \bar{t}$ and the introduction rules for μ establish the logical equivalence of these two expressions. The inference rules for the μ connective are:

$$\frac{\Sigma : \Gamma, B(\mu B)\ \bar{t} \vdash C}{\Sigma : \Gamma, \mu B\ \bar{t} \vdash C}\ \mu_l \qquad \frac{\Sigma : \Gamma \vdash B(\mu B)\ \bar{t}}{\Sigma : \Gamma \vdash \mu B\ \bar{t}}\ \mu_r$$

Example 3.5. Assume that we have a primitive type i and that there are two typed constants $z : i$ and $s : i \to i$. We shall abbreviate the terms z, $(s\ z)$, $(s\ (s\ z))$, $(s\ (s\ (s\ z)))$, etc by $\mathbf{0}$, $\mathbf{1}$, $\mathbf{2}$, $\mathbf{3}$, etc. The following two named fixed point expressions define the natural number predicate and the ternary relation of addition.

$$\mathsf{nat} = \mu\lambda N \lambda n (n = \mathbf{0} \vee \exists n' (n = s\ n' \wedge N\ n'))$$
$$\mathsf{plus} = \mu\lambda P \lambda n \lambda m \lambda p ((n = \mathbf{0} \wedge m = p) \vee \exists n' \exists p' (n = s\ n' \wedge p = s\ p' \wedge P\ n'\ m\ p'))$$

The following theorem states that the plus relation describes a (total) functional dependency between its first two arguments and its third.

$$\forall m, n (\mathsf{nat}\ m \supset \exists k (\mathsf{plus}\ m\ n\ k)) \wedge \forall m, n, p, q (plus\ m\ n\ p \supset plus\ m\ n\ q \supset p = q)$$

Fig. 1.3 shows the complete $\mathcal{H}$ proof system. With only unfolding on fixpoints and no induction mechanism very few proofs of properties about all numbers are achievable using $\mathcal{H}$. It is a simplification we assume for the design we will present in Chapter 2. Indeed induction is important to deduction but plays an irrelevant role in computation which is our main subject of study.

IDENTITY RULE

$$\frac{}{\Sigma : \Gamma, A \vdash A}\ \text{ax}$$

PROPOSITIONAL INTUITIONISTIC LOGIC RULES

$$\frac{}{\Sigma : \Gamma, f \vdash A}\ f \qquad \frac{}{\Sigma : \Gamma \vdash t}\ t$$

$$\frac{\Sigma : \Gamma, A \vdash C \quad \Sigma : \Gamma, B \vdash C}{\Sigma : \Gamma, A \vee B \vdash C}\ \vee_l \qquad \frac{\Sigma : \Gamma \vdash B_i}{\Sigma : \Gamma \vdash B_1 \vee B_2}\ \vee_{r, i\in\{1,2\}}$$

$$\frac{\Sigma : \Gamma, B_1, B_2 \vdash C}{\Sigma : \Gamma, B_1 \wedge B_2 \vdash C}\ \wedge_l \qquad \frac{\Sigma : \Gamma \vdash A \quad \Sigma : \Gamma \vdash B}{\Sigma : \Gamma \vdash A \wedge B}\ \wedge_r$$

$$\frac{\Sigma : \Gamma, A \vdash B}{\Sigma : \Gamma \vdash A \supset B}\ \supset_r \qquad \frac{\Sigma : \Gamma \vdash A \quad \Sigma : \Gamma, B \vdash C}{\Sigma : \Gamma, A \supset B \vdash C}\ \supset_l$$

TYPED FIRST-ORDER QUANTIFICATION RULES

$$\frac{y : \tau, \Sigma : \Gamma, B[y/x] \vdash C}{\Sigma : \Gamma, \exists x_\tau.\ B \vdash C}\ \exists_l \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma : \Gamma \vdash B[t/x]}{\Sigma : \Gamma \vdash \exists x_\tau.\ B}\ \exists_r$$

$$\frac{\Sigma \vdash t : \tau \quad \Sigma : \Gamma, B[t/x] \vdash C}{\Sigma : \Gamma, \forall x_\tau.\ B \vdash C}\ \forall_l \qquad \frac{y : \tau, \Sigma : \Gamma \vdash B[y/x]}{\Sigma : \Gamma \vdash \forall x_\tau.\ B}\ \forall_r$$

EQUALITY RULES

$$\frac{}{\Sigma : \Gamma, s = t \vdash B}\ =_l,\ s \text{ and } t \text{ not unifiable} \qquad \frac{}{\Sigma : \Gamma \vdash t = t}\ =_r$$

$$\frac{\Sigma\theta : \Gamma\theta \vdash B\theta}{\Sigma : \Gamma, s = t \vdash B}\ =_l,\ \theta \text{ the mgu of } s \text{ and } t$$

FIXED-POINT RULES

$$\frac{\Sigma : \Gamma, B(\mu B)\ \bar{t} \vdash C}{\Sigma : \Gamma, \mu B\ \bar{t} \vdash C}\ \mu_l \qquad \frac{\Sigma : \Gamma \vdash B(\mu B)\ \bar{t}}{\Sigma : \Gamma \vdash \mu B\ \bar{t}}\ \mu_r$$

Figure 1.3: The proof system $\mathcal{H}$ for intuitionistic logic

Chapter 2

Separating functional computation from relations

The development of the logical foundations of arithmetic generally starts with the first-order logic of relations to which constructors for zero and successor have been added. Various axioms (such as Peano's axioms) are then added to that framework in order to define the natural numbers and various relations among them. Of course, it is often natural to think of some computations, such as say, the addition and multiplication of natural numbers, as being *functions* instead of relations.

A common way to introduce functions into the relational setting is to enhance the equality theory. For example, Troelstra in [Tro73, Section I.3] presents an intuitionistic theory of arithmetic in which all primitive recursive functions are treated as black boxes and every one of their instances, for example $23 + 756 = 779$, is simply added as an equation. A modern and more structured version of this approach is that of the λΠ-*calculus modulo* framework proposed by Cousineau & Dowek [CD07]: in that framework, the dependently typed λ-calculus (a presentation of intuitionistic predicate logic) is extended with a rich set of terms and rewriting rules on them. When rewriting is confluent, it can be given a functional programming implementation: the Dedukti proof checker [Ass+16] is based on this hybrid approach to treating functions in a relational setting.

A predicate can, of course, encode a function. For example, assume that we have a $(n+1)$-ary ($n \geq 0$) predicate R for which we can prove that the first n arguments uniquely determine the value of its last argument. That is, assume that the following formula is provable (here, $\bar{x}$ denotes the list of variables $x_1, \ldots, x_n$):

$$\forall \bar{x}([\exists y.\mathrm{R}(\bar{x}, y)] \wedge \forall y \forall z[\mathrm{R}(\bar{x}, y) \supset \mathrm{R}(\bar{x}, z) \supset y = z]).$$

In this situation, an n-ary function f_{R} exists such that $f_{\mathrm{R}}(\bar{x}) = y$ if and only if $\mathrm{R}(\bar{x}, y)$. In order to formally describe the function f_{R}, Hilbert [HB39] and Church [Chu40] evoked *choice operators* such as ε and ι which (along with appropriate axioms) are able to take a singleton set and return the unique element in that set. For example, in Church's Simple Theory of Types [Chu40], a definition of f_{R} is provided by the expression $\lambda x_1 \ldots \lambda x_n \iota(\lambda y.\mathrm{R}(x_1, \ldots, x_n, y))$.

In this chapter, we present a different approach to separating functional computations from more general reasoning with relations. We shall not extend the equational theory beyond the minimal equality on terms and we shall not use choice principles.

Although this approach to separating functions from relations is novel, it does not need any new theoretical results: we simply make direct use of several recent results in proof theory.

Chapter outline

1. We introduce *focused proof systems* as developed by Andreoli, Baelde, and Liang & Miller [And92; LM09; Bae12]. Such inference systems structure proofs into two *phases*: the *negative* phase organizes *don't-care nondeterminism* while the *positive* phase organizes *don't-know nondeterminism*. This introduction to focusing will be based on the progressive transformation of $\mathcal{H}$ into $\mathcal{F}$, its focused counterpart.
2. Having our focused system, we notice that the construction of a negative phase (reading it as a mapping from its conclusion to its premises) determines a function and the construction of the positive phase determines a more general nondeterministic relation.
3. Since $\forall x[\mathrm{P}(x) \supset \mathrm{Q}(x)] \equiv \exists x[\mathrm{P}(x) \wedge \mathrm{Q}(x)]$ whenever predicate P denotes a singleton set, the resulting *ambiguity of polarity* makes it possible to position such singleton predicates always into the negative phase. As mentioned above, a suitable treatment of singleton sets allows for a direct treatment of functions.
4. The resulting proof system provides a means to take the specification of a relation and use it directly to compute a function (something that is not available directly when applying choice operators).

These various steps lead to the systematic construction of a single, expressive proof system in which functional computation is abstracted away from quantificational logic. Finally we propose in Section 4 two additions to the Abella proof assistant that could automate some of the computations happening in proofs.

1 An introduction to focusing

Gentzen's style sequent calculus has encountered a lot of success in giving a proof theory to classical, intuitionistic and linear logics. However its main feature, tiny low-level building blocks, is also a major drawback for proof generation. Simple algorithms based on it give rise to an unbearable amount of non-determinism. Moreover, this chaotic behavior allows for numerous proofs of the same formula implying a lack of canonicity: multiple proofs exist for the same formula, and these proofs often seem essentially the same: a lot of "administrative" work can be done in many different orders thus leading to proofs of which macro structure is the same but where internal differences appear.

Focusing techniques appeared as a means to guide the proving process and to reduce mayhem in the application of inference rules in LJ, a sequent calculus for intuitionistic logic and LK, its classical counterpart. One of these early techniques, uniform proofs [Mil+91], consists in an alternation of two phases: goal-directed search and backchaining. It was developed for the purpose of enriching logic programming languages. The result of this operation is the λProlog language [MN12] which we will use as a prototyping tool in Chapter 3.

Then Andreoli extended the same two-phases technique to linear logic giving birth to focusing [And92]. Later, several systems appeared with different flavors of

focusing such as Herbelin's LJT [Her95] and Dyckhoff & Lengrand's LJQ [DL06] and the LJF system elaborated by Liang and Miller [LM07].

In this section we will illustrate the focusing discipline on a very simple proof system with the implication as sole connective. We call this system $\mathcal{F}_{\supset}$ and we will take it as a basis to build the $\mathcal{F}$ proof system for arithmetic, a focused version of $\mathcal{H}$.

1.1 The $\mathcal{F}_{\supset}$ calculus

The main principle of focusing is to classify inference rules in two groups, invertible ones and non-invertible ones. Then it is possible to organize proofs in an alternation of two phases: the *asynchronous phase* during which are applied invertible rules only and the *synchronous phase* when non-invertible rules are applied.

Definition 1.1. We call *invertible* the rules whose conclusion and premises are equiprovable.

Two invertible rules can always be permuted in the tree of inferences. Thus the application order of a series of invertible rules has no impact on the derivability of a formula and such rule applications are guaranteed not to affect the provability of the goal formula. In the two-phases process these sequences of invertible rules, eagerly applied, constitute the *asynchronous* phase. In Section 2 and Section 3 we will show how invertible phases can provide for a natural treatment of computational behavior. In contrast, the *synchronous* phases, where non-invertible rules are used, require choices to be made in order to progress in the proof. One example of such a choice is the necessity to exhibit a witness for an existentially quantified formula. When a synchronous phase is entered, the proof is focused on a particular formula which is then used to indicate followup synchronous rules, maintaining focus as long as possible. This drastically reduces the choice points in the proof, since after a focus has been decided the choices are constrained to those relevant to that chosen formula.

Connectives whose right introduction rules are invertible will be called *negative* or *asynchronous* whereas the others will be called *positive* or *synchronous*. We will use the following set of notations in all our focused systems:

Definition 1.2. Notations

- A, B, C denote formulas
- N_a and P_a respectively range over negatively polarized atoms and positively polarized atoms
- E denotes either a positive formula or a negative atom
- C denotes either a negative formula or a positive atom
- t, u, v denote terms in $\beta\eta$-long normal form
- τ denotes a term type
- Γ is a multiset of formula
- Θ is a list of formulas
- Σ is a list of typed term variables, called the *signature*, which are considered bound over a sequent. To lighten the sequents the signature will be omitted when not necessary.
- Δ_1 and Δ_2 are two multisets of formulas such that $\Delta_1 \cup \Delta_2$ contains one and unique formula.
- We use dots $(\cdot)$ to materialize empty zones.

STRUCTURAL RULES

$$\frac{\Gamma, N \Downarrow N \vdash \cdot \Downarrow E}{\Gamma, N \Uparrow \cdot \vdash \cdot \Uparrow E}\ D_l \qquad \frac{C, \Gamma \Uparrow \Theta \vdash \Delta_1 \Uparrow \Delta_2}{\Gamma \Uparrow C, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ S_l \qquad \frac{\Gamma \Uparrow P \vdash \cdot \Uparrow E}{\Gamma \Downarrow P \vdash \cdot \Downarrow E}\ R_l$$

$$\frac{\Gamma \Downarrow \cdot \vdash P \Downarrow \cdot}{\Gamma \Uparrow \cdot \vdash \cdot \Uparrow P}\ D_r \qquad \frac{\Gamma \Uparrow \cdot \vdash \cdot \Uparrow E}{\Gamma \Uparrow \cdot \vdash E \Uparrow \cdot}\ S_r \qquad \frac{\Gamma \Uparrow \cdot \vdash N \Uparrow \cdot}{\Gamma \Downarrow \cdot \vdash N \Downarrow \cdot}\ R_r$$

NEGATIVE PHASE INTRODUCTION RULES

$$\frac{\Gamma \Uparrow A \vdash B \Uparrow \cdot}{\Gamma \Uparrow \cdot \vdash A \supset B \Uparrow \cdot}\ \supset_r$$

POSITIVE PHASE INTRODUCTION RULES

$$\frac{\Gamma \Downarrow \cdot \vdash A \Downarrow \cdot \qquad \Gamma \Downarrow B \vdash \cdot \Downarrow E}{\Gamma \Downarrow A \supset B \vdash \cdot \Downarrow E}\ \supset_l$$

$$\frac{}{\Gamma \Downarrow N_a \vdash \cdot \Downarrow N_a}\ I_l \qquad \frac{}{\Gamma, P_a \Downarrow \cdot \vdash P_a \Downarrow \cdot}\ I_r$$

Figure 2.1: The $\mathcal{F}_{\supset}$ proof system

Note that atoms can be assigned any bias (positive or negative) without modifying the provability of the formula. But formulas' polarity depends on the invertibility of the right introduction rule of their topmost connective.

The formulas (denoted A, B ...), positive formulas (P, Q ...) and negative formulas (N, M ...) of $\mathcal{F}_{\supset}$ are:

- $A, B ::= P \mid N$
- $P, Q ::= P_a$
- $N, M ::= N_a \mid A \supset B$

The implementation of the focusing mechanism requires the use of a set of six "structural" rules that mediate between phases. By restricting ourselves to the implicative fragment of intuitionistic we will be able to focus efficiently on these new features.

We momentarily exit the "all predicates (such as nat, plus, and times) be defined" setting, and again consider the usual approach to propositional logic where formulas can contain "undefined" atoms. The provability of a formula is not changed by the choice of polarities attributed to its atoms. That is, erasing the polarities of atoms and connective of a provable formula in LJF always leads to a provable formula in LJ.

Definition 1.3. The $\mathcal{F}_{\supset}$ calculus makes use of three kinds of sequents:

unfocused SEQUENTS	$\Sigma : \Gamma \Uparrow \Theta \vdash \Delta_1 \Uparrow \Delta_2$
focused SEQUENTS	$\Sigma : \Gamma \Downarrow \Theta \vdash \Delta_1 \Downarrow \Delta_2$
typing SEQUENTS	$\Sigma \vdash t : \tau$

Since we are considering only *single-focused* proof systems (as opposed to *multifocused* proof systems [CMS08]), we require that sequents of the form $\Sigma : \Gamma \Downarrow \Theta \vdash \Delta_1 \Downarrow \Delta_2$ have the property that the union of Δ_1 and Θ is always a singleton. Moreover, because we are working with intuitionistic logic, the union of Δ_1 and Δ_1 is always a singleton.

A derivation that contains only negative (unfocused) sequents is a *negative phase*: such a phase contains introduction rules for negative connectives, and the storage rules (S_l and S_r). A derivation that contains only positive (focused) sequents is a *positive phase*: such a phase contains introduction rules for positive connectives.

An unfocused sequent of the form $\Sigma : \Gamma \Uparrow \cdot \vdash \cdot \Uparrow E$ is also called a *border sequent*. A *bipole* is a derivation whose conclusion and premises are all border sequents: also, when reading the inference rules from the bottom to the top, the first inference rule is a decide rule (either D_l or D_r); the next rules are positive introduction rules; then there is a release rule (either R_l or R_r); followed by negative introduction rules and storage rules (either S_l or S_r). In other words, a bipole is the joining of a single positive phase and possibly several negative phases.

Fig. 2.1 shows the $\mathcal{F}_{\supset}$ proof-system. It's rules are divided in three groups:

- Asynchronous rules used during the negative phases;
- Synchronous rules used during the focusing phases;
- Structural rules which mediate between the phases: decide rules start focusing and release rules stop it.

In the asynchronous phase we chose to consume the list Θ in the left-most order until it is empty and then Δ_1 (this choice does not impact the provability of the formula). In this system the phases are maximal, that is to say it is not possible to focus on the left if the right formula is not positive (or a negative atom) and the foci can't be released until it turns positive. Therefore Γ will always contain only negative formulas or positive atoms.

1.2 Polarity and proof search

One of the noteworthy features of focusing based on polarity is that it is possible to change the proof search strategy with different choices of atom's polarities. We will illustrate this with the standard forward/backward-chaining example.

Let us consider the following start sequent:

$$\Gamma \Uparrow \cdot \vdash \cdot \Uparrow c \qquad \text{where } \Gamma = \{a, a \supset b, b \supset c\}$$

Forward chaining On the one hand, choosing positive polarities for both a, b and c forces to adopt the forward chaining style (where $a \supset b$ is processed before $b \supset c$). Indeed starting by focusing over $b \supset c$ and then applying $\supset_l$ would require the use of the rule I_l with atom c which would fail because c is positive. This leads to the proof derivation shown in Fig. 2.2. Note that, given the sequent $\Gamma \Downarrow \cdot \vdash a \Downarrow \cdot$, it is not allowed to use a release on the right rule (in order to store, decide on the left and finish with initial on the left) because a is positive.

$$\dfrac{\dfrac{}{a \in \Gamma \Downarrow \cdot \vdash a \Downarrow \cdot}\,\mathrm{I}_r \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, b \Downarrow \cdot \vdash b \Downarrow \cdot}\,\mathrm{I}_r \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, b, c \Downarrow \cdot \vdash c \Downarrow \cdot}\,\mathrm{I}_r}{\Gamma, b, c \Uparrow \cdot \vdash \cdot \Uparrow c}\,\mathrm{D}_r}{\Gamma, b \Downarrow c \vdash \cdot \Downarrow c}\,\mathrm{R}_l + \mathrm{S}_l}{\Gamma, b \Downarrow b \supset c \vdash \cdot \Downarrow c}\,\supset_l}{\Gamma, b \Uparrow \cdot \vdash \cdot \Uparrow c}\,\mathrm{D}_l}{\Gamma \Downarrow b \vdash \cdot \Downarrow c}\,\mathrm{R}_l + \mathrm{S}_l}{\Gamma \Downarrow a \supset b \vdash \cdot \Downarrow c}\,\supset_l}{\Gamma \Uparrow \cdot \vdash \cdot \Uparrow c}\,\mathrm{D}_l$$

Figure 2.2: Forward chaining proof

Backward chaining On the other hand, and for a similar reason to the previous case, with negative atoms the proof becomes:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \Downarrow a \vdash \cdot \Downarrow a}\,\mathrm{I}_l}{\Gamma \Uparrow \cdot \vdash \cdot \Uparrow a}\,\mathrm{D}_l}{\Gamma \Downarrow \cdot \vdash a \Downarrow \cdot}\,\mathrm{R}_r + \mathrm{S}_r \qquad \dfrac{}{\Gamma \Downarrow b \vdash \cdot \Downarrow b}\,\mathrm{I}_l}{\Gamma \Downarrow a \supset b \vdash \cdot \Downarrow b}\,\supset_l}{\Gamma \Uparrow \cdot \vdash \cdot \Uparrow b}\,\mathrm{D}_l}{\Gamma \Downarrow \cdot \vdash b \Downarrow \cdot}\,\mathrm{R}_r + \mathrm{S}_r \qquad \dfrac{}{\Gamma \Downarrow c \vdash \cdot \Downarrow c}\,\mathrm{I}_l}{\Gamma \Downarrow b \supset c \vdash \cdot \Downarrow c}\,\supset_l}{\Gamma \Uparrow \cdot \vdash \cdot \Uparrow c}\,\mathrm{D}_l$$

In fact these two strategies are the ones implemented in the focused sequent calculi LJQ [DL06] (Dyckhoff & Lengrand) and LJT (Herbelin [Her95]) which turned out to be fragments of LJF[1]. However, in the scope of this work, this is mostly a matter of folklore. Our main application to arithmetic will not show this kind of proof behavior because of the systematic employ of fixed-points definitions instead of atomic formulas. We presented it for pedagogical purposes, as it can be helpful to give a working understanding of how focusing works.

2 The $\mathcal{F}$ calculus for Heyting arithmetic

In this section we give a focused version of the $\mathcal{H}$ system. To do that we start by discussing the polarities of the connectives we used in $\mathcal{H}$.

2.1 Polarities of connectives

Propositional intuitionistic logic As we have seen in Chapter 1 propositional intuitionistic logic formulas are constructed using the logical connectives $\wedge$, $\vee$, and $\supset$, the logical constants t and f, and atomic formulas. The focused system in Fig. 2.3

1. It is interesting to note that the λ-calculus associated with these systems (seeing these as typing systems) will respectively adopt the call-by-name and call-by-value reduction strategies.

STRUCTURAL RULES

$$\frac{\Gamma, \mathrm{N} \Downarrow \mathrm{N} \vdash \cdot \Downarrow \mathrm{E}}{\Gamma, \mathrm{N} \Uparrow \cdot \vdash \cdot \Uparrow \mathrm{E}}\ \mathrm{D}_l \qquad \frac{\mathrm{C}, \Gamma \Uparrow \Theta \vdash \Delta_1 \Uparrow \Delta_2}{\Gamma \Uparrow \mathrm{C}, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ \mathrm{S}_l \qquad \frac{\Gamma \Uparrow \mathrm{P} \vdash \cdot \Uparrow \mathrm{E}}{\Gamma \Downarrow \mathrm{P} \vdash \cdot \Downarrow \mathrm{E}}\ \mathrm{R}_l$$

$$\frac{\Gamma \Downarrow \cdot \vdash \mathrm{P} \Downarrow \cdot}{\Gamma \Uparrow \cdot \vdash \cdot \Uparrow \mathrm{P}}\ \mathrm{D}_r \qquad \frac{\Gamma \Uparrow \cdot \vdash \cdot \Uparrow \mathrm{E}}{\Gamma \Uparrow \cdot \vdash \mathrm{E} \Uparrow \cdot}\ \mathrm{S}_r \qquad \frac{\Gamma \Uparrow \cdot \vdash \mathrm{N} \Uparrow \cdot}{\Gamma \Downarrow \cdot \vdash \mathrm{N} \Downarrow \cdot}\ \mathrm{R}_r$$

NEGATIVE PHASE INTRODUCTION RULES

$$\frac{\Gamma \Uparrow \Theta \vdash \Delta_1 \Uparrow \Delta_2}{\Gamma \Uparrow t^+, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ t^+ \qquad \frac{}{\Gamma \Uparrow \cdot \vdash t^- \Uparrow \cdot}\ t^- \qquad \frac{}{\Gamma \Uparrow f^+, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ f^+$$

$$\frac{\Gamma \Uparrow \mathrm{A}, \Theta \vdash \Delta_1 \Uparrow \Delta_2 \qquad \Gamma \Uparrow \mathrm{B}, \Theta \vdash \Delta_1 \Uparrow \Delta_2}{\Gamma \Uparrow \mathrm{A} \vee \mathrm{B}, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ \vee_l \qquad \frac{\Gamma \Uparrow \mathrm{A} \vdash \mathrm{B} \Uparrow \cdot}{\Gamma \Uparrow \cdot \vdash \mathrm{A} \supset \mathrm{B} \Uparrow \cdot}\ \supset_r$$

$$\frac{\Gamma \Uparrow \mathrm{A}, \mathrm{B}, \Theta \vdash \Delta_1 \Uparrow \Delta_2}{\Gamma \Uparrow \mathrm{A} \wedge^+ \mathrm{B}, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ \wedge_l^+ \qquad \frac{\Gamma \Uparrow \cdot \vdash \mathrm{A} \Uparrow \cdot \qquad \Gamma \Uparrow \cdot \vdash \mathrm{B} \Uparrow \cdot}{\Gamma \Uparrow \cdot \vdash \mathrm{A} \wedge^- \mathrm{B} \Uparrow \cdot}\ \wedge_r^-$$

POSITIVE PHASE INTRODUCTION RULES

$$\frac{\Gamma \Downarrow \mathrm{B}_i \vdash \cdot \Downarrow \mathrm{E}}{\Gamma \Downarrow \mathrm{B}_1 \wedge^- \mathrm{B}_2 \vdash \cdot \Downarrow \mathrm{E}}\ \wedge_{l,i\in\{1,2\}}^- \qquad \frac{}{\Gamma \Downarrow \cdot \vdash t^+ \Downarrow \cdot}\ t^+ \qquad \frac{\Gamma \Downarrow \cdot \vdash \mathrm{B}_i \Downarrow \cdot}{\Gamma \Downarrow \cdot \vdash \mathrm{B}_1 \vee \mathrm{B}_2 \Downarrow \cdot}\ \vee_{r,i\in\{1,2\}}$$

$$\frac{\Gamma \Downarrow \cdot \vdash \mathrm{B}_1 \Downarrow \cdot \qquad \Gamma \Downarrow \cdot \vdash \mathrm{B}_2 \Downarrow \cdot}{\Gamma \Downarrow \cdot \vdash \mathrm{B}_1 \wedge^+ \mathrm{B}_2 \Downarrow \cdot}\ \wedge_r^+ \qquad \frac{\Gamma \Downarrow \cdot \vdash \mathrm{A} \Downarrow \cdot \qquad \Gamma \Downarrow \mathrm{B} \vdash \cdot \Downarrow \mathrm{E}}{\Gamma \Downarrow \mathrm{A} \supset \mathrm{B} \vdash \cdot \Downarrow \mathrm{E}}\ \supset_l$$

Figure 2.3: The propositional fragment of cut-free *LJF*

contains not formulas but *polarized* formulas. Such polarized formulas differ from unpolarized formulas in two ways. First, in the same way as in $\mathcal{F}_{\supset}$, every atomic formula is assigned either a positive or negative polarity in an arbitrary but fixed fashion. Thus, one can fix the polarity of atomic formulas (propositional variables) such that they are all positive or all negative or some mixture of positive and negative. Second, the conjunction is replaced with two conjunctions $\wedge^+$ and $\wedge^-$ and the unit of conjunction t with t^+ and t^-. The disjunction connective only has a positive occurrence, $\vee$, because of the intuitionistic setting (but the $\supset$ connective can be seen to be a form of $\vee^-$ that needs to be "controlled"). A polarized formula is *positive* if it is a positive atomic formula or its top-level logical connective is either t^+, f, $\wedge^+$, or $\vee$. A polarized formula is *negative* if it is a negative atomic formula or its top-level logical connective is either t^-, $\wedge^-$, or $\supset$.

Fig. 2.3 contains the structural and introduction rules for the propositional fragment of the LJF focused proof system [LM09]. An invariant in the construction of LJF proofs is that Γ will be a multiset that can contain only negative formulas and positive atoms. Every sequent in LJF denotes a standard sequent in LJ: simply replace $\Uparrow$ and $\Downarrow$ with commas.

Figure 2.3 contains neither the initial rule nor the cut rule. As we stated before, even if the cut rule and the cut-elimination theorem play important roles in justifying the design of focused proof systems, they play a minor role in this thesis (as cut-elimination is not part of our notion of computation). The initial rule will be important but not globally: we will introduce it later when we need (variants of) it.

Quantification and term equality The following rules show the focused versions of the rules for quantification and term equality seen in Chapter 1.

TERM EQUALITY

$$\frac{\Sigma\theta : \Gamma\theta \Uparrow \Theta\theta \vdash \Delta_1\theta \Uparrow \Delta_2\theta \qquad \theta \text{ mgu of } s \text{ and } t}{\Sigma : \Gamma \Uparrow s = t, \Theta \vdash \Delta_1 \Uparrow \Delta_2} =_{l\theta}$$

$$\frac{t \text{ and } s \text{ not unifiable}}{\Sigma : \Gamma \Uparrow s = t, \Theta \vdash \Delta_1 \Uparrow \Delta_2} =_l \qquad \frac{}{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash t = t \Downarrow \cdot} =_r$$

QUANTIFICATION

$$\frac{y : \tau, \Sigma : \Gamma \Uparrow \mathrm{B}[y/x], \Theta \vdash \Delta_1 \Uparrow \Delta_2}{\Sigma : \Gamma \Uparrow \exists x_\tau.\, \mathrm{B}, \Theta \vdash \Delta_1 \Uparrow \Delta_2} \exists_l \qquad \frac{\Sigma \vdash t : \tau \qquad \Sigma : \Gamma \Downarrow \cdot; \Omega \vdash \mathrm{B}[t/x] \Downarrow \cdot}{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash \exists x_\tau.\, \mathrm{B} \Downarrow \cdot} \exists_r$$

$$\frac{\Sigma \vdash t : \tau \qquad \Sigma : \Gamma \Downarrow \mathrm{B}[t/x]; \Omega \vdash \cdot \Downarrow \mathrm{E}}{\Sigma : \Gamma \Downarrow \forall x_\tau.\, \mathrm{B}; \Omega \vdash \cdot \Downarrow \mathrm{E}} \forall_l \qquad \frac{y : \tau, \Sigma : \Gamma \Uparrow \cdot \vdash \mathrm{B}[y/x] \Uparrow \cdot}{\Sigma : \Gamma \Uparrow \cdot \vdash \forall x_\tau.\, \mathrm{B} \Uparrow \cdot} \forall_r$$

Formulas with a top-level $\forall$ have negative polarity while formulas with a top-level $\exists$ or equality have positive polarity. This polarity assignment follows the intuitive thought: to prove a goal starting with forall requires only the introduction of a new variable, it's an invertible operation while the proof of a goal starting with a there exist needs a witness term to be chosen. This choice can be hard and clearly belong to the positive phase.

Fixed-point Momentarily ignoring the † and ‡ provisos, Fig. 2.4 shows the natural rules for unfolding μ-expressions. Here, we have assigned to such expressions the

FIXED POINT RULES

$$\dagger\ \frac{\Sigma : \Gamma \Uparrow \mathrm{B}(\mu\mathrm{B})\ \bar{t} \vdash \Delta_1 \Uparrow \Delta_2}{\Sigma : \Gamma \Uparrow \mu\mathrm{B}\ \bar{t} \vdash \Delta_1 \Uparrow \Delta_2}\ \mu_l \qquad \frac{\Sigma : \Gamma \Downarrow \cdot ; \Omega \vdash \mathrm{B}(\mu\mathrm{B})\ \bar{t} \Downarrow \cdot}{\Sigma : \Gamma \Downarrow \cdot ; \Omega \vdash \mu\mathrm{B}\ \bar{t} \Downarrow \cdot}\ \mu_r$$

MODIFIED VERSIONS OF THE DECIDE AND RELEASE RULES

$$\ddagger\ \frac{\Sigma : \Gamma, \mathrm{N} \Downarrow \mathrm{N}; \Omega \vdash \cdot \Downarrow \mathrm{E}}{\Sigma : \Gamma, \mathrm{N} \Uparrow \Omega \vdash \cdot \Uparrow \mathrm{E}}\ \mathrm{D}_l \qquad \ddagger\ \frac{\Sigma : \Gamma \Downarrow \cdot ; \Omega \vdash \mathrm{P} \Downarrow \cdot}{\Sigma : \Gamma \Uparrow \Omega \vdash \cdot \Uparrow \mathrm{P}}\ \mathrm{D}_r$$

$$\frac{\Sigma : \Gamma \Uparrow \mathrm{P}, \Omega \vdash \cdot \Uparrow \mathrm{E}}{\Sigma : \Gamma \Downarrow \mathrm{P}; \Omega \vdash \cdot \Downarrow \mathrm{E}}\ \mathrm{R}_l \qquad \frac{\Sigma : \Gamma \Uparrow \Omega \vdash \mathrm{N} \Uparrow \cdot}{\Sigma : \Gamma \Downarrow \cdot ; \Omega \vdash \mathrm{N} \Downarrow \cdot}\ \mathrm{R}_r$$

INITIAL RULE

$$\frac{\mathrm{P} \in \Omega}{\Sigma : \Gamma \Downarrow \cdot ; \Omega \vdash \mathrm{P} \Downarrow \cdot}\ \mathrm{I}_r$$

The proviso † requires that $\mu\mathrm{B}.\bar{t}$ does not satisfy $\mathcal{S}$. The proviso ‡ requires Ω to be a multiset of μ-expressions that satisfy $\mathcal{S}$.

Figure 2.4: Rules governing fixed point unfolding, suspensions, and initial sequents in $\mathcal{F}$

positive polarity. Since the left-introduction and right-introduction rules for μ-expressions are the same (i.e., they are unfolded), they could have been polarized negatively as well.

Focused sequent calculus proof systems were originally developed for quantificational logic—as opposed to arithmetic—and in that setting the bottom-up construction of the negative phase causes sequents to get strictly smaller (counting, for example, the number of occurrences of logical connectives). As a result, it was possible to design focused proof systems in which decide rules were not applied until *all* invertible rules were applied. We shall say that such proofs systems are *strongly* focused proof systems: examples of such systems can be found in [And92; LM09].

As is obvious from the first two inference rules in Figure 2.4, the size of the formulas in the negative phase can increase when μ-expressions are unfolded. Thus, a more flexible approach to building negative phases should be considered. Some focused proof systems have been designed in which a decide rule can be applied without consideration of whether all or some of the invertible rules have been applied. Following [SP11], such proof systems are called *weakly* focused proof systems: an early example of such a proof system is Girard's LC [Gir91]. Since we wish to use the negative phase to do functional style, determinate computation, a weakly focused system—with its possibility to stop in many different configurations—cannot provide the foundations that we need.

2.2 Suspensions

Instead of strongly and weakly focused proof systems, we modify the notion of strongly focusing by allowing certain explicitly described μ-expressions appearing

in the negative phase to be *suspended*. In that case, one can switch from a negative phase to a positive phase (using a decide rule) when the only remaining formulas in the negative phase are suspendable. In that case, those formulas are "put aside" (in a zone) during the processing of the positive phase and are reinstated when the positive phase switches to the negative phase (using a release rule). In more detail, let $\mathcal{S}$ denote a *suspension* predicate: this predicate is defined only on μ-expressions and if $\mathcal{S}$ holds for $(\mu \mathrm{B} \bar{t})$ then we say that this expression is suspended. The μ_l rule in Figure 2.4 has the proviso that $\mathcal{S}$ does not hold for the μ-expression that is the subject of that inference rule. In order to accommodate suspended formulas, $\Downarrow$-sequents need to contain a new multiset zone, denoted by the syntactic variable Ω: in particular, they now have the structure $\Sigma : \Gamma \Downarrow \Theta; \Omega \vdash \Delta_1 \Downarrow \Delta_2$. All positive introduction rules ignore this new zone: for example, the left-introduction of $\wedge^-$ will now be written as

$$\frac{\Sigma : \Gamma \Downarrow \mathrm{B}_i; \Omega \vdash \cdot \Downarrow \mathrm{E}}{\Sigma : \Gamma \Downarrow \mathrm{B}_1 \wedge^- \mathrm{B}_2; \Omega \vdash \cdot \Downarrow \mathrm{E}} \wedge^-_{l, i \in \{1,2\}}$$

Moreover, *border sequent* are now of the form $\Sigma : \Gamma \Uparrow \Omega \vdash \cdot \Uparrow \mathrm{E}$ where all μ-expressions of the multiset Ω satisfy $\mathcal{S}$.

The suspension property $\mathcal{S}$ is defined at the meta-level and, as a result, can make use of syntactic details about μ-expressions. For example, this property could be defined to hold for a μ-expression that contains more than, say, 100 symbols or when the first term in the list $\bar{t}$ is an eigenvariable. However, in order to guarantee that the negative phase is determinate, we need to require the following property:

> $(*)$ For all μ-expressions $(\mu \mathrm{B} \bar{t})$ and for all substitutions θ defined on the eigenvariables free in that μ-expression, if $\mathcal{S}$ holds for $(\mu \mathrm{B} \bar{t})\theta$ then $\mathcal{S}$ holds for $(\mu \mathrm{B} \bar{t})$.

That is, if an instance of a μ-expression satisfies $\mathcal{S}$ after a substitution is applied, it must satisfy $\mathcal{S}$ before it was applied. This condition rules out the possible suspension condition "holds if it contains 100 symbols" but it allows the condition "holds if the first term in $\bar{t}$ is an eigenvariable".

In addition, suspension properties should not, in general, be invariant under substitution since otherwise a suspended formula could remain suspended during the construction of a proof: it can only be used within the initial rule.

Example 2.1. Consider the suspension predicate that is true of μ-expressions $\mu \mathrm{B}\, t_1 \ldots t_n$ if and only if $n \geq 2$ and t_1 and t_2 are the same variable. Clearly, property $(*)$ does not hold and the construction of the negative phase can be non-confluent. For example, let A be $\mu\lambda p \lambda x \lambda y.x = a$ (where a is a constant) and consider the sequent $\Gamma \Uparrow u = v, \mathrm{A}\ u\ v \vdash \cdot \Uparrow (\mathrm{E}\ u)$. Since $\mathrm{A}uv$ is a μ-expression for which $\mathcal{S}$ does not hold, unfolding is applicable and yields the sequent $\Gamma \Uparrow u = v, u = a \vdash \cdot \Uparrow (\mathrm{E}\ u)$ which then leads to the border sequent $\Gamma \Uparrow \cdot \vdash \cdot \Uparrow (\mathrm{E}\ a)$. However, the first step in the negative phase of the original sequent could have been the equality introduction, which yields $\Gamma \Uparrow \mathrm{A}\ u\ u \vdash \cdot \Uparrow (\mathrm{E}\ u)$ and this must mark the end of the negative phase since $\mathrm{A}\ u\ u$ is a suspended formula.

Fortunately, this non-confluent behavior is ruled out by the $(*)$ property above. To see this, let $\mathcal{C}$ be a $\Uparrow$-sequent and let Ξ be a negative phase that has $\mathcal{C}$ as its end sequent and with premises that are border sequents. If we collect the premises

of Ξ into a set, say, $\mathcal{P}$, then we call $\mathcal{P}$ an *invertible decomposition* of $\mathcal{C}$. It is easy to show, via permutations of inference rules, that if $\mathcal{C}$ has $\mathcal{P}_1$ and $\mathcal{P}_2$ as invertible decompositions, then $\mathcal{P}_1 = \mathcal{P}_2$. The $(*)$ condition enables the permutation of the equality left-introduction rule and the μ_l rule.

More examples of suspension predicates will be found in Section 4.2.

Theorem 2.2. *Under a suspension predicate $\mathcal{S}$ satisfying the $(*)$ property, the decomposition of a given asynchronous sequent into border sequents is independent of the order in which the asynchronous rules are applied.*

Without suspensions, this standard property of focused proof systems stems from the fact that all asynchronous rules are invertible and thus can permute over each over. Thus, any sequence of inference rules that comprises the asynchronous phase (starting with the release rule and ending with a border sequent) can be permuted to any other sequent. Thus, the border sequent is the same no matter how these inference rules are applied.

However, when suspensions are involved, border sequents can have formulas in their asynchronous context that all satisfy the suspension predicate. Let's assume that there are two formulas, B and C that are in that asynchronous zone. If they are both introducible (that is if corresponding rules can be applied), we should be able to introduce B or C in any order. Let's suppose we cannot:

Imagine we select B and apply an introduction rule. Then we will move to a sequent containing B' and $C\theta$ (since some eigenvariable might get bound). Now, let's suppose that we cannot do the introduction on $C\theta$ because it is suspended. Then the condition $(*)$ ensures that C *was also suspended*. Thus, if a permutation is not possible after applying the inference rule, it was not possible before applying the inference rule. That is, if we could have introduced either B or C, then picking one means that the other one can still be done afterwards.

Definition 2.3. [Purely positive formula] A polarized formula in which all occurrences of logical connectives are polarized positively is called a *purely positive* formula. A μ-expression that is also purely positive will also be called a purely positive fixed point expression.

Horn clauses[2] (such as in Prolog) can provide immediate examples of purely positive fixed points as illustrated in Example 3.5 (using the positive version of conjunction). Let B be a purely positive formula. If $\Sigma : \Gamma \Uparrow \cdot \vdash B \Uparrow \cdot$ is provable then all proofs of that sequent are built of only positive right-introduction rules for t^+, $\wedge^+$, $\vee$, $\exists$, μ (unfolding) and equality. Similarly, if $\Sigma : \Gamma \Downarrow B \vdash \cdot \Downarrow \cdot$ is provable then all proofs of that sequent are built of only negative left-introduction rules for t^+, $\wedge^+$, $\vee$, $\exists$, μ (unfolding), and equality. Thus, focused proofs of B and $B \supset f$ are achieved by using only one phase. In particular, such proofs do not contain structural rules nor the initial rule. As a result, synthetic inference rules that abstract away the construction of phases are not decidable since they can encode arbitrary Horn clause specifications.

2.3 The complete $\mathcal{F}$ calculus

Following the previous discussions about the polarity of connectives, the complete grammar of formulas of $\mathcal{F}_\supset$ is the following:

2. Horn clauses are clauses, that is disjunctions of literals, with at most one unnegated literal.

Identity and structural rules

$$\ddagger\ \frac{\Sigma : \Gamma, N \Downarrow N; \Omega \vdash \cdot \Downarrow E}{\Sigma : \Gamma, N \Uparrow \Omega \vdash \cdot \Uparrow E}\ D_l \qquad \frac{\Sigma : C, \Gamma \Uparrow \Theta \vdash \Delta_1 \Uparrow \Delta_2}{\Sigma : \Gamma \Uparrow C, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ S_l \qquad \frac{\Sigma : \Gamma \Uparrow P, \Omega \vdash \cdot \Uparrow E}{\Sigma : \Gamma \Downarrow P; \Omega \vdash \cdot \Downarrow E}\ R_l$$

$$\ddagger\ \frac{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash P \Downarrow \cdot}{\Sigma : \Gamma \Uparrow \Omega \vdash \cdot \Uparrow P}\ D_r \qquad \frac{\Sigma : \Gamma \Uparrow \cdot \vdash \cdot \Uparrow E}{\Sigma : \Gamma \Uparrow \cdot \vdash E \Uparrow \cdot}\ S_r \qquad \frac{\Sigma : \Gamma \Uparrow \Omega \vdash N \Uparrow \cdot}{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash N \Downarrow \cdot}\ R_r$$

$$\frac{P \in \Omega}{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash P \Downarrow \cdot}\ I_r$$

Negative phase introduction rules

$$\frac{\Sigma : \Gamma \Uparrow \Theta \vdash \Delta_1 \Uparrow \Delta_2}{\Sigma : \Gamma \Uparrow t^+, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ t^+ \qquad \frac{}{\Sigma : \Gamma \Uparrow \cdot \vdash t^- \Uparrow \cdot}\ t^-$$

$$\frac{}{\Sigma : \Gamma \Uparrow f^+, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ f^+ \qquad \frac{\Sigma : \Gamma \Uparrow A \vdash B \Uparrow \cdot}{\Sigma : \Gamma \Uparrow \cdot \vdash A \supset B \Uparrow \cdot}\ \supset_r$$

$$\frac{\Sigma : \Gamma \Uparrow A, B, \Theta \vdash \Delta_1 \Uparrow \Delta_2}{\Sigma : \Gamma \Uparrow A \wedge^+ B, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ \wedge_l^+ \qquad \frac{\Sigma : \Gamma \Uparrow \cdot \vdash A \Uparrow \cdot \quad \Sigma : \Gamma \Uparrow \cdot \vdash B \Uparrow \cdot}{\Sigma : \Gamma \Uparrow \cdot \vdash A \wedge^- B \Uparrow \cdot}\ \wedge_r^-$$

$$\frac{\Sigma : \Gamma \Uparrow A, \Theta \vdash \Delta_1 \Uparrow \Delta_2 \quad \Sigma : \Gamma \Uparrow B, \Theta \vdash \Delta_1 \Uparrow \Delta_2}{\Sigma : \Gamma \Uparrow A \vee B, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ \vee_l \qquad \dagger\ \frac{\Sigma : \Gamma \Uparrow B(\mu B)\ \bar{t} \vdash \Delta_1 \Uparrow \Delta_2}{\Sigma : \Gamma \Uparrow \mu B\ \bar{t} \vdash \Delta_1 \Uparrow \Delta_2}\ \mu_l$$

$$\frac{y : \tau, \Sigma : \Gamma \Uparrow B[y/x], \Theta \vdash \Delta_1 \Uparrow \Delta_2}{\Sigma : \Gamma \Uparrow \exists x_\tau.\ B, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ \exists_l \qquad \frac{y : \tau, \Sigma : \Gamma \Uparrow \cdot \vdash B[y/x] \Uparrow \cdot}{\Sigma : \Gamma \Uparrow \cdot \vdash \forall x_\tau.\ B \Uparrow \cdot}\ \forall_r$$

$$\frac{\Sigma\theta : \Gamma\theta \Uparrow \Theta\theta \vdash \Delta_1\theta \Uparrow \Delta_2\theta \qquad \theta \text{ mgu of } s \text{ and } t}{\Sigma : \Gamma \Uparrow s = t, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ =_{l\theta} \qquad \frac{t \text{ and } s \text{ not unifiable}}{\Sigma : \Gamma \Uparrow s = t, \Theta \vdash \Delta_1 \Uparrow \Delta_2}\ =_l$$

Positive phase introduction rules

$$\frac{}{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash t = t \Downarrow \cdot}\ =_r \qquad \frac{}{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash t^+ \Downarrow \cdot}\ t^+ \qquad \frac{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash B_i \Downarrow \cdot}{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash B_1 \vee B_2 \Downarrow \cdot}\ \vee_{r, i\in\{1,2\}}$$

$$\frac{\Sigma : \Gamma \Downarrow B_i; \Omega \vdash \cdot \Downarrow E}{\Sigma : \Gamma \Downarrow B_1 \wedge^- B_2; \Omega \vdash \cdot \Downarrow E}\ \wedge_{l, i\in\{1,2\}}^- \qquad \frac{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash B_1 \Downarrow \cdot \quad \Sigma : \Gamma \Downarrow \cdot; \Omega \vdash B_2 \Downarrow \cdot}{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash B_1 \wedge^+ B_2 \Downarrow \cdot}\ \wedge_r^+$$

$$\frac{\Sigma \vdash t : \tau \quad \Sigma : \Gamma \Downarrow B[t/x]; \Omega \vdash \cdot \Downarrow E}{\Sigma : \Gamma \Downarrow \forall x_\tau.\ B; \Omega \vdash \cdot \Downarrow E}\ \forall_l \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma : \Gamma \Downarrow \cdot; \Omega \vdash B[t/x] \Downarrow \cdot}{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash \exists x_\tau.\ B \Downarrow \cdot}\ \exists_r$$

$$\frac{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash A \Downarrow \cdot \quad \Sigma : \Gamma \Downarrow B; \Omega \vdash \cdot \Downarrow E}{\Sigma : \Gamma \Downarrow A \supset B; \Omega \vdash \cdot \Downarrow E}\ \supset_l \qquad \frac{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash B(\mu B)\ \bar{t} \Downarrow \cdot}{\Sigma : \Gamma \Downarrow \cdot; \Omega \vdash \mu B\ \bar{t} \Downarrow \cdot}\ \mu_r$$

Figure 2.5: The $\mathcal{F}$ proof system. The proviso † requires that $\mu B\bar{t}$ does not satisfy $\mathcal{S}$. The proviso ‡ requires Ω to be a multiset of μ-expressions that satisfy $\mathcal{S}$.

- $A, B ::= P \mid N$
- $P, Q ::= P_a \mid t \mid f \mid A \wedge^+ B \mid A \vee B \mid \exists x.A \mid \mu(\lambda P.\, \lambda \bar{x}.\, B)\bar{t}$
- $N, M ::= N_a \mid A \supset B \mid A \wedge^- B \mid \forall x.A$

Fig. 2.5 shows the full $\mathcal{F}$ system.

3 Juggling with phases

3.1 Phases as abstractions

Focused proof systems make it possible to define new inference rules by abstracting away details used in the construction of phases. Such composite rules are sometimes called synthetic or macro rules. The positive phase allows a simple abstraction since there is exactly one formula under focus in a positive sequent. A positive phase can be seen as the (derived) inference rule with a conclusion that is a border sequent and with premises that are marked by release rules.

There are, however, at least two challenges to making abstractions of negative phases. First, the premises of a negative phase may repeat the same sequents many times since there can be many paths to compute the result of a function. We shall choose to denote as the collections of premises of the negative phase the *set* of border sequents (instead of as a *multiset*). Second, there are many ways to process the don't-care nondeterminism that is possible when applying invertible rules. We will abstract away from those differences by simply ignoring *how* a phase is constructed since all constructions yield the same border sequents.

The first challenge above stems from the same motivation used in confluent rewriting systems: once a path to a normal form is found, no other paths need to be considered since all other paths must yield the same normal form.

3.2 The polarity ambiguity of singleton sets

As we mentioned in the introduction, singleton sets can be used to help convert relations to functions: if the $(n+1)$-ary relation R describes a function from its first n arguments to its last argument then the expression $(\lambda y.\mathrm{R}(x_1, \ldots, x_n, y))$ denotes a singleton set (given fixed values for $x_1, \ldots, x_n$). The choice operators ϵ or ι can then be applied to this singleton set to extract that element, resulting in a proper function $\lambda x_1 \ldots \lambda x_n \iota(\lambda y.\mathrm{R}(x_1, \ldots, x_n, y))$.

Singleton sets play a role here as well. In fact, let P be a predicate of one argument so that it is provable that P is a singleton, namely,

$$(\exists x.\mathrm{P}(x)) \wedge (\forall x, y.\mathrm{P}(x) \supset \mathrm{P}(y) \supset x = y)$$

As a consequence, the formulas $\exists x.\mathrm{P}(x) \wedge \mathrm{Q}(x)$ and $\forall x.\mathrm{P}(x) \supset \mathrm{Q}(x)$ are equivalent. If we used the ι-operator, these formulas would also be equivalent to $\mathrm{Q}(\iota \mathrm{P})$.

Note that the sequent calculus treatments of $\exists x.\mathrm{P}(x) \wedge \mathrm{Q}(x)$ and $\forall x.\mathrm{P}(x) \supset \mathrm{Q}(x)$ are strikingly different. In particular, a proof of $\Sigma : \Gamma \Downarrow \cdot \vdash \exists x.\mathrm{P}(x) \wedge \mathrm{Q}(x) \Downarrow \cdot$ proceeds by guessing a term t and then attempting to prove $\Sigma : \Gamma \Downarrow \cdot \vdash \mathrm{P}(t) \Downarrow \cdot$ and $\Sigma : \Gamma \Downarrow \cdot \vdash \mathrm{Q}(t) \Downarrow \cdot$. Of course, since P denotes a singleton, there is at most one correct guess t and that guess is confirmed after it is inserted into the proof.

On the other hand, a proof of $\Sigma : \Gamma \Uparrow \cdot \vdash \forall x.\mathrm{P}(x) \supset \mathrm{Q}(x) \Uparrow \cdot$ can be seen as computing the value that satisfies P. Proof construction for that sequent leads to

proving $\Sigma : \Gamma \Uparrow \mathrm{P}(y) \vdash \mathrm{Q}(y) \Uparrow \cdot$. As mentioned before, this phase will move to completion by repeatedly unfolding fixed points and if the phase completes, the eigenvariable y will be instantiated to be the unique term t. Thus, the premises of this completed phase will have the shape $\Sigma : \Gamma \Uparrow \cdot \vdash \cdot \Uparrow \mathrm{Q}(t)$ (assuming for the sake of argument that $\mathrm{Q}(t)$ is a positive formula).

Example 3.1. Using the definitions in Example 3.5, consider the construction of a negative phase of the form $x, \Sigma : \Gamma \Uparrow \mathtt{plus}\ 2\ 3\ x \vdash \cdot \Uparrow (\mathrm{Q}\ x)$. Since plus is a μ-expression, this sequent is proved by a μ_l inference rule (assuming that $\mathcal{S}$ is false for all μ-expressions, i.e., nothing should be suspended). Unfolding yields an expression with a top-level disjunction, namely, $x, \Sigma : \Gamma \Uparrow ((2 = 0 \wedge^+ 3 = x) \vee \exists n' \exists x' (2 = s\ n' \wedge^+ x = s\ x' \wedge^+ \mathtt{plus}\ n'\ 3\ x')) \vdash \cdot \Uparrow (\mathrm{Q}\ x)$. Following the left-introduction for that disjunction, we are left with proving two sequents: the left premise, $x, \Sigma : \Gamma \Uparrow 2 = 0 \wedge^+ 3 = x \vdash \cdot \Uparrow (\mathrm{Q}\ x)$ is proved immediately since $2 = 0$ is not unifiable. A proof of the second premise must proceed as follows

$$\dfrac{\dfrac{x', \Sigma : \Gamma \Uparrow \mathtt{plus}\ 1\ 3\ x' \vdash \cdot \Uparrow (\mathrm{Q}\ (s\ x'))}{x, n', x', \Sigma : \Gamma \Uparrow 2 = s\ n' \wedge^+ x = s\ x' \wedge^+ \mathtt{plus}\ n'\ 3\ x' \vdash \cdot \Uparrow (\mathrm{Q}\ x)}}{x, \Sigma : \Gamma \Uparrow \exists n' \exists x' (2 = s\ n' \wedge^+ x = s\ x' \wedge^+ \mathtt{plus}\ n'\ 3\ x') \vdash \cdot \Uparrow (\mathrm{Q}\ x)}$$

(Here, the double line between sequents denotes the application of possibly several inference rules.) After several more inference steps, the negative phase terminates with the border premise $\Sigma : \Gamma \Uparrow \cdot \vdash \cdot \Uparrow (\mathrm{Q}\ 5)$. By ignoring the internal structure of phases, we have just the synthetic inference rule

$$\dfrac{\Sigma : \Gamma \Uparrow \cdot \vdash \cdot \Uparrow (\mathrm{Q}\ 5)}{x, \Sigma : \Gamma \Uparrow \mathtt{plus}\ 2\ 3\ x \vdash \cdot \Uparrow (\mathrm{Q}\ x)}$$

Note that the actual specification of the relation plus is used to *compute* the addition as a function.

Example 3.2. Employing the suspension mechanism makes it possible for functional computation to be mixed with symbolic computation. For example, let multiplication be defined as the following fixed point expression.

$$\begin{aligned}\mathtt{times} = \mu\lambda \mathrm{P} \lambda n \lambda m \lambda p ((n = 0 \wedge^+ p = 0) \\ \vee \exists n' \exists p' (n = s\ n' \wedge^+ \mathrm{P}\ n'\ m\ p' \wedge^+ \mathtt{plus}\ p'\ m p))\end{aligned}$$

The theorem that states that $(0 \times (x + 1)) + y = y$ can be encoded and proved in this setting by taking two steps. First we translate this expression into the following sequent.

$$y, \Sigma : \Gamma \Uparrow \cdot \vdash \forall u.\ \mathtt{times}\ 0\ (s\ x)\ u \supset \forall v.\ \mathtt{plus}\ u\ y\ v \supset v = y \Uparrow \cdot$$

Here, we assume the (rather typical) suspension mechanism that classifies μ-expressions as suspendable if they are built from plus and times and their first argument is an eigenvariable. Thus, when this sequent is reduced to

$$u, v, y, \Sigma : \Gamma \Uparrow \mathtt{times}\ 0\ (s\ x)\ u,\ \mathtt{plus}\ u\ y\ v \vdash v = y \Uparrow \cdot$$

only the times-expression can be unfolded. After that unfolding, the eigenvariable u will be instantiated and the plus-expression can then also be unfolded.

> Finally, the negative phase ends with the border sequent $y, \Sigma : \Gamma \Uparrow \cdot \vdash \cdot \Uparrow y = y$ which is proved by a D_r rule followed by the right-introduction rule for equality.

3.3 An extension to equivalence classes

Equivalence relations play important roles in computation and reasoning. Occasionally, we have a relation that is not functional but all the possible outcomes are equivalent, for some specific equivalence relation. For example, if two lists are considered equivalent when they are permutations of each other, then the equivalence class of lists modulo that relation encodes multisets. Similarly, if two pairs of integers (x, y) and (w, z) (where y and z are not zero) are considered equivalent when $xz = wy$ then equivalence classes encode rational numbers.

The ambiguity of singletons can be lifted to computation with equivalence classes in the following sense. Let ρ be an equivalence relation. The familiar notation $[x]_\rho$ for the ρ-equivalence class containing x is just syntactic sugar for $\lambda y.x\rho y$. (We define logical equivalence in the usual way: $\mathrm{A} \equiv \mathrm{B}$ is an abbreviation for $(\mathrm{A} \supset \mathrm{B}) \wedge (\mathrm{B} \supset \mathrm{A})$.)

Assume that ρ is an equivalence relation and that the following holds if $\mathrm{Q} : i \to o$.

$$\forall x \forall y.\ x\ \rho\ y \supset [\mathrm{Q}(x) \equiv \mathrm{Q}(y)]$$

(Note that this theorem is immediate for all $\mathrm{Q} : i \to o$ when ρ is equality.) The following equivalence holds.

$$[\forall x.\ x \in [y]_\rho \supset \mathrm{Q}(x)] \equiv [\exists x.\ x \in [y]_\rho \wedge \mathrm{Q}(x)]$$

In a more informal mathematical notation, one might replace either the above existential or universal expression with $\mathrm{Q}([y]_\rho)$. While we shall not use this expression (it involves a typing error), it conveys the usual mathematical sense of this ambiguity: if we show that one member of an equivalence class satisfies such a property Q then all members of that equivalence class satisfy Q.

Obviously, we can generalize the notion of functional dependency to the following

$$\forall \bar{x}([\exists y.\mathrm{R}(\bar{x}, y)] \wedge \forall y \forall z[\mathrm{R}(\bar{x}, y) \supset \mathrm{R}(\bar{x}, z) \supset y\rho z]),$$

which states that the n-ary relation is a total function up to ρ. Thus, during the construction of a proof where one is asked to pick a term t that makes $\mathrm{R}(x_1, \ldots, x_n, t)$ true, one can instead compute just any term t' such that $\mathrm{R}(x_1, \ldots, x_n, t')$ (as long as the established property Q is ρ-invariant). In that setting, we can also extend the phase-abstraction mechanism to exclude border premises that differ up to ρ.

4 A practical use: automation in Abella

Currently, the Abella proof assistant has rather limited forms of automation. Our work on separating computation from deduction using focusing appears to be an interesting framework for exploring possible means of adding more automation to Abella.

To achieve this we will apply the method described in the previous sections and use focusing to compose the ordinary sequent rules that operate on single connectives at a time into compound rules that work on a collection of connectives, called *synthetic connectives*, that have similar properties. In the presence of inductive and coinductive definitions, such synthetic connectives can involve the unbounded unfolding

of fixed points, thereby incorporating arbitrary deterministic and nondeterministic computation within synthetic inference rules.

Thus, even if full focused proof search as a broad basis for the automation of Abella is an interesting project, in this section we only explore a limited application of focusing to recover *computation*, following the method described in the previous sections. We learned that focusing in an intuitionistic logic with fixed points (essentially, Heyting arithmetic) can be used to turn *relational* specifications into *functional* computations. We now present a concrete proposal for a slight and orthogonal extension of Abella that allows it to perform such computations without any change to its underlying logical basis.

We will first give a short introduction to the inner workings of Abella which is built around a two logics system. Then we describe two proposals that automate the proof process.

4.1 The $\mathcal{G}$ Logic and the Abella Implementation

The Abella system implements two logics. The *specification logic* is a simple fragment of intuitionistic logic that is rich enough to specify many λProlog logic programs. This aspect of Abella is not the major concern of this work and we ignore it here. Instead we concentrate on the *reasoning* logic of Abella, known as $\mathcal{G}$ [GMN11], which is an extension of intuitionistic first-order logic with: (1) higher-order λ-terms together with the equational theory induced by αβη-equivalence, (2) inductive and coinductive fixed point definitions, and (3) nominals, nominal abstraction, and generic (∇) quantification. We give a brief introduction to $\mathcal{G}$ using the concrete syntax of Abella [Bae+14].[3]

The *terms* in $\mathcal{G}$ are well-typed terms of Church's simple theory of types [Chu40], where a given *type signature* declares a collection of basic types and constants that are interpreted as constructors for these declared basic types. For instance, the following is a declaration of two basic types, `nat` and `bool`, that are both declared to be types using the `Kind` keyword, and their constructors are indicated with `Type` declarations.

```
Kind bool type.
Type tt, ff  bool.

Kind nat type.
Type z nat.
Type s nat -> nat.
```

Formulas of $\mathcal{G}$ are terms of type `prop`, built from the constructors `/\` (for conjunction), `\/` (for disjunction), and `->` (for implication), all of type `prop -> prop -> prop` and written as infix; `true` and `false` of type `prop` for the constants; and `forall`, `exists`, and `nabla` of type `(α -> prop) -> prop` (for every type α not containing `prop`). The term abstraction λx. `t` is written concretely as `x\ t`, and quantified formulas are written in a more natural style rather than using abstractions, i.e., as `(forall x, f)` instead of `forall (x\ f)`.

Atomic formulas can be created from predicates of target type `prop` that may be declared with a `Type` declaration. More interestingly, $\mathcal{G}$ also allows atomic formulas to be built using inductively or coinductively defined fixed points. For instance, the following inductive definition characterizes all terms of type `nat` built from `z` and `s`:

3. $\mathcal{G}$ logic will also be a crucial tool for the next chapter of this thesis.

```
Define nat : nat -> prop by
  nat z ;
  nat (s X) := nat X.
```

Such definitions consist of a list of clauses where each clause begins with a *head* and is optionally followed by a *body* separated by `:=`. (An omitted body is understood to stand for `true`.) The head is always atomic using the predicate being defined, but the body can be any arbitrary formula; moreover, the head and body can share *variables* that are universally quantified over the entire clause and written using capital letters. Thus, the way to read the second clause above is: *for every `X`, the atom `nat (s X)` holds if and only if `nat X` holds.*

Note that in $\mathcal{G}$ and Abella the only form of induction or coinduction is with such defined predicates. There is no induction principle for the types — indeed, there is no reasoning principle of any kind for the types. Types are just used to enforce syntactic categories. As a consequence, we cannot prove the formula `forall (X:nat), nat X`: when we want to prove a theorem by structural induction on natural numbers, we need to explicitly use the `nat` predicate as a corresponding assumption. To illustrate this, let us introduce the predicate `plus` that relates two numbers to their sum and proceeds by structural induction on its first argument.

```
Define plus : nat -> nat -> nat -> prop by
  plus z X X ;
  plus (s X) Y (s Z) := plus X Y Z.
```

Here is a simple theorem that would need to be proved by structural induction on the first argument.

```
Theorem plus_z2 : forall X, nat X -> plus X z X.
```

Such a theorem would be proved by means of the `induction` tactic. In this case, we would proceed by `induction on 1`, i.e., on the first antecedent of the chain of implications in the theorem. This would generate an *inductive hypothesis* `IH`:[4]

```
IH : forall X, (nat X)* -> plus X z X
```

This is apparently the same as the theorem itself, except the inductive argument is marked with a size restriction `*`. The meaning of `(nat X)*` is that it can be applied to any derivation of `(nat X)` that is strictly smaller than that of the `(nat X)` we started the induction on originally. That original derivation is itself indicated with `(nat X)@`, which is to say that the result of the `induction` tactic is to change the goal to the following after assuming the `IH`.

```
forall X, (nat X)@ -> plus X z X
```

This goal is proved by means of ordinary logical reasoning, together with the `case` tactic[5] that explores all the ways in which an inductively defined assumption may have been derived, i.e., it performs an *inversion* on its definition. This `case` step in turn changes the `@` annotation to a `*` to indicate that it has strictly reduced the size of the derivation; this reduction makes the `IH` applicable. More precisely, inverting `(nat X)@` produces two subgoals; in the first, `X` is instantiated to `z`, and in the other `X` is instantiated to `(s X1)` for a new variable `X1`, and we get the additional

4. The parentheses we use here for didactic reasons are omitted by Abella, i.e., Abella writes `nat X*` instead of `(nat X)*`.

5. Notice that several Abella tactics share common names with Coq tactics. This is intended even if they work in quite different ways.

assumption `(nat X1)*`. Abella also has a collection of lower level tactics such as `unfold`, `witness`, `split`, `apply`, etc. for ordinary logical reasoning.

4.2 Proposal: Computation and Suspension

Our first proposed extension of Abella is rather simple: the addition of a `compute` tactic that performs unfolding and subsequent asynchronous steps for assumptions involving predicates with a fully positive definition. Thus, for instance, if we have an assumption

```
H : plus (s z) (s z) X
```

then the invocation `compute H` would repeatedly *unfold* the definition of `plus` and handle the resulting subgoals eagerly if it can use purely asynchronous steps. In this particular case, the effect will be the removal of `H` entirely and the instantiation of `X` with `(s (s z))`. The `compute` tactic is allowed to produce multiple branches. For instance, in the following case:

```
H : plus X Y (s (s z))
```

the invocation `compute H` would produce three subgoals, one each for the three ways there are to decompose 2 into a sum of two natural numbers.

This kind of feature has long been recognized as an important need in Abella.[6] A very common form is encountered in meta-theoretic proofs involving memberships in contexts, which are represented as lists in Abella, where we have an assumption such as:

```
H : member X (E1 :: E2 :: Rest)
```

In this case we would like `compute H` to yield three subgoals (in addition to the assumption): the first with `X = E1`, the second with `X = E2`, and the last with `member X Rest`.

A more interesting scenario is when the `compute` tactic is used on a purely positive predicate that cannot be fully solved. For instance, given:

```
H : nat (s (s X))
```

where `X` is an eigenvariable, it can be asynchronously simplified to `(nat X)` by just using the second clause of the definition of `nat`. However, to go further we would need to consider the cases where `X = z` and the case for `X = s X1`, and we would be left with a further assumption `nat X1`. We can repeat this process now with `X1` and so on. This *eager* treatment of `nat` not only leads to non-terminating search (which will eventually be forcefully terminating because it reaches a depth bound), but may be unwarranted before we know anything else about `X`. In this case, it would be useful to *suspend* the eager unfolding of `nat`.

To account for this premature unfolding of definitions when the inductive structure is already a variable, we add a new kind of `Suspend` declaration that will make Abella stop the asynchronous phase prematurely as we described in Section 2.2. The following declaration declares that `(nat X)` should not be unfolded if `X` is a variable; we call this a *suspension condition*.

```
Suspend nat X on X.
```

6. See, for instance, https://github.com/abella-prover/abella/issues/35.

A suspension condition can list more than one argument: unfolding is suspended if any of the indicated arguments is a variable. For example:

```
Suspend plus X Y _ on X, Y.
```

Note this declaration means that `compute` would terminate early even on a situation such as:

```
H : plus (s z) Y Z
```

even though we could have finished the phase with `Z` instantiated with `(s Y)`, even though `Y` isn't ground. This is fine because we could have left out the `Y` from the suspension conditions. Also note that although the suspension condition mentions variables, the suspension declaration itself can be any arbitrary pattern. For instance:

```
Suspend plus (s X) _ _ on X.
```

suspends unfolding on `plus` before its first argument is a variable. The pattern can also have repetitions such as:

```
Suspend plus X X _ on X.
```

Finally, a given predicate can have multiple suspension declarations: unfolding is suspended if any suspension declaration matches or if the predicate has no suspension declarations at all. The following pair is equivalent to the first `Suspend` declaration above.

```
Suspend plus X _ _ on X.
Suspend plus _ Y _ on Y.
```

The `compute` tactic has been implemented in prototype form already in Abella —which in fact led to the discovery of the need for `Suspend`—but the full proposal is still being debated by the Abella community. One obvious extension would be to perform the asynchronous phase on *all* assumptions instead of a specific one. Another issue to consider is whether we should allow `compute` to operate on goals as well. What would be the equivalent notion of `Suspend` for goals? Another open question is if the `Suspend` declarations can be inferred from the form of the definition itself.

4.3 Proposal: Deterministic Computation using Singleton Predicates

A monadic predicate `p` that holds for exactly one argument is a *singleton*. As we have seen in Section 3, singletons are interesting from the perspective of focusing. The formula `forall x, p x -> Q x` and `exists x, p x /\ Q x` are equivalent if and only if `p` is a singleton. That is, the following is a theorem of higher-order logic:

```
(forall q, ((forall x, p x -> q x)
   <-> (exists x, p x /\ q x)))
   <-> singleton p
```

where `singleton` has the following definition:

```
Define singleton : (A -> prop) -> prop by
  singleton P :=
      (exists X, P X)
   /\ (forall X Y, P X -> P Y -> X = Y).
```

As a consequence, the formulas `forall x, p x -> Q x` and `exists x, p x /\ Q x` may be freely converted into each other in the course of proof search.

Now, since Abella does not allow predicate quantification, the definition and theorem above are not acceptable. The theorem is explicitly ruled out because Abella does not allow universal quantification over terms whose types contain `prop`. The definition is accepted with a *stratification warning*, because the higher-order parameter `P` is used in a negative position, to the left of `->`. Such definitions can be used in trivial ways to prove `false` and hence for consistency Abella refuses to certify developments using such definitions.

Both the proof theory of sequent calculus and the tactics of Abella require that to make progress on proving `exists x, p x /\ Q x`, we must first supply a witness term `t` such that `(p t)` is true, and then the goal can become `(Q t)`. If we know that `p` is a singleton, then this requirement is unfortunate since one might hope that we could use Abella to actually *compute* this witness term `t` by means of the `compute` tactic in the previous section. It is tempting to extend Abella with *logic variables* or *placeholder variables* such as `?X` so that we can change the query to `p ?X /\ Q ?X`, and then in the course of proving the first conjunct `p ?X` we would replace the variable with the witness term. Such variables have been a part of Isabelle and Agda from the very beginning and have also been introduced to Coq (see, for example, [ZS15]).

We propose a more lightweight treatment, admitting the definition `singleton` to Abella (and syntactically preventing its abuse such as applying `singleton` to itself). Then, the issue of computing the witness term `t` is no different from transforming the goal `exists x, p x /\ Q x` to `forall x, p x -> Q x`, introducing the variable and its hypothesis (using `intros`), and then using `compute` on that hypothesis. Thus, we switch from "guess `t` and check `(p t)`" to "compute the `t` for which `(p t)`."

As we have seen before, singleton predicates arise whenever a relation is actually a function. In particular, the fact that an n-ary predicate `R` actually specifies a function from its, say, first $n-1$ arguments to its nth argument can be captured by:

```
forall x1 x2 … xn-1,
  singleton (x\ R x1 x2 ⋯ xn-1 x)
```

Note that we could have η-contracted the argument to `singleton` above to just `(R x1 ⋯ xn-1)`. More generally, the relation `R` may be a singleton only under certain conditions on its "input" arguments, in which case we would add them as antecedents in an implication chain. For example, consider the `plus` relation from before; its third argument is always uniquely determined by its first two, assuming that they are natural numbers. Hence, we can prove the following theorem.

```
Theorem plus_funct: forall X Y, nat X -> nat Y ->
    singleton (plus X Y).
```

This is an ordinary Abella theorem that can be readily proved by `induction on 1`. As another illustration, consider the partial relation `pred` for predecessors that relates natural numbers greater than 0 to their predecessor.

```
Define pred : nat -> nat -> prop by
  pred (s X) X.
```

To show that it is a function, we have to supply the precondition that its first argument is a natural number greater than z, which we can do as follows.

```
Define nat_gt : nat -> nat -> prop by
  nat_gt (s X) z := nat X ;
  nat_gt (s X) (s Y) := nat_gt X Y.

Theorem pred_funct: forall X, nat_gt X z ->
  singleton (pred X).
```

To make use of `singleton` to convert between the two `exists` and `forall` forms, we add new tactic forms to `witness` and `apply`. When the goal has the form:

```
========================================
 exists X, P X /\ Q X
```

then the invocation `witness compute` first attempts to prove `singleton P` from the same context, and then continues with modified goals of the form:

```
H : P X
========================================
 Q X
```

and follows up with `compute H`. Dually, whenever we have a hypothesis of the form:

```
H : forall X, P X -> Q X
```

then an invocation `apply compute H` has the effect of first trying to prove `(singleton P)` and then continuing with the modified hypotheses

```
H1 : P X
H  : Q X
```

following up with `compute H1`.

In both cases, the proof of `(singleton P)` must be trivial: the way it will be implemented is that the proved lemmas such as `plus_funct` will be searched for a predicate that matches `P`, and if so the antecedents of that lemma will be attempted to be proved with simple proofs. An important consideration in these simple proofs is that assumptions on predicates such as `nat` or `nat_gt` are attempted eagerly first to reduce them to their simplest forms. This will be done with the `compute` tactic as defined in Section 4.2. Note that it is important to supply suitable `Suspend` declarations for such antecedents to prevent infinite loops in the implicitly invoked `compute` invocations.

The above could have been done with a weaker assumption than `singleton`; it would have sufficed for the predicate `p` to be non-empty, which is just the first conjunct in the definition of `singleton`. The real power of the `singleton` assumption comes from the fact that it makes the computations *deterministic*. This means that whenever we perform `compute` on a singleton predicate, we never need to consider any but a single possibility. In other words, *conjunctive* branches in the search space caused by unfolding the singleton predicate can be pruned eagerly. To illustrate this, suppose we had the following variant definition of `plus`:

```
Define plus : nat -> nat -> nat -> prop by
  plus z X X ;
  plus (s X) Y (s Z) := plus X Y Z ;
  plus X Y Z := plus Y X Z.
```

It is still a function from its first two arguments to its third one, but unfolding the definition of `plus` is not unitary: the third clause overlaps with the first two. The

compute tactic should be satisfied (and thus stop) with the first unfolding sequence it finds, and not get distracted computing variants that have different numbers of uses of the third clause.

4.4 Possible extensions

There are a few ways in which these proposals can be generalized further. First, as we have seen in Section 3.3 the notion of singleton can be relaxed to a notion of *singleton up to equivalence* as hinted by the discussion in Section 3.3. For instance, we can say:

```
Define singleton_upto :
  (A -> A -> prop) -> (A -> prop) -> prop by
  singleton_upto Eq P :=
      (exists X, P X)
   /\ (forall X Y, P X -> P Y -> Eq X Y).
```

As long as Eq is an equivalence relation, we get all of the benefits of the singleton definition, such as the free conversion of exists goals into forall goals. This more general definition can be very useful in meta-theoretic proofs that reason about contexts: ordinarily they are represented as lists, but two contexts-as-lists that are merely permutations are considered to represent the same context. It has been observed in [CLR16] that a majority of the effort in formalizing standard meta-theorems such as cut-elimination is due to the complications resulting from reasoning about lists up to permutations.

A second obvious extension has to do with data defined by higher-order type signatures, such as terms represented using λ-tree syntax (sometimes known as higher-order abstract syntax). Many common relations that are defined on such higher-order data can be seen as functions, but it takes a bit more care to use the singleton relations. In particular, with higher-order representations the "typing relation" such as nat are no longer a natural fit for the reasoning logic; in this case, it is usually simpler to write these relations using the specification logic, using the *two-level logic approach* [GMN12]. Recent extensions of Abella to handle the full hereditary Harrop specification language [Wan+13] have allowed the expression of arbitrary higher-order (and even dependently typed) relations in terms of the specification language (see, e.g.,[SC14] for the LF dependent type theory). In these cases, not only the antecedents but also the *argument* to the singleton relation may well be a specification-language sequent.

5 Conclusion and perspectives

We presented in Section 3 a treatment of functional computation based on relations. Principles in proof theory provided a means of organizing Gentzen-style introduction rules so that functional computations can be identified as one specific phase of computation (the negative phase). Since this view of computation is based on the construction of cut-free proofs, it is rather different from, say, the Curry-Howard correspondence.

While we have illustrated most of this mechanism using first-order term structures (such as Peano's numerals), the proof theory behind *LJF* (on which $\mathcal{F}$ is based) works at all finite types. As a result, this approach to functional computation is a possible

avenue to explore how functional programming might be extended to treat terms containing λ-bindings. Furthermore, the proof theory presented here is compatible with the proof theory for least and greatest fixed points that has been developed in a series of papers [MM00; GMN08; GMN11; TM12] and in the Abella theorem prover [Bae+14; 12; Gac08] that we described in Section 4.

The small extensions to Abella's tactics we proposed could enable it to perform deterministic computation without step-by-step guidance by the user. We leave the kernel and the core tactics of Abella untouched, but adding a new `compute` tactic that is designed to perform the asynchronous phase of focused proof search for inductively defined predicates whose definitions are fully positive. Together with this mechanism is a new declaration that allows eager unfolding of definitions to be suspended when it is premature to continue unfolding, for instance where the arguments involve variables in "input" positions. Finally, we propose to allow Abella to express specific lemmas that prove that a given relation on a given collection of inputs determines a singleton on its output, meaning that the output both exists and is uniquely determined. Such lemmas can be used to transform an existential goal to a universal goal and move from a *guess and check* to a *compute and use* paradigm. A crucial feature of this use of singletons is that it treats computations as deterministic functions. The community of Abella users has still to agree on these basic extensions and to implement them.

Chapter 3

A functional programming language using λ-tree syntax

In Chapter 2 we described a way to identify computational behavior in proofs and to guide proof-search in using as much computation as possible with the focusing discipline. In the remainder of this thesis we will approach the relationship between logic and computation from another point-of-view and see how recent developments in logic programming can guide us toward the design of a new functional programming language featuring a novel approach to encode bindings that appear within data structures.

1 Introduction

Nowadays, programming languages are the most common way to give instructions to a computing device. Some of these languages reflect closely the hardware architecture they pilot like assembly languages: writing programs in such languages is a tedious and error prone task, because only a very small set of generic low-level instructions can be used. This situation is analogical to the exercise of building a proof of a complex theorem with only a basic set of axioms and inference rules, without cuts. It is possible, but the interesting parts of the program (or the proof) will be obfuscated by all the low-level administrative work one has to do. To cope with this difficulty, so called high-level languages provide abstractions that allow a simpler expression of the programmer's problems and then translate these high-level instructions to their lower-level equivalent which is usually much more verbose. This translation process is called compilation.

Today there exists a large variety of more-or-less-high-level programming languages, often designed explicitly to ease the creation of programs answering a specific set of problems. Like so, functional programming languages have been used from their earliest days to build systems that manipulate the syntax of other programming languages and logics. Actually, the ML language (which stands for "Meta Language") was initially developed as a tactic language for an interactive theorem prover [GMW79]. Other functional languages such as Lisp were common to build theorem provers, interpreters, compilers, parsers, etc. While these various tasks involve the manipulation of syntax, none of these earliest functional programming languages provided support for a key feature of almost all programming languages

and logics: variable binding. For instance, bindings are involved in logical quantification operators (x is bound in the formula $\forall x.\ \mathrm{P}(x)$), in programming languages functions (x is bound in `int f(int x) = { return 2 * x }`), etc.

Of course, due to their omnipresence, bindings in syntactic expressions have been given a range of different treatments within the functional programming setting. Each of these treatments has its advantages and disadvantages when dealing with standard problems such as testing equality of terms or to perform capture avoiding substitution.

In this chapter we present a new functional programming language with native treatment of bindings in datastructures, inspired by Abella, named MLTS. This acronym stands for Mobility and λ-tree syntax.

Chapter outline

- Section 1 provides an overview of some techniques used to deal with bindings and what led us to design a new system.
- Sections 2 and 3 presents the new structures for MLTS, our new language, along with numerous example programs in an attempt to give a working understanding of the language.
- Sections 4 to 7 contain the material needed to understand the foundations of MLTS. We give a typing system and a natural semantic of MLTS along with a few formal results. However this work is primarily focused on trying to get the design right.
- In Section 8 we present the current prototype implementation of MLTS.
- Finally in Section 9 we consider some works related to this design and highlight how they compare to our new system (when they can be compared).

But before all this we need to describe the syntax we are going to use in our examples.

An overview of the syntax

In this chapter, examples of ML-style programs are given in the OCaml [OCa18] concrete syntax, and the new language we present uses an extended version of this syntax. Here is a list of some key-points of the syntax of OCaml, illustrated with short examples:

- Comments are multiline, and delimited by `(*` and `*)`. They can be nested.
- Instructions like type and function definitions and single expressions are terminated by `;;`

  (In OCaml these double semi-colons can be omitted most of the time because they are optional before another definition or at the end of a file.)
- Global definition of an *immutable* variable: `let x = 4;;`
- Local definition of an immutable variable: `let x = 4 in 2 + x;;`
- Global (/ Local) definition of a function: `let f x y = 2 * x + y (in ..);;`
- Recursive functions must be declared with the keyword `rec`: `let rec f ..`
- Function parameters can be annotated with types: `(x : int)`

- Function application is done without englobing parentheses: `f 2 (1 + 2)` will yield the result `8`.
- Functions always return the value of the last expression of their body. The unit value is `()`.
- Expressions formed of sequences of expressions can be constructed using the binary operator `;` which type is `unit -> 'a -> 'a` (where `'a` is a type variable).
- If-expressions also return a value and both branches must have the same type: `let x = (if 3 = 4 then 42 else 36);;`. The operator `=` in `3 = 4` tests the *structural equality* (and may dive in the recursive definition of a type) while `==` tests the physical equality.
- Recursive types with constructors can be defined in the following way:

```
type int_list = (* a list of |-separated constructors *)
  | EmptyList
  | Cons of int * int_list
```

  This should be read: a `int_list` is either the empty list or the pair of an `int` and an `int_list`.

- Such types can be scrutinized under pattern matching:

```
match l with (* a list of |-separated match clauses *)
  | EmptyList -> printf("The list is empty")
  | Cons(i, tail) -> printf("The list is an int and a tail")
```

  This should be read: if `l` is the empty list then do something, else if l is a list made of a int (named `i`) and a list (named `tail`) then do some other thing.

Of course this is merely a glimpse of the complete OCaml syntax, but it is enough to understand the examples populating the rest of this thesis. The interested reader can find valuable documentation for the complete OCaml system on the main website: `https://ocaml.org`

1.1 A common example: substitution

To illustrate possible treatments for binders in datastructures, let's consider the implementation of substitution in the untyped λ-calculus. Our goal is to write in an ML-style language a function `subst : tm -> var -> tm` such that, given two terms `t` and `u` of type `tm`, `x` a variable, `subst t x u` is $t[x\backslash u]$, that is, the result of the substitution of x by u in t. The following discussion will list different approaches one can take in dealing with the "variable case".

Named variable A common and straightforward way to represent variable is to use names which are strings of characters. Consider the following ML type:

```
type tm =
  | Var of string
  | App of tm * tm
  | Abs of string * tm
```

It seems natural to use strings to encode variables, and the term $\lambda x.x\ y$ would then be represented as `Abs(Var "x", App(Var "x", Var "y"))`. Then the substitution function could be naively defined as follows:

```
let rec subst (t : tm) (x : string) (u : tm) =
  match t with
  | Var y        -> if x = y then u else Var y
  | App(m, n)    -> App(subst m x u, subst n x u)
  | Abs(y, body) -> Abs(y, subst body x u)
```

But this design is flawed because variable capture can happen in the last case of the match if, for example, a variable named `x` also appears in `u`. A more subtle mechanism is needed to ensure capture-free substitution, such as performing α-conversion over `t` when a collision with `x` occurs.

This is one of the many caveats of this naive representation for variables and more generally binders in datastructures. It needs to be very carefully handled to ensure the consistency of programs. It is a well known issue and so are its mitigations, but implementing them is an inexhaustible source of errors.

De Bruijn's nameless dummies In this representation bound variables are represented by an index giving the number of binders between the variable and its own binding term [Bru79]. Name collisions do not appear anymore but a very careful implementation is also needed to keep the indexes updated. It is a widely used way of representing bindings and numerous libraries and languages are built upon it. This technic is forgetful of the names of variables and several variants exist such as the *locally nameless* approach [Cha11; Gor94; MM04] in which bound variables are represented by de Bruijn indices and free variables by names.

Higher order abstract syntax Another way to encode bindings is to use some sort of Higher Order Abstract Syntax (HOAS). The trick here is to make use of the built-in support of functions (and thus binders) of the host language to represent abstraction. A possible ml type for this would be:

```
type tm =
  | App of tm * tm
  | Abs of tm -> tm
```

Where `tm -> tm` is the type of a function from terms to terms. Because there is no variable constructors, we cannot directly build terms with free variables in this setting. Instead, we need to bind such variable outside of the term using the host language functions. For example, the term $\lambda x.x\ y$ would be encoded as `fun y -> (Abs(fun x -> App(x, y)))`.

In that case, the substitution function is nothing else than the application in the host language. We can define the following delegating `subst` function:

```
let subst (f : tm -> tm) (t : tm) = (f t);;
```

It is a clever way to make use of the host language features. Unfortunately, such encoding technique often lacks adequacy (since "exotic terms" can appear [DFH95]), and structural recursion can slip away [GP99]. Moreover a defect of this technique when using the OCaml language is that we loose the ability to test for the structural equality of two terms or to pattern-match efficiently on them. Our approach in MLTS looks similar to this one, but allows to recover these important properties.

1.2 A new language, MLTS

Extending a functional programming language with features that support bindings in data has also already been considered before: for example, there have been the FreshML [SPG03; Pot07] and CαML [Pot06] extensions to ML-style functional programming languages. There also exists libraries such as Bindlib [LR18] for OCaml that ease the use of bindings. Also, entirely new functional programming languages, such as the dependently typed Beluga [PD10] language, have been designed and implemented with the goal to support bindings in syntax. In the domain of logic programming and theorem provers conception, several designs and implemented systems exist that incorporate approaches to binding: such systems include Isabelle's generic reasoning core [Pau89], Nominal Isabelle [Urb08], λProlog [NM88; MN12], Qu-Prolog [CRS91], Twelf [PS99], αProlog [CU04], the Minlog prover [Sch06], and the Abella theorem prover [Bae+14].

In the rest of this thesis, we present the design of a new functional programming language, MLTS, that extends (the core of) ML and incorporates the λ-*tree syntax* approach to encoding the abstract syntax of data structures containing binders. Briefly, the λ-tree syntax approach to syntax can be defined as following the next three tenets:

1. Syntax is encoded as simply typed λ-terms in which the primitive types are identified with syntactic categories.
2. Equality of syntax must include αβη-conversion (see Section 5.2).
3. Bound variables never become free: instead, their binding scope can move.

This latter tenet introduces the most characteristic aspect of λ-tree syntax which is often called *binder mobility*. In this setting, bindings never become free nor escape their scope: instead, binders in data structures are permitted to *move* into binders within programs. MLTS is, in fact, an acronym for *mobility and λ-tree syntax*.

2 The new features of MLTS

We chose the concrete syntax of MLTS to be an extension of that of the OCaml programming language (a program in MLTS not using the new language features should be accepted by the `ocamlc` compiler). We assume that the reader is familiar with basic syntactic conventions of OCaml [OCa18], many of which are shared with most ML-like programming languages. MLTS contains the following five new language features.

1. Datatypes can be extended to contain new *nominal* constants and the `(new X in body)` program phrase provides a binding that declares that the nominal `X` is new within the lexical scope given by `body`.
2. A new typing constructor `=>` is used to type bindings within term structures. This constructor is an addition to the already familiar constructor `->` used for the typing of functional expressions.
3. The *backslash* (`\` as an infix symbol that associates to the right) is used to form an abstraction of a nominal over its scope. For example, `(X\body)` is a syntactic expression that hides the nominal `X` in the scope `body`. Thus the backslash *introduces* an abstraction.

4. The `@` *eliminates* an abstraction: for example, the expression `((X\body) @ t)` denotes the result of substituting the abstracted nominal `X` with the term `t` in `body`.
5. Clauses within match-expressions can also contain the `(nab X in rule)` binding expression: in the scope of this binder, the symbol `X` can match existing nominals introduced by the `new` binder and the `\` operator. Note that `X` is bound over the entire match rule (including both the left and right-side of the rule).

These new term operators have the following precedence from highest to lowest: `@`, `new` and `\`. Other operators have the same precedences and associativity than in OCaml. Thus the expression `fun r -> X\ new Y in r @ X` reads as: `fun r -> (X\ (new Y in (r @ X)))`.

All three binding expressions—`(X\body)`, `(new X in body)` and `(nab X in rule)`—are subject to α-renaming of bound variables, just as the names of variables bound in `let` declarations and function definitions. As we shall see, nominals are best thought of as constructors: as a consequence, we follow the OCaml convention of capitalizing the name of their binders. We are assuming that, in all parts of MLTS, the names of nominals (or bound variables in general) are not available to programs since α-conversion (the alphabetic change of bound variables) is always applicable. Thus, compilers are free to implement nominals in any number of ways, even ways in which they do not have, say, print names.

Expressions involving `@` are greatly restricted within patterns of match expressions: in particular the expression `(m @ X1 ... Xj)` is restricted so that `m` is a pattern variable and `X1`, ..., `Xj` are distinct nominals bound within the scope of the pattern variable binding off `m`. This restriction is essentially the same as the one required by *higher-order pattern unification* [Mil91]: as a result, pattern matching in this setting is a simple generalization of usual first-order pattern matching.

We note that the expression `(X\ r @ X)` is interchangeable with the simple expression `r`: that is, when `r` is of `=>` type, η-equality holds.

3 MLTS examples

We now present several sets of examples of MLTS programs and the Appendix contains an additional longer example. We hope that the informal semantics given above plus the simplicity of the examples will give a working understanding of the semantics of MLTS. We delay the formal definition of the operational semantics of MLTS until Section 5.

3.1 The untyped λ-calculus

The untyped λ-terms can be defined in MLTS as the following datatype:

```
type tm =
     | App of tm * tm
     | Abs of tm => tm;;
```

The use of the `=>` type constructor here indicates that the argument of `Abs` is a *binding abstraction* of a `tm` over a `tm`. It should be reminiscent of the HOAS example in Section 1.1 but we don't use the host language function type constructor `->`

anymore and the type `tm` is now implicitly extended to contain nominal constant. It is said to be an *open* type and by default all user-defined types are open.

Just as the type `tm` denotes a syntactic category of untyped λ-terms, the type `tm => tm` denotes the syntactic category of terms abstracted over such terms.

Following usual conventions, expressions whose concrete syntax have nested binders using the same name are disambiguated by the parser by linking the named variable with the closest binder. Thus, the concrete syntax `(Abs(X\ Abs(X\ X)))` is parsed as a term α-equivalent to `(Abs(Y\ Abs(X\ X)))`. Similarly, the expression `(let n = 2 in let n = 3 in n)` is parsed as an expression α-equivalent to `(let m = 2 in let n = 3 in n)`: this expression has value `3`.

Size Fig. 3.1 shows the MLTS program that computes the size of an untyped λ-term: For example, `(size (App(Abs(X\X), Abs(X\X))))` evaluates to `5`. In the second match rule, the match-variable `r` will be bound to an expression of type `tm => tm` built using the backslash. On the right of that rule, `r` is applied to a single argument which is a newly provided nominal constructor of type `tm`. The third match rule contains the `nab` binder that allows the token `X` to match any nominal: alternatively, that last clause could have matched any non-`App` and non-`Abs` term by using the clause `| _ -> 1`. Note that as written, the three match rules used to define `size` could have been listed in any order. The following sequence of expressions shows the evolution of a computation involving the `size` function[1]:

```
size (Abs (X\ Abs (Y\ App(X,Y))));;
1 + new X in size (Abs (Y\ App(X,Y)));;
1 + new X in 1 + new Y in size (App(X,Y));;
1 + new X in 1 + new Y in 1 + size X + size Y;;
1 + new X in 1 + new Y in 1 + 1 + 1;;
```

The first call to `size` will bind the pattern variable `r` to `X\ Abs(Y\ App(X,Y))`. It is important to note that the names of bound variables within MLTS programs and data structures are fictions: in the expressions above, binding names are chosen for readability. It is also important to know that all nominals should be always bound in an MLTS program. Escaping nominals (like in the expression `new X in X`) will be treated as stuck terms and trigger an evaluation error.

Equality Fig. 3.2 shows the MLTS program that checks the equality of two terms of type `tm`. Both terms are recursively matched together: the application case is standard and self explanatory but the abstraction case is more interesting: the pattern variables `r1` and `r2` will be bound to two expressions of type `tm => tm`, in the body of that rule we declare a new nominal `X` and use it to "open" the bodies of `r1` and `r2`. That is, to move the top level bindings of `r1` and `r2` at the `new X in` binder level. The nominal cases are then simply checking if both nominals are the same of if they are different. Here is another sequence of expressions showing a computation involving the `eq` function:

1. It does not intend to reflect the precise control-flow of the program, that is call-by-value execution, it is simply a series of equivalent expressions illustrating the meaning of `size`

```
let rec size t =
        match t with
        | App(n, m)  -> 1 + size n + size m
        | Abs(r)     -> 1 + new X in size (r @ X)
        | nab X in X -> 1;;
```

Figure 3.1: The function that computes the size of a term.

```
let rec eq t1 t2 =
    match (t1, t2) with
    | (App(l1, l2), App(r1, r2)) ->
        eq l1 r1 && eq l2 r2
    | (Abs r1, Abs r2)           ->
        new X in eq (r1 @ X) (r2 @ X)
    | nab X   in (X, X)          -> true
    | nab X Y in (X, Y)          -> false
    | _                          -> false ;;
```

Figure 3.2: The function that checks the equality (modulo α-renaming) of two terms.

```
let rec subst t x u = match (x,t) with
  | nab X   in (X,X) -> u
  | nab X Y in (X,Y) -> Y
  | (x, Abs r)       -> Abs(Y\ subst (r @ Y) x u)
  | (x, App(m,n))    -> App(subst m x u,  subst n x u)
;;
```

Figure 3.3: The function for computing the substitution $u[x/t]$ where x is free in u.

```
let subst t u = new X in
    let rec aux t = match t with
      | X -> u
      | nab Y in Y -> Y
      | App(u, v) -> App(aux u, aux v)
      | Abs r -> Abs(Y\ aux (r @ Y))
    in aux (t @ X);;
```

Figure 3.4: The function for computing the substitution $u \,@\, t$ for u of type tm ⇒ tm

```
eq (Abs(X\ App(X, Abs(Y\ X)))) (Abs(U\ App(U, Abs(V\ V))));;
new X in eq (App(X, Abs(Y\ X))) (App(X, Abs(V\ V)));;
new X in eq X X && eq (Abs(Y\ X)) (Abs(V\ V));;
new X in true && eq (Abs(Y\ X)) (Abs(V\ V));;
new X in true && new Y in eq X Y;;
new X in true && new Y in false;;
true && false;;
false;;
```

Substitution Figure 3.3 shows a first version of the substitution function (of type `tm -> tm -> tm -> tm`) where `(subst t X u)` replaces "free" occurrences in `u` of the nominal `X` by `t`. Of course, since there are no free nominals in MLTS, `X` must be bound outside of the scope of the call to `subst`. This version of the substitution function is mostly self-explanatory: the function dives recursively in the term `t` and when a subterm is a nominal, it is replaced by `u` if it is an occurrence of `X`.

A more idiomatic version of the substitution can be provided: Figure 3.4 defines the function `(subst t u)` that takes an abstraction over a term `t` and a term `u` and returns the result of substituting the (top-level) bound variable of `t` with `u`. This function works by first introducing a new nominal `X` and then defining an auxiliary function that replaces that nominal in a term with the term `u`. Finally, that auxiliary function is called on the expression `(t @ X)` which is the result of "moving" the top-level bound variable in `t` to the binding occurrence of the expression `new X in`. This substitution function has the type `(tm => tm) -> (tm -> tm)`: that is, it is used to inject the abstraction type `=>` into the function type `->`.

β-reduction Substitution is then used by the function of Figure 3.5, `beta`, to compute the β-normal form of a given term of type `tm`. Given the following Church numeral for 2 and operations for addition and multiplication on Church numerals (see Section 1.3).

```
let two   = Abs(F\ Abs(X\ App(F, App(F, X))));;
let plus  = Abs(M\ Abs(N\ Abs(F\ Abs(X\
                      App(App(M, F), App(App(N, F),
                         X))))));;
let times = Abs(M\ Abs(N\ Abs(F\ Abs(X\
                               App(App(M, App(N, F)),
                                  X)))));;
```

In the resulting evaluation context, the values computed by `(beta (App(App(plus, two), two)))` and `(beta (App(App(times, two), two)))` are both the Church numeral for 4.

Vacuous For another example, consider a program that returns `true` if and only if its argument, of type `tm => tm`, is such that its top-level bound variable is a "vacuous" binding, that is, it does not appear in the body of the term. Figure 3.6 contains three implementations of this boolean-valued function. The first implementation proceeds by matching patterns with the prefix `X\`, thereby, matching expressions of type `tm => tm`. The second implementation uses a different style: it creates a new nominal `X` and proceeds to work on the term `t @ X`, in the same fashion as the `size` example. The internal `aux` function is then defined to search for occurrences of `X`

```
let rec beta t = match t with
  | nab X in X -> X
  | Abs r      -> Abs(Y\ beta (r @ Y))
  | App(m, n)  -> let m = beta m in let n = beta n in
      begin match m with
        | Abs r -> beta (subst r n)
        | _ -> App(m, n)
      end ;;
```

Figure 3.5: The (partial) function that computes the β-normal form of its argument.

```
let rec vacp1 t = match t with
  | X\ X                 -> false
  | nab Y in X\ Y        -> true
  | X\ App(m @ X, n @ X) -> vacp1 m && vacp1 n
  | X\ Abs(Y\ r @ X Y) -> new Y in vacp1 (X\ r @ X Y);;

let rec vacp2 t = new X in
    let rec aux term = match term with
        | X          -> false
        | nab Y in Y -> true
        | App(m, n)  -> aux m && aux n
        | Abs(r)     -> new Y in aux (r @ Y)
    in aux (t @ X);;

let vacp3 t = match t with
    | X\ s  -> true
    | _ -> false;;
```

Figure 3.6: Three implementations for determining if an abstraction is vacuous.

```
let rec assoc x alist = match alist with
  | (u,y)::tl -> if (u = x) then y else assoc x tl;;

type tm' = | App' of tm' * tm' | Abs' of tm' => tm';;

let rec id g term = match term with
  | App(m,n) -> App'(id g m, id g n)
  | Abs(r) -> new X in Abs'(Y\ id ((X, Y)::g) (r @ X))
  | nab X in X -> assoc X g;;
```

Figure 3.7: Translating from `tm` to its mirror version `tm'`.

in that term. The third implementation, `vacp3`, is not (overtly) recursive since the entire effort of checking for the vacuous binding is done during pattern matching. The first match rule of this third implementation is essentially asking the question: is there an instantiation for the (pattern) variable s so that the $\lambda x.s$ equals t? This question can be posed as asking if the logical formula $\exists s.(\lambda x.s) = t$ can be proved. In this latter form, it should be clear that since substitution is intended as a logical operation, the result of substituting for s never allows for variable capture. Hence, every instance of the existential quantifier yields an equation with a left-hand side that is a vacuous abstraction. Of course, this kind of pattern matching requires a recursive analysis of the term t.

Mirror For another simple example of computing on the untyped λ-calculus, consider introducing a mirror version of `tm`, as is done in Figure 3.7, and writing the function that constructs the mirror term in `tm'` from an input term `tm`. This computation is achieved by adding a context (an association list) as an extra argument that maintains the association of bound variables of type `tm` and those of type `tm'`. The value of `id [] (Abs(X\ Abs(Y\ App(X,Y))))` is `(Abs'(X\ Abs'(Y\ App'(X,Y))))` (the types of `X` and `Y` in these two expressions are, of course, different).

De Bruijn indices Figure 3.8 presents a datatype for the untyped λ-calculus in De Bruijn's style nameless dummies [Bru72] as well as the functions that can convert between that syntax and the one with explicit bindings. The auxiliary functions `nth` and `index` take a list of nominals as their second argument: `nth` takes also an integer `n` and returns the n^{th} nominal in that list while `index` takes a nominal and returns its ordinal position in that list. For example, the value of

```
trans [] (Abs(X\ Abs(Y\ Abs(Z\ App(X, Abs(W\ Z))))));;
```

is the term `DAbs(DAbs(DAbs(DApp(Dvar 2, DAbs(Dvar 1)))))` of type `deb`.

If `dtrans []` is applied to this second term, the former term is returned (modulo α-renaming, of course).

3.2 Higher-order programming examples

In this section we present several classic examples involving higher-order programming. That is, functions which take other functions as arguments and/or return functions. Recall the familiar "fold-right" higher-order function:

```
let rec foldr f a lst = match lst with
  | [] -> a
  | x :: xs -> f x (foldr f a xs);;
```

This function can be viewed as replacing all occurrences of `::` with the binary function `f` and `[]` with `a`. The higher-order program `maptm` in Figure 3.9 does the analogous operation on the datatype of untyped λ-terms `tm`. In particular, the constructors `App` and `Abs` are replaced by functions `fapp` and `fabs`, respectively. In addition, the function `fvar` is applied to all nominals encountered in the term. This higher-order function can be used to define a number of other useful and familiar functions. For example, `mapvar` function is a specialization of the `maptm` function that just applies a given function to all nominals in an untyped λ-term.

```
type deb =
     | Dapp of deb * deb
     | Dabs of deb
     | Dvar of int;;

let rec nth n l = match (n, l) with
  | (0, x::k) -> x
  | (c, x::k) -> nth (c - 1) k;;

let index x l =
    let rec aux c x k = match (x, k) with
          | nab X   in (X, X::(l @ X))   -> c
          | nab X Y in (X, Y::(l @ X Y)) ->
                                    aux (c + 1) x (l @
                                       X Y)
     in aux 0 x l;;

let rec trans prefix term = match term with
  | App(m, n)  -> Dapp(trans prefix m, trans prefix n)
  | Abs r      -> new X in Dabs(trans (X::prefix) (r @
     X))
  | nab Y in Y -> Dvar (index Y prefix);;

let rec dtrans prefix term = match term with
  | Dapp(m, n) -> App(dtrans prefix m, dtrans prefix n)
  | Dabs r     -> Abs(X\ dtrans (X::prefix) r)
  | Dvar c     -> nth c prefix;;
```

Figure 3.8: De Bruijn's nameless dummy syntax and its conversions with type `tm`.

The application of a substitution (an expression of type `(tm * tm) list`) to a term of type `tm` can then be seen as the result of applying the `lookup` function to every variable in the term (using `mapvar`). Using the functions in Figure 3.9, the three expressions:

```
Abs(X\
  mapvar (fun x -> X) (Abs(U\ Abs(V\ App(U, V))))
);;

new X in new Y in
 lookup ((X, Abs(U\U))::(Y, Abs(U\ App(U,U)))::[]) X;;

new X in new Y in
 lookup ((X, Abs(U\U))::(Y, Abs(U\ App(U,U)))::[]) Y;;
```

evaluate respectively to the following three λ-terms.

```
Abs(X\ Abs(Y\ Abs(Z\ App(X, X))))
Abs(X\ X)
Abs(X\ App(X, X))
```

```
let rec maptm fapp fabs fvar t = match t with
    | App(m,n) -> fapp (maptm fapp fabs fvar m)
                       (maptm fapp fabs fvar n)
    | Abs r  ->
        fabs (fun x -> maptm fapp fabs fvar (r @ x))
    | nab X in X -> fvar X;;


let lookup sub var = match var with
  | nab X in X ->
    let rec aux s = match s with
                    | []         -> X
                    | (X,t)::sub -> t
                    | (y,t)::sub -> aux sub
     in aux sub;;

let mapvar = maptm (fun m -> fun n -> App (m, n))
                   (fun r -> Abs (X \ r X));;

let rec mem x l = match l with
    | [] -> false
    | n::tl -> if n = x then true else mem x tl;;

let rec union l k = match l with
    | []    -> k
    | h::tl -> if mem h k then union tl k else
       h::(union tl k);;

let rec remove x l = match l with
    | []    -> []
    | h::tl -> if h = x then remove x tl else
       h::(remove x tl);;

let fv term =
  maptm union (fun r -> new X in remove X (r X))
              (fun x -> x::[]) term;;

let size term = maptm (fun x -> fun y -> 1 + x + y)
                      (fun r -> new X in 1 + (r X))
                      (fun x -> 1) term;;

let terminals term = maptm (fun x -> fun y -> x + y)
```

Figure 3.9: Various computations on untyped λ-terms using higher-order programs.

Three additional functions are defined in Figure 3.9: `fv` constructs the list of free variables in a term; `size` is a re-implementation of the `size` function presented in Section 3; and `terminals` counts the number of variable occurrences (terminal nodes) in its argument.

3.3 Normalization by Evaluation (NBE)

Fig. 3.10 shows one possible implementation of NBE in MLTS. NBE is a two steps process. Terms of type `tm` are first evaluated into values and functions (type `sem`) encoded via the ones of the host language, here MLTS functions. Then these values are *reified* back to the term syntax level at any time.

The implementation we provide is mostly inspired by the one given as an example program for the FreshML language in [SPG03]. The main difference in our implementation is that, because MLTS will check for escaping nominals when evaluating a function's arguments, intermediate values of type `sem` must explicitly bind locally "free" nominals that will disappear at the time of reification. To do that we added a constructor "New" to the type "sem" (see lines 2, 15, 36, 39).

We see this added complexity as a feature of MLTS, in fact, it is a crucial aspect of the language that variables can never be free at the program level. This forces the user to have a more precise understanding of what is happening in his programs and keep track of bound variables at all times, and move their binders as necessary until throughout their whole lifetime.

3.4 The π-calculus

The π-calculus [MPW92; Mil90b] is a language for modeling processes in which interactions are name-based. In particular, this calculus permits communication via named channels, including the communication of the names of the channels themselves. The basic calculus has two syntactic categories: names and processes.

Process expressions are defined by the following syntax rule.

$$\mathrm{P} := 0 \;\mid\; \mathrm{P} \mid \mathrm{P} \;\mid\; \mathrm{P} + \mathrm{P} \;\mid\; x(y).\mathrm{P} \;\mid\; \bar{x}y.\mathrm{P} \;\mid\; [x = y].\mathrm{P} \;\mid\; \tau.\mathrm{P} \;\mid\; (y)\mathrm{P} \;\mid\; !\,\mathrm{P}.$$

Here, x and y range over names. The process 0 cannot perform any actions. The expressions $\mathrm{P} \mid \mathrm{P}$ and $\mathrm{P} + \mathrm{P}$ denote, respectively, the parallel composition and the choice of two processes. The next four expressions are *prefixed* processes:

- $x(y).\mathrm{P}$ represents a process that can accept a name on the channel x and will then become P with y bound to the input name;
- $\bar{x}y.\mathrm{P}$ is a process that can output the name y on the channel x;
- $[x = y].\mathrm{P}$ is a process that can become P provided that the names x and y are equal;
- $\tau.\mathrm{P}$ is a process that can evolve through a silent action.

The expression $(y)\mathrm{P}$ represents the restriction of the name y to P: interactions can take place internally to P through this name but the process cannot communicate externally along the channels $\bar{y}$ or y. Finally, ! P denotes the parallel composition of any number of copies of P.

To represent expressions of the π-calculus in MLTS, we define the two datatypes `name` and `proc` for names and processes that are given in Figure 3.11. Note that the

```
type sem =
  | New of tm => sem
  (* functions *)
  | L of (unit -> sem) -> sem
  (* neutral values *)
  | N of neu
and neu =
  | V of tm
  (* neutral app *)
  | A of neu * sem;;

(* sem -> tm *)
let rec reify s =
    match s with
    | New r -> new Z in reify (r @ Z)
    | L f   -> Abs(X\ reify (f (fun () -> N (V X))))
    | N n   -> reifyn n

(* neu -> tm *)
and reifyn n =
    match n with
    | nab Y in V Y -> Y
    | A (n, s)     -> App (reifyn n, reify s);;

(* (tm * (unit -> sem)) list -> tm -> sem *)
let rec evals env term =
    match term with
    | nab X in X  ->
        begin match env with
        | []                    -> N (V X)
        | (X, v)::env           -> v ()
        | nab Y in (Y, v)::env  -> evals env X
        end
    | Abs t       ->
        L (fun v ->
            New (Z\ evals ((Z,v)::env) (t @ Z)))
    | App(t1, t2) ->
        let rec apply s = match s with
            | New r -> New (Z\ apply (r @ Z))
            | L f   -> f (fun () -> evals env t2)
            | N n   -> N (A (n, evals env t2))
        in
        apply (evals env t1);;

(* tm -> sem *)
let eval t = evals [] t;;

(* tm -> tm *)
let norm t = reify (eval t);;
```

Figure 3.10: Normalization By Evaluation.

```
type name = | A | B | C;;

type proc =
  | Null                              (* 0        *)
  | Plus of proc * proc               (* P + P    *)
  | Par  of proc * proc               (* P|P      *)
  | In   of name * (name => proc)     (* x(y).P   *)
  | Out  of name * name * proc        (* xy.P     *)
  | Eqn  of name * name * proc        (* [x=y].P  *)
  | Taup of proc                      (* tau.P    *)
  | Bang of proc                      (* !P       *)
  | Nu   of name => proc;;            (* (y)P     *)
```

Figure 3.11: Two data types for encoding the π-calculus.

```
let rec trans gamma term = match term with
  | App(m, n) ->
    let p = trans gamma m in
    let q = trans gamma n in
    (U\ Nu(V\ Par(
      p @ V,
      Nu(X\ Out(V, X, Out(V, U, Bang(In(X, q))))))))
  | Abs(m) ->
    new X in (U\ In(U, Y\
      let p = trans ((X,Y)::gamma) (m @ X) in
      In(U, V\ p @ V)))
  | nab X in X -> (U\ Out(assoc X gamma, U, Null));;
```

Figure 3.12: Encoding of the call-by-name evaluation of untyped λ-terms into the π-calculus.

two process expressions $x(y).\mathrm{P}$ and $(y)\mathrm{P}$ embody a binding notion. The λ-terms for these expressions will accordingly include an explicit abstraction. For example, the two π-calculus expressions

$$(y)\bar{a}y.((y(w).0) \mid (\bar{b}b.0)) \qquad \text{and} \qquad (y)\bar{a}y.((y(w).\bar{b}b.0) + (\bar{b}b.y(w).0))$$

are encoded in MLTS with the terms, respectively.

```
Nu(Y\ Out(A,Y,Par(In(Y, W\ Null),Out(B,B,Null))))
Nu(Y\ Out(A,Y,Plus(In(Y, W\ Out(B,B,Null)),
                   Out(B,B, In(Y, W\ Null)))))
```

In this encoding of the π-calculus (Figure 3.11), the type `name` must be considered open (in the sense described in Section 4.2) while the type `proc` is not open. The operational semantics of the π-calculus is generally described using a non-deterministic, labeled transition systems. That semantics is easily specified in λProlog [MN12] and reasoned with in Abella [Bae+14].

One way to demonstrate the expressiveness of the π-calculus is to encode within it the call-by-name evaluation of the untyped λ-calculus. Such a translation function

was given by Milner in [Mil90b] and it can be written as follows.

$$\begin{aligned} [\![x]\!](u) &= \bar{x}u.0 \\ [\![\lambda x\ \mathrm{M}]\!](u) &= u(x).u(v).[\![\mathrm{M}]\!](v) \\ [\![(\mathrm{M}\ \mathrm{N})]\!](u) &= (v).([\![\mathrm{M}]\!](v) \mid (x).(\bar{v}x.\bar{v}u.!x(w).[\![\mathrm{N}]\!](w))) \end{aligned}$$

Here, the translation function $[\![\mathrm{M}]\!](u)$ takes an untyped λ-term M and a name u and returns a process that encodes the λ-term M in such a way that it expects to receive its arguments on channel u. In Figure 3.12, we provide an MLTS implementation of this translation: in particular, if $[\![\mathrm{M}]\!](u)$ is the process calculus expression P, then the function `trans`, when applied to (the encoding of) M would yield (the encoding of) λu.P. (The function `assoc` used here is defined in Figure 3.7.) For example, the value of `(transf [] Abs(X\X))` is

```
(U\ In(U,X\ In(U,(Y\ Out(X,Y, Null))))).
```

4 Types and syntax

Three different syntaxes coexist in our actual description of MLTS. The first one is the *concrete syntax*, the one that users should use to write their programs and that is understood by the parser. It is quite verbose and not very suited to formal reasoning. We provide a more appropriate syntax, mostly isomorphic to it that we call *source syntax*. In both the concrete syntax and the source syntax, following standard convention in ML, languages pattern variables are *implicitly* quantified at the beginning of a pattern (and thus outside of the nabla-bound nominals). However, our formal reasoning will be done on another slightly different abstract syntax that we call *explicit syntax*. The major difference with the source syntax is that pattern variables must be *explicitly* bound before using them in explicit syntax. For example, the following pattern matching clause in source syntax:

```
nab X in (X, y) -> y
```

would correspond in explicit syntax to:

```
all y. nab X in (X, y) -> y
```

Fig. 3.13 shows the (combined) source syntax and explicit syntax for the interpretation of MLTS. Source syntax can be immediately deduced from it by removing the `all` rule. When proving formal results for MLTS we will use the explicit syntax.

4.1 Abstract syntax as untyped λ-calculus

Although MLTS is designed as a strongly typed functional programming language, evaluation for this language is fundamentally untyped. The abstract syntax for MLTS is based on the untyped λ-calculus along with a few extensions to capture the new features of MLTS.

Recall the semantic description of the untyped λ-calculus given by Scott in [Sco70]. Scott was able to present a semantic domain D that was isomorphic to its own function space: that is, $\mathrm{D} \equiv [\mathrm{D} \to \mathrm{D}]$. This equivalence is witnessed by the two continuous mappings $\Phi : \mathrm{D} \to (\mathrm{D} \to \mathrm{D})$ (encoding application) and $\Psi : (\mathrm{D} \to \mathrm{D}) \to \mathrm{D}$ (encoding abstraction). For example, the untyped λ-term $\lambda x \lambda y((x\ y)\ y)$ is encoded as a value in domain D using the expression $(\Psi(\lambda \mathrm{X}(\Psi(\lambda \mathrm{Y}(\Phi(\Phi\ \mathrm{X}\ \mathrm{Y})\ \mathrm{X})))))$.

M, N	$::=$		terms
	$\mid$	x, y, z	term variables
	$\mid$	X	nominals
	$\mid$	$\lambda x.\, \mathrm{M}$	function abstraction
	$\mid$	$\mathrm{M}\ \mathrm{N}$	application
	$\mid$	$\mathrm{X} \backslash \mathrm{M}$	name abstraction
	$\mid$	$\mathrm{M}\ @\ \mathrm{X}$	name application
	$\mid$	$\texttt{new}\ \mathrm{X}\ \texttt{in}\ \mathrm{M}$	name generation
	$\mid$	$\texttt{rec}\ x\ \texttt{in}\ \mathrm{M}$	Fixpoints
	$\mid$	$\mathrm{C}(\mathrm{M}_1 \ldots \mathrm{M}_n)$	data constructor
	$\mid$	$\texttt{match}\ \mathrm{M}\ \texttt{with}\ \mathrm{R}_1 \mid \cdots \mid \mathrm{R}_n$	pattern matching
R	$::=$		pattern-matching clause
	$\mid$	$\texttt{nab}\ \mathrm{X}\ \texttt{in}\ \mathrm{R}$	nabla-bound nominal
	$\mid$	$\texttt{all}\ x.\, \mathrm{R}$	pattern-bound variable
	$\mid$	$p \to \mathrm{M}$	match arm
p	$::=$		patterns
	$\mid$	x	variable
	$\mid$	X	nominal
	$\mid$	$\mathrm{C}(p_1 \ldots p_n)$	constructor
	$\mid$	$\_$	wildcard
	$\mid$	$\mathrm{X} \backslash p$	name abstraction
	$\mid$	$x\ @\ \mathrm{X}_1 \ldots \mathrm{X}_n$	name application
v	$::=$	$\lambda x.\, \mathrm{M}$	values
	$\mid$	$\mathrm{C}(v_1 \ldots v_n)$	
	$\mid$	$\mathrm{X} \backslash v$	

Figure 3.13: explicit syntax of MLTS terms (removing the red rule one obtains the source syntax for MLTS)

Note that syntactically, application in the untyped λ-calculus is captured by two domain-level features: function application and the mapping Φ. Similarly, abstraction is captured by two domain-level features: function abstraction (the creation of an element of [D → D]) and the mapping Ψ. We can thus identify two different *syntactic categories* in this encoding: those denoted by the domain D and those identified by the domain of (continuous) functions D → D. In what follows, we need to make a similar distinction between (λx.T) of type D → D and ($\Psi(\lambda x$.T)) of type D. In order to give suggestive names for this distinction, we shall borrow a bit of terminology from Martin-Löf's notion of *arity typing* [NPS90]. In particular, we will say that a term of type D has arity type **0** while a term of type D → D has arity type **0** → **0**. Other arity constructors are possible but they are not needed in the current design of MLTS.

In most formalizations of ML-style programming languages, expressions of non-zero arity generally only arise in the application of a function to its argument: all other features of the language only take arguments of arity type **0**. In MLTS, expressions of non-zero arity play extended roles: for example, in MLTS, pattern matching variables can have non-primitive arity while in most ML-languages, pattern variables are always of primitive arity. In fact, higher arity expressions are *first class citizen* in MLTS since they can be passed as parameters and returned as values. It is important to keep arity typing and ML-style typing separated. For example, the type of `subst` in Section 3.2 can be inferred to be `(tm => tm) -> tm -> tm`. The arity typing of `subst` is, however, the simple expression (**0** → **0**) → **0** → **0**: that is, the first argument given to `subst` must be a binding at the level of the abstract syntax. As we shall see in the following sections, the arity typing is used in the specification of the operational semantics of MLTS.

4.2 Typing for the concrete syntax

While most of the formalism we will introduce later relies on the explicit syntax, we think that giving typing rules for the source syntax users interact with is important to have a good understanding of the intended usage of MLTS. This typing system will only differ from the one for explicit syntax when dealing with pattern matching, which is slightly more complex for the source syntax because of the implicit pattern variables.

Given that MLTS is a rather mild extension to OCaml at the syntax level, a typing system for MLTS is quite simple to present and follows standard practices. Figure 3.14 contains the rules for typing the new features of MLTS: additional rules for encoding `let` and `let rec` constructions (as well as for built-in types such as integers) must also be added, but these follow the usual pattern. The inference rules in this figure involve the following typing judgments.

$$\Gamma \vdash \mathtt{M} : \mathtt{A} \qquad \Gamma \vdash \mathtt{A} : \mathtt{R} : \mathtt{B} \qquad \Gamma \vdash \mathtt{M} : \mathtt{A} \vdash \Delta \qquad \text{open } \mathtt{A}$$

In all of these rules, Γ is the usual association between bound variables and a type: in our situation, Γ will associate both variables and nominals to type expressions. (We also assume that the order of pairs in Γ is not important.) The first of these judgments is the usual typing judgment between a program expression `M` and `A`. The second of these judgments is used to type a pattern-matching clause `R` that has a left-hand side of type `A` and a right-hand side of type `B`. For example, the following typing judgment should be provable.

$$\Gamma \vdash \texttt{tm} : \texttt{Abs(r) -> 1 + (new X in size (r @ X))} : \texttt{int}$$

Expression typing judgment $\Gamma \vdash \mathrm{C} : \mathrm{A}$

$$\frac{}{\Gamma, x : \mathrm{C} \vdash x : \mathrm{C}} \qquad \frac{}{\Gamma, \mathrm{X} : \mathrm{A} \vdash \mathrm{X} : \mathrm{A}} \qquad \frac{\Gamma \vdash \mathrm{M} : \mathrm{A} \to \mathrm{B} \qquad \Gamma \vdash \mathrm{N} : \mathrm{A}}{\Gamma \vdash \mathrm{M\ N} : \mathrm{B}}$$

$$\frac{\Gamma, x : \mathrm{A} \vdash \mathrm{M} : \mathrm{B}}{\Gamma \vdash (\lambda x.\, \mathrm{M}) : \mathrm{A} \to \mathrm{B}} \qquad \frac{\mathrm{C} : \mathrm{A}_1, \ldots, \mathrm{A}_n \to \mathrm{B} \qquad \Gamma \vdash \mathrm{M}_1 : \mathrm{A}_1 \quad \ldots \quad \Gamma \vdash \mathrm{M}_n : \mathrm{A}_n}{\Gamma \vdash \mathrm{C}(\mathrm{M}_1, \ldots, \mathrm{M}_n) : \mathrm{B}}$$

$$\frac{\Gamma, x : \mathrm{B} \vdash \mathrm{M} : \mathrm{B}}{\Gamma \vdash \mathtt{rec}\ x\ \mathtt{in}\ \mathrm{M} : \mathrm{B}} \qquad \frac{\Gamma, \mathrm{X} : \mathrm{A} \vdash \mathrm{M} : \mathrm{B} \qquad \mathsf{open}\ \mathrm{A}}{\Gamma \vdash \mathrm{X} \backslash \mathrm{M} : \mathrm{A} \Rightarrow \mathrm{B}}$$

$$\frac{\Gamma \vdash \mathrm{M} : \mathrm{A} \Rightarrow \mathrm{B} \quad \Gamma \vdash \mathrm{X} : \mathrm{A}}{\Gamma \vdash \mathrm{M} \mathbin{@} \mathrm{X} : \mathrm{B}} \qquad \frac{\Gamma, \mathrm{X} : \mathrm{A} \vdash \mathrm{M} : \mathrm{B} \qquad \mathsf{open}\ \mathrm{A}}{\Gamma \vdash \mathtt{new}\ \mathrm{X}\ \mathtt{in}\ \mathrm{M} : \mathrm{B}}$$

$$\frac{\Gamma \vdash \mathrm{M} : \mathrm{A} \qquad \Gamma \vdash \mathrm{A} : \mathrm{R}_1 : \mathrm{B} \qquad \ldots \qquad \Gamma \vdash \mathrm{A} : \mathrm{R}_n : \mathrm{B}}{\Gamma \vdash \mathtt{match}\ \mathrm{M}\ \mathtt{with}\ \mathrm{R}_1 \mid \cdots \mid \mathrm{R}_n : \mathrm{B}}$$

Pattern-matching clauses typing judgment $\Gamma \vdash \mathrm{A} : \mathrm{R} : \mathrm{B}$

$$\frac{\Gamma, \mathrm{X} : \mathrm{C} \vdash \mathrm{A} : \mathrm{M} : \mathrm{B} \qquad \mathsf{open}\ \mathrm{C}}{\Gamma \vdash \mathrm{A} : \mathtt{nab}\ \mathrm{X}\ \mathtt{in}\ \mathrm{M} : \mathrm{B}} \qquad \frac{\Gamma \vdash \mathrm{L} : \mathrm{A} \dashv \Delta \qquad \Gamma, \Delta \vdash \mathrm{R} : \mathrm{B}}{\Gamma \vdash \mathrm{A} : \mathrm{L} \to \mathrm{R} : \mathrm{B}}$$

Pattern typing judgment $\Gamma \vdash p : \mathrm{A} \dashv \Delta$

$$\frac{}{\Gamma \vdash x : \mathrm{C} \dashv x : \mathrm{C}} \qquad \frac{}{\Gamma, \mathrm{X} : \mathrm{A} \vdash \mathrm{X} : \mathrm{A} \dashv \varnothing}$$

$$\frac{\mathrm{C} : \mathrm{A}_1, \ldots, \mathrm{A}_n \to \mathrm{B} \qquad \Gamma \vdash p_1 : \mathrm{A}_1 \dashv \Delta_1 \quad \ldots \quad \Gamma \vdash p_n : \mathrm{A}_n \dashv \Delta_n}{\Gamma \vdash \mathrm{C}(p_1, \ldots, p_n) : \mathrm{B} \dashv \Delta_1, \ldots, \Delta_n}$$

$$\frac{\Gamma, \mathrm{X} : \mathrm{A} \vdash p : \mathrm{B} \dashv \Delta \qquad \mathsf{open}\ \mathrm{A}}{\Gamma \vdash \mathrm{X} \backslash p : \mathrm{A} \Rightarrow \mathrm{B} \dashv \Delta}$$

$$\frac{\Gamma \vdash \mathrm{X}_1 : \mathrm{A}_1 \ \ldots\ \Gamma \vdash \mathrm{X}_n : \mathrm{A}_n \quad \mathsf{open}\ \mathrm{A}_1 \ldots \mathsf{open}\ \mathrm{A}_n}{\Gamma \vdash x \mathbin{@} \mathrm{X}_1 \ldots \mathrm{X}_n : \mathrm{A} \dashv x : \mathrm{A}_1 \Rightarrow \ldots \Rightarrow \mathrm{A}_n \Rightarrow \mathrm{A}}$$

Figure 3.14: Typing rules based on the source syntax for the new features of MLTS.

Since this rule expression is intended to be closed (that is, the variable `r` is quantified implicitly around this rule), the actual value of Γ will not impact this particular typing judgment. The third typing judgment above is used to analyze the left-hand-side of a match rule: in particular, $\Gamma \vdash \texttt{M} : \texttt{A} \vdash \Delta$ holds if during the process of analyzing the pattern `M`, pattern variables are produced (since these are implicitly quantified) and placed into the typing context Δ. For example, the following should be provable.

$$\Gamma \vdash \texttt{Abs(r)} : \texttt{tm} \vdash \{\texttt{r} : \texttt{tm} \Rightarrow \texttt{tm}\}$$

Some of the inference rules in Figure 3.14 contain premises of the form (open `A`) where `A` is a primitive type. Types for which this judgment holds are called *open types* and are the types of bindings in the `new` and backslash expressions: equivalently, open types can contain nominals. For our purposes here, we can assume that every type that is defined in a program (using the `type` command) is presumed to be open. For example, the judgment (open `tm`) needs to be true so that the type `tm => tm` can be formed in the various typing rules. On the other hand, the built-in type for integers `int` should not be considered open in this sense. Clearly a keyword could be added to datatype declarations to indicate if a type is intended as open in this sense. Our current implementation does not use one.

In the inference rules in Figure 3.14, whenever we extend the typing context Γ to, say, $\Gamma, \texttt{X} : \texttt{A}$, we always assume that `X` is not declared with a type in Γ already. Since α-conversion is always possible within terms, this assumption can always be satisfied. Note that since pattern variables are restricted (as is usual) so that they have at most one occurrence in a given pattern, the union of contexts, in the form $\Delta_1, \ldots, \Delta_n$ never attributes more than one type to the same variable.

4.3 Typing for the explicit syntax

Complementarily, Fig. 3.15 show the typing rules for the explicit syntax that we will be using in formal proofs later in the thesis. Notice the simplification of the pattern-matching rules, the environment Δ not being needed anymore now that pattern variables are declared at the beginning of patterns.

5 Formalizing the design of MLTS

As we have seen before, bindings are such an intimate part of the nature of syntax that we should expect that our high-level programming languages account for them directly in. For example, any built-in notion of equality or matching should respect at least α-conversion. (The paper [Mil18a] contains an extended argument of this point in the setting of logic programming and proof assistants.) Another reason to include binders as a primitive within a functional programming language is that their semantics have a well understood declarative and operational treatment. For example, Church's higher-order logic STT [Chu40] contains an elegant representation of bindings in both terms and formulas. His logic also identifies equality for both terms and formulas with $\alpha\beta\eta$-conversion. Church's representation is also a popular one in theorem proving—being the core logic of the Isabelle [Pau94], HOL [Har09; Gor91], and Abella [Bae+14] theorem provers—as well as the logic programming language λProlog [MN12]. Given the existence of these provers, a good literature now exists that describes how to effectively implement STT and

Expression typing judgment $\Gamma \vdash \mathrm{C} : \mathrm{A}$

$$\frac{}{\Gamma, x : \mathrm{C} \vdash x : \mathrm{C}} \qquad \frac{}{\Gamma, \mathrm{X} : \mathrm{A} \vdash \mathrm{X} : \mathrm{A}} \qquad \frac{\Gamma \vdash \mathrm{M} : \mathrm{A} \to \mathrm{B} \qquad \Gamma \vdash \mathrm{N} : \mathrm{A}}{\Gamma \vdash \mathrm{M\ N} : \mathrm{B}}$$

$$\frac{\Gamma, x : \mathrm{A} \vdash \mathrm{M} : \mathrm{B}}{\Gamma \vdash (\lambda x.\, \mathrm{M}) : \mathrm{A} \to \mathrm{B}} \qquad \frac{\mathrm{C} : \mathrm{A}_1, \ldots, \mathrm{A}_n \to \mathrm{B} \qquad \Gamma \vdash \mathrm{M}_1 : \mathrm{A}_1 \quad \ldots \quad \Gamma \vdash \mathrm{M}_n : \mathrm{A}_n}{\Gamma \vdash \mathrm{C}(\mathrm{M}_1, \ldots, \mathrm{M}_n) : \mathrm{B}}$$

$$\frac{\Gamma, x : \mathrm{B} \vdash \mathrm{M} : \mathrm{B}}{\Gamma \vdash \mathtt{rec}\ x\ \mathtt{in}\ \mathrm{M} : \mathrm{B}} \qquad \frac{\Gamma, \mathrm{X} : \mathrm{A} \vdash \mathrm{M} : \mathrm{B} \qquad \mathsf{open}\ \mathrm{A}}{\Gamma \vdash \mathrm{X} \backslash \mathrm{M} : \mathrm{A} \Rightarrow \mathrm{B}}$$

$$\frac{\Gamma \vdash \mathrm{M} : \mathrm{A} \Rightarrow \mathrm{B} \quad \Gamma \vdash \mathrm{X} : \mathrm{A}}{\Gamma \vdash \mathrm{M} \ @\ \mathrm{X} : \mathrm{B}} \qquad \frac{\Gamma, \mathrm{X} : \mathrm{A} \vdash \mathrm{M} : \mathrm{B} \qquad \mathsf{open}\ \mathrm{A}}{\Gamma \vdash \mathtt{new}\ \mathrm{X}\ \mathtt{in}\ \mathrm{M} : \mathrm{B}}$$

$$\frac{\Gamma \vdash \mathrm{M} : \mathrm{A} \qquad \Gamma \vdash \mathrm{A} : \mathrm{R}_1 : \mathrm{B} \qquad \ldots \qquad \Gamma \vdash \mathrm{A} : \mathrm{R}_n : \mathrm{B}}{\Gamma \vdash \mathtt{match}\ \mathrm{M}\ \mathtt{with}\ \mathrm{R}_1 \mid \cdots \mid \mathrm{R}_n : \mathrm{B}}$$

Pattern-matching clauses typing judgment $\Gamma \vdash \mathrm{A} : \mathrm{R} : \mathrm{B}$

$$\frac{\Gamma, \mathrm{X} : \mathrm{C} \vdash \mathrm{A} : \mathrm{M} : \mathrm{B} \qquad \mathsf{open}\ \mathrm{C}}{\Gamma \vdash \mathrm{A} : \mathtt{nab}\ \mathrm{X}\ \mathtt{in}\ \mathrm{M} : \mathrm{B}} \qquad \frac{\Gamma \vdash \mathrm{L} : \mathrm{A} \qquad \Gamma \vdash \mathrm{R} : \mathrm{B}}{\Gamma \vdash \mathrm{A} : \mathrm{L} \to \mathrm{R} : \mathrm{B}}$$

$$\frac{\Gamma, x : \mathrm{C} \vdash \mathrm{A} : \mathrm{R} : \mathrm{B}}{\Gamma \vdash \mathrm{A} : \mathtt{all}\ x.\mathrm{R} : \mathrm{B}}$$

Pattern typing judgment $\Gamma \vdash p : \mathrm{A}$

$$\frac{}{\Gamma, x : \mathrm{C} \vdash x : \mathrm{C}} \qquad \frac{}{\Gamma, \mathrm{X} : \mathrm{A} \vdash \mathrm{X} : \mathrm{A}}$$

$$\frac{\mathrm{C} : \mathrm{A}_1, \ldots, \mathrm{A}_n \to \mathrm{B} \qquad \Gamma \vdash p_1 : \mathrm{A}_1 \quad \ldots \quad \Gamma \vdash p_n : \mathrm{A}_n}{\Gamma \vdash \mathrm{C}(p_1, \ldots, p_n) : \mathrm{B}}$$

$$\frac{\Gamma, \mathrm{X} : \mathrm{A} \vdash p : \mathrm{B} \qquad \mathsf{open}\ \mathrm{A}}{\Gamma \vdash \mathrm{X} \backslash p : \mathrm{A} \Rightarrow \mathrm{B}}$$

$$\frac{\Gamma \vdash \mathrm{X}_1 : \mathrm{A}_1 \ \ldots\ \Gamma \vdash \mathrm{X}_n : \mathrm{A}_n \quad \mathsf{open}\ \mathrm{A}_1 \ldots \mathsf{open}\ \mathrm{A}_n}{\Gamma, x : \mathrm{A}_1 \Rightarrow \ldots \Rightarrow \mathrm{A}_n \Rightarrow \mathrm{A} \vdash x\ @\ \mathrm{X}_1 \ldots \mathrm{X}_n : \mathrm{A}}$$

Figure 3.15: Typing rules based on the explicit syntax of MLTS.

closely related logics. Below, we describe what that literature can tell us about the meaning and implementation of the novel features of MLTS.

5.1 Equality modulo α, β, η conversion

The abstract syntax behind MLTS is essentially a simply typed λ-term that encodes untyped λ-calculus, as described in Section 4.1. Furthermore, the equality theory of such terms is given by the familiar α, β, η conversion rules. As a result, a programming language that adopts this notion of equality cannot take an abstraction and return, say, the name of its bound variable: since that name can be changed via the α-conversion, such an operation would not be a proper function. Thus, it is not possible to decompose the untyped λ-term $\lambda x.t$ into the two components x and t. Not being able to retrieve a bound variable's name might appear as a serious deficiency but, in fact, it can be a valuable feature of the language: for example, a compiler does not need to maintain such names and can choose any number of different, low-level representations of bindings to exploit during execution. Since the names of bindings seldom have semantically meaningful value, dropping them entirely is an interesting design choice. That choice is similar to one taken in ML-style languages in which the location in memory of a reference cell is not maintained as a value in the language.

The relation of λ-conversion is invoked when evaluating the expression `(t @ s1 ... sn)`. If we assume that expressions `s1`, ... , `sn` have arities $\rho_1, \ldots, \rho_n$, respectively, then `t` must have arity $\rho_1 \to \cdots \to \rho_n \to \mathbf{0}$. Thus, `t` is η-equivalent to a term with n abstractions, for example, `X1\...Xn\ t'` and the value of the expression `(t @ s1 ... sn)` is the result of performing λ-normalization of `(X1\...Xn\ t')` to the arguments `s1`, ... , `sn`.

As we illustrated in Section 3, it is possible to implement both substitution and λ-conversion in MLTS. Thus, it is possible to limit the occurrences of `@` to appear only within the scope of match clauses and only then with a pattern variable as the first argument of `@`. For the sake of the rest of this work, we will not enforce that restriction.

5.2 Pattern unification and matching

Since we are not able to decompose bindings into their bound variable and body, we need to find alternative means for analyzing the structure of terms containing bindings. As our earlier examples illustrated, matching within patterns can be used to probe terms and their bindings. If we do not place restrictions on the use of pattern variables, then patterns can have complex behaviors.

No repeated pattern variable occurrences

We impose a familiar restriction on the match rules: a pattern variable cannot have more than one occurrence within a match pattern. The main reason this is done in ML-style languages is that it relieves pattern matching from the need to check equality of terms. Since terms can be large, pattern matching could involve a costly recursive descent of terms. It is far more common to forbid repeated occurrences of pattern variables and force the programmer to insert equality checking outside the pattern matching operation. Thus, instead of defining `memb : tm -> tm list -> bool` with the following code using a repeated match variable

```
let rec memb x l = match (x,l) with
    | (x,[])     -> false
    | (x,(x::l)) -> true
    | (y,(x::l)) -> memb x l;;
```

we can require the programmer to write an equality predicate for type `tm` and then rewrite the program above as follows.

```
let rec eqtm t s = match (t,s) with
    | (App(m1,m2),
       App(n1,n2))     -> eqtm m1 n1 && eqtm m2 n2
    | (Abs r, Abs s)   -> new X in eqtm (r @ X) (s @ X)
    | nab X in (X, X)  -> true
    | _                -> false;;

let rec memb x l = match (x,l) with
    | (x,[])     -> false
    | (x,(y::l)) -> if (eqtm x y) then true
                                  else (memb x l);;
```

Given the definition of the `tm` datatype, it is clear that a compiler for MLTS could define its own equality predicate for this type. In that case, repeated variable occurrences in patterns could be allowed since resolving such patterns could be done using these equality predicates.

Restricted use of higher-order pattern variables.

Since pattern variables within match rules can have higher-order arity (and higher-order types), occurrences of those variables within patterns need to be restricted: otherwise, undesirable features of higher-order matching could appear. Fortunately, there is a natural restriction on occurrences of pattern variables that guarantees that a match either fails or succeeds with at most one solution. That restriction is the following: every occurrence of an expression of the form `(r @ X1 ... Xn)` in the left-hand side of a match rule must be such that the pattern variable `r` is applied to $n \geq 0$ *distinct* nominals `X1 ... Xn` and those nominals are bound *within* the scope of the binding for `r`. For example, the following expression is not well formed

```
Abs(X\ (match Abs(Y\ App(X,Y)) with
            | Abs(Z\ r @ Z X) -> Abs(Z\ r @ X Z)))
```

since the scope of the nominal `X` contains the (implicit) scope of the pattern variable `r`, which is around the rule `(Abs(Z\ r @ Z X) -> Abs(Z\ r @ X Z))`.

This restriction can be motivated within a purely logical setting as follows. Let j be a primitive type and let $f : j \to j \to j$ be a simply typed constant. The formula $\exists G : j \to j \, \forall x : j \, [G\ x = (f\ x\ x)]$ has a unique proof in which G is instantiated by the term $\lambda w.(f\ w\ w)$. Note that the binding scope of the variable x is inside the binding scope of the variable G. If, however, one switches the order of the quantifiers, yielding $\forall x : j \, \exists G : j \to j \, [G\ x = (f\ x\ x)]$, then there are four different proofs of this equation: if one replaces the outermost universal quantifier with an eigenvariable, say a, then there are four different solutions for G, namely, $\lambda w.(f\ a\ a)$, $\lambda w.(f\ a\ w)$, $\lambda w.(f\ w\ a)$, and $\lambda w.(f\ w\ w)$.

The subset of higher-order unification in which unification variables (a.k.a., logic variables, meta-variables, pattern variables) are applied to distinct bound variables

restricted as described above, is called *higher-order pattern unification* or L_λ *unification* [Mil91]. (Nipkow provides a functional programming implementation of such unification in [Nip93].) This particular subset of higher-order unification is commonly implemented in theorem provers such as Abella [Bae+14], Minlog [Sch06], and Twelf [PS99] as well as recent implementations of λProlog [Dun+15; Qi+15].

The following results about higher-order pattern unification are proved in [Mil91].

1. It is decidable and unitary, meaning that if there is a unifier then there exists a most general unifier.
2. It does not depend on typing (or on arity). As a result, it is possible to add it to the evaluator for MLTS based on untyped terms.
3. The only form of β-conversion that is needed to solve such unification problems is what is called β_0-conversion which is a form of the β rule that equates $(\lambda x.t)x$ with t.

An equivalent way to write the β_0-conversion rule (assuming the presence of α-conversion) is that $(\lambda x.t)y$ converts to $t[y/x]$ *provided* that y is not free in $\lambda x.t$. Notice that applying β_0 reduction actually makes a term smaller and does not introduce new β redexes: as a result it is not a surprise that such unification (and, hence, matching) has low computational complexity (the paper [Qia96] claims that such unification is, in fact, solvable in linear time).

All nab-bound variables must have a rigid occurrence.

There is an additional restriction on match rules that is associated to the nab quantifiers that appear in such rules. We say that an occurrence of a nab-quantified nominal is *flexible* if it is in the scope of an @. For example, in the code

```
Abs(X\ (match Abs(Y\ App(X,Y)) with
            | nab W in Abs(Z\ r @ Z W) ->
                       Abs(Z\ r @ W Z)));;
```

the nominal binding `W` has two occurrences that are flexible: one each within `(r @ Z W)` and `(r @ W Z)`. All other occurrences of a nab quantified nominal is *rigid*. For example, in the match rule `| nab X in X -> 1`, `X` has a binding occurrence and a rigid occurrence. In the auxiliary function used by the `index` function in Figure 3.8, namely,

```
let rec aux c x k = match (x, k) with
    | nab X   in (X, X::(l @ X))   -> c
    | nab X Y in (X, Y::(l @ X Y)) ->
                                aux (c + 1) x (l @ X Y)
```

the nominals `X` and `Y` have both rigid and flexible occurrences within their scope.

The one additional restriction that we need is the following: every nab quantified variable must have at least one rigid occurrence in the left part of the match rule (the pattern) that falls within the scope of its binder. For example, the code listed above (for an expression of type `tm`) does not satisfy this restriction since every occurrence of `W` in the pattern is flexible (there is just one such occurrence). The necessity of this restriction can be seen when we consider a pattern of the form

```
    | nab X Y in  (r @ X Y) -> term
```

In the event that a nominal, say `U`, is matched with the pattern in this rule, there are two possible instantiations for `r` that could succeed, namely, the terms `X\Y\X` and `X\Y\Y`: we wish to avoid multiple successful matches of the same rule. The following clause is also ruled out by this restriction

```
| nab X in 1 -> X
```

since `X` has no rigid occurrence in the expression `1`. Discarding this match rule makes sense since the nominal that is returned as the result of this match is not constrained by the input to the match.

5.3 β_0 versus β

As we described in Section 5.2, in order to ensure that matching a rule either fails or has a unique, most general solution, we will insist that in the left-hand side of a match rule (that is in patterns), all subexpressions of the form `(r @ X1 ... Xn)` are such that the scope of the binding for `r` contains the scopes of the bindings for the distinct variables in `X1`, ..., `Xn`. On the right-hand side of a match rule, however, it seems that one has an interesting choice. If on the right, we have an expression of the form `(r @ t1 ... tn)` then clearly, the terms `t1`, ..., `tn` are intended to be substituted into the abstraction that is instantiated for the pattern variable `r`: that is, we need to use β-conversion on this redex. One design choice is that we restrict the terms `t1`, ..., `tn` to be distinct nominals just as on the left-hand-side: in this case, β-reduction of the expression `(r @ t1 ... tn)` requires only β_0 reductions. A second choice is that we allow the terms `t1`, ..., `tn` to be unrestricted: in this case, β-reduction of the expression `(r @ t1 ... tn)` requires more general (and costly) β-reductions.

A similar trade-off between allowing β-conversion or just β_0 conversion has also been studied within the theory and design of the π-calculus. In particular, the full π-calculus allows the substitution of arbitrary names into input prefixes (modeled by β-conversion) while the π_I-calculus (π-calculus with internal mobility [San96]) is restricted in such a way that the only instances of β-conversions are, in fact, β_0-conversions (see Chapter 11 in [MN12]).

Another reason to identify the β_0 fragment of β-conversion is that β_0 reduction provides support for binder mobility and it can be given effective implementations, sometimes involving only constant time (see Section 8.3).

5.4 Match rule quantification

Match rules in MLTS contain two kinds of quantification. The familiar quantification of pattern variables can be interpreted as being universal quantifiers. For example, the first rule defining the `size` function in Section 3, namely,

```
| App(n, m)  -> 1 + size n + size m
```

uses two pattern variables `n` and `m` and can be encoded as the logical statement

$$\forall m \forall n[(\texttt{size (App(n, m))} = \texttt{1 + size n + size m}].$$

On the other hand, the third match rule for `size` contains the binder `nab`

```
| nab X in X -> 1
```

which corresponds approximately to the *generic* ∇-quantifier (pronounced nabla) that is found in various efforts to formalize the metatheory of computational systems [MT05; Bae+14]. Particularly, this rule can be encoded as the quantified equation $\nabla x.(\texttt{size x} = 1)$: that is, the size of a nominal constant is 1.

Although there are two kinds of quantifiers around such match rules, the ones corresponding to the universal quantifiers are implicit while the ones corresponding to the ∇-quantifiers are explicit. Our design for MLTS places the implicit quantifiers at outermost scope: that is, the quantification over a match rule is of the form $\forall\nabla$. Another choice might be to allow some (all) universal quantifiers to be explicitly written and placed among any nab bindings. While this is a sensible choice, the $\forall\nabla$-prefixes is, in fact, a reduction class in the sense that if one has a $\forall$ quantifier inside a ∇-quantifier, it is possible to rotate that ∇-quantifier inside using a technique called *raising* [Mil91; MT05]. That is, the formula $\nabla x : \gamma,\ \forall y : \tau,\ (\mathrm{B}\ x\ y)$ is logically equivalent to the formula $\forall h : (\gamma \to \tau),\ \nabla x : \gamma,\ (\mathrm{B}\ x\ (h\ x))$: note that as the ∇-quantifier of type γ is moved to the right over a universal quantifier, the type of that quantifier is raised from τ to $\gamma \to \tau$. Thus, it is possible for an arbitrary mixing of $\forall$ and ∇ quantifiers to be simplified to be of the form $\forall\nabla$.

5.5 Nominal abstraction

Before we can present the formal operational semantics of MLTS, we need to introduce one final logical concept: *nominal abstraction* which allows implicit bindings represented by nominals to be moved into explicit abstractions over terms [GMN11]. The following notation is useful for defining this relationship.

Let t be a term, let $c_1, \ldots, c_n$ be distinct nominals that possibly occur in t, and let $y_1, \ldots, y_n$ be distinct variables not occurring in t and such that, for $1 \le i \le n$, y_i and c_i have the same type. Then we write $\lambda c_1 \ldots \lambda c_n.t$ to denote the term $\lambda y_1 \ldots \lambda y_n.t'$ where t' is the term obtained from t by replacing c_i by y_i for $1 \le i \le n$. There is an ambiguity in this notation in that the choice of variables $y_1, \ldots, y_n$ is not fixed. However, this ambiguity is harmless: the terms that are produced by acceptable choices are all equivalent under a renaming of bound variables.

Let $n \ge 0$ and let s and t be terms of type $\tau_1 \to \cdots \to \tau_n \to \tau$ and τ, respectively; notice, in particular, that s takes n arguments to yield a term of the same type as t. The formula $s \trianglerighteq t$ is a *nominal abstraction of degree n* (or, simply, a *nominal abstraction*). The symbol $\trianglerighteq$ is used here in an overloaded way in that the degree of the nominal abstraction it participates in can vary. The nominal abstraction $s \trianglerighteq t$ of degree n is said to hold just in the case that s is λ-convertible to $\lambda c_1 \ldots c_n.t$ for some distinct nominals $c_1, \ldots, c_n$.

Clearly, nominal abstraction of degree 0 is the same as equality between terms based on λ-conversion, and we will use $=$ to denote this relation in that case. In the more general case, the term on the left of the operator serves as a pattern for isolating occurrences of nominals. For example, if p is a binary constructor and c_1 and c_2 are nominals, then the nominal abstractions of the first row below hold while those in the second row do not.

$$\begin{array}{lll} \lambda x.x \trianglerighteq c_1 & \lambda x.p\ x\ c_2 \trianglerighteq p\ c_1\ c_2 & \lambda x.\lambda y.p\ x\ y \trianglerighteq p\ c_1\ c_2 \\ \lambda x.x \not\trianglerighteq p\ c_1\ c_2 & \lambda x.p\ x\ c_2 \not\trianglerighteq p\ c_2\ c_1 & \lambda x.\lambda y.p\ x\ y \not\trianglerighteq p\ c_1\ c_1 \end{array}$$

A logic with equality generalized to nominal abstraction has been studied in [Gac09; GMN11] where a logic, named $\mathcal{G}$, that contains fixed points, induction, coinduction,

∇-quantification, and nominal abstraction is given a sequent calculus presentation. Cut-elimination for $\mathcal{G}$ is proved in [GMN11] and algorithms and implementations for nominal abstraction are presented in [Gac09; Wan+13]. An important feature of the Abella prover—∇ in the head of a definition—can be explained and encoded using nominal abstraction [GMN08].

6 Natural semantic specification

We can now define the operational semantics of MLTS by giving inference rules in the style of natural semantic (a.k.a. big-step semantic) following Kahn [Kah87]. The semantic definition for the core of MLTS is defined in Figure 3.16. Since those inference rules are written using the explicit syntax for MLTS, we need to describe briefly how that syntax is derived from the concrete syntax we have used in our examples.

Instead of detailing the translation from concrete to abstract syntax, we illustrate this translation with an example. There is an implementation of MLTS that includes a parser and a transpiler into λProlog code: this system is available for online use and for download at `https://trymlts.github.io`. For example, the λProlog code in Figure 3.17 is the abstract syntax for the MLTS program for `size` given in Section 3.

The backslash (as infix notation) is also used in λProlog to denote binders. It is the only λProlog primitive in Figure 3.17. The other constructors were defined by us to represent MLTS abstract syntax trees.

The constant `rec` represents anonymous fixpoints, to which recursive functions are translated (we also have a n-ary fixpoint for mutually-recursive functions). Note that `rec` x `in` t will correspond to the λProlog code `rec x\ t` which is idiomatic λProlog syntax for the application `rec(x\ t)`, omitting parentheses to use `rec` in the style of a binder. The expression (λx. ...) (or `lam x \ ...` in the λProlog interpreter) represents the MLTS expression `fun x -> ...`. Similarly, the expression (`new` X `in` ...) (or `new X \ ...`) encodes `new X in ...`. The expression-former `match` represents pattern-matching, it expects a scrutinee and a list of clauses. Clauses are built from the infix operator ⟹, taking a pattern on the left and a term on the right, and from quantifiers `all`, to introduce universally-quantified variables (implicit in MLTS programs), and `nab` to introduce nominals. `all`-bound variables and `nab`-bound nominals have the type of expressions; in the interpreter they are injected in patterns by `pvar` and `pnom`. `pvariant` (in patterns) and `variant` (in expressions) denote datatype constructor applications, they expect a datatype constructor and a list of arguments. `special` expects the name of a run-time primitive (arithmetic operations, polymorphic equality...) and a list of arguments. `int` represents integer literals. Finally, we use explicit AST expression-formers `backslash` and `arobase` and pattern-formers `pbackslash` and `parobase` to represent the constructions `\` and `@` of MLTS. Only `arobase` is present in this example.

It is intended that the inference rules given in Figure 3.16 are, in fact, notations for formulas in the logic $\mathcal{G}$. For example, schema variables of the inference figure are universally quantified around the intended formula; the horizontal line is an implication; the list of premises is a conjunction; and ⇓ is a binary (infix) predicate, etc. Some features of $\mathcal{G}$ are exploited by some of those inference rules: those features are enumerated below.

Existential quantification is written explicitly into the first rule for patterns. It is possible (as is done in other rules) to drop the explicit existential quantifier and

Evaluation judgment $\vdash \mathrm{M} \Downarrow v$

$$\frac{}{\vdash \lambda x.\,\mathrm{M} \Downarrow \lambda x.\,\mathrm{M}} \qquad \frac{\vdash \forall i \in [1;n],\ \mathrm{M}_i \Downarrow v_i}{\vdash \mathrm{C}(\mathrm{M}_1,\ldots,\mathrm{M}_n) \Downarrow \mathrm{C}(v_1,\ldots,v_n)} \qquad \frac{\vdash \texttt{rec}\ x\ \texttt{in}\ \mathrm{M}[\mathrm{M}/x] \Downarrow v}{\vdash \texttt{rec}\ x\ \texttt{in}\ \mathrm{M} \Downarrow v}$$

$$\frac{\vdash \mathrm{M} \Downarrow \lambda x.\,\mathrm{R} \quad \vdash \mathrm{N} \Downarrow \mathrm{U} \quad \vdash \mathrm{R}[\mathrm{U}/x] \Downarrow v}{\vdash \mathrm{M}\ \mathrm{N} \Downarrow v} \qquad \frac{\vdash \mathrm{M} \Downarrow v \qquad \mathrm{X} \notin v}{\vdash \texttt{new}\ \mathrm{X}\ \texttt{in}\ \mathrm{M} \Downarrow v}$$

$$\frac{\vdash \mathrm{M} \Downarrow v' \quad \vdash \mathrm{R}[v'/x] \Downarrow v}{\vdash \texttt{let}\ x = \mathrm{M}\ \texttt{in}\ \mathrm{R} \Downarrow v} \qquad \frac{\vdash \mathrm{M} \Downarrow \mathrm{Y}\backslash\mathrm{N} \quad \vdash \mathrm{N}[\mathrm{X}/\mathrm{Y}] \Downarrow v}{\vdash \mathrm{M}\ @\ \mathrm{X} \Downarrow v} \qquad \frac{\vdash \mathrm{M} \Downarrow v}{\vdash \mathrm{X}\backslash\mathrm{M} \Downarrow \mathrm{X}\backslash v}$$

$$\frac{\vdash \mathrm{M} \Downarrow v_\mathrm{M} \quad \mathsf{clause}\ v_\mathrm{M}\ \mathrm{R}_1\ \mathrm{N} \quad \mathrm{N} \Downarrow v}{\vdash \texttt{match}\ \mathrm{M}\ \texttt{with}\ \mathrm{R}_1 \mid \ldots \mid \mathrm{R}_n \Downarrow v}$$

$$\frac{\vdash \mathrm{M} \Downarrow v_\mathrm{M} \quad \neg(\exists \mathrm{N}, \mathsf{clause}\ v_\mathrm{M}\ \mathrm{R}_1\ \mathrm{N}) \quad \vdash \texttt{match}\ \mathrm{M}\ \texttt{with}\ \mathrm{R}_2 \mid \ldots \mid \mathrm{R}_n \Downarrow v}{\vdash \texttt{match}\ \mathrm{M}\ \texttt{with}\ \mathrm{R}_1 \mid \ldots \mid \mathrm{R}_n \Downarrow v}$$

Match clause judgment $\vdash \mathsf{clause}\ v\ \mathrm{R}\ \mathrm{N}$

$$\frac{\vdash \exists w, \mathsf{clause}\ v\ (\mathrm{R}\ w)\ \mathrm{N}}{\vdash \mathsf{clause}\ v\ (\texttt{all}\ x.\,\mathrm{R}\ x)\ \mathrm{N}} \qquad \frac{\vdash \mathsf{matches}\ v\ \mathrm{P} \qquad \vdash (\lambda z_1 \ldots z_m.\ p \Longrightarrow u) \trianglerighteq (\mathrm{P} \Longrightarrow \mathrm{U})}{\vdash \mathsf{clause}\ v\ (\mathsf{nab}\ z_1\ \texttt{in}\ \ldots \mathsf{nab}\ z_m\ \texttt{in}\ p \Longrightarrow u)\ \mathrm{U}}$$

Value / Pattern identity judgment $\vdash \mathsf{matches}\ \mathrm{M}\ \mathrm{P}$

$$\frac{}{\vdash \mathsf{matches}\ v\ v} \qquad \frac{}{\vdash \mathsf{matches}\ \mathrm{X}\ \mathrm{X}} \qquad \frac{\vdash \forall i \in [1,n], \mathsf{matches}\ v_i\ p_i}{\vdash \mathsf{matches}\ (\mathrm{C}(v_1 \ldots v_n))\ (\mathrm{C}(p_1 \ldots p_n))}$$

$$\frac{}{\vdash \mathsf{matches}\ v\ \_} \qquad \frac{\vdash \mathsf{matches}\ v\ p}{\vdash \mathsf{matches}\ (\mathrm{X}\backslash v)\ (\mathrm{X}\backslash p)} \qquad \frac{\vdash \mathsf{matches}\ (\mathrm{X}\backslash v)\ p}{\vdash \mathsf{matches}\ v\ (p\ @\ \mathrm{X})}$$

Figure 3.16: A natural semantic specification of evaluation.

```
(rec size \ lam term \
  match term
  [(all m \ all n \
     (pvariant c_App [(pvar n), (pvar m)]) ==>
     (special add [(special add [(int 1),
                    (app size n)]), (app size m)])),
   (all r \ (pvariant c_Abs [pvar r]) ==>
      (special add [(int 1),
                (new X \ app size (arobase r X))])),
   (nab X \ (pnom X) ==> (int 1))])
```

Figure 3.17: The translation in λProlog of the `size` program.

instead have the quantification be implicitly universally quantified around the entire rule. We write it explicitly here to highlight the fact that solving the problem of finding instances of pattern variables in matching rules is lifted to the general problem of finding substitution terms in $\mathcal{G}$.

The proof rules for natural semantics are nondeterministic in principle. Consider attempting to prove that t, a term of arity type $\mathbf{0}$, has a value: that is, $\exists v, t \Downarrow v$. It can be the case that no proof exists or that there might be several proofs with different values for v. No proofs are possible if, for example, the condition in a conditional phrase does not evaluate to a boolean or if there are insufficient match rules provided to cover all the possible values given to a match expression. Ultimately, we will want to provide a static check that could issue a warning if the rules listed in a match expression are not exhaustive. Conversely, the variables introduced by `all` and `nab` in patterns may have several satisfying values, if they are not used in the pattern itself, or only in flexible occurrences (see Section 5.2).

The *nominal abstraction* of $\mathcal{G}$ is directly invoked to solve pattern matching in which nominals are explicitly abstracted using the nab binding construction. When attempting to prove the judgment $\vdash$ clause T R*ule* U, the inference rules in Figure 3.16 eventually lead to an attempt to prove in $\mathcal{G}$ an existentially quantified nominal abstraction of the form

$$\exists x_1 \ldots \exists x_n[(\lambda z_1 \ldots \lambda z_m.(\underbrace{p \Longrightarrow u}_{t})) \trianglerighteq (\underbrace{\mathrm{P} \Longrightarrow \mathrm{U}}_{s})].$$

Here, the arrow $\Longrightarrow$ is simply a formal (syntactic) pairing operator, expecting a pattern on the left and a term on the right. The schema variables $x_1, \ldots, x_n$ can appear free only in p and u: furthermore, if any of these variables are free in s they must be free in t. Also, if any of the variables $z_1, \ldots, z_m$ are free in s they are also free in t. While the variables $x_1, \ldots, x_n$ cannot appear more than once in t, the variables $z_1, \ldots, z_m$ are not restricted in this fashion. In order to prove the formula $\exists \bar{x}(\lambda \bar{z}.t) \trianglerighteq s$, one must find a collection of distinct nominals $\bar{c}$ and witness terms $\bar{t}$ that do not contain any of the elements of $\bar{c}$ such that $[\bar{t}/\bar{x}, \bar{c}/\bar{z}]t = s$ [Wan+13].

The last ingredient of our pattern-matching rule is the judgment ($\vdash$ matches T P) that checks that a term or value T is indeed matched by a pattern P. Patterns and terms form two distinct syntactic categories, the judgment relates pattern-formers to the corresponding term-formers. Nominals are embedded in patterns by the pnom(c) pattern-former, which matches a corresponding nominal—the condition

$$\begin{array}{rcll} \pi & ::= & \Box & \text{paths} \\ & | & \mathrm{C}(i.\pi) & \\ & | & \mathrm{X}\backslash\pi & \\ & | & \pi \ @ \ \mathrm{X} & \end{array}$$

Rigid occurrence in a value $v' \in_\pi v$

$$\frac{}{v' \in_\Box v'} \qquad \frac{v' \in_\pi v_k}{v' \in_{\mathrm{C}(k.\pi)} \mathrm{C}(v_1, \ldots, v_n)} \qquad \frac{v' \in_\pi v}{v' \in_{(\mathrm{Y}\backslash\pi)} (\mathrm{Y}\backslash v)} \qquad \frac{v' \in_\pi \mathrm{Y}\backslash v}{v' \in_{(\pi \ @ \ \mathrm{Y})} (v)}$$

Rigid occurrence in a pattern $p' \in_\pi p$

$$\frac{}{p' \in_\Box p'} \qquad \frac{p' \in_\pi p_k}{p' \in_{\mathrm{C}(k.\pi)} \mathrm{C}(p_1, \ldots, p_n)} \qquad \frac{p' \in_\pi p}{p' \in_{(\mathrm{Y}\backslash\pi)} (\mathrm{Y}\backslash p)} \qquad \frac{p' \in_\pi p}{p' \in_{(\pi \ @ \ \mathrm{Y})} (p \ @ \ \mathrm{Y})}$$

Rigid occurrence in a clause $p' \in_\pi \mathrm{R}$

$$\frac{p' \in_\pi \mathrm{R}}{p' \in_\pi \mathsf{nab}\ \mathrm{Y}\ \mathsf{in}\ \mathrm{R}} \qquad \frac{p' \in_\pi \mathrm{R}}{p' \in_\pi \mathtt{all}\ x.\mathrm{R}} \qquad \frac{p' \in_\pi p}{p' \in_\pi p \to \mathrm{M}}$$

Figure 3.18: Paths in values and patterns

nominal(c) can be expressed in terms of nominal abstraction $(\lambda x.\, x) \trianglerighteq c$. Term variables introduced by all are embedded in patterns by the pvar pattern-former, and they can match any term x—note that in this rule, x denotes an arbitrary term, substituted for a term variable by the all-handling rule.

7 Formal properties of MLTS

We list here some formal properties about MLTS. Some of the proofs refer to substitutions lemmas and we do not provide detailed proofs of these. However, when type checking is specified using λ-tree syntax and implemented in λProlog, then it is typical for substitution lemmas to be provable using simple arguments as it is done in Section 6.4 of Miller's JAR paper "Mechanized metatheory revisited" [Mil18b].

The properties we are going to prove are *Type preservation*, *Determinacy of evaluation* and the fact that *Nominals never escape their scopes*.

Lemma 7.1 (Same path same type)**.** *Consider the definition of* paths *given in Fig. 3.18. For any given* Γ, π, A*, all possibles values, clauses and patterns of type* A *that are well-typed under* Γ *agree on the type of the values at path* π*. In other words, there exists a type* C *such that all of the following hold:*

$$\begin{array}{lrcl} \forall v, v', & \Gamma \vdash v : \mathrm{A} \ \wedge \ v' \in_\pi v & \Longrightarrow & \Gamma \vdash v' : \mathrm{C} \\ \forall p', \mathrm{R}, \mathrm{B}, & \Gamma \vdash \mathrm{A} : \mathrm{R} : \mathrm{B} \ \wedge \ p' \in_\pi \mathrm{R} & \Longrightarrow & \Gamma \vdash p' : \mathrm{C} \\ \forall p, p', & \Gamma \vdash p : \mathrm{A} \ \wedge \ p' \in_\pi p & \Longrightarrow & \Gamma \vdash p' : \mathrm{C} \end{array}$$

Proof Let A be a type, π a path and Γ a typing environment. We will proceed by induction over π. We only show the first subcase of each case, the others being similar.

- Case $\pi = \square$
  - Let v and v' be two values such that $\Gamma \vdash v : \mathrm{A}$ and $v' \in_{\square} v$. Thus $v = v'$ and $\Gamma \vdash v' : \mathrm{A}$.
- Case $\pi = \mathrm{C}(i.\pi')$ where C is a type constructor of type $\mathrm{A}_1 \to ... \to \mathrm{A}_n \to \mathrm{A}$ and π' another path.
  - For any values v and v' such that $\Gamma \vdash v : \mathrm{A}$ and $v' \in_{\mathrm{C}(i.\pi')} v$, we can deduce from this last fact that there exist values $v_1, ..., v_n$ of types $\mathrm{A}_1, ..., \mathrm{A}_n$ such that $v = \mathrm{C}(v_1, ..., v_n)$. Having $\Gamma \vdash v_i : \mathrm{A}_i$ and $v' \in_{\pi'} v_i$ we can use the induction hypothesis with the path π' and obtain $\Gamma \vdash v' : \mathrm{A}_i$.
- Case $\pi = \mathrm{Y} \backslash \pi'$
  - Let v and v' be two values such that $\Gamma \vdash v : \mathrm{A}$ and $v' \in_{\mathrm{X} \backslash \pi'} v$. Following rules in Fig. 3.18, there exists some value v'' such that v is of the form $\mathrm{Y} \backslash v''$ and $v' \in_{\pi'} v''$. Moreover, by inversion of typing we have that there exists types B and B′ such that $\mathrm{A} = \mathrm{B} \Rightarrow \mathrm{B}'$ and $\Gamma, \mathrm{Y} : \mathrm{A} \vdash v'' : \mathrm{B}$.
- Case $\pi = \pi' \ @ \ \mathrm{Y}$
  - For any values v and v' such that $\Gamma \vdash v : \mathrm{A}$ and $v' \in_{(\pi' \ @ \ \mathrm{Y})} v$, we know from this last fact that $v' \in_{\pi'} \mathrm{Y} \backslash v$ and by induction hypothesis there exists a type C such that $\Gamma \vdash v' : \mathrm{C}$.

□

Lemma 7.2 (Matches matches)**.** *For any values v, v' and a pattern p such that all variables of p have been replaced by values (it is always the case in* matches*' definition), if $v' \in_{\pi} p$ and* matches v p *holds, then $v' \in_{\pi} v$.*

Proof Let v and v' be values and p a pattern such that $v' \in_{\pi} p$. If $\pi = \square$ then the result is immediate, otherwise we proceed by induction on the definition of matches.

- Cases
$$\frac{}{\vdash \mathsf{matches}\ v\ v} \qquad \text{and} \qquad \frac{}{\vdash \mathsf{matches}\ \mathrm{X}\ \mathrm{X}}$$
  Then $p = v$ and the result is immediate.
- Case
$$\frac{}{\vdash \mathsf{matches}\ v\ \_}$$
  It is possible to have $\_ \in_{\square} \_$ but never $v \in_{\square} \_$ for a value v.
- Case
$$\frac{\vdash \forall i \in [1, n], \mathsf{matches}\ v_i\ p_i}{\vdash \mathsf{matches}\ (\mathrm{C}(v_1 \ldots v_n))\ (\mathrm{C}(p_1 \ldots p_n))}$$
  We have $\pi = \mathrm{C}(i.\pi_i)$ and $v' \in_{\pi_i} p_i$ and by induction hypothesis, $v' \in_{\pi_i} v_i$ and thus $v' \in_{\pi = \mathrm{C}(i.\pi_i)} v$.

- Case
$$\frac{\vdash \mathsf{matches}\ v''\ p}{\vdash \mathsf{matches}\ (\mathrm{X}\backslash\, v'')\ (\mathrm{X}\backslash\, p)}$$
In that case $v = \mathrm{X}\backslash\, v''$ and $v' \in_\pi \mathrm{X}\backslash\, p$. By inversion of the path judgment, we have that $\pi = \mathrm{X}\backslash\, \pi'$ and $v' \in_{\pi'} p$. By induction hypothesis we have $v' \in_{\pi'} v''$ and going back: $v' \in_\pi \mathrm{X}\backslash\, v'' = v$
- Case
$$\frac{\vdash \mathsf{matches}\ (\mathrm{Y}\backslash\, v)\ p}{\vdash \mathsf{matches}\ v\ (p\ @\ \mathrm{Y})}$$
We have $\pi = \pi'\ @\ \mathrm{Y}$ and $v' \in_{(\pi'\ @\ \mathrm{Y})} p\ @\ \mathrm{Y}$ implies $v' \in_{\pi'} p$. By induction hypothesis we have that $v' \in_{\pi'} \mathrm{Y}\backslash\, v$, that is $v' \in_\pi v$.

□

Lemma 7.3 (Clause matches). *For any values v and v', rule* R *and term* U*, if $v' \in_\pi$* R *and* clause v R U *holds, then $v' \in_\pi v$*

Proof Le v and v' be values, R a rule and U a term such that $v' \in_\pi \mathrm{R}$. We proceed by induction on the definition of clause.

- Case
$$\frac{\vdash \exists \mathrm{W}, \mathsf{clause}\ v\ (\mathrm{R}'\ \mathrm{W})\ \mathrm{U}}{\vdash \mathsf{clause}\ v\ (\mathtt{all}\ x.\, \mathrm{R}'\ x)\ \mathrm{U}}$$
Here $v' \in_\pi \mathtt{all}\ x.\, \mathrm{R}'\ x$. By definition of paths, $v' \in_\pi \mathrm{R}'$ and we can use the induction hypothesis, obtaining that $v' \in_\pi v$.
- Case
$$\frac{\vdash \mathsf{matches}\ v\ \mathrm{P} \qquad \vdash (\lambda z_1 \ldots z_m.\ p \Longrightarrow u) \trianglerighteq (\mathrm{P} \Longrightarrow \mathrm{U})}{\vdash \mathsf{clause}\ v\ (\mathsf{nab}\ z_1\ \mathtt{in}\ \ldots \mathsf{nab}\ z_m\ \mathtt{in}\ p \Longrightarrow u)\ \mathrm{U}}$$
Here $v' \in_\pi (\mathsf{nab}\ z_1\ \mathtt{in}\ \ldots \mathsf{nab}\ z_m\ \mathtt{in}\ p \Longrightarrow u)$. By the definition of paths, we have that $v' \in_\pi p$. By nominal abstraction definition, we have that for each $i \in [1..m]$ there exists c_i such that $\mathrm{P} = p[c_i/z_i]$ for all the i. Thus, by substitution, we have that $v' \in_\pi \mathrm{P}$ and because matches v P, Lemma 7.2 gives us that $v' \in_\pi v$.

Theorem 7.1 (Nominals do not escape). *Let* E *be the explicit syntax of an MLTS program that does not contain any free nominal. If* $\vdash \mathrm{E} \Downarrow \mathrm{V}$ *is provable then* V *does not contain any free nominals.*

Proof We proceed by induction over the structure of proofs in our natural semantic. Only three cases are "dangerous" with regard to nominal escaping: new, \ and nab:

- The case
$$\frac{\vdash \mathrm{M} \Downarrow v \qquad \mathrm{X} \notin v}{\vdash \mathsf{new}\ \mathrm{X}\ \mathtt{in}\ \mathrm{M} \Downarrow v}$$
is immediate as an explicit check is specified that the nominal X should not appear free in the value v.
- Case
$$\frac{\vdash \mathrm{M} \Downarrow v}{\vdash \mathrm{X}\backslash\, \mathrm{M} \Downarrow \mathrm{X}\backslash\, v}$$
is also quite easy: the binder is still present in the value so X is not free in it.

- The case

$$\frac{\vdash \mathsf{matches}\ v\ \mathrm{P} \qquad \vdash (\lambda z_1 \ldots z_m.\ p \Longrightarrow u) \trianglerighteq (\mathrm{P} \Longrightarrow \mathrm{U})}{\vdash \mathsf{clause}\ v\ (\mathsf{nab}\ z_1\ \mathtt{in}\ \ldots \mathsf{nab}\ z_m\ \mathtt{in}\ p \Longrightarrow u)\ \mathrm{U}}$$

  is more subtle. We need to make sure that all the nominals that will be introduced by the nominal abstraction process are already nominals of v. But we can indeed show that this is the case: because each pattern nominal z_i must have a rigid occurrence (see Section 5.2), in p, there exists π_i such that $z_i \in_{\pi_i} p$. In addition, we know by the definition of nominal abstraction that there exist $c_1, \ldots, c_n$ such that $\mathrm{P} = p[c_1, \ldots, c_n/z_1, \ldots, z_n]$. By substitution, $z_i \in_{\pi_i} p$ implies $c_i \in_{\pi_i} \mathrm{P}$. And with $\mathsf{matches}\ v\ \mathrm{P}$ we can use Lemma 7.2 and we obtain $c_i \in_{\pi_i} v$. Thus all nominals used in nominal abstractions are already in v and no new free nominal will be created.

□

Theorem 7.2 (Type preservation). *If the typing judgment* $\Gamma \vdash \mathrm{E} : \mathrm{A}$ *and the evaluation judgment* $\vdash \mathrm{E} \Downarrow \mathrm{V}$ *holds, then so does* $\Gamma \vdash \mathrm{V} : \mathrm{A}$.

Proof This proof is referring to the natural semantic of Fig. 3.16 and the corresponding typing system of Fig. 3.15. We proceed by induction over the evaluation derivation $\vdash \mathrm{E} \Downarrow \mathrm{V}$.

- Case

$$\frac{\vdash \mathrm{M} \Downarrow \lambda x.\mathrm{R} \quad \vdash \mathrm{N} \Downarrow \mathrm{U} \quad \vdash \mathrm{U}[\mathrm{R}/x] \Downarrow v}{\vdash \mathrm{M}\ \mathrm{N} \Downarrow v}$$

  By hypothesis, $\vdash (\mathrm{M}\ \mathrm{N}) : \mathrm{B}$ and given the corresponding typing rule we know (by inversion) that there exists a type A such that $\vdash \mathrm{M} : \mathrm{A} \to \mathrm{B}$ and $\vdash \mathrm{N} : \mathrm{B}$. Thus we know by induction hypothesis that $\vdash (\lambda x.\mathrm{R}) : \mathrm{A} \to \mathrm{B}$ and $\vdash \mathrm{U} : \mathrm{B}$. An appropriate substitution lemma allows to conclude that $\vdash \mathrm{U}[\mathrm{R}/x] : \mathrm{B}$. Another use of the induction hypothesis gives us $\vdash v : \mathrm{B}$.

- Case

$$\frac{\vdash \mathrm{M} \Downarrow v \qquad \mathrm{X} \notin v}{\vdash \mathsf{new}\ \mathrm{X}\ \mathtt{in}\ \mathrm{M} \Downarrow v}$$

  By hypothesis, $\Gamma \vdash (\mathsf{new}\ \mathrm{X}\ \mathtt{in}\ \mathrm{M}) : \mathrm{A}$ and $\vdash \mathsf{new}\ \mathrm{X}\ \mathtt{in}\ \mathrm{M} \Downarrow v$. By inversion of typing: $\Gamma, \mathrm{X} : \mathrm{A} \vdash \mathrm{M} : \mathrm{A}$ and thus by induction hypothesis $\Gamma, \mathrm{X} : \mathrm{A} \vdash v : \mathrm{A}$.

  Eventually, because $\mathrm{X} \notin v$, we obtain by strengthening that $\Gamma \vdash v : \mathrm{A}$.

- Like the two previous cases, most others follows the standard outline of a proof of type preservation. We jump to the most complex case: pattern matching.

- Case

$$\frac{\vdash \mathrm{M} \Downarrow v_{\mathrm{M}} \quad \mathsf{clause}\ v_{\mathrm{M}}\ \mathrm{R}_1\ \mathrm{N} \quad \mathrm{N} \Downarrow v}{\vdash \mathtt{match}\ \mathrm{M}\ \mathtt{with}\ \mathrm{R}_1 \mid \ldots \mid \mathrm{R}_n \Downarrow v}$$

  By hypothesis, $\vdash \mathtt{match}\ \mathrm{M}\ \mathtt{with}\ \mathrm{R}_1 \mid \ldots \mid \mathrm{R}_n : \mathrm{B}$ and given the corresponding typing rule we know (by inversion) that there exists a type A such that $\vdash \mathrm{M} : \mathrm{A}$ and $(\vdash \mathrm{A} : \mathrm{R}_i : \mathrm{B})_{i \in [1..n]}$ and by induction hypothesis we have $\vdash v_{\mathrm{M}} : \mathrm{A}$. We now need to prove the following lemma:

  Lemma 7.4. *If* $\mathsf{clause}\ v\ \mathrm{R}\ \mathrm{U}$ *and* $\Gamma \vdash v : \mathrm{A}$ *and* $\Gamma \vdash \mathrm{A} : \mathrm{R} : \mathrm{B}$ *then* $\Gamma \vdash \mathrm{U} : \mathrm{B}$

Proof: We proceed by induction on the definition of clause

– Case $\texttt{all}\ x.\,\mathrm{R}\ x$, we have:

$$\frac{\vdash \exists w, \mathsf{clause}\ v\ (\mathrm{R}\ w)\ \mathrm{N}}{\vdash \mathsf{clause}\ v\ (\texttt{all}\ x.\,\mathrm{R}\ x)\ \mathrm{N}} \qquad \frac{\Gamma, x : \mathrm{C} \vdash \mathrm{A} : \mathrm{R} : \mathrm{B}}{\Gamma \vdash \mathrm{A} : \texttt{all}\ x.\,\mathrm{R} : \mathrm{B}}$$

Because all pattern variables must occur once in a pattern (in concrete syntax pattern variable are implicitly quantified, thus a non-appearing pattern variable will not appear at all), we have $x \in_\pi (\mathrm{R}\ x)$ and thus $w \in_\pi (\mathrm{R}\ w)$. Then Lemma 7.3 gives us that $w \in_\pi v$ and the Lemma 7.1 allows us to conclude that $\Gamma \vdash x : \mathrm{C}$

– Case

$$\frac{\vdash \mathsf{matches}\ v\ \mathrm{P} \qquad \vdash (\lambda z_1 \ldots z_m.\ p \Longrightarrow u) \trianglerighteq (\mathrm{P} \Longrightarrow \mathrm{U})}{\vdash \mathsf{clause}\ v\ (\mathsf{nab}\ z_1\ \texttt{in}\ \ldots \mathsf{nab}\ z_m\ \texttt{in}\ p \Longrightarrow u)\ \mathrm{U}}$$

By hypothesis, $\Gamma \vdash v : \mathrm{A}$ and $\Gamma \vdash \mathrm{A} : \mathrm{R} : \mathrm{B}$. Because matches preserves the type we have $\Gamma \vdash \mathrm{P} : \mathrm{A}$, and by using the typing rule for $\mathsf{nab}\ z_1 \ldots\ \texttt{in}\ (p \Longrightarrow u)$ we also know that: $\Gamma, z_1 : \mathrm{C}_1 \ldots z_m : \mathrm{C}_m \vdash p : \mathrm{A}$ and $\Gamma, z_1 : \mathrm{C}_1 \ldots z_m : \mathrm{C}_m \vdash u : \mathrm{B}$.
In addition, the definition of *nominal abstraction* gives us that there exist $c_1 \ldots c_m$ such that

$$(\mathrm{P} \Longrightarrow \mathrm{U}) = (p \Longrightarrow u)[c_1 \ldots c_m / z_1 \ldots z_m] \qquad (\star)$$

To conclude (our goal is to show that $\Gamma \vdash \mathrm{U} : \mathrm{B}$) we need to prove that the $c_1 \ldots c_m$ have the same types in Γ than the $z_1 \ldots z_m$ in $\Gamma, z_1 : \mathrm{C}_1 \ldots z_m : \mathrm{C}_m$. Because all ∇-bound variables have a *rigid occurrence* (again, see Section 5.2), for each i there must exist a path π_i such that $z_i \in_{\pi_i} p$ and $c_i \in_{\pi_i} \mathrm{P}$. Applying the Lemma 7.1 we have that $(\Gamma, z_1 \ldots z_m)(z_i) = (\Gamma, z_1 \ldots z_m)(c_i)$ thus $\Gamma \vdash c_i : \mathrm{C}_i$.
Finally, another substitution lemma and equality $(\star)$ gives us $\Gamma \vdash \mathrm{U} : \mathrm{B}$, proving the lemma. □

By applying Lemma 7.3 to clause $v_\mathrm{M}\ \mathrm{R}_1\ \mathrm{N}$, we obtain that $\Gamma \vdash \mathrm{N} : \mathrm{B}$. Thus, using the induction hypothesis, $\Gamma \vdash v : \mathrm{B}$ □

Theorem 7.3 (Determinacy of evaluation)**.** *If* $\vdash \mathrm{E} \Downarrow \mathrm{V}$ *and* $\vdash \mathrm{E} \Downarrow \mathrm{U}$ *then* $\mathrm{V} = \mathrm{U}$.

Proof The proof of this theorem follows the usual outline. The main complicating difference from the standard approach is the more complex nature of pattern matching. The restrictions on patterns have been designed, however, to ensure determinacy: this is particularly true for the restriction on pattern variable (i.e., they have exactly one occurrence in the scope of the pattern variable's scope) and the restriction that every nab bound variable has at least one rigid occurrence in the left-hand-side of the pattern. We proceed by induction on the evaluation derivation. Most cases are simple so we focus only on the pattern matching here.

- Case

$$\frac{\vdash \mathrm{M} \Downarrow v_\mathrm{M} \quad \mathsf{clause}\ v_\mathrm{M}\ \mathrm{R}_1\ \mathrm{N} \quad \mathrm{N} \Downarrow v}{\vdash \texttt{match}\ \mathrm{M}\ \texttt{with}\ \mathrm{R}_1 \mid \ldots \mid \mathrm{R}_n \Downarrow v}$$

By hypothesis, $\vdash$ `match` M `with` R_1 | ... | $R_n \Downarrow v$ and $\vdash$ `match` M `with` R_1 | ... | $R_n \Downarrow u$. In both cases $\vdash$ M $\Downarrow v_M$ by induction hypothesis. But to go on we need to prove that given v and R, N is uniquely determined by clause v_M R N.

Lemma 7.5. *For any given value v and rule* R*, if there exist two terms* M *and* N *such that* clause v R N *and* clause v R M *then* N = M*.*

Proof We proceed by induction on the definition of clause .

- Case
$$\frac{\vdash \exists W, \mathsf{clause}\ v\ (R\ W)\ N}{\vdash \mathsf{clause}\ v\ (\mathtt{all}\ x.\ R\ x)\ N}$$
By hypothesis there exist M and N such that clause v (`all` x. R x) M and clause v (`all` x. R x) N. And thus there exist W and W′ such that clause v R W M and clause v R W′ N.
There exists a path pi such that x has path ?in R (that is, $x \in_\pi$ R). Thus, by substitution, we have that W $\in_\pi$ R W and W′ $\in_\pi$ R W′. Lemma 7.3 gives us that W $\in_\pi v$ and W′ $\in_\pi v$. Thus we must have W = W′ and we can use the induction hypothesis on (clause v (R W) ?) and obtain M = N.
- Case
$$\frac{\vdash \mathsf{matches}\ v\ P \qquad \vdash (\lambda z_1 \ldots z_m.\ p \Longrightarrow u) \trianglerighteq (P \Longrightarrow U)}{\vdash \mathsf{clause}\ v\ (\mathsf{nab}\ z_1\ \mathtt{in}\ \ldots \mathsf{nab}\ z_m\ \mathtt{in}\ p \Longrightarrow u)\ U}$$
By nominal abstraction definition, we have that for each $i \in [1..m]$ there exists c_i such that P = $p[c_i/z_i]$ and U = $u[c_i/z_i]$ for all the i. And because every nominal has a rigid occurrence in a pattern, for each i there exists a path π_i such that $z_i \in_{\pi_i} p$ and thus $c_i \in_{\pi_i}$ P. By using Lemma 7.2, we have that $c_i \in_{\pi_i} v$, meaning that all c_i are uniquely determined and thus U(= $u[c_i/z_i]$) is too. □

The lemma we just proved allows us to apply our induction hypothesis on $\vdash$ N $\Downarrow v$ and finish the proof of determinacy. □

8 Interpreters for MLTS

We have a prototype implementation of MLTS. A parser from our extended OCaml syntax and a transpiler that generates λProlog code are implemented in OCaml. A simple evaluator and type checker written in λProlog can then be used to execute and type check MLTS code. The implementation of the evaluator in λProlog is rather compact but not completely trivial since neither ∇-quantification nor nominal abstraction are native to λProlog: they needed to be implemented. Commented fragments of this interpreter can be found in Appendix B. Both the Teyjus [Qi+15] and the Elpi [Dun+15] implementations of λProlog can be used to execute the MLTS interpreter.

There is little about our prototype implementation that is focused on providing an efficient implementation of MLTS. Instead, the prototype is a useful device for exploring the exact meaning and possible uses of the new program features. Nevertheless, we can comment here briefly on some costs of the underlying system that will likely appear in any implementation of MLTS.

8.1 Nominal-escape checking

As we have mentioned in Section 6, nominals are not allowed to escape their scope during evaluation and quantifier alternation can be used to enforce this restriction at the logic level. When one implements the logic, one needs to implement (parts of) the unification of simply typed λ-terms [Hue75] and such unification is constantly checking that bound variable scopes are properly restricted. There are times, however, when the expensive check for escaping nominals are not, in fact, needed. In particular, it is possible to rewrite the inference rule in Figure 3.16 for the `new` binding operator as the following rule.

$$\frac{\vdash \nabla x.(\mathrm{E}\ x) \Downarrow (\mathrm{U}\ x) \qquad \mathrm{U} = \lambda x.\mathrm{V}}{\vdash \mathrm{new}\ \mathrm{E} \Downarrow \mathrm{V}}$$

Here, both U and V are quantified universally around the inference rule. Attempting a proof of the first premise can result in the construction of some (possibly large) value, say t such that $\vdash (\mathrm{E}\ x) \Downarrow t$ holds. We can immediately form the binding of $\mathrm{U} \mapsto \lambda x.t$ without checking the structure of t. The second premise is where the examination of t may need to take place: if x is free in t, then there is no substitution for V that makes $\lambda x.t$ equal to $\lambda x.\mathrm{V}$. This check can be expensive, of course, since one might in principle need to examine the entire structure of t to solve this second premise. There are many situations, however, where such an examination is not needed and they can be revealed by the typing system. For example, if the type of U is, say, `tm => int`, there should not be any possible way for an untyped λ-term to have an occurrence inside an integer. Furthermore, there are static methods for examining type declarations in order to describe if a type $\tau_1 \to \tau_2$ (for primitive types τ_1 and τ_2) can be inhabited by only vacuous λ-terms (see, for example, [Mil92]). Of course, if the types of τ_1 and τ_2 are the same (say, `tm`), then type information is not useful here and a check of the entire structure t might be necessary. Other static checks and program analyses might be possible as a way to reduce the costs of checking for escaping nominals: the paper [Pot07] includes such static checks albeit for a technically different functional programming language, namely FreshML [SPG03]. The actual implementation does not try to check for escaping nominals but will trigger a runtime error when it happens.

8.2 Binder mobility

We started this programming language project with the desire to treat binders in syntax as directly and naturally as possible. We approached this project by designing the MLTS language with more binders than, say, OCaml: it has not only the usual binders for building functions and for refactoring computation (via the `let` construction) but also new binders that are directly linked to binders in data (via the `new X in`, `nab X in`, and `X\` operators). Finally, the natural semantic of MLTS in $\mathcal{G}$ and its implementation in λProlog are all based on using logics that contain rich binding operators that go beyond the usual universal and existential quantifiers. It is worth noting that if one were to write MLTS programs that do not need to manipulate data structures containing bindings, then the new binding features of MLTS would not be needed and neither would the novel features of both $\mathcal{G}$ and λProlog. Thus, in a sense, binders have not been formally implemented in this story: instead, binders of one kind have been implemented and specified using binders in another system. We were able to complete a prototype implementation of MLTS

since we know how to implement the high-level logics, and those techniques can be applied directly to the natural semantic specification.

One way to view the processing of a binder is that one needs to first *open* the abstraction, process the result (by "freshening" the newly freed names), and then *close* the abstraction [Pot06]. In the setting of MLTS, it is better to view such processing as the *movement* of a binder: that is, the binder in a data structure actually gets re-identified with an actual binder in the programming language. As we illustrated in Section 3 with the following step-by-step evaluation

```
size (Abs (X\ (Abs (Y\ (App(X,Y))))));;
new X in 1 + (size (Abs (Y\ (App(X,Y)))));;
new X in 1 + new Y in 1 + (size (App(X,Y)));;
new X in 1 + new Y in 1 + 1 + (size X) + (size Y);;
new X in 1 + new Y in 1 + 1 + 1 + 1;;
```

the bound variable occurrences for `X` and `Y` simply move. It is never the case that a bound variable actually becomes free: instead, it just becomes bound elsewhere. Thus, our strategy for strengthening the expressiveness of MLTS over other ML-style languages has been to add to the language more binding sites to which bindings can move.

8.3 Costs of moving binders

As we have mentioned before, binders are able to move from, say, a term-level binding to a program-level binding by the use of β_0. In particular, if y is a binder that does not appear free in the abstraction $\lambda x.\mathrm{B}$ then the β_0 reduction of $(\lambda x.\mathrm{B})y$ causes the x binding in B to move and to be identified with the y binder in $\mathrm{B}[y/x]$. If one must actually do the substitution of y for x in B, a possibly large term (at least its spine) must be copied. However, there are some situations where this movement of a binding can be inexpensive. For example, consider again the following match rule for `size`.

```
| Abs(r) -> 1 + (new X in size (r @ X))
```

If we assume that the underlying implementation of terms use De Bruijn's nameless dummies, it is possible to understand the rewriting needed in applying this match clause to be a constant time operation. In particular, if `r` is instantiated with an abstraction then it's top-level constructor would indicate where a binder of value 0 points. If we were to compile the syntax `(r @ X)` as simply meaning that that top-level constant is stripped away, then a binder of value 0 in the resulting term would automatically point (move) to being bound by the `new X in` binder. While such a treatment of binder mobility without doing substitution is possible in many of our examples, it does not cover all cases. In general, a more involved scheme for implementing binder mobility must be considered. This kind of analysis and implementation of binder mobility is used in the Elpi implementation of λProlog [Dun+15].

8.4 A web frontend for the interpreter

When prototyping MLTS, our goal was to allow people interested in it to experiment with the language in the easiest way possible. As of today's date, the most obvious way to do that is to provide a website where one could write MLTS code and have

the results of evaluation appear directly in the browser. This kind of easy in-browser playground is more and more common. For example OCaml (try.ocamlpro.com), Rust (play.rust-lang.org), Scala (scastie.scala-lang.org) and the Abella prover (abella-prover.org) all provide such playgrounds.

The main difficulty in the design of such interface is the execution of the compiler, and two strategies exist to solve that issue. One possible way (used by Scala and Rust) is to send the source code entered in the client to a server which then compiles and executes the program and sends the answer back to the client. This method can work for any language or tool because any native compiler can be used on the server-side, but it has several drawbacks: it can be slow to send the code via the network and wait for the answer, the client needs to always be online and the server needs to be solid (or throttled) enough to not be overwhelmed by computational intensive code started by several clients. Another option (used by OCaml and Abella) is to compile and execute the code directly in the browser. Of course, to do so, one need to have a compiler written in javascript and thus this solution won't work with any arbitrary language. This option has the advantage that the server only serves static files and thus does not care about the potentially expensive computations. However computation power will be limited by the user's own machine power and the fact that everything will run in the browser's javascript sandbox.

We chose the second path for MLTS because it seemed the most user-friendly one and the existence of two other projects allowed us to do so: Elpi [Dun+15], a λProlog interpreter written in OCaml and `js_of_ocaml` [**js-of-ocaml**], an OCaml to javascript compiler. This allows one to interpret λProlog programs (such as our interpreter for MLTS) directly in the browser via javascript. This led to the design of the TryMLT website (`https://trymlts.github.io/`). Here is a list of its main components.

- A graphical interface (see Fig. 3.19) written in HTML and CSS using Bootstrap (getbootstrap.com) and jQuery (jquery.com) and relying on Ace (ace.c9.io) for the editor part.
- A transpiler from MLTS' concrete syntax to the explicit syntax syntax in λProlog. This transpiler is written in OCaml and compiled to javascript with `js_of_ocaml`.
- The Elpi λProlog interpreter, compiled to javascript with `js_of_ocaml` and bundled with our type-checking algorithm and interpreter for MLTS written in λProlog.

When the user clicks the "Run" button his code is translated to the abstract syntax using the transpiler in javascript and then it is type-checked and interpreted using the interpreter written in λProlog, itself interpreted by Elpi in javascript. All these computationally intensive tasks run in a separate "Web worker", that is an independent javascript thread, thus preventing the graphical interface from freezing while the code is executed. The codebase is hosted on GitHub and the static files are served from GitHub Pages.

9 Related work

The term *higher-order abstract syntax (HOAS)* was introduced in [PE88] to describe an encoding technique available in λProlog. A subsequent paper identified HOAS as a technique "whereby variables of an object language are mapped to variables in the metalanguage" [PS99]. When applied to functional programming, this latter

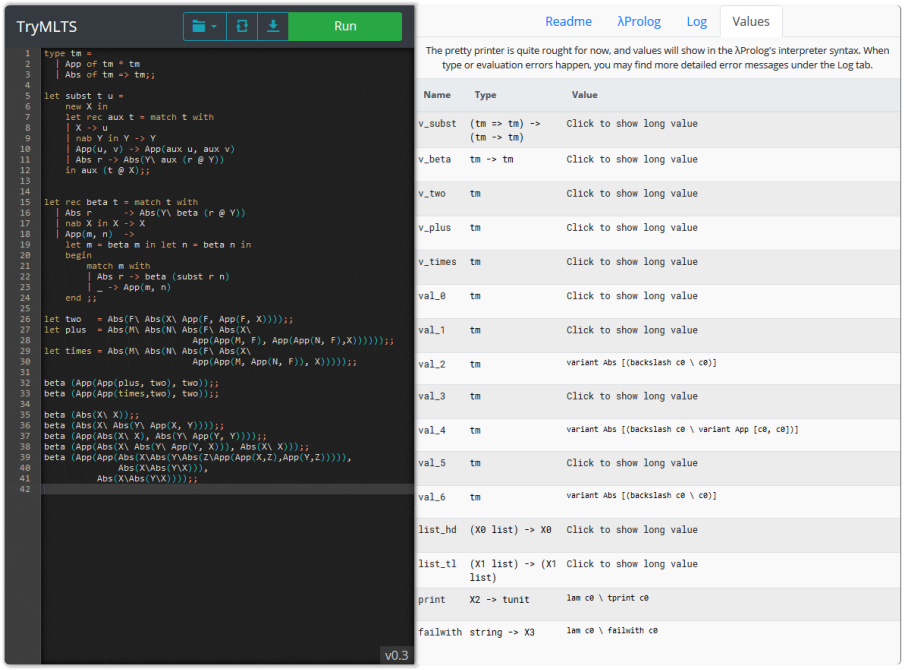

Figure 3.19: The graphical interface of TryMLTS

description implies the mapping of bindings in syntax to the bindings that create functions. Unfortunately, such encoding technique often lacks adequacy (since "exotic terms" can appear [DFH95]), and structural recursion can slip away [GP99]. The terms λ*-tree syntax* [MP99] and *binder mobility* [Mil04] were later introduced to describe the different and more syntactic approach that we have used here.

9.1 Systems with two arrow type constructors

The ML_λ [Mil90a] extension to ML is similar to MLTS in that it also contains two different arrow type constructors (-> and =>) and pattern matching was extended to allow for pattern variables to be applied to a list of distinct bound variables. The new operator of MLTS could be emulated by using the backslash operator and a "discharge" function. Critically missing from that language was anything similar to the nab binding of MLTS. Also, no formal specification and no implementation were ever offered. Licata & Harper [LH09] have used the universe feature of Agda 2 to provide an implementation of bindings in data structures that also relies on supporting two different implications-as-types that is found in MLTS and in ML_λ.

Nominals and nominal abstraction, in the sense used in this paper, were first conceived, studied, and implemented as part of the Abella theorem prover [Bae+14]. Although the design of Abella does not use the $\trianglerighteq$ relation directly, the notion of "∇ in the head" of definitions is essentially equivalent to having the $\trianglerighteq$ relation in the logic. While Abella only has one arrow type constructor, that arrow type essentially maps to the => of MLTS: this is possible in Abella since computation is performed at the level of relations and not functions. As a result, the function type arrow -> of MLTS and OCaml is not needed. Thus the distinction mentioned in [LH09] between

an arrow for computation and an arrow for binding is, in fact, also present in Abella, although computations are not represented functionally.

9.2 Systems with one arrow type constructor

The Delphin design is probably the closest to MLTS, in particular [SPS05] introduced a programming-language version of the ∇ quantifier from [MT05], whose usage is related to the ∇ of MLTS. In Delphin, ∇ introduces normal term variables (there is no separate class of nominal constants), while MLTS presents nominals as closer to datatype constructors, with a natural usage in pattern-matching.

Delphin makes nominal-escape errors impossible at runtime by imposing a static discipline to prevent them, while MLTS allows runtime failure in order to allow for more experimentation. The original proposal in [SPS05] uses a type modality that imposes a strict FIFO discipline on free variables. This discipline was found too constraining; [PS08a] completely eschews a `new` construct (its $\nu x.\, e$ binder actually corresponds to nominal abstraction `X\e` in MLTS), and [PS08b] uses a type-based restriction (type subordination), only allowing to introduce a fresh nominal in expressions whose return types only contains values that cannot contain this nominal. This discipline accepts some examples from our paper, for example `size` in Figure 3.17 and `id` in Figure 3.7, but rejects other (safe) programs, such as the second and third one-liner examples of Section 3.2. Richer static disciplines have been proposed by the FreshML community [PG00; Pot07], but they add complexity, and for example interact poorly with the introduction of mutable state; MLTS is an experimental design aiming for expressivity, so we decided to allow dynamic escape failures instead.

Beluga allows the programmer to use both dependent types and recursive definitions as well as an integrated notion of context (along with a method to describe certain invariants using *context schema*). Static checks of Beluga programs can be used to prove that formal correctness of Beluga programs (commonly by proving that a given piece of program code is, in fact, a total function). As a result, a checked Beluga program is often a formal proof. Since a wide range of formal systems can be encoded naturally using LF terms, Beluga programs can be used for both programming with and reasoning about the meta-theory of those formal systems. Since bindings and contexts are part of the vocabulary of Beluga, these formal proofs can capture the metatheory of logical and computational systems (such as natural deduction proof systems and the operational semantics of rich programming languages). The goal of MLTS is intended only to support programming and not directly reasoning: just as with, say, OCaml, the intent of new features of MLTS is only to support the manipulation of syntax containing bindings. A possibly interesting comparison between MLTS and Beluga might be explored by using typing and contexts in the latter in a mostly trivial way. It is likely that Beluga could code most MLTS programs although using different primitives.

9.3 Systems using nominal logic

The FreshML [SPG03] and CαML [Pot06] functional programming languages provide an approach to names based on nominal logic [Pit03]. In a sense, these two programming languages provide for an abstract treatment of names and naming. Once naming is available, binding structures can also be implemented. In a sense, the design of these two ML-variants are also more ambitious than the design goal intended for MLTS: in the latter, we were not focused on naming but just bindings.

The recent paper [FP17] introduces a syntactic framework that treats bindings as primitives. That framework is then integrated with various tools and with the framework of contextual types (similar to that found in Beluga) in order to provide a programmer of, say, OCaml with sophisticated tools for the manipulation of syntax and binders. A possible future target for MLTS could be to provide such tools more directly in the language itself.

9.4 Challenge problems and benchmarks

Genuine comparisons between different programming languages are generally hard to achieve. For example, in the area of logical frameworks and related theorem provers, there are also a number of formal systems and computer implementations. In order to understand the relative merits of these different systems, challenge problems and benchmarks [Ayd+05; FMP15] have been proposed to help people sort out specific merits and challenges of one system relative to another. In depth comparisons of the programming languages described above will probably require similar in-depth comparisons on representative programming tasks.

10 Perspectives for MLTS

The λ-tree syntax approach to computing with syntax containing bindings has been successfully developed within the logic programming setting (in particular, in λProlog and Twelf). In this chapter we provided another example of how binding can be captured in a functional programming language. Most of the expressiveness of MLTS arises from its increased use of program-level binding. The sophistication needed to correctly exploit binders and quantifiers in MLTS is a skill most people have learned from using quantification in, for example, predicate logic.

We have presented a number of MLTS programs and we note that they are both natural and unencumbered by concerns about managing bound variable names. The few extensions to the standard ML syntax that are needed to provide such treatment of binders are simple and local by nature: we only added scoped sites where bindings can move. We also presented a typing discipline for MLTS as well as a formal specification of its natural semantic: this latter task was aided by being able to directly exploit a rich logic, called $\mathcal{G}$, that incorporates λ-tree syntax principles within quantificational logic. Finally, this natural semantic specification was (almost) directly implementable in a few hundred lines of λProlog. As a consequence, a prototype implementation is available for helping to judge the expressiveness of MLTS programs.

Of course, these are only the first steps of the development of a new language. We aim next to refine the description of the semantics and are actively working on a small-step version. We also hope to provide an abstract machine for MLTS following the method given in [HM90]. A more future-sighted project is the creation of a better reference implementation as a native compiler.

Conclusion

The logic $\mathcal{G}$ is a recent extension to Heyting arithmetic. In this thesis we gave two different examples of applying that logic and its proof theory to computations in logic, according to the declarative programming tradition: one in the logic programming settings and one in the functional programming setting.

In the first part we showed how recent developments in proof theory and especially proof search, namely the focusing technique and the $\mathcal{G}$-logic, can allow for a natural treatment of how functional computations can be performed using relational specification when those relational specifications are known to compute functions. This led us to propose a number of possible automatization for the Abella theorem prover.

The second part of this thesis leveraged $\mathcal{G}$-logic in a different way: inspiring the design of a new functional programming language which provides a natural treatment of bindings in datastructures. Adding new binding sites in datastructures and patterns, it allows for bindings to move across the program in a flexible manner, and the possibility of having "higher arity values" compensate for the obligation of having all variables bound at all times.

There is a more direct connection between these two parts than the use of $\mathcal{G}$-logic. In Chapter 2 we used focusing to describe how functional computations can be performed using relational specification when those relational specifications are known to compute functions. A different question to consider in this setting, however, is: can we transform the syntax of a relational specification of a function and produce a functional program in MLTS of that encoded function? A possible additional ingredient to such a transformation might also be a formal proof that the relational specification actually determines a function when we view the first two arguments as inputs and the third argument as output. Then, if the Abella specification contains `nabla`-in-the-head, the corresponding clause in the MLTS program would contains a `nab` binder. If the Abella specification contains a `nabla` in the body of a clause, then the corresponding clause in the MLTS program would contains either a `new` binder or an explicit λ-binder (the backslash). An interesting project would be to formalize such a transformation, generalizing to nominal-aware relations the work that have been done for Coq and Focalize [Tol13].

Finally, both of these designs followed the same process: they illustrate how staying close to logic and proof search make it possible to design useful computation strategies and imagine new frameworks for syntax and computation that feel natural and sound. Because the logic they are based on is clean, no hacks were involved in this process and correctness has been easier to establish because we could use higher level arguments.

Bibliography

[12] *The Abella Prover*. Available at http://abella-prover.org/. 2012.

[And09] Peter Andrews. "Church's Type Theory". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2009. Stanford University, 2009.

[And92] Jean-Marc Andreoli. "Logic Programming with Focusing Proofs in Linear Logic". In: *J. of Logic and Computation* 2.3 (1992), pp. 297–347. DOI: 10.1093/logcom/2.3.297.

[Ass+16] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. "Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory". http://www.lsv.ens-cachan.fr/~dowek/Publi/expressing.pdf. 2016. URL: http://www.lsv.ens-cachan.fr/~dowek/Publi/expressing.pdf.

[Ayd+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. "Mechanized Metatheory for the Masses: The POPLmark Challenge". In: *Theorem Proving in Higher Order Logics: 18th International Conference*. LNCS 3603. Springer, 2005, pp. 50–65.

[Bae+14] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. "Abella: A System for Reasoning about Relational Specifications". In: *Journal of Formalized Reasoning* 7.2 (2014), pp. 1–89. DOI: 10.6092/issn.1972-5787/4650. URL: http://jfr.unibo.it/article/download/4650/4137.

[Bae08] David Baelde. "A linear approach to the proof-theory of least and greatest fixed points". PhD thesis. Ecole Polytechnique, Dec. 2008.

[Bae12] David Baelde. "Least and greatest fixed points in linear logic". In: *ACM Trans. on Computational Logic* 13.1 (Apr. 2012), 2:1–2:44. DOI: 10.1145/2071368.2071370. URL: http://tocl.acm.org/accepted/427baelde.pdf.

[Bar84] Henk P. Barendregt. *The Lambda Calculus*. North Holland, 1984.

[Bar92] Henk Barendregt. "Lambda calculus with types". In: *Handbook of Logic in Computer Science*. Ed. by S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum. Vol. 2. Oxford, England: Oxford University Press, 1992, pp. 117–309.

[Bru72] Nicolaas Govert de Bruijn. "Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with an Application to the Church-Rosser Theorem". In: *Indagationes Mathematicae* 34.5 (1972), pp. 381–392.

[Bru79] N. G. de Bruijn. "Lambda Calculus Notation with Namefree Formulas Involving Symbols that Represent Reference Transforming Mappings". In: *Indag. Math.* 40.3 (1979), pp. 348–356.

[CD07] Denis Cousineau and Gilles Dowek. "Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo". In: *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*. Ed. by Simona Ronchi Della Rocca. Vol. 4583. LNCS. Springer, 2007, pp. 102–117.

[Cha11] Arthur Charguéraud. "The Locally Nameless Representation". In: *Journal of Automated Reasoning* (May 2011), pp. 1–46. DOI: 10.1007/s10817-011-9225-2.

[Chu40] Alonzo Church. "A Formulation of the Simple Theory of Types". In: *J. of Symbolic Logic* 5 (1940), pp. 56–68. DOI: 10.2307/2266170.

[CLR16] Kaustuv Chaudhuri, Leonardo Lima, and Giselle Reis. "Formalized Meta-Theory of Sequent Calculi for Substructural Logics". In: *Workshop on Logical and Semantic Frameworks, with Applications (LSFA)*. Post proceedings version to appear; Formalization https://github.com/meta-logic/abella-reasoning. 2016.

[CMS08] Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. "Canonical Sequent Proofs via Multi-Focusing". In: *Fifth International Conference on Theoretical Computer Science*. Ed. by G. Ausiello, J. Karhumäki, G. Mauri, and L. Ong. Vol. 273. IFIP. Springer, Sept. 2008, pp. 383–396.

[CRS91] Anthony S. K. Cheng, Peter J. Robinson, and John Staples. "Higher Level Meta Programming in Qu-Prolog 3: 0". In: *Logic Programming, Proceedings of the Eigth International Conference, Paris, France, June 24-28, 1991*. Ed. by Koichi Furukawa. MIT Press, 1991, pp. 285–298.

[CU04] James Cheney and Christian Urban. "Alpha-Prolog: A Logic Programming Language with Names, Binding, and Alpha-Equivalence". In: *Logic Programming, 20th International Conference*. Ed. by Bart Demoen and Vladimir Lifschitz. Vol. 3132. LNCS. Springer, 2004, pp. 269–283.

[DFH95] Joëlle Despeyroux, Amy Felty, and Andre Hirschowitz. "Higher-order abstract syntax in Coq". In: *Second International Conference on Typed Lambda Calculi and Applications*. Apr. 1995, pp. 124–138.

[DL06] R. Dyckhoff and S. Lengrand. "LJQ: a strongly focused calculus for intuitionistic logic". In: *Computability in Europe 2006*. Ed. by A. Beckmann and *et al.* Vol. 3988. LNCS. Springer, 2006, pp. 173–185.

[Dun+15] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. "ELPI: Fast, Embeddable, λProlog Interpreter". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Ed. by Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov. Vol. 9450. LNCS. Springer, 2015, pp. 460–468. DOI:

10.1007/978-3-662-48899-7_32. URL: http://dx.doi.org/10.1007/978-3-662-48899-7%5C_32.

[FMP15] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. "The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 2–A Survey". In: *J. of Automated Reasoning* 55.4 (2015), pp. 307–372.

[FP17] Francisco Ferreira and Brigitte Pientka. "Programs Using Syntax with First-Class Binders". In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Hongseok Yang. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 504–529. ISBN: 978-3-662-54433-4; 978-3-662-54434-1.

[Gac08] Andrew Gacek. "The Abella Interactive Theorem Prover (System Description)". In: *Fourth International Joint Conference on Automated Reasoning*. Ed. by A. Armando, P. Baumgartner, and G. Dowek. Vol. 5195. LNCS. Springer, 2008, pp. 154–161. URL: http://arxiv.org/abs/0803.2305.

[Gac09] Andrew Gacek. "A Framework for Specifying, Prototyping, and Reasoning about Computational Systems". PhD thesis. University of Minnesota, 2009.

[Gen35] Gerhard Gentzen. "Investigations into Logical Deduction". In: *The Collected Papers of Gerhard Gentzen*. Ed. by M. E. Szabo. Amsterdam: North-Holland, 1935, pp. 68–131. DOI: 10.1007/BF01201353.

[Gir91] Jean-Yves Girard. "A new constructive logic: classical logic". In: *Math. Structures in Comp. Science* 1 (1991), pp. 255–296. DOI: 10.1017/S0960129500001328.

[Gir92] Jean-Yves Girard. "A Fixpoint Theorem in Linear Logic". An email posting to the mailing list linear@cs.stanford.edu. Feb. 1992.

[GMN08] Andrew Gacek, Dale Miller, and Gopalan Nadathur. "Combining generic judgments with recursive definitions". In: *23th Symp. on Logic in Computer Science*. Ed. by F. Pfenning. IEEE Computer Society Press, 2008, pp. 33–44. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/lics08a.pdf.

[GMN11] Andrew Gacek, Dale Miller, and Gopalan Nadathur. "Nominal abstraction". In: *Information and Computation* 209.1 (2011), pp. 48–73. DOI: 10.1016/j.ic.2010.09.004.

[GMN12] Andrew Gacek, Dale Miller, and Gopalan Nadathur. "A two-level logic approach to reasoning about computations". In: *J. of Automated Reasoning* 49.2 (2012), pp. 241–273. DOI: 10.1007/s10817-011-9218-1. URL: http://arxiv.org/abs/0911.2993.

[GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Vol. 78. LNCS. Springer, 1979.

[Gor91] Michael J. C. Gordon. "Introduction to the HOL System". In: *Proceedings of the International Workshop on the HOL Theorem Proving System and its Applications*. Ed. by Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley. IEEE Computer Society, 1991, pp. 2–3.

[Gor94] A. Gordon. "A Mechanisation of Name-Carrying Syntax up to Alpha-Conversion". In: *International Workshop on Higher Order Logic Theorem Proving and its Applications*. Vol. 780. Lecture Notes in Computer Science. 1994, pp. 414–426.

[GP99] M. J. Gabbay and A. M. Pitts. "A new approach to abstract syntax involving binders". In: *14th Symp. on Logic in Computer Science*. IEEE Computer Society Press, 1999, pp. 214–224.

[Har09] John Harrison. "HOL Light: an overview". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 60–66.

[HB39] D. Hilbert and P. Bernays. *Grundlagen der Mathematik II*. Springer Verlag, 1939.

[Her95] Hugo Herbelin. "Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes". PhD thesis. Université Paris 7, 1995.

[HM17] Quentin Heath and Dale Miller. "A Proof Theory for Model Checking: An Extended Abstract". In: *Proceedings Fourth International Workshop on Linearity (LINEARITY 2016)*. Ed. by Iliano Cervesato and Maribel Fernández. Vol. 238. EPTCS. Jan. 2017. DOI: 10.4204/EPTCS.238.1.

[HM90] John Hannan and Dale Miller. "From Operational Semantics to Abstract Machines: Preliminary Results". In: *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*. Ed. by M. Wand. ACM. ACM Press, 1990, pp. 323–332.

[Hue75] Gérard Huet. "A Unification Algorithm for Typed λ-Calculus". In: *Theoretical Computer Science* 1 (1975), pp. 27–57.

[Kah87] Gilles Kahn. "Natural Semantics". In: *Proceedings of the Symposium on Theoretical Aspects of Computer Science*. Ed. by Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing. Vol. 247. LNCS. Springer, Mar. 1987, pp. 22–39.

[LH09] Daniel R. Licata and Robert Harper. "A Universe of Binding and Computation". In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP '09. Edinburgh, Scotland: ACM, 2009, pp. 123–134. ISBN: 978-1-60558-332-7. DOI: 10.1145/1596550.1596571. URL: http://doi.acm.org/10.1145/1596550.1596571.

[LM07] Chuck Liang and Dale Miller. "Focusing and Polarization in Intuitionistic Logic". In: *CSL 2007: Computer Science Logic*. Ed. by J. Duparc and T. A. Henzinger. Vol. 4646. LNCS. Springer, 2007, pp. 451–465. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/csl07liang.pdf.

[LM09] Chuck Liang and Dale Miller. "Focusing and Polarization in Linear, Intuitionistic, and Classical Logics". In: *Theoretical Computer Science* 410.46 (2009), pp. 4747–4768. DOI: 10.1016/j.tcs.2009.07.041.

[LR18] Rodolphe Lepigre and Christophe Raffalli. "Abstract Representation of Binders in OCaml using the Bindlib Library". In: Proceedings of the 13th International Workshop on *Logical Frameworks and Meta-Languages: Theory and Practice,* Oxford, UK, 7th July 2018. Ed. by Frédéric Blanqui and Giselle Reis. Vol. 274. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2018, pp. 42–56. DOI: 10.4204/EPTCS.274.4.

[Mil+91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. "Uniform Proofs as a Foundation for Logic Programming". In: *Annals of Pure and Applied Logic* 51.1–2 (1991), pp. 125–157.

[Mil04] Dale Miller. "Bindings, mobility of bindings, and the ∇-quantifier". In: *18th International Conference on Computer Science Logic (CSL) 2004*. Ed. by Jerzy Marcinkowski and Andrzej Tarlecki. Vol. 3210. LNCS. 2004, p. 24. DOI: 10.1007/978-3-540-30124-0_4.

[Mil18a] Dale Miller. "Mechanized Metatheory Revisited". In: *Journal of Automated Reasoning* (Oct. 2018). ISSN: 1573-0670. DOI: 10.1007/s10817-018-9483-3.

[Mil18b] Dale Miller. "Mechanized metatheory revisited". In: *Journal of Automated Reasoning* (2018). URL: https://hal.inria.fr/hal-01884210.

[Mil90a] Dale Miller. "An Extension to ML to Handle Bound Variables in Data Structures: Preliminary Report". In: *Proceedings of the Logical Frameworks BRA Workshop*. Available as UPenn CIS technical report MS-CIS-90-59. Antibes, France, June 1990, pp. 323–335. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/mll.pdf.

[Mil90b] Robin Milner. "Functions as processes". In: *Automata, Languages and Programming 17th Int. Coll.* Vol. 443. LNCS. Warwick University, UK: Springer, July 1990, pp. 167–180.

[Mil91] Dale Miller. "A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification". In: *J. of Logic and Computation* 1.4 (1991), pp. 497–536.

[Mil92] Dale Miller. "Unification under a mixed prefix". In: *Journal of Symbolic Computation* 14.4 (1992), pp. 321–358.

[MM00] Raymond McDowell and Dale Miller. "Cut-elimination for a logic with definitions and induction". In: *Theoretical Computer Science* 232 (2000), pp. 91–119. DOI: 10.1016/S0304-3975(99)00171-1.

[MM04] Conor McBride and James McKinna. "Functional pearl: I am not a number - I am a free variable". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*. Ed. by Henrik Nilsson. ACM, 2004, pp. 1–9. URL: http://doi.acm.org/10.1145/1017472.1017477.

[MN12] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012. DOI: 10.1017/CBO9781139021326.

[MP99] Dale Miller and Catuscia Palamidessi. "Foundational Aspects of Syntax". In: *ACM Computing Surveys* 31 (Sept. 1999). Ed. by Pierpaolo Degano, Roberto Gorrieri, Alberto Marchetti-Spaccamela, and Peter Wegner.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. "A Calculus of Mobile Processes, Part I". In: *Information and Computation* 100.1 (Sept. 1992), pp. 1–40.

[MT05] Dale Miller and Alwen Tiu. "A proof theory for generic judgments". In: *ACM Trans. on Computational Logic* 6.4 (Oct. 2005). Ed. by Phokion Kolaitis, pp. 749–783. DOI: 10.1145/1094622.1094628. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/tocl-nabla.pdf.

[Nip93] Tobias Nipkow. "Functional Unification of Higher-Order Patterns". In: *8th Symp. on Logic in Computer Science*. Ed. by M. Vardi. IEEE, June 1993, pp. 64–74.

[NM88] Gopalan Nadathur and Dale Miller. "An Overview of λProlog". In: *Fifth International Logic Programming Conference*. Seattle: MIT Press, Aug. 1988, pp. 810–827. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/iclp88.pdf.

[NPS90] Bengt Nordstrom, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory : an introduction*. International Series of Monographs on Computer Science. Oxford: Clarendon, 1990.

[OCa18] OCaml. http://ocaml.org/. 2018.

[Pau89] Lawrence C. Paulson. "The Foundation of a Generic Theorem Prover". In: *Journal of Automated Reasoning* 5 (Sept. 1989), pp. 363–397.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828. Springer Verlag, 1994.

[PD10] Brigitte Pientka and Joshua Dunfield. "Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)". In: *Fifth International Joint Conference on Automated Reasoning*. Ed. by J. Giesl and R. Hähnle. LNCS 6173. 2010, pp. 15–21.

[PE88] Frank Pfenning and Conal Elliott. "Higher-Order Abstract Syntax". In: *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, June 1988, pp. 199–208.

[PG00] A. M. Pitts and M. J. Gabbay. "A Metalanguage for Programming with Bound Names Modulo Renaming". In: *Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings*. Ed. by R. Backhouse and J. N. Oliveira. Vol. 1837. LNCS. Springer, Heidelberg, 2000, pp. 230–255.

[Pit03] Andrew M. Pitts. "Nominal Logic, A First Order Theory of Names and Binding". In: *Information and Computation* 186.2 (2003), pp. 165–193.

[Pot06] François Pottier. "An Overview of Cαml". In: *Proceedings of the ACM-SIGPLAN Workshop on ML (ML 2005)*. Vol. 148. Electr. Notes Theor. Comput. Sci. 2006, pp. 27–52. DOI: 10.1016/j.entcs.2005.11.039.

[Pot07] François Pottier. "Static name control for FreshML". In: *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. IEEE. 2007, pp. 356–365.

[PS08a] Adam Poswolsky and Carsten Schürmann. "Practical programming with higher-order encodings and dependent types". In: *Proceedings of the European Symposium on Programming (ESOP '08)*. 2008.

[PS08b] Adam Poswolsky and Carsten Schürmann. "System Description: Delphin - A Functional Programming Language for Deductive Systems". In: *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*. Ed. by A. Abel and C. Urban. Vol. 228. 2008, pp. 113–120.

[PS99] Frank Pfenning and Carsten Schürmann. "System Description: Twelf — A Meta-Logical Framework for Deductive Systems". In: *16th Conf. on Automated Deduction (CADE)*. Ed. by H. Ganzinger. LNAI 1632. Trento: Springer, 1999, pp. 202–206. DOI: 10.1007/3-540-48660-7_14.

[Qi+15] Xiaochu Qi, Andrew Gacek, Steven Holte, Gopalan Nadathur, and Zach Snow. *The Teyjus System – Version 2*. http://teyjus.cs.umn.edu/. 2015. URL: http://teyjus.cs.umn.edu/.

[Qia96] Zhenyu Qian. "Unification of higher-Order patterns in linear time and space". In: *J. of Logic and Computation* 6.3 (1996), pp. 315–341.

[San96] Davide Sangiorgi. "π-calculus, internal mobility and agent-passing calculi". In: *Theoretical Computer Science* 167.2 (1996), pp. 235–274.

[SC14] Mary Southern and Kaustuv Chaudhuri. "A Two-Level Logic Approach to Reasoning about Typed Specification Languages". In: *34th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Ed. by Venkatesh Raman and S. P. Suresh. Vol. 29. Leibniz International Proceedings in Informatics (LIPIcs). New Delhi, India: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dec. 2014, pp. 557–569. DOI: 10.4230/LIPIcs.FSTTCS.2014.557. URL: http://abella-prover.org/papers/fsttcs14lf.pdf.

[Sch06] Helmut Schwichtenberg. "Minlog". In: *The Seventeen Provers of the World*. Ed. by Freek Wiedijk. Vol. 3600. LNCS. Springer, 2006, pp. 151–157. DOI: 10.1007/11542384_19.

[Sch93] Peter Schroeder-Heister. "Rules of Definitional Reflection". In: *8th Symp. on Logic in Computer Science*. Ed. by M. Vardi. IEEE Computer Society Press. IEEE, June 1993, pp. 222–232. DOI: 10.1109/LICS.1993.287585.

[Sco70] Dana Scott. "Outline of a Mathematical Theory of Computation". In: *Proceedings, Fourth Annual Princeton Conference on Information Sciences and Systems*. Also, Programming Research Group Technical Monograph PRG–2, Oxford University. Princeton University, 1970, pp. 169–176.

[SP11] Robert J. Simmons and Frank Pfenning. *Weak Focusing for Ordered Linear Logic*. Tech. rep. CMU-CS-10-147. Carnegie Mellon University, Apr. 2011.

[SPG03] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. "FreshML: Programming with Binders Made Simple". In: *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*. ACM Press, Aug. 2003, pp. 263–274.

[SPS05] Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. "The nabla-calculus. Functional Programming with Higher-order Encodings". In: *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*. TLCA'05. 2005. DOI: 10.1007/11417170_25.

[TM12] Alwen Tiu and Alberto Momigliano. "Cut elimination for a logic with induction and co-induction". In: *Journal of Applied Logic* 10.4 (2012), pp. 330–367. DOI: 10.1016/j.jal.2012.07.007.

[TNM05] Alwen Tiu, Gopalan Nadathur, and Dale Miller. "Mixing Finite Success and Finite Failure in an Automated Prover". In: *Empirically Successful Automated Reasoning in Higher-Order Logics (ESHOL'05)*. Dec. 2005, pp. 79–98.

[Tol13] Pierre-Nicolas Tollitte. "Extraction of Certified Functional Code from Inductive Specifications". Theses. Conservatoire national des arts et metiers - CNAM, Dec. 2013. URL: https://tel.archives-ouvertes.fr/tel-00968607.

[Tro73] Anne Sjerp Troelstra, ed. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Vol. 344. Lecture Notes in Mathematics. Springer, 1973.

[Urb08] Christian Urban. "Nominal Reasoning Techniques in Isabelle/HOL". In: *Journal of Automated Reasoning* 40.4 (2008), pp. 327–356.

[VM10] Alexandre Viel and Dale Miller. "Proof search when equality is a logical connective". Presented to the International Workshop on Proof-Search in Type Theories. July 2010. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/unif-equality.pdf.

[Wan+13] Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek, and Gopalan Nadathur. "Reasoning about Higher-Order Relational Specifications". In: *Proceedings of the 15th International Symposium on Princples and Practice of Declarative Programming (PPDP)*. Ed. by Tom Schrijvers. Madrid, Spain, Sept. 2013, pp. 157–168. DOI: 10.1145/2505879.2505889. URL: http://chaudhuri.info/papers/draft13hhw.pdf.

[ZS15] Beta Ziliani and Matthieu Sozeau. "A unification algorithm for Coq featuring universe polymorphism and overloading". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 179–191. DOI: 10.1145/2784731.2784751.

Appendix A

A unification algorithm in MLTS

The following commented example presents a direct implementation of the Robinson unification algorithm in MLTS. Due to its exponential nature and to the relative inefficiency of the current toy implementation of MLTS, even the small examples present on the website can take a long time to execute.

```
(* List concatenation *)
let rec concat l k =
    match l with
    | [] -> k
    | h::tl -> h::(concat tl k);;

(* Checks if a given predicate is true for
   at least one element of a list.
   ('a -> bool) * ('a list) -> bool *)
let rec forsome args = match args with
    | (pred,[])     -> false
    | (pred,(x::l)) -> if (pred x) then true else (forsome(pred,l));;

(* First-order terms to unify  *)
type folterm =
    | A
    | B
    | H of folterm
    | G of folterm * folterm;;

(* subst x t s replaces x by t in s *)
let rec subst x t s = match (x, s) with
     | nab X   in (X,X)  -> t
     | nab X Y in (X,Y)  -> Y
     | (x,A)             -> A
     | (x,B)             -> B
     | (x, H u)          -> H (subst x t u)
     | (x, G(u,v))       -> G (subst x t u, subst x t v);;

(* Unification problems
    are lists of "disagreement pairs"

    unif =
        Coerce(disagreementPairs, substitutions)
      | Some(X\ unif @ X)
*)
type unif =
```

```
    | Coerce of ((folterm * folterm) list) * ((folterm * folterm)
        list)
    | Some   of (folterm => unif) ;;

(* Results are lists of substitutions *)
type subst =
    | Subs of (folterm * folterm) list
    | Var  of (folterm => subst) ;;

let rec getSubs uni =
     match uni with
     | Some(X\ u @ X) -> Var(X\ getSubs (u @ X))
     | Coerce(dpl, subl) -> Subs(subl)
;;

type result =
    | Fail
    | Success of subst
;;

(* The simpl function simplifies a list
  of disagreement pairs by removing not
  disagreeing pairs or not disagreeing heads *)
let rec simpl dpairs = match dpairs with
     | [] -> []
     | nab X in (X, X)::(l @ X) -> simpl (l @ X)
     | nab X in (t, X)::(l @ X) -> (X, t)::simpl (l @ X)
     | (A, A)::l -> simpl l
     | (B, B)::l -> simpl l
     | (H x, H u)::l -> simpl ((x,u)::l)
     | (G(x, y), G(u, v))::l -> simpl ((x, u)::(y, v)::l)
     | dpair::l -> dpair::(simpl l);;

let rec simplify unif = match unif with
  | (Some r)               -> (Some (X\ (simplify (r @ X))))
  | (Coerce (dpairs, subs)) -> (Coerce (simpl dpairs, subs));;

(* A term is rigid if it's not a nominal *)
let rigidp term = match term with
           | nab X in X -> false
           | y          -> true;;

(* occursp (nominal, term) checks if the
   nominal appears in the term.
   it's needed by the "variable elimination" phase *)
let rec occursp p = match p with
  | nab X in (X,X) -> true
  | nab X in (X,A) -> false
  | nab X in (X,B) -> false
  | nab X in (X,H(t @ X)) -> occursp (X,(t @ X))
  | nab X in (X,G((t @ X),(s @ X))) -> occursp (X,t @ X) || occursp
      (X,s @ X)
  | x -> false;;

(* a SIMPLIFIED list of disagreement pairs is failing
   if one of the pairs contains two rigid terms
   or if t appears in s or vice versa. *)
let rec failuredp dpairs = match dpairs with
  | [] -> false
  | (t,s)::l ->
      if (rigidp t) && (rigidp s) then true
      else if (occursp (t,s)) || (occursp (s,t)) then true
           else (failuredp l);;
```

```
let rec failurep unif = match unif with
  | (Some r) -> (new X in (failurep (r @ X)))
  | (Coerce (dps,subs)) -> (failuredp dps);;

let rec endp unif = match unif with
  | (Some r) -> (new X in (endp (r @ X)))
  | (Coerce (dps,subs)) ->
      match dps with
      | [] -> true
      | d -> false;;

let rec varElimFirst uni =
    (* This auxillary replace a var by a term
        in all disagreement pairs *)
    let rec aux_subst x t dpairs =
        match dpairs with
        | [] -> []
        | (lf, rf)::tl ->
            (subst x t lf, subst x t rf)::(aux_subst x t tl)
    in

    let rec doFirst dpairs subs =
            (* Because simpl and failure check are
            called before VarElim we can assume that:
               The dpair list is non-empty
               We only have f / f or f / r dpairs
             and in the later case f is not in r *)
            match dpairs with
            | nab X in (X, t)::(tl @ X) ->
                      Coerce(
                        aux_subst X t (tl @ X),
                        ((X, t)::(aux_subst X t subs))
                      )
    in
    match uni with
    | Some r -> Some(X\ (varElimFirst (r @ X)))
    | Coerce(dpl, subl) -> doFirst dpl subl
;;


(* We successively :
    simplify
    eliminate variables (todo)
    check failure
  until success or stale state *)
let rec unify uni =
    let simpleUni = simplify uni in
    if (failurep simpleUni)
    then Fail
    else if (endp simpleUni)
         then Success(getSubs simpleUni)
         else unify (varElimFirst simpleUni)
;;
```

Appendix B

A prototype implementation for MLTS

This appendix presents commented fragments of MLTS' prototype interpreter in λProlog. For example, mutually recursive definitions and builtins have been stripped from the code to increase readability. The full interpreter's code is available at https://github.com/voodoos/mlts/tree/master/lib/data/core.

1 Terms and types

```
% Units
kind unit           type.
type unit           unit.

% Booleans
kind bool           type.
type btrue          bool.
type bfalse         bool.

% Literals
kind literal        type.
type unit           literal.
type i              int -> literal.
type b              bool -> literal.
type s              string -> literal.

% Constructors
kind constructor    type.

% Terms
kind tm             type.

% Function (abstraction)
type lam            (tm -> tm) -> tm.
% Fixpoints
type fix            (tm -> tm) -> tm.
% Application
type app            tm -> tm -> tm.
% Definition
type let            tm -> (tm -> tm) -> tm.
% Nominal abstraction
type backslash      (tm -> tm) -> tm.
% Nominal application
type arobase        tm -> tm -> tm.
% Fresh nominal
type new            (tm -> tm) -> tm.
% Literal terms
type lit            literal -> tm.
% Variants
type variant        constructor -> list tm -> tm.
type match          tm -> list clause -> tm.

% patterns
kind pat            type.

% Wildcard "_"
type pany           pat.
% Pattern literal
type plit           literal -> pat.
% Pattern variable
type pvar           tm -> pat.
% Pattern nominal
type pnom           tm -> pat.
% Pattern nominal abstraction
type pbackslash     (tm -> pat) -> pat.
% Pattern nominal application
type parobase       pat -> tm -> pat.
% Pattern variant
type pvariant       constructor -> list pat -> pat.

% Matching clauses
kind clause         type.

% Simple match rule
type arr            pat -> tm -> clause.
% Pattern variable quantification
type all            (tm -> clause) -> clause.
% Nominal quantification (nab .. in)
type nab            (tm -> clause) -> clause.
```

2 Typing

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Kind and types definitions %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
kind ty type.

% Raw types for integers,
% strings, boolean and unit
type int, bool, string, tunit ty.
% Lists of some type
type lst                ty -> ty.
% Functional arrow type
type arr                ty -> ty -> ty.
% Higher-arity arrow type (=>)
type bigarr             ty -> ty -> ty.

% Top-level typing predicate
type typeof             tm -> ty -> prop.

%%%%%%%%%%%%%%%%%%%%
%%% Typing rules %%%
%%%%%%%%%%%%%%%%%%%%
% Literals
typeof (lit unit) tunit.
typeof (lit (i _I)) int.
typeof (lit (b _B)) bool.
typeof (lit (s _S)) string.

% Recursive definitions
typeof (fix M) A :-
    pi x\ typeof x A => typeof (M x) A.

% Let-expressions
typeof (let M R) A :-
    typeof M B, pi x\ typeof x B => typeof (R x) A.

% New nominal
typeof (new R) B :-
    pi x\ typeof x A_ => typeof (R x) B.

% Function abstraction
typeof (lam R) (arr A B)  :-
    pi x\ typeof x A => typeof (R x) B.

% Nominal abstraction
typeof (backslash R) (bigarr A B) :-
    pi x\ typeof x A => typeof (R x) B.

% Application
typeof (app M N) B :-
    typeof M (arr A B),
    typeof N A.

% Nominal application
typeof (arobase M N) A :-
    typeof M (bigarr B A),
    typeof N B.

% Constructors (variants)
% built-in pairs
type t_pair ty -> ty -> ty.
type pair constructor.
type_constr pair [A, B] (t_pair A B).

% built-in lists
type t_list ty -> ty.
type list_cons constructor.
type list_empty constructor.
type_constr list_empty [] (t_list A_).
type_constr list_cons [A, (t_list A)] (t_list A).

typeof (variant Constr Args) A :-
    type_constr Constr Typs A,
    type_check Args Typs.

% Two by two check of a list of
% expressions and a list of types.
type_check [] [].
type_check [E|TL] [Ty|TyL] :-
    typeof E Ty,
    type_check TL TyL.

% Patterns
type_pat pany A_.
type_pat (plit I) A :- typeof (lit I) A.
type_pat (pvar T) A :- typeof T A.
type_pat (pnom N) A :- typeof N A.
type_pat (pbackslash R) (bigarr A B) :-
    pi x\ typeof x A => type_pat (R x) B.

type_pat (parobase P T) B :-
    type_pat P (bigarr A B),
    typeof T A.

type_pat (pvariant Constr Pats) A :-
    type_constr Constr Typs A,
    type_checkp Pats Typs.

% Two by two check of a list of
% patterns and a list of types.
type_checkp [] [].
type_checkp [P|TL] [Ty|TyL] :-
    type_pat P Ty,
    type_checkp TL TyL.

% Matching
typeof (match Exp Rules) A :-
    typeof Exp B, type_match B Rules A.

type_match _ [] _.
type_match A (R::Rs) B :-
    type_match_rule A R B, type_match A Rs B.

type_match_rule A (arr Pat Result) B :-
    type_pat Pat A, typeof Result B.
type_match_rule A (nab   R) B :-
    pi x\ typeof x C_ => type_match_rule A (R x) B.
type_match_rule A (all   R) B :-
    pi x\ typeof x C_ => type_match_rule A (R x) B.
```

3 Interpreter

```
% Value and evaluation props
type val            tm -> prop.
type eval           tm -> tm -> prop.

% Tools to introduce nominals and variables
pin G :- pi x \ nom x => G x.
piv G :- pi x \ abstract_value x => G x.

%%%%%%%%%%%%%%
%%% Values %%%
%%%%%%%%%%%%%%
val (lam _).
val (lit L) :- val_lit L.

val N :- nom N.
val A :- abstract_value A.

val_lit unit.
val_lit (i _).
val_lit (s _).
val_lit (b btrue).
val_lit (b bfalse).

val (backslash R) :- pin x \ val (R x).
val (variant _C Vs) :- foreach val Vs.

%%%%%%%%%%%%%%%%%%
%%% EVALUATION %%%
%%%%%%%%%%%%%%%%%%
eval (lam R) (lam R).
eval (lit L) (lit L) :- val_lit L.

eval N N :- nom N.
eval A A :- abstract_value A.

eval (app T U) V :-
    eval T (lam R),
    eval U VU,
    eval (R VU) V.

eval (let Def Body) V :-
    eval Def VDef,
    eval (Body VDef) V.

eval (new R) V :-
    sigma W \ pin x \ eval (R x) V.

eval (backslash R) (backslash VR):-
    pin x \ eval (R x) (VR x).

eval (arobase T N) V :-
    nom N,
    eval T (backslash R),
    eval (R N) V.

eval (variant C Ts) (variant C Vs) :-
    foreach2 eval Ts Vs.

eval (fix F) V :- eval (F (fix F)) V.

% Evaluation of pattern matching
eval (match T Cls) V :-
    eval T VT,
    eval_clauses VT Cls V.

% Evaluation of a list of clauses
type eval_clauses
    tm -> list clause -> tm -> prop.

eval_clauses _VL [] _VR :- false.
eval_clauses VL (Cl::Cls) VR :-
    if (eval_clause VL Cl [] R)
      (eval R VR)
      (!, eval_clauses VL Cls VR).

% Evaluation of a single clause
% We need to carry a list of bindings
% appearing in the pattern of the clause
type eval_clause
    tm -> clause -> list binding -> tm -> prop.

eval_clause L (all RCl) Sigma (R Vx) :-
  pi x \ pattern_variable x =>
  eval_clause L (RCl x) (Sigma_in x) (R x),
  extract (Sigma_in x) x Vx Sigma.

eval_clause L (nab RCl) Sigma (R Y) :-
  pin x \
  locate_rigid_clause x Y L (RCl x), !,
  subst copy [bind Y x] L (Lx x),
  eval_clause (Lx x) (RCl x) Sigma (R x).

eval_clause L (arr P R) Sigma R :-
  matches L P Sigma.

% Matches (used by eval_clause)
type matches
    tm -> pat -> list binding -> prop.

matches _ pany [].

matches (lit L) (plit L) [].

matches X (pvar Y) [bind Y X] :-
    pattern_variable Y.

matches X (pnom X) [].

matches (variant C Vs) (pvariant C Ps) TmSubst :-
    matches_all Vs Ps TmSubst.

matches_all [] [] [].
matches_all (V :: Vs) (P :: Ps) TmSubst_all :-
    matches V P TmSubst,
    matches_all Vs Ps TmSubst_rest,
    concat_bindings TmSubst TmSubst_rest TmSubst_all.

matches (backslash V) (pbackslash P) TmSubst :-
```

```
    pin x \ matches (V x) (P x) TmSubst.

matches V (parobase P X) TmSubst :-
    nom X,
    sigma Vx \
    pin x \
    subst copy [bind X x] V (Vx x),
    matches (backslash (x \ Vx x)) P TmSubst.


% Utilities on binding lists
type extract
    list binding -> tm -> tm -> list binding -> prop.

extract (bind X Y :: Sigma) X Y Sigma :- !.
extract (bind X1 Y1 :: Sigma1)
        X2 Y2 (bind X1 Y1 :: Sigma2) :-
    not (X1 = X2), !,
    extract Sigma1 X2 Y2 Sigma2.

type concat_bindings    list binding ->
    list binding -> list binding -> prop.

concat_bindings [] Subst2 Subst2.
concat_bindings (bind X Y :: Subst1) Subst2 (bind X
    Y :: Subst3) :-
    concat_bindings Subst1 Subst2 Subst3.

type subst      (A -> A -> prop) ->
     list binding -> A -> A -> prop.

subst Copy Subst Tin Tout :-
    if (current_subst _)
      (print "assert failure in subst", !, false)
      (current_subst Subst => Copy Tin Tout).

binds [] X1 Y1 :- X1 = Y1.
binds (bind X1 Y1 :: Subst) X2 Y2 :-
    if (X1 = X2)
       (Y1 = Y2)
       (binds Subst X2 Y2).

% Paths
kind list_path type.
type now       list_path.
type next      list_path -> list_path.

type in_list  A -> list_path -> list A -> prop.
in_list X now      (Y :: _Ys) :- X = Y.
in_list X (next P) (_Y :: Ys) :- in_list X P Ys.

kind value_path         type.
type here               value_path.
type under_cons         constructor ->
    list_path -> value_path -> value_path.
type under_backslash
    (tm -> value_path) -> value_path.
type under_arobase
    value_path -> tm -> value_path.

% Finding rigid occurences (used by eval_clause)
type rigid_in_clause
    tm -> value_path -> clause -> prop.
rigid_in_clause X Path (all Cl) :-
    piv x \ rigid_in_clause X Path (Cl x).
rigid_in_clause X Path (nab Cl) :-
    pin x \ rigid_in_clause X Path (Cl x).
rigid_in_clause X Path (arr P _) :-
    rigid_in_pat X Path P.

type rigid_in_pat
    tm -> value_path -> pat -> prop.
rigid_in_pat X here (pnom X).
rigid_in_pat X here (pvar X).
rigid_in_pat X (under_cons C N Path) (pvariant C Ps)
    :-
    in_list P N Ps,
    rigid_in_pat X Path P.
rigid_in_pat X (under_backslash Pathx) (pbackslash
    Px)
    :- pin x \ (rigid_in_pat X) (Pathx x) (Px x).
rigid_in_pat X (under_arobase Path Y) (parobase P Y)
    :- not (X = Y),
    rigid_in_pat X Path P.

type rigid_in_val
    tm -> value_path -> tm -> prop.
rigid_in_val X here X.
rigid_in_val X (under_cons C N Path) (variant C Vs)
    :- in_list V N Vs,
    rigid_in_val X Path V.
rigid_in_val X (under_backslash Pathx) (backslash R)
    :- pin x \ rigid_in_val X (Pathx x) (R x).
rigid_in_val X (under_arobase Path Y) (Vx Y) :-
    rigid_in_val X Path (backslash (x \ Vx x)).

type locate_rigid_clause
    tm -> tm -> tm -> clause -> prop.
type locate_rigid   tm -> tm -> tm -> pat -> prop.

type silence-rigid-occurrence-constraint prop.
locate_rigid_clause Xin Xout V Cl :-
    (rigid_in_clause Xin Path Cl, !;
     not silence-rigid-occurrence-constraint, !,
     term_to_string Xin XinStr,
     term_to_string Cl PStr,
     ErrMsg is "Nominal " ^ XinStr ^
     " has no rigid occurrence in " ^ PStr,
     eval_error ErrMsg,
     false),
    rigid_in_val Xout Path V.

%%% Copy clauses %%%
% Substitution is realized using
% "copy clauses" as described in
% Programming with Higher Order Logic
% By D. Miller and G. Nadathur.
% The full definition of these
% copy clauses can be found on the git.
copy (lit L1) (lit L2) :- L1 = L2.
copy (lam R1) (lam R2) :- underv copy R1 R2.
copy (app M1 N1) (app M2 N2) :-
    copy M1 M2, copy N1 N2.
% etc.
```

INSTITUT POLYTECHNIQUE DE PARIS

Titre: Calculer avec des relations, des fonctions et des lieurs

Mots clés: Calcul, logique, lieurs, languages de programmation

Résumé: Cette thèse s'inscrit dans la longue tradition de l'étude des relations entre logique mathématique et calcul et plus spécifiquement de la programmation déclarative. Le document est divisé en deux contributions principales. Chacune d'entre-elles utilise des résultats récents de la théorie de la démonstration pour développer de techniques novatrices utilisant déduction logique et fonctions pour effectuer des calculs.
La première contribution de ce travail consiste en la description et la formalisation d'une nouvelle technique utilisant le mécanisme de la focalisaton (un moyen de guider la recherche de preuve) pour distinguer les calculs fonctionnels qui se dissimulent dans les preuves déductives. À cet effet nous formulons un calcul des séquents focalisé pour l'arithmétique de Heyting où points-fixes et égalité sont définis comme des connecteurs logiques et nous décrivons une méthode pour toujours placer les prédicats singletons dans des phases négatives de la preuve, les identifiant ainsi avec un calcul fonctionnel. Cette technique n'étend en aucune façon la logique sous-jacente: ni opérateur de choix, ni règles de réécritures ne sont nécéssaires. Notre logique reste donc purement relationnelle même lorsqu'elle calcule des fonctions.
La seconde contribution de cette thèse est la conception d'un nouveau langage de programmation fonctionnelle: MLTS. De nouveau, nous utilisons des travaux théoriques récents en logique: la sémantique de MLTS est ainsi une théorie au sein de la logique $\mathcal{G}$, la logique de raisonnement de l'assistant de preuve Abella. La logique $\mathcal{G}$ utilise un opérateur spécifique: ∇, qui est un quantificateur sur des noms "frais" et autorise un traitement naturel des preuves manipulant des objets pouvant contenir des lieurs. Ce traitement s'appuie sur la gestion naturelle des lieurs fournie par le calcul des séquents. La syntaxe de MLTS est basée sur celle du langage de programmation OCaml mais fournit des constructions additionnelles permettant aux lieurs présents dans les termes de se déplacer au niveau du programme. De plus, toutes les opérations sur la syntaxe respectent l'alpha et la béta conversion. Ces deux aspects forment l'approche syntaxique des lieurs appelée λ-tree syntax. Un prototype d'implémentation du langage est fourni, permettant à chacun d'expérimenter facilement en ligne (`https://trymlts.github.io`).

Title: Computing with relations, functions, and bindings

Keywords: Computation, logic, binders, programming languages

Abstract: The present document pursues the decades-long study of the interactions between mathematical logic and functional computation, and more specifically of declarative programming. This thesis is divided into two main contributions. Each one of those make use of modern proof theory results to design new ways to compute with relations and with functions.
The first contribution of this work is the description and formalization of a new technique that leverages the focusing mechanism (a way to guide proof-search) to reveal functional computation concealed in deductive proofs. To that extent we formulate a focused sequent calculus proof system for Heyting arithmetic where fixed points and term equality are logical connectives and describe a means to always drive singleton predicates into negative phases of the proof, thus identifying them with functional computation. This method does not extend the underlying logic in any way: no choice principle nor equality theory nor rewriting rules are needed. As a result, our logic remains purely relational even when it is computing functions.
The second contribution of this thesis is the design of a new functional programming language: MLTS. Again, we make use of recent work in logic: the semantics of MLTS is a theory inside $\mathcal{G}$-logic, the reasoning logic of the Abella interactive theorem prover. $\mathcal{G}$-logic uses a specific operator: ∇, a fresh-name quantifier which allows for a natural treatment of proofs over structures with bindings based on the natural handling of bindings of the sequent calculus. The syntax of MLTS is based on the programming language OCaml but provides additional sites so that term-level bindings can move to programming level bindings. Moreover, all operations on syntax respect $\alpha\beta$-conversion. Together these two tenets form the λ-tree syntax approach to bindings. The resulting language was given a prototype implementation that anyone can conveniently try online (`https://trymlts.github.io`).

Institut Polytechnique de Paris
91120 Palaiseau, France