Object-Oriented Mechanisms for Interoperability between Proof Systems

Raphaël Cauderlier

CONSERVATOIRE NATIONAL
DES ARTS ET MÉTIERS | le cnam

École Doctorale Informatique, Télécommunications et Électronique (Paris)

Centre d'Études et de Recherche en Informatique et Communications

THÈSE DE DOCTORAT

présentée par : **Raphaël CAUDERLIER**

soutenue le : **10 octobre 2016**

pour obtenir le grade de : **Docteur du Conservatoire National des Arts et Métiers**

Spécialité : **Informatique**

Object-Oriented Mechanisms for Interoperability between Proof Systems

THÈSE DIRIGÉE PAR

Mme. Dubois Catherine	*Professeur des universités, ENSIIE*

RAPPORTEURS

Mme. Kesner Delia	*Professeur des universités, Université Paris Diderot*
M. Miller Dale	*Directeur de recherche, Inria Saclay et LIX*

PRÉSIDENT

M. Rémy Didier	*Directeur de recherche, Inria Paris-Rocquencourt*

EXAMINATEURS

M. Dowek Gilles	*Directeur de recherche, LSV*
M. Pessaux François	*Docteur, ENSTA ParisTech*

Remerciements

Le travail présenté dans cette thèse n'aurait pas pu aboutir sans l'aide des personnes qui m'ont entourées ces dernières années. La recherche est un effort collectif et je souhaite rendre justice à ceux qui m'ont permis d'aller au bout de cette aventure.

La première des personnes que je souhaite remercier est naturellement Catherine pour sa lourde tâche de direction de mes travaux de thèse. Malgré ma tendance à partir dans tous les sens pour résoudre tous les problèmes de la Terre à la fois, elle a su canaliser mes efforts pour que nous arrivions à proposer des solutions à certains d'entre eux.

Je tiens à remercier chaleureusement Delia Kesner pour avoir accepté de rapporter cette thèse. Ses remarques m'ont permis d'améliorer ce document. L'introduction et le Chapitre 6 ont particulièrement bénéficié de ses conseils.

I would also like to thank Dale Miller for accepting to review this thesis. His report brought me a lot of relief and encouragement when I needed them.

Je suis extrêmement reconnaissant envers Gilles pour m'avoir permis d'effectuer un stage de Master dans son équipe. Il a suivi mes travaux de près depuis malgré son emploi du temps chargé. Je tiens à le remercier pour ses encouragements, ses conseils et ses anecdotes. Je remercie aussi François Pessaux pour l'outil merveilleux qu'est FoCaLiZe, pour la qualité de son code sans laquelle mon travail sur Focalide aurait été beaucoup plus difficile. Mes remerciements vont aussi à Didier Rémy pour avoir accepté d'évaluer ma thèse à mi-parcours. Je suis honoré qu'ils aient tous les trois accepté de faire partie de mon jury.

Je remercie particulièrement Catherine et François pour avoir traqué les fautes qui jonchaient ce manuscrit et je tiens à m'excuser auprès des lecteurs pour celles qui continuent

certainement d'y figurer.

J'ai beaucoup apprécié travailler au sein de Deducteam, une équipe qui m'a beaucoup apporté tant sur le plan professionnel que sur le plan personnel dans laquelle on discute facilement de travail comme de choses moins sérieuses (Now, I know how to divide a $\lambda\Pi$ in n equal parts).

Mon travail de thèse a été grandement enrichi par les interactions que j'ai pu avoir avec mes collègues. Ali a toujours eu de bonnes idées d'encodages dans Dedukti qui ont grandement influencé toutes les parties de cette thèse. J'ai pris beaucoup de plaisir à travailler avec lui, en particulier sur l'interopérabilité des systèmes Coq et HOL et sur les énigmes qu'il nous posait régulièrement. Ma thèse doit également beaucoup à Pierre Halmagrand et Guillaume Bury qui m'ont initié à l'art de modifier Zenon. Merci aussi à l'expert OCaml local de Deducteam, Simon Cruanes ; même si nos travaux se sont peu intersectés, il a, malgré la concurrence de Pierre et de ses commandes groupées de capsules, réussi à imposer le café filtre à la pause déjeuner. Merci aussi à Quentin et Ronan pour les deux versions de Dedukti avec lesquelles j'ai travaillé au cours de ma thèse. La réimplantation de Dedukti par Ronan a permis à la fois un gain de performance très impressionnant qui a ouvert de nouvelles possibilités d'utilisation de Dedukti et enrichi Dedukti de fonctionnalités qui m'ont été extrêmement utiles : la vérification de la confluence et la réécriture d'ordre supérieur. Je tiens à le remercier pour son travail et pour la réactivité dont il a fait preuve tout au long de sa thèse dans le maintient de Dedukti. Je remercie aussi Frédéric Gilbert pour les discussions passionnantes que j'ai eu avec lui en particulier celles sur la constructivisation autour d'un Bibimbap lorsque nous fuyions le RU ainsi que Simon Martiel pour avoir supporté sans broncher des débats interminables sur des sujets éloignés de son domaine. Merci à tous les autres membres de Deducteam que j'ai eu la chance de croiser place d'Italie et au LSV pour leur convivialité: Hélène Milome, Virginie Collette et Thida Iem qui m'ont soulagé de ma phobie administrative; Achille qui a refait le site et fait baisser la moyenne d'âge de l'équipe; Robert, ses moustaches et son parapluie; Raphaël Bost, qui a contribué à vérifier la théorie des prénoms doubles; Quentin; Gaëtan; Frédéric Blanqui, et ses doodles; Guillaume Burel; Olivier, qui a encadré mon stage de Master; Jean-Pierre; David; Thérèse; François Thiré, qui intègre continuement mes erreurs

dans Dedukti; Kailiang; Bruno, qui a soutenu; Pierre Néron; Arnaud; Hugo; Benoit et Alejandro.

Je tiens à remercier tous les membres du LSV pour la bienveillance avec laquelle ils ont accueilli Deducteam. Merci en particulier aux doctorants du LSV pour la convivialité de leurs goûters. Cette thèse s'appuie sur un grand nombre de notions de logique et d'informatique qui m'ont été enseignées à l'ENS Cachan et au MPRI. J'ai pris grand plaisir à retrouver les enseignants qui m'ont apporté ces connaissances dans mon jury et comme collègues au LSV.

Je suis reconnaissant envers mes parents pour leur soutien infaillible, leur maison a toujours été un refuge pour moi entre l'intérêt et la curiosité de mon père pour mon travail et le don de ma mère pour me changer les idées. Je les remercie pour avoir toujours cru en moi et je remercie mes frères de m'avoir supporté. Je remercie aussi mes grands-parents qui m'ont accueilli à Paris durant mes études. Je remercie aussi ma belle-famille qui m'a toujours accueilli très gentiment et à laquelle j'ai beaucoup de plaisir à rendre visite.

Je n'oublie pas mes amis qui m'ont beaucoup apporté sur le plan humain: Imago, Hortense, Antoine; les docteurs cru 2016, Poussinet, Émile, Ara, Trollin qui m'ont montré la voie; Shadoko, NeK, 20-100, Ping, Eguel, b2moo, Levity, CPA, Adèle, Nicolas, Nolwenn, Maud et tous les A♡.

Enfin, je remercie du fond du coeur ma femme Marie-Noëlle pour l'amour, la tendresse et le soutien qu'elle n'a cessé de m'apporter depuis notre rencontre. Elle m'a réconforté dans les moments de doute, elle a supporté mes horaires indécents, elle m'a encouragé sans relâche, elle a relu mes articles, elle m'a fait travailler mes présentations, elle a crue en moi, en elle et en nous, elle nous a lancé dans des projets merveilleux et elle a su me rendre heureux en toutes circonstances. Il me tarde de découvrir ce que l'avenir nous réserve.

Contents

Introduction

When Bolzano and Cantor discovered in the second half of the 19th century that the mathematical notions of set and infinity were worth studying [30, 39], it had two very important impacts on logic. First it launched the study of foundations of mathematics which happens to be a very rich field of research and is still active nowadays. It also led to a series of paradoxes, the very problem of which was not the absence of workarounds but on the contrary the existence of several possible corrections. One such correction was Russell and Whitehead's *Principia Mathematica* [173] solving the paradoxes by introducing a type system, another one was the axiomatization of set theory ZFC by Zermelo and Fraenkel [175, 3].

Both systems introduced axioms whose self-evidence was not obvious for all logicians: Russell and Whitehead introduced the axiom of reducibility, a highly technical device which seemed mandatory to formulate real analysis and Zermelo introduced the axiom of choice which quickly led to counter-intuitive results such as Hausdorff Paradox [92] and Banach-Tarski Paradox [21].

But the coexistence of several logical systems was only at its very beginning. At the time where these new logics were elaborated, Brouwer rejected the principle of excluded middle [34] and entered a long controversy with Hilbert about the validity of non-constructive proofs. Brouwer's vision of constructivism was adapted by his student Heyting as a new logic called intuitionistic logic [94]. The study of intuitionistic logic culminated with the discovery by Scott and Martin-Löf that the constructions of type theory closely correspond to intuitionistic logical connectives [159, 123]. This discovery leads to a presentation of type theory much cleaner than *Principia Mathematica*: Intuitionistic Type Theory. Many other logics have since been invented such as modal logic, minimal logic, linear logic, temporal

logic, and fuzzy logic just to name a few.

But the real explosion in the number of logical systems comes from the development of computer science. Computers have been used very soon to solve mathematical and logical problems. We can distinguish three kinds of mathematical software with respect to their logical foundations:

- Some systems such as computer algebra systems are very imprecise with respect to their foundations and offer very few logical guarantees. They are intended to help mathematicians to perform algebraic computations and guide the mathematical intuition but the mathematical proofs remain to the charge of the mathematician.
- At the other extremity of the spectrum, proof checkers implement well-defined logics and require extremely precise proofs that they verify step by step. They typically implement rich logics in which complex mathematics can be formulated.
- Finally, some systems are designed to solve particular kinds of problems without requiring human interaction.

The development of systems of the second and third categories have required the study of a lot of new logics.

Quite often, the best choice for solving a particular class of problems is to design a specific logic which is simple enough to be efficiently automatized and powerful enough to express the problems of interest. For example, temporal logics and separation logics are useful for modeling and verifying respectively the evolution of finite systems and the memory constraints of imperative programs.

Proof checkers, have been used to formalize and check mathematical theorems since the 60s [132]. This led to a higher level of confidence in these mathematical results and even allowed the resolution of some mathematical problems for which the usual peer-reviewing process was inapplicable due to the length of the proof such as proofs of the four-color theorem [84] and Kepler conjecture [89].

Another important application of implementations of logical systems in computers is program deductive verification consisting in the formalization of program behaviour and

the formal proof that programs respect their specifications. Formal correctness proof of programs is mandatory when human lives depend on them; in particular in the transport, medical, and energy industries. Actually, formalization of mathematics and program deductive verification are closely connected. On one side, formally solving a mathematical problem such as the four-color theorem or Kepler conjecture amounts to develop a computer program for solving the problem and prove it correct with respect to its mathematical specification. On the other side, rich libraries of mathematical results are needed for proving some programs; for example, aircraft safety relies on real analysis and correctness of cryptographic protocols relies on arithmetic.

The implementations of logical systems have served as experimentation platforms for new logical ideas such as the Curry-Howard correspondence [134], automation [25], mixing set theory with typing [4], and extensional type theory [54]. Because of this habit, logical systems are getting more and more complex. To increase their trustability, most of these logical systems are built on top of rather small kernels [87, 63, 10] which are supposed to be close to published logical systems for which good meta properties such as consistency have been proved. In practice, this means that for each new feature, implementors of logical systems have to choose whether it should be added to the kernel at the price of making the logic more complicated or to an upper layer and compiled to kernel code which might impact performances.

All these new features lead to high diversity of logically incompatible systems, especially in the world of interactive proof assistants. Because of this diversity, formal proofs are usually associated with the system in which they have been developed; while the foundational choices are usually left implicit in mathematics, we almost systematically precise that the four-color theorem has been proved **in Coq** logic, that Kepler conjecture has been proved **in higher-order logic** etc...

We believe however that until we have understood why the proof of a theorem requires a precise logic, the user should be allowed to use whatever systems suits her best or even that she should be allowed to use any combination of existing systems. Each logical system has its own benefits and we would like to combine them. The questions of integration and interoperability between logical systems is a longstanding demand of the users of these

systems. The most successful results in this field concern the integration of automatic tools in proof assistants. In [24], three styles of integration of automatic tools are distinguished: the accepting style, the skeptical style, and the autarkic style.

- In the accepting style, problems are delegated to automatic tools which are blindly trusted. This style is easy to implement because we only have to communicate the problem to the automatic tool. The proof assistant PVS [139] and the Why3 system [72] are examples of this style of delegation.

- In the skeptical style, problems are still delegated to automatic tools but they are asked to provide a formal proof of their answers that is independently checked. Examples of skeptical-style delegation are the FoCaLiZe environment [143] and the Mizar integration of the automatic theorem prover E [6]. The skeptical style is much more trustworthy than the accepting style because proof checkers and proof assistant kernels are much smaller systems than automatic theorem provers. This approach is however very limited in the number of automatic tools that can be used because most automatic tools provide only partial justification of their results.

- Finally, the autarkic approach pragmatically combines the benefits of the accepting style and the skeptical style. In this approach, we take whatever justification the automatic tool accepts to provide and we use this proof certificate to guide the reconstruction of a fully formal justification in a trustworthy system. In [24], Barendregt and Barendsen advocate for the omission of computation steps in proof certificate and this is the choice made by Deduction modulo provers to which proof obligations from Atelier B are delegated in the BWare project [62]. This notion of incomplete proof certificate is further pushed by the ProofCert project [126, 127] which even allows some reasoning steps to be omitted in the certificate. The most successful integration of automatic reasoners in a proof assistant is probably achieved by the Sledgehammer tool [25] which combines the power of state-of-the-art theorem provers and SMT solvers with the trust level of Isabelle by reconstructing the proofs using a certifying prover called Metis.

What makes integration of automatic tools such as automatic theorem provers feasible

is that they essentially all share the same logic (first-order logic) which happens to be a fragment of the logic of the interactive systems calling them so the proofs found by the automatic tool are readable as proofs of the interactive system. The only logical drawback is that most automatic theorem provers reason in classical logics whereas some proof assistants use constructive logics. We will propose an original and pragmatic solution to this difficulty in Chapter 12.

Interoperability between interactive theorem provers is logically much harder because, as we already mentioned, there are almost as many incomparable logics as there are interactive provers implementing them. Even provers claiming to implement the same logic such as Coq [63] and Matita [10] differ on some details such as proof irrelevance and universe polymorphism so automatically translating a formal development from a system to another one is often considered harder than redoing the development from scratch in the second one. The hard question of interoperability between interactive proof systems has been attacked many times for different pairs of systems [104, 131, 135, 103, 8] but it is still far from being solved because the number of translations to define is naturally quadratic in the number of existing proof systems.

This interoperability problem is the main topic of the Deducteam research team in which most of the work presented in this thesis has been conducted. In order to solve this problem, we take inspiration from the close field of programming languages. Taking inspiration from programming languages is very common in the type-theoretical community since the Brouwer-Heyting-Kolmogorov interpretation of intuitionistic logic as solving programming tasks.

So how is interoperability achieved in the programming world? Programming languages are built on various paradigms and provide different libraries so the need for interoperability is equally present for programming languages and for logical systems. Typical programming projects are built by combining a few languages, interoperability is achieved by compiling code written in these different high-level languages into a low-level language such as bytecode or assembly. All the code is then linked together at low level to obtain a runnable program. Making the high-level languages agree on a single low-level language requires only a linear number of compilers instead of a quadratic number in the case of one-to-one

correspondences.

The requirements for good low-level and high-level languages are different. Among other requirements, high-level languages should focus on readability and modularity whereas low-level languages should be efficient but are generally hard to read.

Deducteam proposes to follow the same path for logical systems. A low-level proof system called Dedukti [155] has been developed together with many translators from logical systems to Dedukti playing the role of compilers. As a low-level system, Dedukti does not focus on readability nor modularity but on expressiveness to encode many logics: it claims to be a universal proof format [29].

At the beginning of this thesis, only a few logical systems were translated to Dedukti. Moreover Dedukti did not really help to exchange proofs between these systems because no support for the linking operation is provided. It seemed that a companion tool for doing the linking work was needed but the priority was to develop translators for new systems and improve the existing ones.

We developed a translator from the FoCaLiZe formal environment to Dedukti. FoCaLiZe is both a logical system and a programming language so this led us to the compilation of programming languages to Dedukti and more particularly object-oriented mechanisms. We also realized that FoCaLiZe, thanks to its object-oriented mechanisms, could provide the missing linking operation that was needed to achieve interoperability in Dedukti.

In this thesis, we translate to Dedukti the formal environment FoCaLiZe and two popular programming paradigms, object-oriented and functional programming. In order to better understand how formal developments can be linked once they have been translated in Dedukti, we conduct experiments with interoperability between the proof assistants Coq and HOL and we propose FoCaLiZe object-oriented mechanisms to perform the linking.

The rest of thesis is structured as follows. In Part I, we present the notions on which this thesis is built. In Part II, we focus on the compilation of object-oriented programming in Dedukti independently of logical aspects through an object calculus. In Part III, we extend the FoCaLiZe compiler by a backend to Dedukti, and Part IV is a case study of interoperability based on Dedukti and FoCaLiZe.

More precisely, Part I is composed of Chapters 1, 2, and 3. In Chapters 1 and 2, we recall notions from logic, rewriting, and type theory. We present Dedukti and explain how languages are encoded in Dedukti in Chapter 3.

Part II is composed of Chapters 4, 5, and 6. We define a translation from an object calculus to Dedukti in Chapter 4. We extend this translation to subtyping, an important feature of object-oriented type systems, in Chapter 5. We then implement this translation in Chapter 6.

Part III is composed of Chapters 7, 8, and 9. FoCaLiZe is presented in Chapter 7. In Chapter 8, we propose a translation from FoCaLiZe functional programming language to Dedukti and in Chapter 9, we adapt FoCaLiZe automatic theorem prover to use it together with our translation to Dedukti.

Part IV is composed of Chapters 10, 11, and 12. A first proof of concept is conducted in Chapter 10 in which the linking between Coq and HOL logics is performed manually in Dedukti. In order to scale to a bigger example, we rely on FoCaLiZe object-oriented mechanisms in Chapter 11. Finally in Chapter 12, we propose heuristics to eliminate unnecessary uses of classical axioms in proofs found by automated theorem provers.

Part I

Background

Real mathematical proofs typically alternate clever reasoning and computation phases. This distinction is also appropriate for machine-checked proofs. In this context, we expect the machine to perform the computation by itself because, after all, computing is what computers are good at! In [24], Barendregt and Barendsen state the Poincaré principle recommending that reasoning and computing should be separated and that it should not be necessary to record pure computational steps.

Deduction modulo [69] is about generalizing this distinction between logical reasoning and computation in the context of automated proof search for first-order logic. The main practical benefit is a pruning of search space leading to more efficient proof search. In Deduction modulo, computation is formalized as term rewriting, a very powerful formalism which is especially appropriate for program verification because it has close connections with operational semantics.

Theorem provers for Deduction modulo respect the Poincaré principle. They record all the reasoning steps but not the computation steps so a proof checker for Deduction modulo proofs has to be able to reason modulo a given rewrite system. Dedukti is such a system.

Dedukti is also used as a proof checker for proof assistants based on type theory. These proof assistants usually implement higher-order logics which are more expressive than first-order logic. This extra expressivity makes fully automatic proof search unpractical hence the need for human hints.

The fact that Dedukti behaves well as a proof checker for both Deduction modulo and type theory is not surprising because Dedukti implements the $\lambda\Pi$-calculus modulo, the combination of a type system and Deduction modulo.

In this thesis, Dedukti will be used as a proof checkers for both Deduction modulo and type systems. In particular, our translation of FoCaLiZe to Dedukti in Part III will be the occasion to extend FoCaLiZe to Deduction modulo and our interoperability case study in Part IV is based on two translators from proof assistants to Dedukti.

In this first part, we recall the background notions on which this thesis is built. Chapter 1 is devoted to first-order reasoning: we present first-order logic, first-order rewriting and their combination Deduction modulo. In Chapter 2, we consider how types can be as-

signed to λ-calculus in functional programming and type theory. After these preliminaries, we will be able to introduce Dedukti and its type system in Chapter 3.

Chapter 1

First-Order Logic and First-Order Rewriting

First-order logic is the logical formalism which is most often implemented in automatic theorem provers. The traditional way to define a mathematical theory using first-order logic is the axiomatic approach: the theory is defined by a set of primitive axioms and the theorems are the logical consequences of these axioms.

The notion of computation is absent from the axiomatic approach so the Poincare principle is not applicable. In the axiomatic approach, to extend a theory by the definition of a computable function, one adds a symbol and one or more axioms relating the new symbol to the other symbols of the theory. Deduction modulo is an alternative to the axiomatic approach in which functions and predicates are defined by well behaved rewrite systems.

In Section 1.1, we present first-order logic and the traditional axiomatic approach. We then give a quick introduction to first-order term rewriting and we make more precise what "well behaved" rewrite systems are by defining important properties that rewrite systems may or not enjoy in Section 1.2. Deduction modulo, the alternative approach, is presented in Section 1.3. Finally Section 1.4 is devoted to the implementation of Deduction modulo in Zenon, a first-order theorem prover used in FoCaLiZe.

1.1 First-Order Logic

First-order logic [66], also known as predicate logic, is a logic in which a lot of mathematical theories such as Peano arithmetic, Euclid geometry and set theory can be expressed. Contrary to weaker logics such as propositional logic, first-order logic is powerful enough to serve as a logical foundation for mathematics so it is not surprising that provability in first-order logic is not decidable.

First-order logic is however semi-decidable which means that we can build programs that will eventually find proofs for all provable formulae but when run on an unprovable formula they might either reject the formula or run indefinitely. These programs are called automatic theorem provers. Developers of automatic theorem provers form a very active community. New theorem proving techniques can be evaluated thanks to the TPTP database of first-order problems [163] and a competition of automatic theorem provers is organized every year [165, 142].

In this section, we give a traditional presentation of first-order logic. We start by defining the syntactic constructs of the logic: terms, formulae, and theories. We then present natural deduction as an example of a proof system for first-order logic. Finally, we consider a common extension of first-order logic to typing.

1.1.1 Syntax

In first-order logic, a clear distinction is made between the objects of discourse called first-order terms and the logical formulae. Contrary to second-order logic and higher-order logic (that will be presented in Section 2.3.2), quantification in first-order logic is restricted to terms: it is syntactically not possible to quantify over propositions, predicates or functions in first-order logic.

1.1.1.1 First-Order Terms

First-order terms are built from variables and function symbols. We assume that a countable set X of variables has been fixed and we denote variables by the letters x, y, and z possibly decorated by indices if we need more than three names. Function symbols

are denoted by the letter f. Each function symbol comes with a natural number n called its arity indicating the number of arguments of the function symbol. An n-ary function symbol is a function symbol of arity n. The syntax of first-order terms, denoted by the letter t is as follows:

$$\begin{array}{llll} \textbf{term} & t & ::= & x & \textit{Variable} \\ & & & f(t_1, \ldots, t_n) & \textit{Application of an } n\textit{-ary function symbol} \end{array}$$

A 0-ary function symbol is called a constant. Constants are denoted by the letter a. We simply write a instead of the application of the constant a to no argument $a()$.

The set of all the variables occurring in a term t is written $\mathsf{FV}(t)$ and can be formally defined as follows:

- $\mathsf{FV}(x) := \{x\}$
- $\mathsf{FV}(f(t_1, \ldots, t_n)) := \mathsf{FV}(t_1) \cup \ldots \cup \mathsf{FV}(t_n)$

A variable x is said to be fresh in a term t when $x \notin \mathsf{FV}(t)$. Since the set of variables is infinite and the sets of variables of terms are always finite, we assume given a function fresh assigning to each term t a variable x which is fresh in t.

Variables are to be seen as placeholders for terms. The operation consisting of replacing a variable by a term is called substitution. The substitution of the variable x by the term t in the term t' is written $t'\{x\backslash t\}$ and defined as follows:

$$\begin{array}{lcl} x\{x\backslash t\} & := & t \\ y\{x\backslash t\} & := & y \\ f(t_1, \ldots, t_n)\{x\backslash t\} & := & f(t_1\{x\backslash t\}, \ldots, t_n\{x\backslash t\}) \end{array}$$

In the second line, y is assumed different from x otherwise the first line would apply.

The notion of substitution is easy to generalize to the parallel substitution of several variables by terms. A substitution is a mapping of variables to terms whose domain (that is, the set of variables not mapped to themselves) is finite. Substitutions are denoted by the letter ρ and applying the substitution ρ to the term t is written $t\rho$ and defined by:

$$\begin{array}{lcl} x\rho & := & \rho(x) \\ f(t_1, \ldots, t_n)\rho & := & f(t_1\rho, \ldots, t_n\rho) \end{array}$$

Or using the notation $\{x_1\backslash t_1, \ldots, x_n\backslash t_n\}$ for the substitution mapping each variable x_i to the term t_i.

$$
\begin{array}{lcl}
x_i\{x_1\backslash t_1, \ldots, x_n\backslash t_n\} & := & t_i \\
y\{x_1\backslash t_1, \ldots, x_n\backslash t_n\} & := & y \\
f(t'_1, \ldots, t'_m)\{x_1\backslash t_1, \ldots, x_n\backslash t_n\} & := & f(t'_1\{x_1\backslash t_1, \ldots, x_n\backslash t_n\}, \ldots, t'_m\{x_1\backslash t_1, \ldots, x_n\backslash t_n\})
\end{array}
$$

1.1.1.2 First-Order Formulae

Terms are related to each others by predicate symbols denoted by the letter P. As for function symbols, each predicate symbol comes with a fixed arity.

Applied predicate symbols are called atomic formulae or atoms; more complex formulae can be built using boolean connectives and quantifiers:

formula	φ	$::=$	$P(t_1, \ldots, t_n)$	*Application of an n-ary predicate symbol*
			$t_1 = t_2$	*Equality*
			$\top$	*Truth*
			$\bot$	*Falsehood*
			$\neg\varphi$	*Negation*
			$\varphi_1 \wedge \varphi_2$	*Conjunction*
			$\varphi_1 \vee \varphi_2$	*Disjunction*
			$\varphi_1 \Rightarrow \varphi_2$	*Implication*
			$\varphi_1 \Leftrightarrow \varphi_2$	*Equivalence*
			$\forall x.\ \varphi$	*Universal quantification*
			$\exists x.\ \varphi$	*Existential quantification*

In the quantified formulae $\forall x.\ \varphi$ and $\exists x.\ \varphi$, the variable x can appear in the formula φ, it is *bound* in φ and can be renamed without changing the meaning of the formula. This means that we want to identify the formulae $\forall x.\ \varphi$ and $\forall y.\ \varphi\{x\backslash y\}$ (and similarly, we want to identify $\exists x.\ \varphi$ and $\exists y.\ \varphi\{x\backslash y\}$).

To define this identification, we first need to define the set of free variables $\mathsf{FV}(\varphi)$ of a formula φ and the substitution of a variable by a term.

A variable x occurring in a formula φ is called a *bound* variable if it is in the scope of a quantifier. Otherwise, it is a *free* variable. The set of all the free variables of a formula φ is written $\mathsf{FV}(\varphi)$ and can be formally defined as follows:

$$
\begin{array}{lcl}
\mathsf{FV}(P(t_1,\ldots,t_n)) & := & \mathsf{FV}(t_1) \cup \ldots \cup \mathsf{FV}(t_n) \\
\mathsf{FV}(t_1 = t_2) & := & \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2) \\
\mathsf{FV}(\top) & := & \emptyset \\
\mathsf{FV}(\bot) & := & \emptyset \\
\mathsf{FV}(\neg\varphi) & := & \mathsf{FV}(\varphi) \\
\mathsf{FV}(\varphi_1 \wedge \varphi_2) & := & \mathsf{FV}(\varphi_1) \cup \mathsf{FV}(\varphi_2) \\
\mathsf{FV}(\varphi_1 \vee \varphi_2) & := & \mathsf{FV}(\varphi_1) \cup \mathsf{FV}(\varphi_2) \\
\mathsf{FV}(\varphi_1 \Rightarrow \varphi_2) & := & \mathsf{FV}(\varphi_1) \cup \mathsf{FV}(\varphi_2) \\
\mathsf{FV}(\varphi_1 \Leftrightarrow \varphi_2) & := & \mathsf{FV}(\varphi_1) \cup \mathsf{FV}(\varphi_2) \\
\mathsf{FV}(\forall x.\ \varphi) & := & \mathsf{FV}(\varphi) \setminus \{x\} \\
\mathsf{FV}(\exists x.\ \varphi) & := & \mathsf{FV}(\varphi) \setminus \{x\}
\end{array}
$$

The distinction between free and bound variables only makes sense in the presence of binding operations such as quantifiers. In first-order terms, all the variables are free hence the notation $\mathsf{FV}(t)$ for the set of all the variables occurring in the term t. The set of bound variables is not very interesting because it does not respect the identification of formulae under renaming.

A formula is *closed* if it has no free variable. A variable x is said to be fresh in a formula φ when $x \notin \mathsf{FV}(\varphi)$.

The substitution of the variable x by the term t in the formula φ is written $\varphi\{x\backslash t\}$ and defined as follows:

$$
\begin{array}{lcll}
P(t_1,\ldots,t_n)\{x\backslash t\} & := & P(t_1\{x\backslash t\},\ldots,t_n\{x\backslash t\}) & \\
(t_1 = t_2)\{x\backslash t\} & := & t_1\{x\backslash t\} = t_2\{x\backslash t\} & \\
\top\{x\backslash t\} & := & \top & \\
\bot\{x\backslash t\} & := & \bot & \\
(\neg\varphi)\{x\backslash t\} & := & \neg\varphi\{x\backslash t\} & \\
(\varphi_1 \wedge \varphi_2)\{x\backslash t\} & := & \varphi_1\{x\backslash t\} \wedge \varphi_2\{x\backslash t\} & \\
(\varphi_1 \vee \varphi_2)\{x\backslash t\} & := & \varphi_1\{x\backslash t\} \vee \varphi_2\{x\backslash t\} & \\
(\varphi_1 \Rightarrow \varphi_2)\{x\backslash t\} & := & \varphi_1\{x\backslash t\} \Rightarrow \varphi_2\{x\backslash t\} & \\
(\varphi_1 \Leftrightarrow \varphi_2)\{x\backslash t\} & := & \varphi_1\{x\backslash t\} \Leftrightarrow \varphi_2\{x\backslash t\} & \\
(\forall x.\ \varphi)\{x\backslash t\} & := & \forall x.\ \varphi & \\
(\forall y.\ \varphi)\{x\backslash t\} & := & \forall y.\ \varphi\{x\backslash t\} & (\text{when } y \text{ is fresh in } t) \\
(\forall y.\ \varphi)\{x\backslash t\} & := & \forall z.\ \varphi\{y\backslash z\}\{x\backslash t\} & (\text{where } z := \mathsf{fresh}(t)) \\
(\exists x.\ \varphi)\{x\backslash t\} & := & \exists x.\ \varphi & \\
(\exists y.\ \varphi)\{x\backslash t\} & := & \exists y.\ \varphi\{x\backslash t\} & (\text{when } y \text{ is fresh in } t) \\
(\exists y.\ \varphi)\{x\backslash t\} & := & \exists z.\ \varphi\{y\backslash z\}\{x\backslash t\} & (\text{where } z := \mathsf{fresh}(t))
\end{array}
$$

The freshness condition is mandatory to avoid capture when performing substitutions: in the formula $\forall y.\ x = y$, substituting x by y should not yield $\forall y.\ y = y$ because we do not want to identify $\forall x.\ \forall y.\ x = y$ and $\forall y.\ \forall y.\ y = y$.

A first-order *signature* declares the function and predicate symbols and provides their arities.

A first-order *theory* is defined by a signature and a set of closed first-order formulae called *axioms*. The set of axioms might be infinite but it is required to be computable (that is, we can always decide whether or not a given formula φ is an axiom or not).

A first-order *problem* is given by a theory and a closed formula called the *goal* of the problem.

Example 1. *Zermelo-Fraenkel set theory ZF can be defined as a first-order theory:*

Signature

- $\emptyset$ is a constant
- $\{\bullet, \bullet\}$ is a binary function symbol (we write $\{t_1, t_2\}$ instead of $\{\bullet, \bullet\}(t_1, t_2)$)
- $\bigcup$ is a unary function symbol
- W is a constant
- $\mathcal{P}$ is a unary function symbol
- $\in$ is a binary predicate symbol (we write $t_1 \in t_2$ instead of $\in (t_1, t_2)$)

Axioms

- Axiom of extensionality: $\forall x.\ \forall y.\ (\forall z.\ z \in x \Leftrightarrow z \in y) \Rightarrow x = y$
- Axiom of the empty set: $\forall x.\ \neg x \in \emptyset$
- Axiom of foundation: $\forall x.\ \neg(x = \emptyset) \Rightarrow (\exists y.\ y \in x \wedge \neg\exists z.\ z \in y \wedge z \in x)$
- Axiom schema of restricted comprehension: for all formula φ whose free variables are among $x, z, w_1, \ldots, w_n$,
  the formula $\forall z.\ \forall w_1.\ \ldots \forall w_n.\ \exists y.\ \forall x.\ x \in y \Leftrightarrow (x \in z \wedge \varphi)$ is an axiom
- Axiom of pairing: $\forall x.\ \forall y.\ \forall z.\ z \in \{x, y\} \Leftrightarrow z = x \vee z = y$

- Axiom of union: $\forall x.\ \forall z.\ z \in \bigcup(x) \Leftrightarrow (\exists y.\ z \in y \wedge y \in x)$
- Axiom schema of replacement: for all formula φ whose free variables are among $x, y, A, w_1, \ldots, w_n$,
  the formula $\forall A.\ \forall w_1.\ \ldots \forall w_n.\ (\forall x.\ x \in A \Rightarrow ((\exists y.\ \varphi) \wedge \forall y'.\ \varphi\{y \backslash y'\} \Rightarrow y = y')) \Rightarrow \exists B.\ \forall x.\ x \in A \Rightarrow \exists y.\ y \in B \wedge \varphi$ is an axiom
- Axiom of infinity: $\emptyset \in W \wedge \forall n.\ n \in W \Rightarrow \bigcup(\{n, \{n, n\}\}) \in W$
- Axiom of the power set: $\forall x.\ \forall y.\ y \in \mathcal{P}(x) \Leftrightarrow (\forall z. z \in y \Rightarrow z \in x)$

1.1.2 A Proof System for First-Order Logic: Natural Deduction

Automatic theorem provers implement various techniques based on various proof systems for first-order logic [152, 161]. We are not going to detail all of them but only present a proof system called natural deduction which is of interest for relating first-order theorem provers with proof assistants.

The derivation rules for natural deduction are given in Figure 1.1. The deduction judgment $\Gamma \vdash \varphi$ where Γ is a set of formulae and φ is a formula represents the derivability of the assertion φ from the list of hypotheses Γ. The deduction rules in Natural Deduction are split in three categories: introduction rules tell us how to derive a complex formula, elimination rules tell us how to use a derived complex formula and the axiom rule allows us to derive a formula when it is one of the hypotheses. Instead of giving rules for the missing connectives negation and equivalence, we consider them as derived connectives defined by $\neg\varphi := \varphi \Rightarrow \bot$ and $\varphi_1 \Leftrightarrow \varphi_2 := (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$.

Example 2. *The formula $(P(t) \wedge \forall x.\ P(x) \Rightarrow Q(x)) \Rightarrow Q(t)$ (where t is any term and P and Q are unary predicate symbols) can be proved by the following derivation where φ abbreviates the formula $P(t) \wedge \forall x.\ P(x) \Rightarrow Q(x)$:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varphi \vdash \varphi}\,(\text{Axiom})}{\varphi \vdash \forall x.\ P(x) \Rightarrow Q(x)}\,(\wedge\text{-elim-2})}{\varphi \vdash P(t) \Rightarrow Q(t)}\,(\forall\text{-elim}) \qquad \dfrac{\dfrac{}{\varphi \vdash \varphi}\,(\text{Axiom})}{\varphi \vdash P(t)}\,(\wedge\text{-elim-1})}{\varphi \vdash Q(t)}\,(\Rightarrow\text{-elim})}{\vdash \varphi \Rightarrow Q(t)}\,(\Rightarrow\text{-intro})$$

$$\frac{(\text{when } \varphi \in \Gamma)}{\Gamma \vdash \varphi}\ (\text{Axiom}) \qquad \frac{}{\Gamma \vdash \top}\ (\top\text{-intro}) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ (\bot\text{-elim})$$

$$\frac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2}\ (\wedge\text{-intro}) \qquad \frac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_1}\ (\wedge\text{-elim-1}) \qquad \frac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_2}\ (\wedge\text{-elim-2})$$

$$\frac{\Gamma \vdash \varphi_1}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee\text{-intro-1}) \qquad \frac{\Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee\text{-intro-2})$$

$$\frac{\Gamma, \varphi_1 \vdash \varphi_3 \qquad \Gamma, \varphi_2 \vdash \varphi_3 \qquad \Gamma \vdash \varphi_1 \vee \varphi_2}{\Gamma \vdash \varphi_3}\ (\vee\text{-elim}) \qquad \frac{\Gamma, \varphi_1 \vdash \varphi_2}{\Gamma \vdash \varphi_1 \Rightarrow \varphi_2}\ (\Rightarrow\text{-intro})$$

$$\frac{\Gamma \vdash \varphi_1 \Rightarrow \varphi_2 \qquad \Gamma \vdash \varphi_1}{\Gamma \vdash \varphi_2}\ (\Rightarrow\text{-elim}) \qquad \frac{\Gamma \vdash \varphi \qquad (\text{when } x \notin \mathsf{FV}(\Gamma))}{\Gamma \vdash \forall x.\ \varphi}\ (\forall\text{-intro})$$

$$\frac{\Gamma \vdash \forall x.\ \varphi}{\Gamma \vdash \varphi\{x \backslash t\}}\ (\forall\text{-elim}) \qquad \frac{\Gamma \vdash \varphi\{x \backslash t\}}{\Gamma \vdash \exists x.\ \varphi}\ (\exists\text{-intro})$$

$$\frac{\Gamma \vdash \exists x.\ \varphi \qquad (\text{when } x \notin \mathsf{FV}(\Gamma))}{\Gamma \vdash \varphi}\ (\exists\text{-elim}) \qquad \frac{}{\Gamma \vdash t = t}\ (=\text{-intro})$$

$$\frac{\Gamma \vdash \varphi\{x \backslash t_1\} \qquad \Gamma \vdash t_1 = t_2}{\Gamma \vdash \varphi\{x \backslash t_2\}}\ (=\text{-elim})$$

Figure 1.1: Natural Deduction

Natural deduction is a constructive proof system; in order to obtain a classical system equivalent to the calculi implemented in classical first-order theorem provers, we need to add an axiom scheme such as the Law of Excluded Middle: for all closed formula φ, the formula $\varphi \vee \neg\varphi$ is an axiom.

1.1.3 Polymorphic First-Order Logic

A weakness of first-order logic is that the only syntactic verification which is made concerns the arity of symbols so assuming parallelism has been declared as a binary predicate symbol, the sentence "the straight line (d) is parallel" is syntactically rejected but meaningless formulae such as "2 is parallel to the empty set" are still allowed albeit hopefully not provable.

To solve this oddity, first-order logic is often extended by introducing a notion of types (usually called sorts [136] in the first-order community). Each term of first-order logic is assigned a type and function and predicate symbols come not only with an arity but also with the expected types for their arguments (and in the case of function symbols, the type of the term obtained by applying the function symbol to arguments of the required types). This extended first-order logic syntactically rejects the formula "2 is parallel to the empty set" by assigning 2 to a type of numbers, the empty set a type of sets and the parallelism binary predicate expects two arguments in the type of straight lines.

When first-order theorem provers are used for program verification of programs written in typed programming languages (this is the situation in the FoCaLiZe environment as we will see in Chapter 7) or integrated into typed proof assistants (such as Isabelle [25]), a richer notion of typing is often useful. A polymorphic extension of first-order logic has been added to the TPTP format used by automatic theorem provers [26], we now present this polymorphic first-order logic.

In polymorphic first-order logic, the first syntactic class to be defined is the class of types. Types are built from type variables and type symbols. Type variables are denoted by the letter α and taken from a countable set of type variables disjoint from the set of term variables. Type symbols are introduced with their arities in the signature of the theory. The notions of free variables and substitutions defined in Section 1.1.1 are straightforwardly

adapted to types. We use the letter ρ to denote substitutions of type variables by types.

Instead of only asking for the arity of symbols, we require that each function symbol is introduced with a closed type scheme of the form $\Pi\alpha_1.\ \ldots \Pi\alpha_k.\ (\tau_1, \ldots, \tau_n) \to \tau_0$ and each predicate symbol is introduced with a type scheme of the form $\Pi\alpha_1.\ \ldots \Pi\alpha_k.\ (\tau_1, \ldots, \tau_n)$. The syntactic construct Π binds a type variable α.

For example, integers should be represented by a constant type and the type of polymorphic lists is introduced by a unary type symbol. When a polymorphic function is applied to arguments, the types needed to instantiate its type scheme are explicitly provided as arguments to the function: for example, if f is a function symbol of scheme $\Pi\alpha.\ \mathbf{int} \to \alpha$, we cannot accept the formula $f(0) = f(1)$ because it is ambiguous so we give the instance of the type α as first argument to f and write $f(\mathbf{list}(\mathbf{int}); 0) = f(\mathbf{list}(\mathbf{int}); 1)$. Also to avoid ambiguity, the type of quantified variables is attached to the quantifier so we write $\forall x : \mathbf{int}.\ x = x$ instead of $\forall x.\ x = x$. Finally, type variables can themselves be quantified by type-level quantifiers $\forall_{\mathbf{type}}$ and $\exists_{\mathbf{type}}$.

The syntax and typing rules of polymorphic first-order logic are given in Figure 1.2.

Example 3. *As an example of a polymorphic theory, we consider a theory of polymorphic lists:*

Signature

- **nat** is a constant type symbol
- $0 : () \to \mathbf{nat}$
- $succ : (\mathbf{nat}) \to \mathbf{nat}$
- **list** is a unary type symbol
- $nil : \Pi\alpha.\ () \to \mathbf{list}(\alpha)$
- $cons : \Pi\alpha.\ (\alpha, \mathbf{list}(\alpha)) \to \mathbf{list}(\alpha)$
- $length : \Pi\alpha.\ (\mathbf{list}(\alpha)) \to \mathbf{nat}$

Syntax

Types	τ	$::=$	α
			$F(\tau_1, \ldots, \tau_n)$
Terms	t	$::=$	x
			$f(\tau_1, \ldots, \tau_k; t_1, \ldots, t_n)$
Typing contexts	Γ	$::=$	$\emptyset$
			Γ, α
			$\Gamma, x : \tau$
			$\Gamma, F : n$
			$\Gamma, f : \Pi\alpha_1.\ \ldots \Pi\alpha_k.\ (\tau_1, \ldots, \tau_n) \to \tau_0$
			$\Gamma, P : \Pi\alpha_1.\ \ldots \Pi\alpha_k.\ (\tau_1, \ldots, \tau_n)$
Formulae	φ	$::=$	$P(\tau_1, \ldots, \tau_k; t_1, \ldots, t_n)$
			$t_1 = t_2$
			$\neg\varphi$
			$\varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2$
			$\forall x : \tau.\ \varphi \mid \exists x : \tau.\ \varphi$
			$\forall_{\mathbf{type}}\alpha.\ \varphi \mid \exists_{\mathbf{type}}\alpha.\ \varphi$

Well-typed terms

$$\frac{(\text{when } (x : \tau) \in \Gamma)}{\Gamma \vdash x : \tau}\ (\text{Var})$$

$$\frac{\Gamma \vdash t_1 : \tau_1\rho \quad \ldots \quad \Gamma \vdash t_n : \tau_n\rho \qquad (\text{when } (f : \Pi\alpha_1.\ \ldots \Pi\alpha_k.\ (\tau_1, \ldots, \tau_n) \to \tau_0) \in \Gamma)}{\Gamma \vdash f(\alpha_1\rho, \ldots, \alpha_k\rho; t_1, \ldots, t_n) : \tau_0\rho}\ (\text{App})$$

Well-typed formulae

$$\frac{\Gamma \vdash t_1 : \tau_1\rho \quad \ldots \quad \Gamma \vdash t_n : \tau_n\rho \qquad (\text{when } (P : \Pi\alpha_1.\ \ldots \Pi\alpha_k.\ (\tau_1, \ldots, \tau_n)) \in \Gamma)}{\Gamma \vdash P(\alpha_1\rho, \ldots, \alpha_k\rho; t_1, \ldots, t_n)\ \mathbf{prop}}\ (\text{Atom})$$

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 = t_2\ \mathbf{prop}}\ (=) \qquad \frac{\Gamma \vdash \varphi\ \mathbf{prop}}{\Gamma \vdash \neg\varphi\ \mathbf{prop}}\ (\neg)$$

$$\frac{\Gamma \vdash \varphi_1\ \mathbf{prop} \qquad \Gamma \vdash \varphi_2\ \mathbf{prop}}{\Gamma \vdash \varphi_1 \wedge \varphi_2\ \mathbf{prop}}\ (\wedge) \qquad \frac{\Gamma \vdash \varphi_1\ \mathbf{prop} \qquad \Gamma \vdash \varphi_2\ \mathbf{prop}}{\Gamma \vdash \varphi_1 \vee \varphi_2\ \mathbf{prop}}\ (\vee)$$

$$\frac{\Gamma \vdash \varphi_1\ \mathbf{prop} \qquad \Gamma \vdash \varphi_2\ \mathbf{prop}}{\Gamma \vdash \varphi_1 \Rightarrow \varphi_2\ \mathbf{prop}}\ (\Rightarrow) \qquad \frac{\Gamma \vdash \varphi_1\ \mathbf{prop} \qquad \Gamma \vdash \varphi_2\ \mathbf{prop}}{\Gamma \vdash \varphi_1 \Leftrightarrow \varphi_2\ \mathbf{prop}}\ (\Leftrightarrow)$$

$$\frac{\Gamma, x : \tau \vdash \varphi\ \mathbf{prop}}{\Gamma \vdash \forall x : \tau.\ \varphi\ \mathbf{prop}}\ (\forall) \qquad \frac{\Gamma, x : \tau \vdash \varphi\ \mathbf{prop}}{\Gamma \vdash \exists x : \tau.\ \varphi\ \mathbf{prop}}\ (\exists) \qquad \frac{\Gamma, \alpha \vdash \varphi\ \mathbf{prop}}{\Gamma \vdash \forall_{\mathbf{type}}\alpha.\ \varphi\ \mathbf{prop}}\ (\forall_{\mathbf{type}})$$

$$\frac{\Gamma, \alpha \vdash \varphi\ \mathbf{prop}}{\Gamma \vdash \exists_{\mathbf{type}}\alpha.\ \varphi\ \mathbf{prop}}\ (\exists_{\mathbf{type}})$$

Figure 1.2: Polymorphic First-Order Logic: Syntax and Typing Rules

Axioms

- $\forall m : \mathbf{nat}.\ \forall n : \mathbf{nat}.\ succ(m) = succ(n) \Rightarrow m = n$
- $\forall n : \mathbf{nat}.\ \neg 0 = succ(n)$
- $\forall_{\mathbf{type}} \alpha.\ length(\alpha; nil(\alpha)) = 0$
- $\forall_{\mathbf{type}} \alpha.\ \forall a : \alpha.\ \forall l : \mathbf{list}(\alpha).\ length(\alpha; cons(\alpha; a, l)) = succ(length(\alpha; l))$

In this theory, we can prove for example the formula $\forall_{\mathbf{type}} \alpha.\ \forall a : \alpha.\ \forall l : \mathbf{list}(\alpha).\ \neg cons(\alpha; a, l) = nil(\alpha)$.

Polymorphism has been added to TPTP rather recently so only a few automatic theorem provers already support polymorphism natively. As far as we know, only the SMT solver Alt-Ergo [27], the supperposition theorem provers SPASS [170] and Zipperposition [56] and Zenon [37] (see also Section 9) support this extension.

1.2 Term Rewriting

Term rewriting [20] is a formal theory of computation presented as a succession of elementary steps, each step being an instance of one of the *rewrite rules* defining the *rewrite system* under consideration. We now focus on first-order rewriting, that is the notion of rewriting on the first-order terms that we introduced in Section 1.1.1.1.

The most general way to define a first-order rewrite rule is by giving two first-order terms. The rewrite rule defined by the first-order terms l and r is written $l \longrightarrow r$. In this rule, l is called the left-hand side and r is called the right-hand side. The intended meaning of the rule $l \longrightarrow r$ is that any instance $l\rho$ (where ρ is a substitution) of the pattern l evolves to the corresponding instance $r\rho$ of the term r. It is natural to expect that close terms only rewrite to close terms so we additionally require $\mathsf{FV}(r) \subseteq \mathsf{FV}(l)$ where $\mathsf{FV}(t)$ denotes the set of the variables occurring in t (for first-order terms, there is no notion of bound variable). Moreover, the case where the left-hand side l is a mere variable x is degenerated since any term matches the left-hand side so we also forbid this case. Hence our revised syntax for rewrite-rules is $f(t_1, \ldots, t_n) \longrightarrow r$ where $\mathsf{FV}(r) \subseteq \mathsf{FV}(t_1) \cup \ldots \cup \mathsf{FV}(t_n)$.

A first-order rewrite system is a finite set of first-order rewrite rules. For the rest of this section, we assume a rewrite system $\mathcal{R}$ has been fixed.

The rewriting relation associated with $\mathcal{R}$ is the smallest binary relation containing the rewrite rules of $\mathcal{R}$ and closed under context, in other words the smallest relation $\longrightarrow_{\mathcal{R}}$ such that:

- for every rewrite rule $l \longrightarrow r \in \mathcal{R}$ and every substitution ρ, $l\rho \longrightarrow_{\mathcal{R}} r\rho$, and
- for every function symbol f of arity n, every $k \leq n$, and every terms $t_1, \ldots, t_n$ and t'_k, if $t_k \longrightarrow_{\mathcal{R}} t'_k$ then $f(t_1, \ldots, t_n) \longrightarrow_{\mathcal{R}} f(t_1, \ldots, t_{k-1}, t'_k, t_{k+1}, \ldots, t_n)$.

We say that the term t_1 *rewrites to* the term t_2 when $t_1 \longrightarrow_{\mathcal{R}} t_2$.

The reflexive and transitive closure of $\longrightarrow_{\mathcal{R}}$ is the smallest relation $\longrightarrow^*_{\mathcal{R}}$ such that:

- if $t_1 \longrightarrow_{\mathcal{R}} t_2$ then $t_1 \longrightarrow^*_{\mathcal{R}} t_2$,
- $\longrightarrow^*_{\mathcal{R}}$ is reflexive: $t \longrightarrow^*_{\mathcal{R}} t$, and
- $\longrightarrow^*_{\mathcal{R}}$ is transitive: if $t_1 \longrightarrow^*_{\mathcal{R}} t_2$ and $t_2 \longrightarrow^*_{\mathcal{R}} t_3$ then $t_1 \longrightarrow^*_{\mathcal{R}} t_3$.

The relation $\longrightarrow^*_{\mathcal{R}}$ models a finite sequence of reductions. We say that the term t_1 *reduces to* the term t_2 when $t_1 \longrightarrow^*_{\mathcal{R}} t_2$.

A term t_1 for which no term t_2 exists such that t_1 rewrites to t_2 is called a *normal* term. If $t_1 \longrightarrow^*_{\mathcal{R}} t_2$ and t_2 is normal then we say that t_1 *normalizes to* t_2 and that t_2 is a *normal form* of t_1. In general, not all terms normalize and normal forms, when they exist, are not unique. If all the terms have a normal form, we say that $\mathcal{R}$ is weakly normalizing. If no term has an infinite reduction sequence $t_0 \longrightarrow_{\mathcal{R}} t_1 \longrightarrow_{\mathcal{R}} t_2 \ldots$ then we say that $\mathcal{R}$ is strongly normalizing. If for all terms t_1, t_2, and t_3 such that t_1 reduces to both t_2 and t_3 there exists a term t_4 such that both t_2 and t_3 reduce to t_4 we say that $\mathcal{R}$ is confluent. Confluence, also known as the Church-Rosser property, plays a very important role in rewriting.

The *congruence* induced by $\mathcal{R}$ is the smallest relation $\equiv_{\mathcal{R}}$ such that:

- if $t_1 \longrightarrow_{\mathcal{R}} t_2$ then $t_1 \equiv_{\mathcal{R}} t_2$,

- $\equiv_{\mathcal{R}}$ is reflexive: $t \equiv_{\mathcal{R}} t$,
- $\equiv_{\mathcal{R}}$ is symmetric: if $t_1 \equiv_{\mathcal{R}} t_2$ then $t_2 \equiv_{\mathcal{R}} t_1$, and
- $\equiv_{\mathcal{R}}$ is transitive: if $t_1 \equiv_{\mathcal{R}} t_2$ and $t_2 \equiv_{\mathcal{R}} t_3$ then $t_1 \equiv_{\mathcal{R}} t_3$.

We say that the terms t_1 and t_2 are *congruent modulo* $\mathcal{R}$ when $t_1 \equiv_{\mathcal{R}} t_2$. In general, the relation $\longrightarrow_{\mathcal{R}}$ is decidable but the relations $\longrightarrow^*_{\mathcal{R}}$ and $\equiv_{\mathcal{R}}$ are not. However, the following properties are trivial:

- If $\mathcal{R}$ is strongly normalizing, then we can effectively compute a normal form for each term.
- If $\mathcal{R}$ is both confluent and strongly normalizing, then every term has a unique normal form.
- If $\mathcal{R}$ is both confluent and strongly normalizing, then $\longrightarrow^*_{\mathcal{R}}$ and $\equiv_{\mathcal{R}}$ are decidable.

These are all the properties about term rewriting that we need to consider for the integration of rewriting in first-order logic as it is exemplified in Deduction modulo.

1.3 Deduction Modulo

Deduction modulo [69] is an extension of first-order logic in which theories are not only composed of a signature and a list of axioms, but also of rewrite rules. A proof requiring to perform some computation using the rewrite rules does not need to explicit the computation steps so proofs in Deduction modulo are smaller (see Example 4). Moreover, contrary to axioms, rewrite rules are oriented so, assuming good properties of the rewrite system, a theorem prover in Deduction modulo can blindly apply the rewrite rules instead of backtracking.

For example, when associativity of a symbol $\Box$ is given as a rewrite rule $x\Box(y\Box z) \longrightarrow (x\Box y)\Box z$, the prover does not need to choose a way to reorder parentheses, this choice is imposed by the rewrite system (which is assumed confluent and strongly normalizing). This leads to a reduction of the proof search.

1.3.1 Presentation

In this section, we do not try to give an exhaustive presentation of Deduction modulo but we only highlight a few results which are relevant for our concerns. A recent survey on Deduction modulo can be found in [67].

In Deduction modulo, there are two kinds of axioms that can be turned into rewrite rules:

- Axioms of the form $\forall x_1.\ \ldots \forall x_m.\ f(t_1, \ldots, t_n) = t_0$ where f is a function symbol of arity n and $t_0, \ldots, t_n$ are terms with $\mathsf{FV}(t_0) = \{x_1, \ldots, x_m\} \subseteq \mathsf{FV}(t_1) \cup \ldots \cup \mathsf{FV}(t_n)$.

  The corresponding rewrite rule is $f(t_1, \ldots, t_n) \longrightarrow t_0$, it is called a term rewrite rule.

- Axioms of the form $\forall x_1.\ \ldots \forall x_m.\ P(t_1, \ldots, t_n) \Leftrightarrow \varphi$ where P is a predicate symbol of arity n, $t_1, \ldots, t_n$ are terms, and φ is a formula with $\mathsf{FV}(\varphi) = \{x_1, \ldots, x_m\} \subseteq \mathsf{FV}(t_1) \cup \ldots \cup \mathsf{FV}(t_n)$.

  The corresponding rewrite rule is $P(t_1, \ldots, t_n) \longrightarrow \varphi$, it is called a proposition rewrite rule.

The way axioms are chosen to be replaced by rewrite rules is out of the scope of this thesis.

The definitions of Section 1.2 are extended in the obvious way to define rewriting, reduction, normalization, and congruence of first-order formulae.

1.3.2 Extending First-Order Logic

There are two equivalent ways to extend to Deduction modulo a proof system for first-order logic: the first one, inspired by the conversion rule in type theory, consists in simply adding a proof rule allowing the replacement of a formula by a congruent one:

$$\frac{\Gamma \vdash \varphi \qquad (\text{when } \varphi \equiv_{\mathcal{R}} \psi)}{\Gamma \vdash \psi}\ (\text{Conv})$$

The computation steps used for going from φ to ψ are not recorded in the proof but proofs are still a bit polluted by the new rule since each occurrence of Conv is recorded in

the proof derivation. The second presentation avoids this by integrating the conversion to every rule; for example, the natural deduction rule for introduction of conjunction:

$$\frac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2} \ (\wedge\text{-intro})$$

is transformed into

$$\frac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2 \qquad (\text{when } \psi \equiv_{\mathcal{R}} \varphi_1 \wedge \varphi_2)}{\Gamma \vdash \psi} \ (\wedge\text{-intro})$$

These two approaches are equivalent [69], the former is preferred when studying the connection between Deduction modulo and type theory, the latter is closer to the implementation of theorem provers for Deduction modulo.

Example 4. *Consider the following rewrite system defining the addition of Peano natural numbers:* $0 + n \longrightarrow n, succ(m) + n \longrightarrow succ(m + n)$.

The formula $2 + 2 = 4$ *where* $2 := succ(succ(0))$ *and* $4 := succ(succ(succ(succ(0))))$ *can be proved by a single application of the reflexivity rule:*

$$\frac{(\text{when } t \equiv_{\mathcal{R}} u)}{\Gamma \vdash t = u} \ (\text{reflexivity})$$

1.3.3 Termination and Consistency

Proving consistency of theories is at least as hard in Deduction modulo than in regular first-order logic.

A common technique to prove consistency in first-order logic relies on cut-elimination: if we can define a notion of cut for the theory such that cuts are eliminable and every cut-free proof starts with an introduction rule, then the theory is consistent (because there is no introduction rule for the false proposition).

Since cut-elimination is a statement about the termination of a certain proof transformation procedure, in Deduction modulo we expect it to be linked with termination of the rewrite system. Dowek and Werner [70] remark however that these two notions have few links:

Lemma 1 (Proposition 3.8 in [70]). *If a theory in Deduction modulo contains only rewrite rules at the level of terms (no axiom and no rewrite rule at the level of propositions), then cuts are eliminable hence the theory is consistent, regardless of the terminating status of the rewrite system.*

Example 5 (Section 2.3 in [70]). *Let A be a constant (a function symbol of arity 0) and $\bullet \in \bullet$ be a binary predicate symbol, the axiom-free theory containing the terminating rewrite rule $A \in A \longrightarrow \forall x, (x = A \Rightarrow \neg(x \in A))$ only is inconsistent.*

1.4 Zenon Modulo

The two main techniques for automated theorem proving in first-order logic are the Tableaux Method [161] and Resolution [152]. Deduction modulo has been implemented on top of both: Zenon Modulo [61] and iProver Modulo [36] are extensions to Deduction modulo of, respectively, the tableaux prover Zenon and the resolution prover iProver.

Zenon [31] is able to read problems both in the standard TPTP format and in (a fragment of) Coq syntax. Zenon is one of the very few theorem provers able to produce an independently checkable proof; the output format of Zenon is Coq.

Zenon Modulo [61] is an extension of Zenon to Deduction modulo, it is developed by Pierre Halmagrand. Coq implements a type theory and can represent some computation using the conversion rule but this is not enough for Deduction modulo: Coq is strongly normalizing so its conversion can not express computation defined by a rewrite system for which termination is unknown. For this reason, Zenon Modulo has been adapted to produce proofs in Dedukti format.

It has been shown that proof search in Zenon is more effective when using Deduction modulo [38].

Chapter 2

λ-Calculus and Type Theory

Type theory was invented by Russell [153] in 1908 to solve the paradoxes of naive set theory that had just been discovered a few years before.

The typing discipline corresponds to the mathematical habit of not interchanging objects of different natures. In planar geometry for example, the parallelism relation applies only to straight lines; statements such as "2 is parallel to the empty set" are not rejected because they are false statements, they must be ruled-out because they carry no meaning.

Types also have a wide range of applications in programming languages. Most languages assign types to data at least to indicate how much size they have in memory. This assignation happens either dynamically during the program evaluation or statically during the compilation of the program. In the case of static typing, types are used to ensure that some dynamic errors such as trying to apply a number as a function will not appear. Types are also of a great help for reporting errors, for guiding compiler optimizations [114], and for ensuring various properties.

In the particular case of the λ-calculus at the heart of the functional languages, many type systems have been studied. We present λ-calculus in Section 2.1, the simplest type system for λ-calculus in Section 2.2, and we extend this system to polymorphism in Section 2.3 to obtain the type system used in real functional programming languages. By adding another feature called dependent typing, type systems can be used to encode logics. We present dependent type systems in Section 2.4 and a particular class of type systems called logical frameworks specialized in the encoding of logical systems in Section 2.5.

2.1 λ-Calculus

The λ-calculus is the core calculus of functional programming languages. It is defined by a syntax describing which terms belong to the language and a semantics describing how programs are evaluated.

The syntax of the pure λ-calculus has only one category: the category of λ-terms. λ-terms are built from variables, binary application, and the original operation of λ-abstraction:

$$\begin{array}{llll} \textbf{λ-terms} & t & ::= & x & \textit{Variable} \\ & & & t_1\ t_2 & \textit{Application} \\ & & & \lambda x.\ t & \lambda\textit{-abstraction} \end{array}$$

Application roughly corresponds to the usual operation consisting of applying a function to its argument and is written $t_1\ t_2$ instead of the more common mathematical notation $t_1(t_2)$ in order to save parentheses. To save more parentheses, we take the convention that application associates to the left: $t_1\ t_2\ t_3$ should be read as $(t_1\ t_2)\ t_3$, not $t_1\ (t_2\ t_3)$.

λ-abstraction is the way functions are defined in λ-calculus: the term $\lambda x.\ t$ corresponds roughly to the function returning t when applied to x. In the term $\lambda x.\ t$, the variable x is bound in the term t (the notion of binding has been introduced in Section 1.1.1.2). In the context of the λ-calculus, the renaming operation is called *α-renaming*, *α-conversion*, or *α-equivalence*. For example, the terms $\lambda x.\ \lambda y.\ x$ and $\lambda z.\ \lambda x.\ z$ are *α-equivalent*. We are usually interested in λ-terms modulo α-equivalence only.

The notions of free variables and substitutions are the same as for first-order formulae. The set of free variables $\mathsf{FV}(t)$ of a term t is defined as follows:

$$\begin{array}{lcl} \mathsf{FV}(x) & := & x \\ \mathsf{FV}(t_1\ t_2) & := & \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2) \\ \mathsf{FV}(\lambda x.\ t) & := & \mathsf{FV}(t) \setminus \{x\} \end{array}$$

It is always possible to α-rename a λ-term in such a way that free and bound variables form disjoint sets.

The substitution of the variable x by the term t_1 in the term t_2 is written $t_2\{x \backslash t_1\}$ and defined as follows:

$$
\begin{array}{lcll}
x\{x\backslash t_1\} & := & t_1 & \\
y\{x\backslash t_1\} & := & y & \text{when } y \neq x \\
(t_2\ t_3)\{x\backslash t_1\} & := & (t_2\{x\backslash t_1\})\ (t_3\{x\backslash t_1\}) & \\
(\lambda x.\ t)\{x\backslash t_1\} & := & \mathrm{t} & \\
(\lambda y.\ t)\{x\backslash t_1\} & := & \lambda y.\ (t\{x\backslash t_1\}) & \text{when } y \notin \mathsf{FV}(t_1)
\end{array}
$$

λ-calculus is given a computational meaning by defining the β rewriting rule:

- $(\lambda x.\ t_2)\ t_1 \longrightarrow t_2\{x\backslash t_1\}$

This rule is not a first-order rewrite rule in the sense discussed in Section 1.2 because of the presence of the λ binder in the left-hand side and the substitution in the right-hand side so we need to generalize the definitions of Section 1.2.

β-reduction is the smallest relation $\longrightarrow_\beta$ such that:

- $(\lambda x.\ t_2)\ t_1 \longrightarrow_\beta t_2\{x\backslash t_1\}$, and
- $\longrightarrow_\beta$ is a closed under context:
  - if $t_1 \longrightarrow_\beta t_2$ then $t_1\ t_3 \longrightarrow_\beta t_2\ t_3$ and $t_3\ t_1 \longrightarrow_\beta t_3\ t_2$, and
  - if $t_1 \longrightarrow_\beta t_2$ then $\lambda x.\ t_1 \longrightarrow_\beta \lambda x.\ t_2$.

For example, the term $\lambda x.\ (\lambda y.\ x)\ x$ β-reduces to $\lambda x.\ x$ and the term $(\lambda x.\ x\ x)\ (\lambda y.\ y\ y)$ β-reduces to itself.

We denote by $\longrightarrow_\beta^*$ the reflexive and transitive closure of $\longrightarrow_\beta$ defined as the smallest relation such that:

- if $t_1 \longrightarrow_\beta t_2$ then $t_1 \longrightarrow_\beta^* t_2$,
- $\longrightarrow_\beta^*$ is reflexive: $t \longrightarrow_\beta^* t$, and
- $\longrightarrow_\beta^*$ is transitive: if $t_1 \longrightarrow_\beta^* t_2$ and $t_2 \longrightarrow_\beta^* t_3$ then $t_1 \longrightarrow_\beta^* t_3$.

Finally, we denote by $\equiv_\beta$ the reflexive, symmetric and transitive closure of $\longrightarrow_\beta$ defined as the smallest relation such that:

- if $t_1 \longrightarrow_\beta t_2$ then $t_1 \equiv_\beta t_2$,

- $\equiv_\beta$ is reflexive: $t \equiv_\beta t$,
- $\equiv_\beta$ is symmetric: if $t_1 \equiv_\beta t_2$, then $t_2 \equiv_\beta t_1$,
- $\equiv_\beta$ is transitive: if $t_1 \equiv_\beta t_2$ and $t_2 \equiv_\beta t_3$ then $t_1 \equiv_\beta t_3$.

A λ-term t_1 for which no λ-term t_2 exists such that $t_1 \longrightarrow_\beta t_2$ is called a *normal* term. For example, the identity $\lambda x.\ x$ is normal. If $t_1 \longrightarrow_\beta^* t_2$ and t_2 is normal then we say that t_1 *normalizes to* t_2 and that t_2 is the *normal form* of t_1. Not all terms normalize, for example the self-reducing term $(\lambda x.\ x\ x)\ (\lambda y.\ y\ y)$ does not normalize but the normal form of a normalizing term is unique. This is a consequence of the Church-Rosser theorem [48] stating that β-reduction is a confluent relation:

Theorem 1 (Church-Rosser). *Let t_1, t_2, and t_3 be λ-terms such that $t_1 \longrightarrow_\beta^* t_2$ and $t_1 \longrightarrow_\beta^* t_3$, there exists a λ-term t_4 such that $t_2 \longrightarrow_\beta^* t_4$ and $t_3 \longrightarrow_\beta^* t_4$.*

We can distinguish three kinds of λ-terms:

- *diverging terms* having no normal form such as $(\lambda x.\ x\ x)\ (\lambda y.\ y\ y)$
- *weakly normalizing* terms which have normal forms but also infinite reduction sequences such as $(\lambda x.\ \lambda y.\ y)\ ((\lambda x.\ x\ x)\ (\lambda y.\ y\ y))$ which reduces both to itself and to the normal term $\lambda y.\ y$
- *strongly normalizing* terms for which all reduction sequences are finite.

Despite its simplicity, λ-calculus is a Turing-complete model of computation. In particular, natural numbers can be encoded in various ways, the simplest of which is probably Church numerals. In this encoding of numerals, the natural number n is represented as the operation consisting of iterating functions n times:

$$\underline{n} := \lambda f.\ \lambda x.\ \overbrace{f\ (f\ (\ldots (f}^{n \text{ times}}\ x) \ldots))$$

Some arithmetic functions are easy to define in this setting:

$$\begin{aligned} 0 &:= \lambda f.\ \lambda x.\ x \\ succ &:= \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x) \\ + &:= \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x) \\ \times &:= \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ (n\ f)\ x \end{aligned}$$

2.2 Simple Types

The most simple way of assigning types to λ-terms is given by the simply-typed λ-calculus. Simple types serve two purposes:

- they are used to forbid some meaningless terms, they can be conceived as a weak form of specification,

- they guarantee termination: all well-typed terms are strongly normalizing.

In this section, we give the syntax and then the basic properties of the simply-typed λ-calculus.

If τ_1 and τ_2 are types, we can build the type $\tau_1 \to \tau_2$ which is the type of functions taking an argument of type τ_1 and returning a value of type τ_2. To save parentheses, we take the convention that $\to$ is right-associative: $\tau_1 \to \tau_2 \to \tau_3$ means $\tau_1 \to (\tau_2 \to \tau_3)$. We also add basic types denoted by the letter i so that at least one type can be constructed. The precise nature of these basic types is not relevant for the simply-typed λ-calculus. In practice, they are often defined by a first-order signature as we defined first-order terms in Section 1.1.1.1.

The terms of the simply-typed λ-calculus are the λ-terms that we introduced in Section 2.1; this is called the presentation of simply-typed λ-calculus *a la Curry* as opposed to the presentation of simply-typed λ-calculus *a la Church* where λ-abstractions are decorated by the type assigned to the abstracted variable.

In order to check that a given λ-term has a given type, we require the type of all the free variables. They are provided by a typing context Γ: a finite list of variable declarations of the form $x : \tau$. When a context Γ is extended by a new declaration $x : \tau$, we write the extended context $\Gamma, x : \tau$. The full syntax of simply-typed λ-calculus is as follows:

Typing

$$\frac{(\text{when } (x : A) \in \Gamma)}{\Gamma \vdash x : A}\ (\text{Var}) \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}\ (\text{App})$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2 \qquad (\text{when } x \notin \Gamma)}{\Gamma \vdash \lambda x.\ t : \tau_1 \to \tau_2}\ (\text{Abs})$$

Figure 2.1: Typing rules for simply-typed λ-calculus

Simple types	τ	::=	i	*Basic type*
			$\tau_1 \to \tau_2$	*Arrow type*
λ-terms	t	::=	x	*Variable*
			$t_1\ t_2$	*Application*
			$\lambda x.\ t$	*Abstraction*
Typing contexts	Γ	::=	$\emptyset$	*Empty context*
			$\Gamma, x : \tau$	*Extended context*

We write $x \in \Gamma$ to indicate that x is declared in Γ and $(x : \tau) \in \Gamma$ to indicate that the declaration $x : \tau$ is present in Γ. Types are associated to terms using an inductive relation: the typing judgment $\Gamma \vdash t : \tau$ defined in Figure 2.1.

A *well-formed* context is a context in which each variable is declared at most once. The side condition in (Abs) rule is here to preserve well-formedness.

This type system satisfies a few important properties such as decidability and termination which are the main motivations for introducing types.

Theorem 2 (Decidability)**.** *Given a typing context Γ, a term t and a type τ, the judgment $\Gamma \vdash t : \tau$ is decidable.*

Theorem 3 (Type inference)**.** *Given a typing context Γ and a term t, we can decide whether or not there exists a type τ such that $\Gamma \vdash t : \tau$.*

This result is constructive in the sense that when such a type τ exists, it can actually be computed from Γ and t.

Theorem 4 (Subject reduction). *If* $\Gamma \vdash t_1 : \tau$ *and* $t_1 \longrightarrow_\beta t_2$ *then* $\Gamma \vdash t_2 : \tau$.

Theorem 5 (Termination). *If* $\Gamma \vdash t_1 : \tau$ *then* t_1 *is strongly normalizing.*

2.3 Polymorphism

Simple types lead to a lot of code duplication because the same untyped λ-term can usually be given several types. For example, the identity function $\lambda x.\ x$ can be assigned all the types of the form $\tau \to \tau$ and the composition of functions $\lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x)$ can be assigned all the types of the form $(\tau_2 \to \tau_3) \to (\tau_1 \to \tau_2) \to \tau_1 \to \tau_3$. When the λ-calculus is used as a basis for a programming language, this duplication is not acceptable. We do not want to duplicate the code of the programs in order to get the right to use them on different types.

This problem is solved by enriching the type system by *polymorphism*. Polymorphism is the ability to define functions acting on several types; two kinds of polymorphism can be distinguished: ad-hoc polymorphism and parametric polymorphism.

An ad-hoc polymorphic function can act differently depending on the type of its argument. Ad-hoc polymorphism is usually defined by overloading a function symbol with several types and definitions. A typical example of ad-hoc polymorphism in programming is the printing function. Many programming languages provide a function `print` which prints its argument and is defined differently depending on the type of the argument: for example, numbers are converted in decimal notation before printing.

Parametric polymorphic functions on the other side are functions whose definitions are generic in one or more types; they do not inspect the type of their arguments. The identity function and the composition of functions are examples of parametric polymorphic functions.

The most common polymorphic type system implemented in functional programming languages such as OCaml and Haskell is the Damas-Hindley-Milner type system [59]. It extends the simply-typed λ-calculus with parametric polymorphism. This type system is also used to define higher-order logic.

We present Damas-Hindley-Milner type system in Section 2.3.1 and higher-order logic

in Section 2.3.2.

2.3.1 Damas-Hindley-Milner Type System

To provide a powerful type system for the functional programming language ML, Damas, and Milner [59] have proposed a polymorphic type system which had already been discovered by Hindley [96] in the context of combinatory logic. Damas-Hindley-Milner type system is also known as ML-like polymorphism, prenex polymorphism, let-polymorphism, and rank-1 polymorphism. There are a few equivalent presentations of this type system, one of the simplest is the syntax-directed presentation [53] that we adopt here.

In order to write the types of the identity function and of the composition of functions, we add type variables denoted by α. The term $\lambda x.\ x$ still accepts multiple types such as $\iota \to \iota$ and $(\iota \to \iota) \to (\iota \to \iota)$ but it has a *principle* type $\alpha \to \alpha$; all the types of $\lambda x.\ x$ are obtained from $\alpha \to \alpha$ by substituting the type variable α. Similarly, $\lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x)$ will have the principle type $(\alpha_2 \to \alpha_3) \to (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_3$. All λ-terms which are well-typed with respect to the simply-typed λ-calculus have principle types and principle types are efficiently computable.

In order to bind the variables occurring in types, we introduce the notion of *type schemes*. Type schemes have the form $\Pi\alpha_1.\ \ldots \Pi\alpha_n.\ \tau$[1], they appear instead of types in typing contexts.

Polymorphic terms are introduced by named local definitions with the syntax $\mathbf{let}\ x\ :=\ t_1\ \mathbf{in}\ t_2$ which is semantically equivalent to $(\lambda x.\ t_2)\ t_1$ but has a more liberal typing rule: in the case of the local definition, t_2 is checked in a typing context in which x is assigned a type scheme whereas in the case of the abstraction $\lambda x.\ t_2$, we are only allowed to assign a type to x while checking t_2. The syntax and typing rules of Damas-Hindley-Milner type system are given in Figure 2.2.

Contrary to other extensions of simply-typed λ-calculus to polymorphism such as System F [82], type inference is decidable in Damas-Hindley-Milner type system.

Theorem 6 (Type inference)**.** *Given a typing context* Γ *and a term* t*, we can decide*

[1]We use the notation Π for prenex quantification instead of the more common notation $\forall$ in order to avoid confusion with the logical universal quantifier.

Syntax

Types	τ	::=	α	*Type variable*
			ι	*Atomic type*
			$\tau_1 \to \tau_2$	*Arrow type*
Type schemes	σ	::=	τ	*Type*
			$\Pi\alpha.\ \sigma$	*Universal scheme*
Terms	t	::=	x	*Variable*
			$t_1\ t_2$	*Application*
			$\lambda x.\ t$	*Abstraction*
			$\textbf{let}\ x\ :=\ t_1\ \textbf{in}\ t_2$	*Local definition*
Typing contexts	Γ	::=	$\emptyset$	*Empty context*
			$\Gamma, x : \sigma$	*Extended context*

Typing

$$\frac{(\text{when } (x : \Pi\alpha_1.\ \ldots \Pi\alpha_n.\ \tau) \in \Gamma)}{\Gamma \vdash x : \tau\{\alpha_1 \backslash \tau_1, \ldots \alpha_n \backslash \tau_n\}}\ (\text{Var}) \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}\ (\text{App})$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\ t : \tau_1 \to \tau_2}\ (\text{Abs})$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \begin{array}{c}\Gamma, x : \Pi\alpha_1.\ \ldots \Pi\alpha_n.\ \tau_1 \vdash t_2 : \tau_2 \\ (\text{when } \{\alpha_1, \ldots, \alpha_n\} = \mathsf{FV}(\tau_1) \setminus \mathsf{FV}(\Gamma))\end{array}}{\Gamma \vdash \textbf{let}\ x\ :=\ t_1\ \textbf{in}\ t_2 : \tau_2}\ (\text{Let})$$

Figure 2.2: Damas-Hindley-Milner type system

whether or not there exists a type τ *such that* $\Gamma \vdash t : \tau$.

As in the simply-typed case, this result is constructive in the sense that when such a type τ exists, it can actually be computed from Γ and t.

2.3.2 HOL

Prenex polymorphism is also the type discipline adopted in Church simple theory of types, also known as Higher-Order Logic (HOL for short). HOL has been implemented in various proof assistants: HOL Light, HOL4, HOL Zero, ProofPower-HOL and Isabelle/HOL. These proof assistants are commonly referred to as the HOL family.

As its name suggests, HOL is a logic in which quantification is allowed at all orders: in HOL we can quantify over logical propositions, predicates and arbitrary λ-terms. The type system of HOL is simply obtained by extending Damas-Hindley-Milner type system by a new atomic type o. Logical propositions are terms of type o and predicates over a type τ are terms of type $\tau \to o$.

As a logical system, HOL is axiomatized by a typing context called the signature and a set of rules for deriving new theorems. All theorem statements should be well-typed terms of type o in the signature.

The usual axiomatization of HOL which is implemented in proof assistants of the HOL family is named $\mathcal{Q}_0$, it has been proposed by Andrews [7]. The signature of $\mathcal{Q}_0$ is $\doteq : \Pi\alpha.\ \alpha \to \alpha \to o, \epsilon : \Pi\alpha.\ (\alpha \to o) \to \alpha$. We write $t_1 = t_2$ for $\doteq\ t_1\ t_2$. The deduction rules of $\mathcal{Q}_0$ are as follows:

Deduction rules

$$\frac{}{p \vdash p}\ (\text{Assume}) \qquad \frac{}{\vdash t = t}\ (\text{Refl}) \qquad \frac{\vdash t_1 = t_2}{\vdash \lambda x.\ t_1 = \lambda x.\ t_2}\ (\text{AbsThm})$$

$$\frac{\vdash t_1 = t_2 \qquad \vdash t_3 = t_4}{\vdash t_1\ t_3 = t_2\ t_4}\ (\text{AppThm}) \qquad \frac{\vdash p = q \qquad \vdash p}{\vdash q}\ (\text{EqMP})$$

$$\frac{q \vdash p \qquad p \vdash q}{\vdash p = q}\ (\text{DeductAntiSym}) \qquad \frac{}{\vdash \lambda x.\ t\ x = t}\ (\eta\text{-equality}) \qquad \frac{}{p\ t \vdash p\ (\epsilon\ p)}\ (\text{Choice})$$

In $\mathcal{Q}_0$, functional extensionality $f\ x = g\ x \vdash f = g$ is provable from AbsThm and η-equality. From extensionality and Choice, the Law of Excluded Middle can be derived by Diaconescu Theorem [65] hence $\mathcal{Q}_0$ defines a classical logic.

If we want to work in an intuitionistic higher-order logic, we can take the universal quantifier and the implication as primitives instead of equality and the choice operator. The signature of this alternative axiomatization is $\Sigma := \dot{\forall} : \Pi\alpha.\ (\alpha \to o) \to o, \dot{\Rightarrow} : o \to o \to o$. We write $\forall x.\ p$ instead of $\dot{\forall}(\lambda x.\ p)$ and $p \Rightarrow q$ instead of $\dot{\Rightarrow}\ p\ q$. The derivation rules are as follows:

Deduction rules

$$\frac{}{p \vdash p}\ (\text{Assume}) \qquad \frac{\vdash p \Rightarrow q \qquad \vdash p}{\vdash q}\ (\Rightarrow\text{-elim}) \qquad \frac{p \vdash q}{\vdash p \Rightarrow q}\ (\Rightarrow\text{-intro}) \qquad \frac{\vdash \dot{\forall} p}{\vdash p\ t}\ (\forall\text{-elim})$$

$$\frac{\vdash p\ x}{\vdash \dot{\forall} p}\ (\forall\text{-intro})$$

In this axiomatization, equality can be defined using Leibnitz definition $\doteq\ :=\ \lambda x.\ \lambda y.\ \forall p.\ p\ x \Rightarrow p\ y$.

2.4 Dependent Types

Types can be seen as weak specifications of λ-terms. If a term t has type $\tau_1 \to \tau_2$ then we know that t will produce values of type τ_2 when applied to arguments of type τ_1.

In order to enrich the expressivity of types as program specifications, we can enrich the type system by types depending on terms called dependent types. For example, in a dependently-typed programming language, we can define the type of lists parameterized by their length, usually called the type of vectors. Adding an element at the head of a vector of length n should return a vector of length $n + 1$ so this operation has type $\texttt{vector}\ n \to \texttt{vector}\ (n + 1)$.

De Bruijn introduced dependent types in the late 60s in the logical framework Automath [132] to encode the judgments of various object logics. This use of dependent types will be the subject of Section 2.5. A few years later, Scott generalized De Bruijn's idea to represent all the intuitionistic logical connectives using dependent types [159] but he did so without distinguishing λ-abstraction and universal quantification so the types themselves could not be quantified upon, which leads to a lot of complications. Martin-Löf introduced the dependent product to form a type theory expressive enough to represent intuitionistic predicate logic through the Curry-Howard correspondence and he proved its consistency [123].

We present Martin-Löf Type Theory in Section 2.4.1, we demonstrate its use as a logical system by an encoding of natural deduction in Section 2.4.2, and we present the

type system implemented by the Coq proof assistant in Section 2.4.3.

2.4.1 Martin-Löf Type Theory

The Curry-Howard correspondence is a one-to-one correspondence between proof systems and computational models. It was first discovered by Curry in [58] in the context of combinator calculus and Hilbert system and then extended by Howard to natural deduction for minimal propositional logic and simply-typed λ-calculus [98].

The Brouwer-Heyting-Kolmogorov (BHK) interpretation [108] is an informal explanation of the meaning of intuitionistic proofs. Following the BHK interpretation, a formula in intuitionistic logic can be read as a programming task and a proof as a program accomplishing this task. Each intuitionistic connective can be associated a meaning in this interpretation: for example, the task $A \wedge B$ consists in accomplishing both tasks A and B and the task $\exists x.\ P(x)$ consists in computing a witness a of P and accomplishing the task $P(a)$.

Because intuitionistic logic features a nice interpretation of proofs as programs, it is a good candidate for the Curry-Howard correspondence. Martin-Löf Type Theory (MLTT) is a type system for an extended λ-calculus in which formulae can be represented as types and proofs as terms.

To represent intuitionistic connectives, MLTT features inductive types freely generated by a set of constructors. The definitions of these inductive types all follow the same pattern: we first give the constructors with their types, then the elimination principle denoted by E_A for each type A states that all values of inductive types start with a constructor. Finally, we add computation rules representing cut-elimination: one rule for each possible way of applying the elimination principle to a constructor. Actual implementations of MLTT such as Agda [134] propose this general scheme as a syntactic construct called inductive definition. The users of these implementations are free to define their own inductive types and the type constructors proposed by Martin-Löf to represent logic through the Curry-Howard correspondence are special cases of inductive definitions. Defining inductive definitions precisely is however a quite subtle topic so we rather adopt a presentation close to the one by Martin-Löf in [122].

Types in MLTT are themselves elements of universes. An infinite hierarchy of universes $\mathbf{Type}_0, \mathbf{Type}_1, \ldots$ is assumed, each universe inhabits the following one and each universe is closed with respect to the type-forming operations that we are about to define.

Apart from the universes, the basic types of MLTT are the empty type 0 and the unit type 1. Two types A and B can be combined by forming their disjoint sum $A + B$. Moreover, two binding constructs for building types are available: the dependent sum $\Sigma x : A.\ B$ and the dependent product $\Pi x : A.\ B$. In both constructs, the variable x is bound in B. Finally, from a type A and two terms a and a' in A, we can build the identity type $\mathbf{Eq}(A, a, a')$.

All these syntactic constructs have two readings. They can either be understood as describing sets of terms of a certain shape or logical propositions:

- The empty type 0 has no inhabitant and corresponds to logical falsehood.
- The unit type 1 is a singleton and corresponds to logical truth.
- The disjoint sum $A + B$ contains terms of the form $\mathbf{inl}(a)$ where a inhabits A and terms of the form $\mathbf{inr}(b)$ where b inhabits B. The disjoint sum corresponds to the logical disjunction $A \vee B$.
- The dependent sum $\Sigma x : A.\ B$ contains all the pairs (a, b) where a inhabits A and b inhabits $B\{x \backslash a\}$. In particular, the type of the second component of these pairs may depend on the value of the first component. The dependent sum corresponds to the logical existential quantification $\exists x : A.\ B$.
- The dependent product $\Pi x : A.\ B$ contains all the functions of the form $\lambda x : A.\ b$ where x is bound in b and b inhabits B. In particular, the variable x representing the argument of the function may appear not only in the returned value b but also in its type B. The dependent product corresponds to the logical universal quantification $\forall x : A.\ B$.
- The identity type $\mathbf{Eq}(A, a, a')$ contains only the reflexivity proof. The term $\mathbf{refl}(A, a)$ inhabits $\mathbf{Eq}(A, a, a)$. The identity type corresponds to the logical equality $a =_A a'$.

The other logical connectives can be derived as special cases:

- The conjunction $A \wedge B$ is represented in MLTT by the Cartesian product $A \times B$ which is defined as the non-dependent case of dependent sum: $(A \times B) := (\Sigma x : A.\ B)$ where x does not occur free in B. Hence inhabitants of $A \times B$ are the pairs (a, b) where a inhabits A and b inhabits B.
- The implication $A \Rightarrow B$ is represented in MLTT by the arrow type $A \to B$ which is defined as the non-dependent case of dependent product: $(A \to B) := (\Pi x : A.\ B)$ where x does not occur free in B. Hence inhabitants of $A \to B$ are the functions $\lambda x : A.\ b$ such that b inhabits B (the variable x may appear in b but not in B).
- Negation is defined as usual in intuitionistic logic: $(\neg A) := (A \to 0)$.
- Equivalence is also defined as usual in intuitionistic logic: $(A \leftrightarrow B) := (A \to B) \times (B \to A)$.

We have explained all the ways by which we can construct inhabitants of types but not yet how to use them. For example, given two types A and B, we are not yet able to construct a term of type $(A + B) \to (B + A)$ which logically reads as commutativity of disjunction. If we try, we start by constructing the term $\lambda x : (A + B).\ c$ where c is a term of type $B + A$ that we have to provide (and which might use the variable x). Now we need to internalize in the type theory the principle that all inhabitants of $A + B$ have one of the following shape : $\mathbf{inl}(a)$ or $\mathbf{inr}(b)$. By internalizing, we mean adding a new syntactic construct in the theory to do this.

This new construct is called the eliminator of disjoint sums and written $E_{A+B}(t, z : A + B.\ C, x : A.\ c, y : B.\ d)$. This eliminator is a new binder, the variables x, y, and z are bound respectively in the terms c, d, and C. The programming reading of the eliminator of disjoint sums is a pattern matching construct. The term t is matched against the two possible shapes $\mathbf{inl}(x)$ and $\mathbf{inr}(y)$. In the first case, the branch defined by the term c is chosen; in the second case, the branch defined by the term d is chosen. The subtlety of this construct comes from its typing rule. The types of both branches do not need to be identical but they may depend on the matched term. This dependency is handled by the

type C, the required type for c is $C\{z\backslash\mathbf{inl}(x)\}$, the required type for d is $C\{z\backslash\mathbf{inr}(y)\}$, and the returned type for the whole expression $E_{A+B}(t, z : A + B.\ C, x : A.\ c, y : B.\ d)$ is $C\{z\backslash t\}$. The logical reading of this new construct is reasoning by case depending on the shape of t; if t has the shape $\mathbf{inl}(x)$ then the first branch (the term c) provides a way to prove $C\{z\backslash t\}$, if on the contrary t has the shape $\mathbf{inr}(y)$ then the second branch (the term d) provides a way to prove $C\{z\backslash t\}$. As a reasoning tool, the eliminator of disjoint sum hence corresponds to the natural deduction rule of elimination of disjunction (see Section 1.1.2). The computational behaviour of the eliminator is provided by the following reduction rules:

- $E_{A+B}(\mathbf{inl}(a), z : A + B.\ C, x : A.\ c, y : B.\ d) \longrightarrow c\{x\backslash a\}$,
- $E_{A+B}(\mathbf{inr}(b), z : A + B.\ C, x : A.\ c, y : B.\ d) \longrightarrow d\{y\backslash b\}$.

Similar eliminators can be added for all the type constructors. We shall not describe them in detail.

Formally, the judgments of MLTT are the following:

- $\Gamma \vdash$ meaning that Γ is a well-formed typing context,
- $\Gamma \vdash t : A$ meaning that t is a term of type A,
- $\Gamma \vdash t \equiv u : A$ meaning that t and u are convertible terms of type A.

The last judgment $\Gamma \vdash t \equiv u : A$ should not be confused with the judgment $\Gamma \vdash v : \mathbf{Eq}(A, t, u)$. The judgment $\Gamma \vdash t \equiv u : A$ implies $\Gamma \vdash \mathbf{refl}(A, t) : \mathbf{Eq}(A, t, u)$ but the converse does not hold because the judgment $\Gamma \vdash t \equiv u : A$ is decidable but the existence of a v such that $\Gamma \vdash v : \mathbf{Eq}(A, t, u)$ is not.

The rules related to disjoint sum are given in Figure 2.3. We will not attempt to list all the other typing rules of MLTT but we only highlight the most interesting rule of the system which is the conversion rule:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B : \mathbf{Type}_i}{\Gamma \vdash t : B}\ (\text{Conv})$$

As in Deduction modulo (see Section 1.3), this rule can be used to let huge computations implicit by following the Poincaré principle [24].

$$\frac{\Gamma \vdash A : \mathbf{Type}_i \qquad \Gamma \vdash B : \mathbf{Type}_i}{\Gamma \vdash A + B : \mathbf{Type}_i}\ (+\text{-formation}) \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : \mathbf{Type}_i}{\Gamma \vdash \mathbf{inl}(a) : A + B}\ (\text{inl})$$

$$\frac{\Gamma \vdash A : \mathbf{Type}_i \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathbf{inr}(b) : A + B}\ (\text{inr})$$

$$\frac{\Gamma, z : A + B \vdash C : \mathbf{Type}_i \quad \Gamma \vdash t : A + B \quad \Gamma, x : A \vdash c : C\{z \backslash \mathbf{inl}(x)\} \quad \Gamma, y : B \vdash d : C\{z \backslash \mathbf{inr}(y)\}}{\Gamma \vdash E_{A+B}(t, z : A + B.\ C, x : A.\ c, y : B.\ d) : C\{z \backslash t\}}\ (+\text{-elim})$$

$$\frac{\Gamma, z : A + B \vdash C : \mathbf{Type}_i \quad \Gamma \vdash a : A \quad \Gamma, x : A \vdash c : C\{z \backslash \mathbf{inl}(x)\} \quad \Gamma, y : B \vdash d : C\{z \backslash \mathbf{inr}(y)\}}{\Gamma \vdash E_{A+B}(\mathbf{inl}(a), z : A + B.\ C, x : A.\ c, y : B.\ d) \equiv c\{x \backslash a\} : C\{z \backslash \mathbf{inl}(a)\}}\ (+\text{-elim-inl})$$

$$\frac{\Gamma, z : A + B \vdash C : \mathbf{Type}_i \quad \Gamma \vdash b : B \quad \Gamma, x : A \vdash c : C\{z \backslash \mathbf{inl}(x)\} \quad \Gamma, y : B \vdash d : C\{z \backslash \mathbf{inr}(y)\}}{\Gamma \vdash E_{A+B}(\mathbf{inr}(b), z : A + B.\ C, x : A.\ c, y : B.\ d) \equiv d\{y \backslash b\} : C\{z \backslash \mathbf{inr}(b)\}}\ (+\text{-elim-inr})$$

Figure 2.3: Typing rules for disjoint sums in MLTT

2.4.2 Curry-Howard Correspondence for Natural Deduction

Through the Curry-Howard correspondence, the type system of the simply-typed λ-calculus that we described in Section 2.2 corresponds exactly to minimal natural deduction, the fragment of natural deduction where the only available connective is implication obtained by taking only the rules Axiom, $\Rightarrow$-intro and $\Rightarrow$-elim from Figure 1.1.

It is possible to extend the Curry-Howard correspondence by embedding natural deduction in MLTT. To each function symbol f we associate a variable f of type $\overbrace{I \to \ldots \to I}^{n \text{ times}} \to I$ where n is the arity of f and I is a fixed type in the first universe $\mathbf{Type}_0$; to each predicate symbol P we associate a variable P of type $\overbrace{I \to \ldots \to I}^{n \text{ times}} \to \mathbf{Type}_0$ where n is the arity of P. In this context, we can define a well-typed translation of first-order terms and formulae to MLTT; first-order terms are translated as terms of type I and first-order formulae are translated as types in the first universe $\mathbf{Type}_0$.

$$
\begin{array}{lcl}
[\![x]\!] & := & x \\
[\![f(t_1,\ldots,t_n)]\!] & := & f\ t_1\ \ldots\ t_n \\
 & & \\
[\![P(t_1,\ldots,t_n)]\!] & := & P\ t_1\ \ldots\ t_n \\
[\![t_1 = t_2]\!] & := & \mathbf{Eq}(I, t_1, t_2) \\
[\![\top]\!] & := & 1 \\
[\![\bot]\!] & := & 0 \\
[\![\neg\varphi]\!] & := & [\![\varphi]\!] \to 0 \\
[\![\varphi_1 \wedge \varphi_2]\!] & := & [\![\varphi_1]\!] \times [\![\varphi_2]\!] \\
[\![\varphi_1 \vee \varphi_2]\!] & := & [\![\varphi_1]\!] + [\![\varphi_2]\!] \\
[\![\varphi_1 \Rightarrow \varphi_2]\!] & := & [\![\varphi_1]\!] \to [\![\varphi_2]\!] \\
[\![\varphi_1 \Leftrightarrow \varphi_2]\!] & := & [\![\varphi_1]\!] \leftrightarrow [\![\varphi_2]\!] \\
[\![\forall x.\ \varphi]\!] & := & \Pi(I, x.\ [\![\varphi]\!]) \\
[\![\exists x.\ \varphi]\!] & := & \Sigma(I, x.\ [\![\varphi]\!])
\end{array}
$$

A derivation of a judgment $\Gamma \vdash \varphi$ can then be translated as a typing derivation of some term t such that $[\![\Gamma]\!] \vdash t : [\![\varphi]\!]$.

2.4.3 The Calculus of Inductive Constructions

Russell invented the first type theory in 1908 [153] to solve Russell paradox. This paradox can be stated as follows in naive set theory: any set might or not belong to itself, if we denote by E the set of all sets not belonging to themselves, then $E \in E$ if and only if $E \notin E$. According to Russell, the source of the paradox is the possibility to quantify over all possible sets, including E, in the definition of E. Such a definition is called an *impredicative* definition and a way to get rid of Russell Paradox is to forbid impredicativity. To construct a predicative theory of sets, Russell invented a system of types. The quantification over all the objects of a certain type is a formula of a greater type.

MLTT is a predicative type theory but HOL (see Section 2.3.2) is impredicative. Indeed, impredicativity is used at the very basis of HOL. For example, conjunction can be defined by the impredicative definition $A \wedge B := \forall C.\ (A \Rightarrow B \Rightarrow C) \Rightarrow C$.

The Calculus of Inductive Constructions (CIC) [141] is an extension of Martin-Löf Type Theory with an impredicative universe **Prop**. The universe hierarchy is maintained as in MLTT: there is a universe $\mathbf{Type}_i$ for each natural number i. In CIC, types are more distinguished from propositions than in MLTT: alongside this hierarchy, the universe **Prop** of type $\mathbf{Type}_1$ is used to represent propositions. **Prop** beeing impredicative, it is possible to

quantify (using the dependent product) over all the propositions and the resulting formula is still a proposition. Thanks to impredicativity, we can define the impredicative encodings of logical connectives in the same way than in HOL; for example, the conjunction of two propositions A and B can be defined as the proposition $A \wedge B := \Pi C : \mathbf{Prop}.\ (A \to B \to C) \to C$. However, the inductive definitions from MLTT are also available and they are usually preferred over the impredicative encodings.

CIC is implemented in two systems, Coq [63] and Matita [10] with some slight differences in the available features that we are not going to detail.

2.5 Logical Frameworks

Implementation of logical systems such as MLTT is both error-prone and critical from the point of view of trust. In order to ease the implementation of proof checkers for new logics, frameworks for expressing logics have been proposed; they are called *Logical Frameworks*. Type theory is a branch of computer science which has been a great source of inspiration for logical frameworks because some of the simplest type theories can be used as efficient logical frameworks; this fact was already observed by de Bruijn in the 60s during the development of Automath [132], the first logical framework.

Dedukti is a logical framework based on type theory, its two main sources of inspiration are the Edinburgh Logical Framework [90] (ELF for short) and Martin-Löf's Logical Framework [133] (MLLF for short). We start with a short discussion on the representation of binding in logical frameworks in Section 2.5.1. We then present ELF and MLLF in Sections 2.5.2 and 2.5.3. In Sections 2.5.4 and 2.5.5 we detail the two main differences between ELF and MLLF and explain the choices that have been made in Dedukti. We postpone the presentation of Dedukti itself to Chapter 3.

2.5.1 Representing Binding

Almost all logical systems require a notion of variable binding. For example, quantifiers are binders in first-order logic (see Section 1.1), λ-abstraction is a binder in HOL (see Section 2.3.2), and a lot of binding constructs have been introduced in our presentation

of MLTT (see Section 2.4.1). While the human reader usually understands easily what binding means, mechanizing binding is a notoriously hard task. Binding is hard to get efficient, readable, and easy to reason about at the same time so compromises are made depending on the use-case [169].

Two main trends of representations of variable binding influence the development of logical frameworks[2]: higher-order abstract syntax [146, 128, 91, 113] and nominal logic [149, 166, 19, 46].

Nominal logic axiomatizes binding from the more primitive notion of name swapping and name freshness from which α-equivalence and capture-avoiding substitution are reasonably easy to define.

The idea behind HOAS is to represent the binding operations of the represented logic by the binding operation of the λ-calculus: λ-abstraction. The capture-avoiding substitution $b\{x\backslash a\}$ can then simply be represented by the β-redex $(\lambda x.\ b)\ a$.

HOAS can be used in any logical framework built on top of λ-calculus, not necessarily a logical framework featuring dependent types. For example, λ-prolog [128] and Isabelle are logical frameworks based on polymorphic λ-calculi and HOL, we can use HOAS to define universal quantification from equality as $(\forall x.\ p) := (p = \lambda x.\ \top)$.

If the logic features quantification, then the propositions need to be represented as λ-terms in the logical framework: the formula $\forall x.\ P(x)$ would be written $\forall\ (\lambda x.\ P\ x)$.

In logical frameworks featuring dependent types, we can build the type of all the proofs of a formula φ as a type depending on the term representing φ. The motivation for using dependent types in logical frameworks is precisely the ability to represent proofs as objects of the framework. Following the Curry-Howard correspondence, we can then use proofs as programs, that is we can compute with proofs.

2.5.2 Edinburgh Logical Framework

The expression *Logical Framework* was first coined in [90] to refer to the $\lambda\Pi$-calculus, the type system underlying the Edinburgh Logical Framework.

[2]Other representations of binding such as explicit names and De Bruijn indices are possible but we do not present them because logical frameworks do not offer more support for them than first-order logic.

The $\lambda\Pi$-calculus is a system of Barendregt's λ-cube [22] and can hence be presented as a Pure Type System [23], this is the presentation that we use because it is very succinct.

The syntax and the typing rules for the $\lambda\Pi$-calculus are presented in Figure 2.4. The $\lambda\Pi$-calculus extends the simply-typed λ-calculus by dependent typing and nothing else. Types are not syntactically distinguished from terms but two particular terms **Type** and **Kind** are distinguished and called sorts. **Type** is the sort of all the types so the typing judgment $\Gamma \vdash t : \mathbf{Type}$ is used to represent the fact that the term t is a type. **Type** itself is not a type to avoid Girard's Paradox [82]; it is a kind, that is a term of type **Kind**. Type families can be introduced as terms of type $\tau_1 \to \ldots \to \tau_n \to \mathbf{Type}$ for some types $\tau_1, \ldots, \tau_n$. All the terms of the form $\tau_1 \to \ldots \to \tau_n \to \mathbf{Type}$ are also kinds but **Kind** itself is not a well-typed term. As in MLTT (see Section 2.4.1), the arrow type $\tau_1 \to \tau_2$ is seen as a particular case of the dependent product: $\tau_1 \to \tau_2 := \Pi x : \tau_1.\ \tau_2$ where the variable x does not occur free in τ_2.

We let the letter s range over sorts, the letter x range over variables, and other letters range over terms. The two forms of judgment are

- $(\Gamma \vdash)$ meaning that the context Γ is well-formed,
- $(\Gamma \vdash t_1 : t_2)$ meaning that the term t_1 has type t_2 in context Γ.

The most interesting typing rule of the $\lambda\Pi$-calculus is the conversion rule which can be used to transparently replace a type A by any type B which is β-convertible to A. The relation $\equiv_\beta$, is defined on untyped terms as in Section 2.1.

2.5.3 Martin-Löf's Logical Framework

Martin-Löf's Logical Framework has been informally introduced by Martin-Löf to define Martin-Löf's Type Theory [122]. This logical framework has then been formalized in [133]. We give here a presentation of this framework where variables are introduced together with their types, this presentation has been proposed by Luo in [118] and is convenient to compare MLLF with other logical frameworks, especially Pure Type Systems such as the $\lambda\Pi$-calculus.

Syntax	s	$:=$	$\mathbf{Type} \mid \mathbf{Kind}$	*sorts*
	t	$:=$	$s \mid x \mid t_1\ t_2 \mid \lambda x : t_1.\ t_2 \mid \Pi x : t_1.\ t_2$	*terms*
	Γ	$:=$	$\emptyset \mid \Gamma, x : t$	*contexts*

Typing

$$\frac{}{\emptyset \vdash}\ (\text{Empty}) \qquad \frac{\Gamma \vdash A : s \qquad (\text{when } x \notin \Gamma)}{\Gamma, x : A \vdash}\ (\text{Decl}) \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{Type} : \mathbf{Kind}}\ (\text{Type})$$

$$\frac{\Gamma \vdash \qquad (\text{when } (x : A) \in \Gamma)}{\Gamma \vdash x : A}\ (\text{Var}) \qquad \frac{\Gamma \vdash f : \Pi x : A.\ B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B\{x \backslash a\}}\ (\text{App})$$

$$\frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.\ b : \Pi x : A.\ B}\ (\text{Abs})$$

$$\frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.\ B : s}\ (\text{Prod})$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad (\text{when } A \equiv_\beta B)}{\Gamma \vdash t : B}\ (\text{Conv})$$

Figure 2.4: The $\lambda\Pi$-calculus

Similarly to the $\lambda\Pi$-calculus, MLLF is a dependent type system. In order to avoid confusion between the types of MLLF and the types of type systems that are to be embedded in MLLF, the former are called *kinds* and the later are the inhabitants of a built-in kind called **Type**. Contrary to the $\lambda\Pi$-calculus, kinds and terms are syntactically distinguished, we shall use uppercase letters to denote kinds and lowercase letters to denote terms. To each inhabitant a of **Type** is associated a kind $\mathsf{El}(a)$, moreover kinds can be built using the dependent product. The terms of MLLF are the usual terms of the λ-calculus where λ-abstraction is annotated by the kind over which the introduced variable ranges.

The syntax of MLLF is given in Figure 2.5 and Figure 2.6 together with the rules defining MLLF judgments:

- $(\Gamma \vdash)$ meaning that the context Γ is well-formed,
- $(\Gamma \vdash K \ \mathbf{kind})$ meaning that K is a kind in context Γ,
- $(\Gamma \vdash t : K)$ meaning that the term t has kind K in context Γ,
- $(\Gamma \vdash A \equiv B)$ meaning that the kinds A and B are convertible in context Γ,
- $(\Gamma \vdash t \equiv u : K)$ meaning that the terms t and u of kind K are convertible in context Γ.

2.5.4 Internal vs. External Conversion

In dependently typed systems, types are usually quotiented by evaluation of the terms occurring inside; the congruence $\equiv$ used for quotienting types is called conversion and it is dealt in two slightly different ways in logical frameworks:

- For some frameworks, such as MLLF [133] and Luo's PAL$^+$ [119], conversion is an **internal** judgment $\Gamma \vdash A \equiv B : T$; this judgment is a congruence by virtue of its derivation rules.
- For other frameworks, such as ELF [90] and Pure Type Systems [23], conversion is an **external** relation on untyped terms, typically β-conversion or $\beta\eta$-conversion; conversion appears as a side condition for the conversion rule

Syntax

t	$:=$	$x \mid t\ t \mid \lambda x : K.\ t$	*terms*
K	$:=$	$\mathbf{Type} \mid \mathsf{El}(t) \mid \Pi x : K.\ K$	*kinds*
Γ	$:=$	$\emptyset \mid \Gamma, x : K$	*contexts*

Context Formation

$$\frac{}{\emptyset \vdash}\ (\text{Empty}) \qquad \frac{\Gamma \vdash A\ \mathbf{kind} \qquad (\text{when } x \notin \Gamma)}{\Gamma, x : A \vdash}\ (\text{Decl})$$

Kinds

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbf{Type}\ \mathbf{kind}}\ (\text{Type}) \qquad \frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash \mathsf{El}(A)\ \mathbf{kind}}\ (\text{El})$$

$$\frac{\Gamma \vdash A\ \mathbf{kind} \qquad \Gamma, x : A \vdash B\ \mathbf{kind}}{\Gamma \vdash \Pi x : A.\ B\ \mathbf{kind}}\ (\text{Prod})$$

Typing

$$\frac{\Gamma \vdash \qquad (\text{when } (x : A) \in \Gamma)}{\Gamma \vdash x : A}\ (\text{Var}) \qquad \frac{\Gamma \vdash f : \Pi x : A.\ B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B\{x \backslash a\}}\ (\text{App})$$

$$\frac{\Gamma \vdash A\ \mathbf{kind} \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.\ b : \Pi x : A.\ B}\ (\text{Abs})$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B\ \mathbf{kind} \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash t : B}\ (\text{Conv})$$

Figure 2.5: Martin-Löf's Logical Framework (part 1/2)

Kind Conversion

$$\frac{\Gamma \vdash A \ \mathbf{kind}}{\Gamma \vdash A \equiv A}\ (\text{Refl}) \qquad \frac{\Gamma \vdash A \equiv B}{\Gamma \vdash B \equiv A}\ (\text{Sym}) \qquad \frac{\Gamma \vdash A \equiv B \qquad \Gamma \vdash B \equiv C}{\Gamma \vdash A \equiv C}\ (\text{Trans})$$

$$\frac{\Gamma \vdash A \equiv A' \qquad \Gamma, x : A \vdash B \equiv B'}{\Gamma \vdash \Pi x : A.\ B \equiv \Pi x : A'.\ B'}\ (\text{Prod}) \qquad \frac{\Gamma \vdash a \equiv a' : \mathbf{Type}}{\Gamma \vdash \mathsf{El}(a) \equiv \mathsf{El}(a')}\ (\text{El})$$

Term Conversion

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t \equiv t : A}\ (\text{Refl}) \qquad \frac{\Gamma \vdash t \equiv u : A}{\Gamma \vdash u \equiv t : A}\ (\text{Sym})$$

$$\frac{\Gamma \vdash t \equiv u : A \qquad \Gamma \vdash u \equiv v : A}{\Gamma \vdash t \equiv v : A}\ (\text{Trans})$$

$$\frac{\Gamma \vdash f \equiv f' : \Pi x : A.\ B \qquad \Gamma \vdash a \equiv a' : A}{\Gamma \vdash f\ a \equiv f'\ a' : B\{x \backslash a\}}\ (\text{App})$$

$$\frac{\Gamma \vdash A \equiv A' \qquad \Gamma, x : A \vdash b \equiv b' : B}{\Gamma \vdash \lambda x : A.\ b \equiv \lambda x : A'.\ b' : \Pi x : A.\ B}\ (\text{Abs})$$

$$\frac{\Gamma \vdash t \equiv u : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash t \equiv u : B}\ (\text{Conv})$$

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x : A.\ b)\ a \equiv b\{x \backslash a\} : B\{x \backslash a\}}\ (\text{Beta})$$

$$\frac{\Gamma \vdash f : \Pi x : A.\ B \qquad (\text{when } x \notin \mathsf{FV}(f))}{\Gamma \vdash \lambda x : A.\ f\ x \equiv f : \Pi x : A.\ B}\ (\text{Eta})$$

Substitution

$$\frac{\Gamma, x : A, \Gamma' \vdash \qquad \Gamma \vdash a : A}{\Gamma, \Gamma'\{x \backslash a\} \vdash}\ (\text{Decl}) \qquad \frac{\Gamma, x : A, \Gamma' \vdash B \ \mathbf{kind} \qquad \Gamma \vdash a : A}{\Gamma, \Gamma'\{x \backslash a\} \vdash B\{x \backslash a\} \ \mathbf{kind}}\ (\text{Kind})$$

$$\frac{\Gamma, x : A, \Gamma' \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma, \Gamma'\{x \backslash a\} \vdash b\{x \backslash a\} : B\{x \backslash a\}}\ (\text{Typing})$$

$$\frac{\Gamma, x : A, \Gamma' \vdash B \ \mathbf{kind} \qquad \Gamma \vdash a = a' : A}{\Gamma, \Gamma'\{x \backslash a\} \vdash B\{x \backslash a\} \equiv B\{x \backslash a'\}}\ (\text{KindConv1})$$

$$\frac{\Gamma, x : A, \Gamma' \vdash B \equiv B' \qquad \Gamma \vdash a : A}{\Gamma, \Gamma'\{x \backslash a\} \vdash B\{x \backslash a\} \equiv B'\{x \backslash a\}}\ (\text{KindConv2})$$

$$\frac{\Gamma, x : A, \Gamma' \vdash b : B \qquad \Gamma \vdash a \equiv a' : A}{\Gamma, \Gamma'\{x \backslash a\} \vdash b\{x \backslash a\} \equiv b\{x \backslash a'\}}\ (\text{Conv1})$$

$$\frac{\Gamma, x : A, \Gamma' \vdash b \equiv b' : B \qquad \Gamma \vdash a : A}{\Gamma, \Gamma'\{x \backslash a\} \vdash b\{x \backslash a\} \equiv b'\{x \backslash a\}}\ (\text{Conv2})$$

Figure 2.6: Martin-Löf's Logical Framework (part 2/2)

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad (\text{when } A \equiv B)}{\Gamma \vdash t : B}\ (\text{Conv})$$

The internal version is more powerful as it allows conversion to depend on the ambient type of terms A and B and on the context Γ, this extra power is useful for the study of undecidable conversion relations such as the conversion of the extensional version of Martin-Löf's Type Theory (in which two terms are convertible if and only if they are provably equal) as it allows to keep decidability of derivation checking. It does however lead to a huge increase in the size of derivations so it is rarely implemented.

As we will see in Section 3.2, the approach taken in Dedukti is intermediate between these two alternatives. In Dedukti, conversion is essentially an external relation which shall be decidable and it does not appear in derivations. It does however depend on the rewrite rules declared in the context Γ.

2.5.5 Proposition-as-Type vs. Judgment-as-Type

Following the Curry-Howard correspondence, logical frameworks types can be used to represent the logical propositions of the embedded logic and logical connectives (such as conjunction) are then interpreted as type-level operations (such as the Cartesian product of two types). This approach leads to a rich conversion that has to be extended when new type constructs are introduced, this is the way MLLF is usually used.

In order to keep the conversion relation simple, an alternative approach has been proposed; it consists in representing the judgments of the logic as types in the logical framework and every other construct, including propositions, only as terms in the logical framework. This is the way ELF is used to represent natural deduction for example.

In Dedukti, both methodologies are commonly used. We will see both a judgment-as-type and a proposition-as-type encoding of Natural Deduction in Section 3.5.1.

Chapter 3

Dedukti: a Universal Proof Checker

Mixing the $\lambda\Pi$-calculus with Deduction modulo leads to the $\lambda\Pi$-calculus modulo, a logical framework able to express proofs modulo rewriting. The combination of dependent types and rewriting is surprisingly powerful and a wide variety of logical systems have been encoded in Dedukti, an implementation of the $\lambda\Pi$-calculus modulo [155].

Since the $\lambda\Pi$-calculus modulo is built on λ-calculus, first-order rewriting as we presented in Section 1.2 has to be generalized to higher-order in order to be included in the $\lambda\Pi$-calculus modulo. This generalization is the topic of Section 3.1. The $\lambda\Pi$-calculus modulo itself is then briefly presented in Section 3.2. Dedukti syntax is used throughout this thesis, it is summarized in Section 3.3.

In order to get a better feeling of what Dedukti can be used for, most of this chapter is devoted to Dedukti from a user point of view. Section 3.4 is devoted to concrete examples of use of programming and logical paradigms in Dedukti and Section 3.5 demonstrates the encoding of logical systems in Dedukti.

3.1 Higher-Order Rewriting

Similarly to the way we have defined first-order rewriting in Section 1.2, we now define rewriting in the case where the terms under consideration are not first-order terms but (simply-typed) λ-terms.

As in Section 1.2, a higher-order rewrite rule [20] $l \longrightarrow r$ is defined by giving two terms, a left-hand side l and a right-hand side r. We continue to require that all the free variables

of r are among those of l and that l is not itself a variable. The intended meaning of the rule $l \longrightarrow r$ is that any term $\beta\eta$-equivalent to an instance $l\rho$ of l evolves to the corresponding instance $r\rho$ of the term r.

Contrary to the first-order case, unification of λ-terms modulo $\beta\eta$ is not decidable so we need to restrict the shape of the left-hand sides if we want the reduction relation to be decidable. Such a constraint is provided by Miller patterns.

A Miller pattern [125] is a λ-term p in η-long β-normal form such that every free occurrence of a variable x is in a subterm of p of the form $x\ y_1\ \ldots\ y_n$ such that the y_i are η-equivalent to distinct bound variables.

In this thesis, we are only interested in higher-order rewrite systems corresponding to rewrite systems of the $\lambda\Pi$-calculus modulo. Moreover, rewriting is used in the $\lambda\Pi$-calculus modulo to define an external convertibility relation which is defined before the typing relation so it is defined on all terms, not only well-typed terms. In particular, the rewrite systems that we consider do not terminate because the β-reduction relation on untyped terms does not terminate.

Confluence however holds for some but not all rewrite systems. As in the first-order case, confluence is not a decidable property of higher-order rewrite systems but useful criteria have been found and implemented to the point that higher-order term rewrite systems became in 2015 a new category in the international confluence competition.

The competitors for this new category were:

- ACPH [137], implementing two criteria:
  - Weakly orthogonal rewrite systems are confluent [168].

    A weakly orthogonal rewrite system is a left-linear rewrite system such that all critical pairs are trivial.
  - Knuth-Bendix theorem: for terminating rewrite systems, confluence is equivalent to local confluence [124].

    Sadly, this criterion is useless because we are interested in non-terminating rewrite systems only.

- CSIˆHO [130]: implementing the same criteria plus:
  - An approximation of Van Oostrom criteria for left-linear rewrite systems [167], which we are not going to explain.

    This criterion can be used to prove confluence for complex examples.

3.2 The λΠ-Calculus Modulo

The λΠ-calculus modulo has been introduced by Cousineau and Dowek in [55] and then improved by Saillard in [154, 155] to integrate several features needed for the translation of complex systems such as CIC. We give here a simplified presentation of this improved λΠ-calculus modulo.

The syntax and the typing rules for the λΠ-calculus modulo are given in Figure 3.2. The λΠ-calculus modulo is obtained from the λΠ-calculus (see Figure 2.4) by the following steps:

- Extending contexts so that they can contain rewrite rules:

$$\Gamma := \ldots \mid \Gamma, t \longrightarrow t$$

- Adding a derivation rule for checking well-formedness of contexts containing rewrite rules:

$$\frac{\Gamma \vdash \qquad \Gamma \vdash l \longrightarrow r}{\Gamma, l \longrightarrow r \vdash} \text{(Rule)}$$

  Defining precisely typing of rewrite rules is a very subtle affair if we want the criterion to be both decidable and powerful enough to be used in practice. The interested reader is refered to [155]. In this thesis, we will not give the definition of the judgment $\Gamma \vdash l \longrightarrow r$. All well-typed rules $l \longrightarrow r$ are higher-order rewrite rules in the sense of Section 3.1: l is not a variable, $\mathsf{FV}(r) \subseteq \mathsf{FV}(l)$, and l is a Miller pattern.

- Changing the conversion relation used in rule (Conv): instead of mere β-conversion, we use the congruence induced by the relation $\beta\Gamma$ consisting of β-reduction together with all the rewrite rules present in Γ.

Syntax

	s	$:=$	$\mathbf{Type} \mid \mathbf{Kind}$	*sorts*
	t	$:=$	$s \mid x \mid t\ t \mid \lambda x : t.\ t \mid \Pi x : t.\ t$	*terms*
	Γ	$:=$	$\emptyset \mid \Gamma, x : t \mid \Gamma, t \longrightarrow t$	*contexts*

Typing

$$\frac{}{\emptyset \vdash}\ (\text{Empty}) \qquad \frac{\Gamma \vdash A : s \qquad (\text{when } x \notin \Gamma)}{\Gamma, x : A \vdash}\ (\text{Decl}) \qquad \frac{\Gamma \vdash \qquad \Gamma \vdash l \longrightarrow r}{\Gamma, l \longrightarrow r \vdash}\ (\text{Rule})$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbf{Type} : \mathbf{Kind}}\ (\text{Type}) \qquad \frac{\Gamma \vdash \qquad (\text{when } (x : A) \in \Gamma)}{\Gamma \vdash x : A}\ (\text{Var})$$

$$\frac{\Gamma \vdash f : \Pi x : A.\ B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B\{x \backslash a\}}\ (\text{App})$$

$$\frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.\ b : \Pi x : A.\ B}\ (\text{Abs})$$

$$\frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.\ B : s}\ (\text{Prod})$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad (\text{when } A \equiv_{\beta\Gamma} B)}{\Gamma \vdash t : B}\ (\text{Conv})$$

Figure 3.1: The λΠ-calculus modulo

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash t : B}\ (\text{Conv})$$

This extension to the λΠ-calculus enhances greatly its power as a logical framework: it is easy to encode any functional Pure Type System in the λΠ-calculus modulo [55]. More complicated logics such as Martin-Löf Type Theory, the Calculus of Inductive Constructions, and PVS are also faithfully encoded in the λΠ-calculus modulo as they would be encoded in MLLF following the proposition-as-type principle.

In [155], Saillard showed that assuming confluence of the relation $\beta\Gamma$ (on untyped terms), the λΠ-calculus modulo enjoys the subject reduction property. If the relation $\beta\Gamma$ is also terminating (on well-typed terms), then $\equiv_{\beta\Gamma}$ and typing are decidable so under these assumptions, the λΠ-calculus modulo can be implemented.

3.3 Dedukti

Dedukti [156] is an implementation of a proof-checking algorithm for the λΠ-calculus modulo. Its input is a context Γ of the λΠ-calculus modulo, its output is either `SUCCESS` if it succeeded in checking $\Gamma \vdash$ or `ERROR` and an error message otherwise.

Dedukti is a free software available at `http://dedukti.gforge.inria.fr`. In this thesis, we use the version v2.5 of Dedukti.

3.3.1 Syntax

A Dedukti file represents a λΠ-modulo context, it is composed of symbol declarations, rewrite rules, and commands.

Dedukti uses ASCII notations, identifiers are composed of letters, digits, and the underscore `_`. The mapping between Dedukti ASCII syntax and the constructs of the λΠ-calculus modulo is given in Figure 3.2; abstractions are written using a double `=>` arrow and products using a simple arrow `->`. The name of the variable in the product can be omitted for the non-dependent arrow.

Dedukti features a very basic module system. Each Dedukti file corresponds to a module, the name of the module is given by the first line. The mandatory line `#NAME module_name.` at the beginning of a Dedukti file defines a module named `module_name`. It is possible to refer to symbols declared in another Dedukti module using the dotted notation: `a.b` refers to the symbol `b` in module `a`. Modules cannot be nested nor opened.

Dedukti distinguishes two kinds of declarations:

- declaration of a *static* symbol `f` of type `A` is written `f : A`,
- declaration of a *definable* symbol `f` of type `A` is written `def f : A`.

The difference between static and definable symbols is that the head symbol of a rewrite rule must always be definable. Because static symbols cannot appear at head of rewrite rules, they are injective with respect to conversion and this information can be exploited by Dedukti when it needs to solve unification problems. Static symbol are so called because

Dedukti	$\lambda\Pi$-calculus modulo	Syntactic Construct
`x : A => b`	$\lambda x : A.\ b$	Abstraction
`x : A -> b`	$\Pi x : A.\ b$	Product
`A -> B`	$A \to B$	Arrow type
`def x : A.`	$x : A$	Definable symbol declaration
`x : A.`	$x : A$	Static symbol declaration
`[`x_1`, ..., `x_n`] l --> r.`	$l \longrightarrow r$	Rewrite rule

Figure 3.2: Correspondance between Dedukti syntax and the $\lambda\Pi$-calculus modulo

Sugar	Meaning
`def f (`x_1 `:` A_1`) ... (`x_n `:` A_n`) : A := a.`	`def f : (`x_1 `:` A_1 `-> ... -> `x_n `:` A_n `-> A).` `[] f -->` x_1 `:` A_1 `=> ... =>` x_n `:` A_n `-> a.`
`thm f (`x_1 `:` A_1`) ... (`x_n `:` A_n`) : A := a.`	`f : (`x_1 `:` A_1 `-> ... -> `x_n `:` A_n `-> A).` (when `a : A`)

Figure 3.3: Dedukti definitions as syntactic sugar

they cannot change during evaluation whereas definable symbols might receive (maybe partial) definitions.

Rewrite rules have the following syntax: `[`x_1`, ..., `x_n`] l --> r.` where the x_i are the matching variables, `l` is a pattern and `r` is a term.

Definitions are a special case of rewrite rules for which a specific syntax is available as syntactic sugar defined in Figure 3.3: `def f (`x_1 `:` A_1`) ... (`x_n `:` A_n`) : A := a.`, where `f` is a symbol, the x_i are variables and `A`, the A_i and `a` are terms, declares the definable symbol `f` and defines it as `a` in which the x_i have been λ-abstracted. The type `A` can be omitted since it can always be inferred from `a`.

Finally, Dedukti features opaque definitions with the `thm` keyword: the opaque definition `thm f (`x_1 `:` A_1`) ... (`x_n `:` A_n`) : A := a.` is equivalent to the static declaration `f : (`x_1 `:` A_1 `-> ... -> `x_n `:` A_n `-> A).` but also checks that the term x_1 `:` A_1 `=> ... =>` x_n `:` A_n `=> a.` has the same type than `f`.

3.3.2 Commands

The primary role of Dedukti is to check that a given $\lambda\Pi$-modulo context is well-formed but Dedukti also gives access to toplevel commands which can be used for requesting normal forms and testing whether or not two terms are convertible. Theses commands are:

- `#STEP` t: print the term obtained by applying one $\beta\Gamma$-reduction step to t.
- `#SNF` (resp. `#HNF`, `#WHNF`) t: print the $\beta\Gamma$ strong- (resp. head-, weak-head-) normal form of term t.
- `#CONV` t_1, t_2: print `YES` if t_1 is $\beta\Gamma$-convertible to t_2 and `NO` otherwise.
- `#CHECK` t_1, t_2: print `YES` is t_1 has type t_2 and `NO` otherwise.
- `#INFER` t: print a type for t.
- `#PRINT "`s`"`: print the string s.

3.3.3 Confluence Checking

For efficiency, Dedukti does not check that typing is preserved each time it reduces a term but, similarly to evaluators for typed functional languages, it relies on the subject-reduction property. As we already mentioned, this property cannot be proved in general but it holds when the (untyped) $\beta\Gamma$-reduction is confluent.

The $\beta\Gamma$-reduction fits the definition of a Higher-Order Term Rewrite System (see Section 3.1) for which confluence checkers such as CSI^HO are available.

These tools implement no criterion for non left-linear rewrite systems so we have no way of automatically ensuring subject reduction for non left-linear systems. Actually, when Γ contains non left-linear rules, the relation $\beta\Gamma$ is almost never confluent on untyped terms. Since confluence is our main tool for proving subject reduction, Dedukti discourages the use of non left-linear rewrite systems; they are only allowed when the option "-nl" (for non-linear) is passed to Dedukti.

Unless otherwise specified, all the examples in this thesis have been type checked by Dedukti and proved confluent by CSI^HO.

Termination is however not checked because the untyped $\beta\Gamma$-reduction is never terminating.

3.4 Proving and Programming in Dedukti

Dedukti can be seen both as a logical framework in which logics and proofs can be developed and as a dependently typed programming language based on rewriting. When combining these two views, we can take benefit of programming techniques to develop logical embeddings, proofs and proof transformations in Dedukti. In this chapter we describe a few such techniques which make Dedukti more powerful than usual implementations of type theory.

3.4.1 Smart Constructors

In ML, data types are useful for representing free structures generated by a set of constructors but some types are better represented by quotienting data types by an equivalence relation. Often enough, this equivalence relation can be defined as the congruence generated by a rewrite system.

For example, the type of Peano natural numbers is generated by zero and successor but integers are harder to represent as a free structure. They are however easy to define as a quotient of pairs of natural numbers. Here is for example a definition of integers in the OCaml programming language.

```
type nat = O | S of nat;;
type int = Diff of nat * nat;;
let rec diff (m, n) =
  match (m, n) with
  | (S m', S n') -> diff (m', n')
  | _ -> Diff (m, n);;
```

The function `diff` is called a smart constructor [5] for the type `int`; if we restrict ourselves to use the function `diff` instead of the constructor `Diff`, an extra invariant holds for the values of type `int`: there are of the form `Diff(m, n)` where at least one from `m` and `n` is zero. This restriction can be automatically enforced by hiding the implementation of `int` in the interface of the file.

In Dedukti, there is no fixed notion of constructor so we do not need to distinguish `diff` from `Diff`. The invariant automatically holds for close normal terms:

```
nat : Type.
0 : nat.
S : nat -> nat.

int : Type.
def Diff : nat -> nat -> int.
[m,n] Diff (S m) (S n) --> Diff m n.
```

3.4.2 Partial Functions

Usually in type theory (for example in the Calculus of Inductive Constructions, in Martin-Löf Type Theory or in NuPRL), symbols can be classified into: constructors, type constructors, functions, and axioms.

Constructors are used to build values, type constructors are used to build types, functions are abbreviations for their definitions, and axioms are symbols assumed to inhabit their types without justification and should be avoided if possible. Constructors, type constructors and axioms have no associated reduction behaviour. Functions however can always be unfolded. In type theory, functions are total.

In the $\lambda\Pi$-calculus modulo however, only one kind of symbol is considered. Dedukti distinguishes static and definable symbols but this is a different scenario since if f is a definable symbol, Dedukti does not enforce that f actually appears as head symbol of some rewrite rules and as we have seen, some definable symbols play the role of smart constructors.

Dedukti does not enforce that definitions are total because this has no meaning in the $\lambda\Pi$-calculus modulo: types can always be extended by declaring new symbols:

Example 6. *Consider again the usual signature defining Peano natural numbers:*

```
nat : Type.
0 : nat.
S : nat -> nat.
```

In this signature, we can define Peano addition as usual:

```
def plus : nat -> nat -> nat.
[n] plus 0 n --> n
```

```
[m,n] plus (S m) n --> S (plus m n).
```

This definition is total in the sense that any normal close term of type `nat` *is either* `0` *or starts with* `S`*.*

However, we can extend the type `nat` *by declaring a new constructor* `infty` *representing infinity:*

```
infty : nat.
```

and now the `plus` *function is partial and the term* `plus infty 0` *is normal.*

In Dedukti, the distinction between constructors, axioms, total functions and partial functions is only in the eye of the user. It is not always possible to split the set of symbols between constructors and total functions.

Example 7. *The following signature defines lists of natural numbers:*

```
list : Type.
Nil : list.
Cons : nat -> list -> list.
```

The functions returning the head and the tail of a constructed list can be partially defined:

```
def head : list -> nat.
def tail : list -> list.
[a] head (Cons a _) --> a.
[l] tail (Cons _ l) --> l.
```

To summarize, there is almost only one kind of symbol in Dedukti; constructors, axioms, and functions are not distinguished; some symbols never reduce, some other reduce on any closed normal terms, but some other sometimes reduce and sometimes do not; smart constructors and partial functions belong to this category.

3.4.3 Encoding Polymorphism

Polymorphism is the ability to define functions acting on several types; two kinds of polymorphism can be distinguished, parametric polymorphism and ad-hoc polymorphism.

Parametric polymorphic functions are functions whose definitions are generic in one or more types; for example, the identity function can be defined using parametric polymorphism as $\lambda A : Type.\ \lambda x : A.\ x$. The $\lambda\Pi$-calculus and the $\lambda\Pi$-calculus modulo do not feature polymorphism[1] but:

- Dedukti has an option, "-coc" which turns Dedukti into a type-checker for the Calculus of Constructions modulo, a very small adaptation of the $\lambda\Pi$-calculus modulo featuring parametric polymorphism.

- Without the "-coc" option, polymorphism can easily be encoded. In order to pass types as arguments, we need to reify types as terms of a fixed type `type` and interpret them as types by an injection `term`. We need to construct products in `type` so we introduce the constant `pi` for this purpose and we add a rewrite rule identifying interpretations of products with products of interpretations:

  ```
  type : Type.
  def term : type -> Type.
  pi : A : type -> (term A -> type) -> type.
  [A,B] term (pi A B) --> x : term A -> term (B x).
  ```

  The identity function can then be defined by:

  ```
  def id (A : type) (x : term A) := x.
  ```

Ad-hoc polymorphism is the ability for a function to act differently depending on the type of its argument. Ad-hoc polymorphism is usually defined by overloading a function symbol with several types and definitions. For example, we might want to dispose of a polymorphic equality `eq : A : type -> term A -> term A -> term bool` whose definition depends on its type argument:

```
bool : type.
True : term bool.
False : term bool.

nat : type.
0 : term nat.
S : term nat -> term nat.
```

[1] This comes from the condition $\Gamma \vdash A$: **Type** in rule (Abs) of Section 3.2. In particular, the term $\lambda A : Type.\ \lambda x : A.\ x$ is not well-typed.

```
def plus : term nat -> term nat -> term nat.
[n] plus 0 n --> n
[m, n] plus (S m) n --> S (plus m n).

int : type.
def Diff : term nat -> term nat -> term int.
[m, n] Diff (S m) (S n) --> Diff m n.

def eq : A : type -> term A -> term A -> term bool.

[] eq nat 0 0 --> True
[] eq nat 0 (S _) --> False
[] eq nat (S _) 0 --> False
[m, n] eq nat (S m) (S n) --> eq nat m n

[m1,m2,n1,n2]
   eq int (Diff m1 m2) (Diff n1 n2)
     -->
   eq nat (plus m1 n2) (plus n1 m2)

[] eq bool True True --> True
[] eq bool True False --> False
[] eq bool False True --> False
[] eq bool False False --> True.
```

Given our encoding of parametric polymorphism, ad-hoc polymorphism is the same as partial definitions for functions on type `type`, whereas parametric polymorphism corresponds to total definitions for functions on type `type`.

3.4.4 Overfull Definitions

The dual feature to partial definition is overfull definition, that is defining a total function with more rules than needed to make it total. This seemingly useless feature actually provides an elegant solution to a common issue in type theory: we cannot state that the empty vector is a neutral element for vector concatenation.

Example 8. *We can define vectors of natural numbers (lists of numbers depending on their length) and concatenation:*

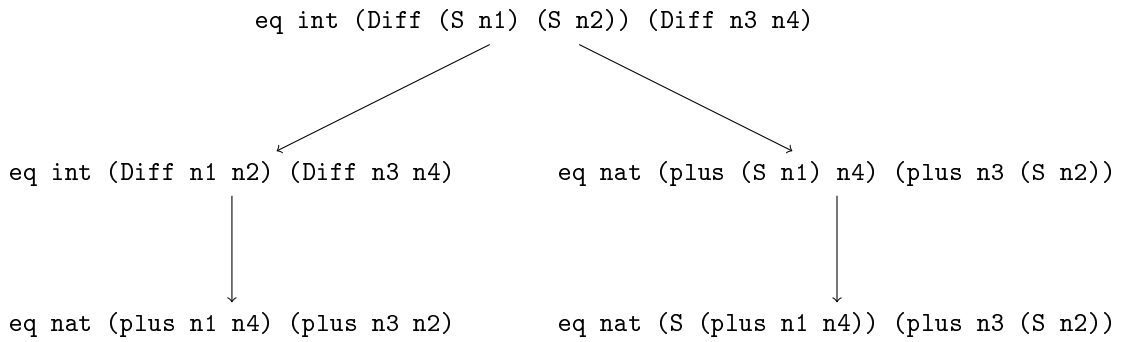
```
vector : nat -> Type.
Nilv : vector 0.
Consv : n : nat -> nat -> vector n -> vector (S n).

def append : m : nat -> n : nat -> vector m -> vector n -> vector (plus m n).
[v] append _ _ Nilv v --> v
[m,n,a,v,w] append _ n (Consv m a v) w -->
```

Consv (plus m n) a (append m n v w).

but for `m : nat` *and* `v : vector m`*, the term {append m 0 v Nilv} has type* `vector (plus m 0)` *which is not convertible to* `vector m` *so we cannot state that* `append m 0 v Nilv` *is equal to* `v`*.*

This can be fixed in Dedukti by adding the rewrite rule `[m] plus m 0 --> m`.

In fact, the definition of equality of integers that we gave in Section 3.4.3 is not confluent: the following counterexample is given by CSI^HO:



This can be fixed by adding the rewrite rule `[m,n] plus m (S n) --> S (plus m n)` which leads to the fully symmetric definition of `plus`:

```
def plus : nat -> nat -> nat.
[n] plus 0 n --> n
[m] plus m 0 --> m
[m,n] plus (S m) n --> S (plus m n)
[m,n] plus m (S n) --> S (plus m n).
```

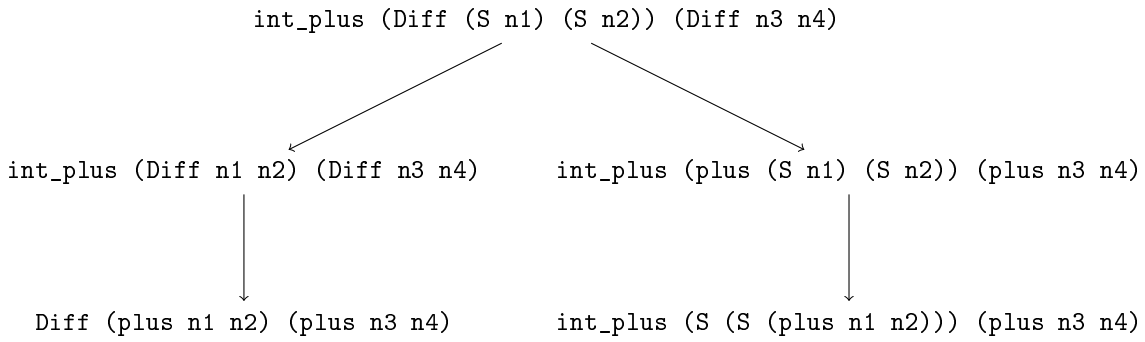
Overfull definitions also naturally appear when translating problems and proofs from Deduction modulo: the more equality axioms are turned into rewrite rules, the simpler and shorter the proof will be so there is no reason to stop when the defined function is total.

Moreover, when we add a rewrite rule such as `[m] plus m 0 --> m` for a total symbol such as our first definition of `plus`, the confluence condition which is automatically checked by CSI^HO guarantees that the reduction relation on ground terms is unchanged. Actually, checking that critical pairs generated by the new rule `plus 0 0` and `plus (S m) 0` are closed is the biggest part of a proof of $\forall$`m. plus m 0` $=$ `m`. We are actually delegating some reasoning to the confluence checker, and it might even provide counter-examples when it fails!

For example, if we mistakenly define addition of integers by

```
def int_plus : int -> int -> int.
[m1, n1, m2, n2]
  int_plus (Diff m1 n1) (Diff m2 n2)
    -->
  Diff (plus m1 n1) (plus m2 n2).
```

then CSIˆHO catches the error and provides the following explanation:



Thanks to the symmetric definition of plus, we can state that `Nilv` is a neutral element for `append` but we get almost the same level of confidence by adding the rewrite rule `[v] append _ _ v Nilv --> v` and requiring a confluence check.

3.4.5 Meta-Programming

Programmers often feel the need of defining syntactic sugar over a programming language such as defining a for loop as syntactic sugar around a while loop.

These definitions of syntactic sugar cannot be achieved by regular function definitions because they have to be performed before evaluation of arguments; they manipulate code snippets, not regular values.

The simplest solution is to add a preprocessor: a programming language designed for manipulating the programs of the initial language. The preprocessing phase happens before the program is evaluated or compiled.

Some languages such as LISP are their own preprocessors, this is known as meta-programming. In meta-programming, two or more evaluation phases can be distinguished, parts of the semantics of the language are available in certain phases only while others are available in all steps.

Rewriting is known to be a nice framework for meta-programming, the Pure programming language [88], a dynamically-typed language based on rewriting and the Maude system [52], an implementation of rewriting logics achieve meta-programming by reflection: declarations and rewrite rules are represented as first-class objects which can be manipulated in the language.

In Dedukti, rewrite rules are not first-class objects so it is not clear if we can achieve reflection simply. Meta programming in Dedukti is however easy by chaining Dedukti invocations.

If two rewrite systems R_1 and R_2 are defined on the same signature, we can ask Dedukti to normalize a term t with respect to R_1 (using the command #SNF). The output of Dedukti is a term in Dedukti syntax so we can check it with respect to R_2.

This process can be iterated, t can be normalized to t_1 with respect to R_1, then t_1 can be normalized to t_2 with respect to R_2, then t_2 can be normalized to t_3 with respect to some other rewrite system R_3 and so on. The term t_{n-1} resulting from this process can finally be checked in the final rewrite system R_n.

Interestingly, the intermediate systems $(R_1, \ldots, R_{n-1})$ need not a degree of confidence as high as the last one R_n. In particular, we will often consider non-linear and non-confluent intermediate systems, this is not a problem as long as we do not break subject reduction.

These unsafe intermediate systems are useful to model non-confluent behaviour but also for efficiency reasons. For example, it is very tempting to define polymorphic equality by the following non-linear rewrite system:

```
bool : type.
True : term bool.
False : term bool.

def eq : A : type -> term A -> term A -> term bool.
[x] eq _ x x --> True
[x,y] eq _ x y --> False.
```

but we usually avoid this definition because it is not confluent, even without β-reduction: the term `eq A x x` reduces to both `True` and `False`. At the meta-level however, we take the liberty of accepting this kind of rewrite systems. It gives a direct access to the conversion check inside the language. Using the rewrite system of Section 3.4.3, the time

taken for computing `eq A t t` is usually proportional to the size of `t`; in the above rewrite system however Dedukti does not need to traverse the term thanks to hash consing.

3.5 Translating Logical Systems in Dedukti

Dedukti is able to check proofs coming from a wide variety of logical systems; in [11], Assaf describes three translators from proof assistants to Dedukti:

- Coqine, a translator for the Calculus of Inductive Constructions, as implemented in Coq;
- Krajono, another translator for the Calculus of Inductive Constructions, as implemented in Matita;
- Holide [13], a translator for HOL as implemented in OpenTheory [99], an exchange format for proof assistant in the HOL family.

A translator for PVS is also under development and several automatic theorem provers (Zenon Modulo [44], iProver Modulo [36], VeriT) use Dedukti as a proof format.

We start in Section 3.5.1 with the translation of natural deduction for (polymorphic) first-order logic which is at the core of this proof format for automatic theorem provers and is a good example of the way logical systems are embedded in Dedukti. More examples are given in [14]. We then quickly review the translators to Dedukti that we use to experiment interoperability between Coq and HOL in Part IV: Coqine in Section 3.5.2 and Holide in Section 3.5.3.

3.5.1 First-Order Logic in Dedukti

We translate natural deduction for polymorphic first-order logic, the system that we presented in Section 1.1.3, in Dedukti to demonstrate the use of Dedukti as a logical framework. To translate a language in Dedukti, we start by representing the syntax of the language using HOAS (see Section 2.5.1).

The representation of the type system is similar to the rewrite system of Section 3.4.3 but we do not need to represent dependent types so it is enough to start with:

```
type : Type.
term : type -> Type.
```

For each n-ary type constructor F in the signature, we declare a symbol F of type $\overbrace{\texttt{type -}\texttt{>}\ldots\texttt{->}}^{n \text{ times}}\ \texttt{type}$. The translation function for types is defined by

$$\begin{array}{lcl} [\![\alpha]\!] & := & \alpha \\ [\![F(\tau_1, \ldots, \tau_n)]\!] & := & F\ [\![\tau_1]\!]\ \ldots\ [\![\tau_n]\!] \end{array}$$

For each function symbol f of type scheme $\Pi\alpha_1.\ \ldots \Pi\alpha_k.\ (\tau_1, \ldots, \tau_n) \to \tau_0$, we declare a symbol f of type α_1`: type ->`$\ldots$`->` α_k`: type ->` `term` $[\![\tau_1]\!]$ `->`$\ldots$ `->` `term` $[\![\tau_n]\!]$ `->` `term` $[\![\tau_0]\!]$. The translation function for terms is defined by

$$\begin{array}{lcl} [\![x]\!] & := & x \\ [\![f(\tau_1, \ldots, \tau_k; t_1, \ldots, t_n)]\!] & := & f\ [\![\tau_1]\!]\ \ldots\ [\![\tau_k]\!]\ [\![t_1]\!]\ \ldots\ [\![t_n]\!] \end{array}$$

In order to translate formulae, we declare a new Dedukti type `prop` and all the connectives:

```
prop : Type.
true : prop.
false : prop.
eq : a : type -> term a -> term a -> prop.
and : prop -> prop -> prop.
or : prop -> prop -> prop.
imp : prop -> prop -> prop.
all : a : type -> (term a -> prop) -> prop.
ex : a : type -> (term a -> prop) -> prop.
all_type : (type -> prop) -> prop.
ex_type : (type -> prop) -> prop.
```

Negation and equivalence are seen as derived connectives:

```
def not (A : prop) : prop := imp A false.
def eqv (A : prop) (B : prop) : prop := and (imp A B) (imp B A).
```

For each predicate symbol P of type scheme $\Pi\alpha_1.\ \ldots \Pi\alpha_k.\ (\tau_1, \ldots, \tau_n)$, we declare a symbol P of type α_1`: type ->`$\ldots$`->` α_k`: type ->` `term` $[\![\tau_1]\!]$ `->`$\ldots$ `->` `term` $[\![\tau_n]\!]$ `->` `prop`.

The translation function for formulae is defined by

$$
\begin{array}{lcl}
[\![P(\tau_1,\ldots,\tau_k;t_1,\ldots,t_n)]\!] & := & P\ [\![\tau_1]\!]\ \ldots\ [\![\tau_k]\!]\ [\![t_1]\!]\ \ldots\ [\![t_n]\!] \\
[\![\top]\!] & := & \texttt{true} \\
[\![\bot]\!] & := & \texttt{false} \\
[\![t_1 =_\tau t_2]\!] & := & \texttt{eq}\ [\![\tau]\!]\ [\![t_1]\!]\ [\![t_2]\!] \\
[\![\neg]\!]\varphi & := & \texttt{not}\ [\![\varphi]\!] \\
[\![\varphi_1 \wedge \varphi_2]\!] & := & \texttt{and}\ [\![\varphi_1]\!]\ [\![\varphi_2]\!] \\
[\![\varphi_1 \vee \varphi_2]\!] & := & \texttt{or}\ [\![\varphi_1]\!]\ [\![\varphi_2]\!] \\
[\![\varphi_1 \Rightarrow \varphi_2]\!] & := & \texttt{imp}\ [\![\varphi_1]\!]\ [\![\varphi_2]\!] \\
[\![\varphi_1 \Leftrightarrow \varphi_2]\!] & := & \texttt{eqv}\ [\![\varphi_1]\!]\ [\![\varphi_2]\!] \\
[\![\forall x:\tau.\ \varphi]\!] & := & \texttt{all}\ [\![\tau]\!]\ (x\ \texttt{:}\ \texttt{term}\ [\![\tau]\!]\ \texttt{=>}\ [\![\varphi]\!]) \\
[\![\exists x:\tau.\ \varphi]\!] & := & \texttt{ex}\ [\![\tau]\!]\ (x\ \texttt{:}\ \texttt{term}\ [\![\tau]\!]\ \texttt{=>}\ [\![\varphi]\!]) \\
[\![\forall_{\mathbf{type}}\alpha.\ \varphi]\!] & := & \texttt{all\_type}\ (\alpha\ \texttt{:}\ \texttt{type}\ \texttt{=>}\ [\![\varphi]\!]) \\
[\![\exists_{\mathbf{type}}\alpha.\ \varphi]\!] & := & \texttt{ex\_type}\ (\alpha\ \texttt{:}\ \texttt{type}\ \texttt{=>}\ [\![\varphi]\!])
\end{array}
$$

Finally, we declare a type `proof` parameterized by a proposition. The type `proof A` is intended to represent the type of the proofs of the formula `A`. For each deduction rule in natural deduction, we declare a corresponding symbol in Figure 3.4

Until now, we have faithfully represented the syntax of natural deduction in Dedukti using the judgment-as-type paradigm (see Section 2.5.5). Actually, we have not yet used rewriting so this encoding uses only the pure $\lambda\Pi$-calculus and the only difference compared to an encoding in an implementation of ELF such as Twelf [147] is purely syntactic. We emphasize this by calling this translation a deep translation as opposed to shallow translations obtained by the proposition-as-type paradigm.

Through the Curry-Howard isomorphism, proofs can be interpreted as programs and the reduction of these programs correspond on the logical side to the process of cut elimination. Cut elimination can be added to our Dedukti signature by adding rewrite rules that simplify elimination rules applied to introduction rules:

```
[p] and_elim_1 _ _ (and_intro _ _ p _) --> p.
[q] and_elim_2 _ _ (and_intro _ _ _ q) --> q.

[p,r] or_elim _ _ _ p _ (or_intro_1 _ _ r) --> p r
[q,s] or_elim _ _ _ _ q (or_intro_2 _ _ s) --> q s.

[p,q] imp_elim _ _ (imp_intro _ _ p) q --> p q.

[p,x] all_elim _ _ (all_intro _ _ p) x --> p x.

[p,x,q] ex_elim _ _ _ p (ex_intro _ _ x q) --> p x q.

[p,a] all_type_elim _ (all_type_intro _ p) a --> p a.
```

```
proof : prop -> Type.

def true_intro : proof true.

def false_elim : A : prop -> proof false -> proof A.

def and_intro : A : prop -> B : prop ->
                proof A -> proof B -> proof (and A B).
def and_elim_1 : A : prop -> B : prop -> proof (and A B) -> proof A.
def and_elim_2 : A : prop -> B : prop -> proof (and A B) -> proof B.

def or_intro_1 : A : prop -> B : prop -> proof A -> proof (or A B).
def or_intro_2 : A : prop -> B : prop -> proof B -> proof (or A B).
def or_elim : A : prop -> B : prop -> C : prop ->
              (proof A -> proof C) -> (proof B -> proof C) ->
              proof (or A B) -> proof C.

def imp_intro : A : prop -> B : prop ->
                (proof A -> proof B) -> proof (imp A B).
def imp_elim : A : prop -> B : prop ->
               proof (imp A B) -> proof A -> proof B.

def all_intro : a : type -> A : (term a -> prop) ->
                (x : term a -> proof (A x)) -> proof (all a A).
def all_elim : a : type -> A : (term a -> prop) ->
               proof (all a A) -> x : term a -> proof (A x).

def ex_intro : a : type -> A : (term a -> prop) -> x : term a ->
               proof (A x) -> proof (ex a A).
def ex_elim : a : type -> A : (term a -> prop) -> B : prop ->
              (x : term a -> proof (A x) -> proof B) ->
              proof (ex a A) -> proof B.

def all_type_intro : A : (type -> prop) ->
                     (a : type -> proof (A a)) -> proof (all_type A).
def all_type_elim : A : (type -> prop) ->
                    proof (all_type A) -> a : type -> proof (A a).

def ex_type_intro : A : (type -> prop) -> a : type ->
                    proof (A a) -> proof (ex_type A).
def ex_type_elim : A : (type -> prop) -> B : prop ->
                   (a : type -> proof (A a) -> proof B) ->
                   proof (ex_type A) -> proof B.

def eq_intro : a : type -> x : term a -> proof (eq a x x).
def eq_elim : a : type ->
              x : term a -> y : term a -> A : (term a -> prop) ->
              proof (A x) -> proof (eq a x y) -> proof (A y).
```

Figure 3.4: Dedukti signature for polymorphic natural deduction

```
[p,a,q] ex_type_elim _ _ p (ex_type_intro _ a q) --> p a q.

[p] eq_elim _ _ _ _ p (eq_intro _ _) --> p.
```

In the case of implication, we can read the introduction and elimination rules as axiomatizing a logical equivalence between the types `proof (imp A B)` and `proof A -> proof B` and the rewrite rule `[p,q] imp_elim _ _ (imp_intro _ _ p) q --> p q.` as stating that the functions `imp_elim` and `imp_intro` are inverses of each other. We can further identify the types `proof (imp A B)` and `proof A -> proof B` thanks to the rewrite rule `[A,B] proof (imp A B) --> proof A -> proof B.` and quite generally, we can encode all the connectives using impredicative encodings:

```
def proof : prop -> Type.
[] proof true --> A : prop -> proof A -> proof A
[] proof false --> A : prop -> proof A
[A,B] proof (imp A B) --> proof A -> proof B
[A,B] proof (and A B) -->
      C : prop -> (proof A -> proof B -> proof C) -> proof C
[A,B] proof (or A B) -->
      C : prop -> (proof A -> proof C) -> (proof B -> proof C) -> proof C
[a,A] proof (all a A) --> x : term a -> proof (A x)
[a,A] proof (ex a A) -->
      B : prop -> (x : term a -> proof (A x) -> proof B) -> proof B
[A] proof (all_type A) --> a : type -> proof (A a)
[A] proof (ex_type A) -->
      B : prop -> (a : type -> proof (A a) -> proof B) -> proof B
[a,x,y] proof (eq a x y) -->
      A : (term a -> prop) -> proof (A x) -> proof (A y).
```

Using these rewrite rules, all the deduction rules for natural deduction can be derived (see Figure 3.5).

The cut-elimination rewrite rules are now superfluous, we can remove them and ask Dedukti to check that cut-reduction still holds (see Figure 3.6) using the `#CONV` command for checking that two terms are convertibles.

This translation of natural deduction is more shallow in the sense that it reuses more features available in Dedukti: implication is mapped to Dedukti arrow, universal quantification is mapped to Dedukti dependent product etc...

Proof terms in this shallow translation are lighter than the ones of the deep translation because less type annotations are needed. For example, the proof of $(P(t) \wedge \forall x : \tau.\ P(x) \Rightarrow$

```
def true_intro : proof true := A => p => p.
def false_elim (A : prop) (p : proof false) : proof A := p A.
def and_intro (A : prop) (B : prop) (p : proof A) (q : proof B)
              : proof (and A B)
:= C : prop => r : (proof A -> proof B -> proof C) => r p q.
def and_elim_1 (A : prop) (B : prop) (p : proof (and A B)) : proof A
:= p A (x => y => x).
def and_elim_2 (A : prop) (B : prop) (p : proof (and A B)) : proof B
:= p B (x => y => y).
def or_intro_1 (A : prop) (B : prop) (p : proof A) : proof (or A B)
:= C : prop =>
   q : (proof A -> proof C) => r : (proof B -> proof C) => q p.
def or_intro_2 (A : prop) (B : prop) (p : proof B) : proof (or A B)
:= C : prop =>
   q : (proof A -> proof C) => r : (proof B -> proof C) => r p.
def or_elim (A : prop) (B : prop) (C : prop)
            (p : proof A -> proof C) (q : proof B -> proof C)
            (r : proof (or A B)) : proof C
:= r C p q.
def imp_intro (A : prop) (B : prop) (p : (proof A -> proof B))
              : proof (imp A B) := p.
def imp_elim (A : prop) (B : prop) (p : proof (imp A B))
             : proof A -> proof B := p.
def all_intro (a : type) (A : (term a -> prop))
              (p : x : term a -> proof (A x)) : proof (all a A)
:= p.
def all_elim (a : type) (A : (term a -> prop)) (p : proof (all a A))
             : x : term a -> proof (A x) := p.
def ex_intro (a : type) (A : (term a -> prop))
             (x : term a) (p : proof (A x)) : proof (ex a A)
:= B : prop => q : (x : term a -> proof (A x) -> proof B) => q x p.
def ex_elim (a : type) (A : (term a -> prop)) (B : prop)
            (p : x : term a -> proof (A x) -> proof B)
            (q : proof (ex a A)) : proof B
:= q B p.
def all_type_intro (A : (type -> prop)) (p : a : type -> proof (A a))
                   : proof (all_type A) := p.
def all_type_elim (A : (type -> prop)) (p : proof (all_type A))
                  : a : type -> proof (A a) := p.
def ex_type_intro (A : (type -> prop)) (a : type) (p : proof (A a))
                  : proof (ex_type A)
:= B : prop => q : (a : type -> proof (A a) -> proof B) => q a p.
def ex_type_elim (A : (type -> prop)) (B : prop)
                 (p : a : type -> proof (A a) -> proof B)
                 (q : proof (ex_type A)) : proof B
:= q B p.
def eq_intro (a : type) (x : term a) : proof (eq a x x)
:= A : (term a -> prop) => p : proof (A x) => p.
def eq_elim (a : type) (x : term a) (y : term a)
            (A : (term a -> prop))
            (p : proof (A x)) (q : proof (eq a x y)) : proof (A y)
:= q A p.
```

Figure 3.5: Shallow embedding of Natural Deduction in Dedukti

```
#CONV (A : prop => B : prop => p : proof A => q : proof B =>
       and_elim_1 A B (and_intro A B p q)),
      (A : prop => B : prop => p : proof A => q : proof B => p).
#CONV (A : prop => B : prop => p : proof A => q : proof B =>
       and_elim_2 A B (and_intro A B p q)),
      (A : prop => B : prop => p : proof A => q : proof B => q).
#CONV (A : prop => B : prop => C : prop => p : proof A =>
      q : (proof A -> proof C) => r : (proof B -> proof C) =>
       or_elim A B C q r (or_intro_1 A B p)),
      (A : prop => B : prop => C : prop => p : proof A =>
       q : (proof A -> proof C) => r : (proof B -> proof C) => q p).
#CONV (A : prop => B : prop => C : prop => p : proof B =>
       q : (proof A -> proof C) => r : (proof B -> proof C) =>
       or_elim A B C q r (or_intro_2 A B p)),
      (A : prop => B : prop => C : prop => p : proof B =>
       q : (proof A -> proof C) => r : (proof B -> proof C) => r p).
#CONV (A : prop => B : prop => p : (proof A -> proof B) =>
       q : proof A => imp_elim A B (imp_intro A B p) q),
      (A : prop => B : prop => p : (proof A -> proof B) =>
       q : proof A => p q).
#CONV (a : type => A : (term a -> prop) =>
       p : (x : term a -> proof (A x)) => x : term a =>
       all_elim a A (all_intro a A p) x),
      (a : type => A : (term a -> prop) =>
       p : (x : term a -> proof (A x)) => x : term a => p x).
#CONV (a : type => A : (term a -> prop) => B : prop =>
       p : (x : term a -> proof (A x) -> proof B) =>
       x : term a => q : proof (A x) =>
       ex_elim a A B p (ex_intro a A x q)),
      (a : type => A : (term a -> prop) => B : prop =>
       p : (x : term a -> proof (A x) -> proof B) =>
       x : term a => q : proof (A x) => p x q).
#CONV (A : (type -> prop) => p : (a : type -> proof (A a)) =>
       a : type => all_type_elim A (all_type_intro A p) a),
      (A : (type -> prop) => p : (a : type -> proof (A a)) =>
       a : type => p a).
#CONV (A : (type -> prop) => B : prop =>
       p : (a : type -> proof (A a) -> proof B) =>
       a : type => q : proof (A a) =>
       ex_type_elim A B p (ex_type_intro A a q)),
      (A : (type -> prop) => B : prop =>
       p : (a : type -> proof (A a) -> proof B) =>
       a : type => q : proof (A a) => p a q).
#CONV (a : type => A : (term a -> prop) => x : term a =>
       p : proof (A x) => eq_elim a x x A p (eq_intro a x)),
      (a : type => A : (term a -> prop) => x : term a =>
       p : proof (A x) => p).
```

Figure 3.6: Checking cut elimination in the shallow embedding of natural deduction in Dedukti

$Q(x)) \Rightarrow Q(t)$ that we gave in 1.1.2 is written as follows in the deep encoding:

```
a : type.
t : term a.
P : term a -> prop.
Q : term a -> prop.

def example_0 : proof (imp (and (P t) (all a (x => imp (P x) (Q x)))) (Q t))
:=
 imp_intro
   (and (P t) (all a (x => imp (P x) (Q x))))
   (Q t)
   (p =>
      imp_elim (P t) (Q t)
         (all_elim a (x => imp (P x) (Q x))
            (and_elim_2 (P t) (all a (x => imp (P x) (Q x))) p)
            t)
         (and_elim_1 (P t) (all a (x => imp (P x) (Q x))) p)).
```

In the shallow encoding, this term reduces to the following much shorter proof term:

```
a : type.
t : term a.
P : term a -> prop.
Q : term a -> prop.

def example_0 : proof (imp (and (P t) (all a (x => imp (P x) (Q x)))) (Q t))
:= p =>
    p
      (all a (x => imp (P x) (Q x)))
      (x => y => y)
      t
      (p (P t) (x => y => x)).
```

3.5.2 Coqine

Coq implements the Calculus of Inductive Constructions (CIC), an extension of Martin-Löf Type Theory with an impredicative universe **Prop** (see Section 2.4.3). Universes in CIC are types of types; alongside **Prop**, there is a universe $\mathbf{Type}_i$ for each natural number i. The type of **Prop** is in $\mathbf{Type}_1$ and for all i, the type of $\mathbf{Type}_i$ is in $\mathbf{Type}_{i+1}$.

Coqine [11], is a translator of Coq to Dedukti, it has been implemented by Assaf as a Coq plugin from the ideas of a previous version [28] and his own improvements in order to handle the universe hierarchy.

Coqine takes as input compiled Coq files using the `.vo` extension and produces Dedukti

files using the `.dk` extension. The Dedukti files produced by Coqine depend on a small hand-written Dedukti file `coq.dk` representing CIC. This file contains in particular the following declarations:

```
#NAME Coq.

(; Natural numbers ;)

Nat : Type.
z : Nat.
s : Nat -> Nat.

(; Universes ;)

Universe : Type.
prop : Universe.
type : Nat -> Universe.

U : Universe -> Type.
def T : s : Universe -> U s -> Type.
```

CIC universes are represented in `coq.dk` by terms of type `Universe`. **Prop** is represented by `prop` and $\mathbf{Type}_i$ is represented by `type i`. The CIC judgment corresponding to the fact that a type A lies in a universe s is represented in Dedukti by the typing judgment `A : U s`. Similarly, the CIC judgment corresponding to the fact that the term t has a type A in a universe s is represented by the typing judgment `t : T s A`.

The file `coq.dk` also contains declarations and rewrite rules to support some features of CIC. One of these features, universe cumulativity (if a type A lies in a universe $\mathbf{Type}_i$ then it also lies in all bigger universes $\mathbf{Type}_{i+j}$), is implemented using a non-linear rewrite rule [11] so Coqine developments require Dedukti non-linearity flag and cannot automatically be checked for confluence. In fact, the file `coq.dk` itself is not confluent but could be extended to a confluent rewrite system if matching modulo associativity, commutativity, and identity would be supported in Dedukti [17, 16].

Unfortunately, Coqine still lacks a few features such as module functors which are used in Coq standard library. It works well on small examples as long as we do not require substantial parts of the standard library.

3.5.3 Holide

Several proof assistants have implemented HOL (see Section 2.3.2) following the LCF approach: a small and simple kernel implements the rules of the logic and exports an abstract type of theorems. The proof assistants of the HOL family usually implement $\mathcal{Q}_0$, a classical presentation of HOL taking only equality and a choice operator as primitives [7].

OpenTheory is a package manager for HOL libraries of proofs developed and maintained by Hurd [99]. Each OpenTheory package is composed of article files containing the proofs. All proof assistants of the HOL family can import OpenTheory article files and most of them can also export their developments to the OpenTheory article format. OpenTheory standard library is generated by exporting most of HOL Light standard library.

Holide [13] is a translator of HOL to Dedukti. It takes OpenTheory article files with the `.art` extension as input and produces Dedukti files with the `.dk` extension as output. The Dedukti files generated by Holide depend on a small hand-written Dedukti file `hol.dk` representing HOL type system and logic:

```
#NAME hol.

(; HOL Types ;)

def type : Type.

bool : type.
ind  : type.
def arr  : type -> type -> type.

(; HOL Terms ;)

def term : type -> Type.
[a,b] term (arr a b) --> term a -> term b.

eq : a : type -> term (arr a (arr a bool)).
select : a : type -> term (arr (arr a bool) a).

(; HOL Proofs ;)

def proof : term bool -> Type.

(; Axioms of Q0 ;)

REFL : a : type -> t : term a -> proof (eq a t t).
```

```
ABS_THM :
  a : type ->
  b : type ->
  f : (term a -> term b) ->
  g : (term a -> term b) ->
  (x : term a -> proof (eq b (f x) (g x))) ->
  proof (eq (arr a b) f g).
APP_THM :
  a : type ->
  b : type ->
  f : term (arr a b) ->
  g : term (arr a b) ->
  x : term a ->
  y : term a ->
  proof (eq (arr a b) f g) ->
  proof (eq a x y) ->
  proof (eq b (f x) (g y)).
PROP_EXT : p : term bool -> q : term bool ->
  (proof q -> proof p) ->
  (proof p -> proof q) ->
  proof (eq bool p q).
EQ_MP : p : term bool -> q : term bool ->
  proof (eq bool p q) ->
  proof p ->
  proof q.
def BETA_CONV (a : type) (b : type)
      (f : term a -> term b) (u : term a) := REFL b (f u).
```

In this signature, `bool` represents the type o of propositions, `ind` represents the type ι of individuals, and `arr a b` represents the arrow type $a \to b$. Thanks to the rewrite rule `[a, b] term (arr a b) --> term a -> term b`, we can translate HOL application by Dedukti application and HOL abstraction by Dedukti abstraction. `eq` represents equality, `select` is the choice symbol and the axioms `REFL` to `EQ_MP` are the axioms of $\mathcal{Q}_0$ presented in Section 2.3.2. Since Dedukti conversion extends β-conversion, the axiom `EQ_MP` of $\mathcal{Q}_0$ can be derived.

OpenTheory and Holide have been extended by Shuai to also take implication and universal quantification as primitives. This extension of OpenTheory and Holide is called Holala [171]. Holala requires a slightly bigger presentation of the logic but it leads to smaller proofs and, most importantly for our purpose, to a shallower translation since implication and universal quantification can directly be mapped to Dedukti arrow type and dependent product. The file `hol.dk` used by Holala extends the previous one as follows:

```
imp : term bool -> term bool -> term bool.
```

```
forall : a : type -> (term (arr a bool) -> (term bool)).

[p,q] proof (imp p q) --> proof p -> proof q.
[a,p] proof (forall a p) --> x : term a -> proof (p x).
```

In Part IV, we use the Holala version of OpenTheory and Holide. The article files can be produced from the modified version of HOL Light available at `https://github.com/airobert/holala` and the Holala version of Holide needed to translate them to Dedukti is available on the `holala` branch at `https://gforge.inria.fr/projects/holide`.

Part II

Object Calculi in Dedukti

Object-oriented programming languages are nowadays the dominant programming paradigm. By regrouping inside objects data and operations on them, object-oriented languages focus on abstraction, maintaining invariants, code reuse through modularity, while also breaking the rigidity of modules thanks to redefinition and overloading.

Despite the popularity of object-oriented languages in the programming world, purely functional languages are often preferred in proof systems because it is easier to enforce termination by a typing discipline in the context of purely functional languages. We are aware of two systems mixing object-oriented mechanisms with proof techniques: Yarrow, a type system based on pure type systems with subtyping [176], and FoCaLiZe, an environment for certified programming that we are going to discuss at length in Parts III and IV.

Dedukti is both a programming language and a dependent type system. Termination in Dedukti is not mandatory and it is a good experimentation platform for the encoding of object-oriented languages. Such encodings can be a starting point for the design of a dependent type system for objects able to express proofs as methods.

With their gain of popularity in the 90s, object-oriented languages raised theoretical interest. They were connected with functional type systems, especially System $\mathrm{F}^{\omega}_{<:}$, through several encodings [148, 35]. However, encoding simple object-oriented languages into a system as complex as System $\mathrm{F}^{\omega}_{<:}$ was found unsatisfactory and foundational calculi for the object-oriented paradigm started to be designed.

The λ-calculus of objects [73] is an extension of λ-calculus with a few object primitives: calling a method (sending a message), updating a method (replacing its definition by another one), and extending an object by a new method. Several, slightly different, type systems have been proposed for this calculus: subtyping is proposed in [74], type annotations (making type-checking decidable) are added in [116], typing of incomplete objects (objects missing some methods but already usable as prototypes) is added in [32], typing of methods extending the object type is added in [64].

Abadi and Cardeli [2] proposed ς-calculi, a family of purely object-oriented calculi which simplify the λ-calculus of objects by restricting the object primitives to selection

and update only and by removing λ-abstraction and application. These calculi correspond to common type systems for functional languages based on λ-calculus. In ς-calculi however, subtyping and recursive types play a more important role than in functional type systems. λ-abstraction and application are easy to encode in ς-calculi but extending an object by a new method cannot be done in a polymorphic way in ς-calculi because the type corresponding to the notion of the current object with an extra method is not expressible (and adding it would lead to a system very similar to λ-calculus of objects). Hence ς-calculi are a form of restriction of the λ-calculus of objects which lead to smaller foundational calculi. This restriction also allows to consider simpler type disciplines such as simple types, whereas type systems for the λ-calculus of objects require polymorphism.

ς-calculi and the λ-calculus of objects are foundations for object-based (or prototype-based) programming language such as Self and Javascript but they can also encode classes. A functional class-based core of the Java language, named Featherweight Java [100], has also emerged as a framework for studying extensions of Java independently.

These calculi have been embedded in proof systems such as Isabelle [75, 93] and Coq [120, 49] but none of these embeddings preserves the reduction behaviour for two reasons:

1. these proof systems require all functions to terminate and object calculi do not try to enforce termination
2. the motivation for most of these encodings was to prove properties of the reduction relation such as confluence and subject reduction, which is not possible to do in a semantics-preserving encoding.

Reduction-preserving encodings of objects have been designed in the context of rewriting logic (in the ρ-calculus [51, 50] and in the Maude system [52]). However, the resulting object-oriented languages are untyped and the encodings seem not to be easily adaptable to typing.

We propose a shallow encoding of the simply-typed ς-calculus in Dedukti. The notion of shallow embedding is generally not precisely defined but in this context we call a translation shallow when it preserves variable binding, typing, and operational semantics.

In Chapter 4, we present the simply-typed ς-calculus and its shallow translation in Dedukti. The simply-typed ς-calculus is the simplest of the typed object calculi. It lacks the most interesting feature of type systems for object-oriented languages: subtyping. In Chapter 5, we extend our shallow embedding of the simply-typed ς-calculus to subtyping. The Chapter 6 is devoted to our implementation.

Chapter 4

Simply-Typed ς-Calculus in Dedukti

The simply-typed ς-calculus is the simplest typed ς-calculus. It has been introduced by Abadi and Cardeli in [1]. In this chapter, we first recall its definition in Section 4.1, then embed it in Dedukti in two steps. The first step is as shallow as possible while remaining strongly terminating. The second step drops the termination restriction to obtain a fully shallow embedding. The terminating translation is defined in Sections 4.2, 4.3, and 4.4 and the shallow translation is defined in Section 4.5.

4.1 Simply-Typed ς-Calculus

The simply-typed ς-calculus is similar to the simply-typed λ-calculus that we presented in Section 2.2 but to be used as a core calculus for object-oriented languages instead of functional languages. In this section, we present the syntax of the types and the terms of the calculus, the typing rules, and the operational semantics. We conclude this section by small examples of encodings of programming constructs in the ς-calculus.

4.1.1 Syntax

The syntax of the simply-typed ς-calculus is given in Figure 4.1; it is composed of types and terms. Types are, possibly empty, records of types in which labels are assumed distinct and their order is irrelevant. Terms are either ground objects given by a, possibly empty, record of methods, each method being a term bound by the self binder ς binding the object itself, given a term a, its method labeled by l can be selected or updated by

Type	$A, B, \ldots$	$::=$	$[l_i : A_i]_{i=1\ldots n}$	*Object type*
Term	$a, b, \ldots$	$::=$	x	*Variable*
			$[l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n}$	*Object*
			$a.l$	*Method selection*
			$a.l \Leftarrow \varsigma(x : A)b$	*Method update*

Figure 4.1: Syntax of the simply-typed ς-calculus

Notation: In this figure, A abbreviates $[l_i : A_i]_{i=1\ldots n}$

$$\frac{(x : A) \in \Delta}{\Delta \vdash x : A}\ \text{(Type Var)} \qquad \frac{\Delta, x_i : A \vdash a_i : A_i \qquad \forall i \in 1 \ldots n}{\Delta \vdash [l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n} : A}\ \text{(Type Obj)}$$

$$\frac{\Delta \vdash a : A \qquad j \in 1 \ldots n}{\Delta \vdash a.l_j : A_j}\ \text{(Type Select)}$$

$$\frac{\Delta \vdash a : A \qquad \Delta, x : A \vdash b : A_j \qquad j \in 1 \ldots n}{\Delta \vdash a.l_j \Leftarrow \varsigma(x : A)b : A}\ \text{(Type Update)}$$

Figure 4.2: Typing rules for simply-typed ς-calculus

a new method body. When the ς binders are unused, we might omit them, for example, we write $[l = []]$ and $[l = \varsigma(x : A)x.l].l \Leftarrow []$ respectively instead of $[l = \varsigma(x : A)[]]$ and $[l = \varsigma(x : A)x.l].l \Leftarrow \varsigma(x : A)[]$ where A is the type $[l : []]$.

4.1.2 Typing

The typing judgment $\Delta \vdash a : A$ means that in context Δ, the term a has type A.

Contexts are composed of variable type assignments, we assume all variables appearing in a context to be distinct:

$$\Delta \quad ::= \quad \emptyset \mid \Delta, x : A$$

The typing rules for the simply-typed ς-calculus are given in Figure 4.2. An object is

well-typed when all its methods have the expected type in the context in which the self variable has the type of the object being defined. Selecting a method returns a term of the expected type and updating a term by a well-typed method returns a term of the same type.

4.1.3 Operational Semantics

Values of the ς-calculus are ground objects. The operational semantics is given by two reduction rules, one defining selection and one defining update on values:

$$\begin{array}{lcl} a.l_j & \leadsto & a_j\{x_j \backslash a\} \\ a.l_j \Leftarrow \varsigma(x:A')b & \leadsto & [l_j = \varsigma(x:A)b, l_i = \varsigma(x_i:A)a_i]_{i=1\ldots n, i\neq j} \end{array}$$

where A is $[l_i : A_i]_{i=1\ldots n}$ and a is $[l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n}$. The type A' in the second rule might be any type, the typing rule of Figure 4.2 enforces that well-typed application of this rule must satisfy $A' = A$ but this will not remain true when we will extend the type system with subtyping in Chapter 5.

4.1.4 Examples

The simply-typed ς-calculus is a bit limited but can already encode a few interesting examples:

- records can obviously be encoded by not using the ς binder
- booleans and conditional expressions can be encoded but since we lack polymorphism, we will have a different copy of the encoding of boolean and conditional for each type A for which we want a conditional:

$$\begin{array}{lcl} \mathsf{Bool}_A & := & [if : A, then : A, else : A] \\ \mathsf{true}_A : \mathsf{Bool}_A & := & [if = \varsigma(self : \mathsf{Bool}_A)self.then, \\ & & \ then = \varsigma(self : \mathsf{Bool}_A)self.then, \\ & & \ else = \varsigma(self : \mathsf{Bool}_A)self.else] \\ \mathsf{false}_A : \mathsf{Bool}_A & := & [if = \varsigma(self : \mathsf{Bool}_A)self.else, \\ & & \ then = \varsigma(self : \mathsf{Bool}_A)self.then, \\ & & \ else = \varsigma(self : \mathsf{Bool}_A)self.else] \\ \mathsf{ifthenelse}(b : \mathsf{Bool}_A, t : A, e : A) & := & ((b.then \Leftarrow t).else \Leftarrow e).if \end{array}$$

For every terms t and e of type A, we have $\mathsf{ifthenelse}(\mathsf{true}_A, t, e) \leadsto^* t$ and $\mathsf{ifthenelse}(\mathsf{false}_A, t, e) \leadsto^* e$.

- simply-typed λ-calculus can also be encoded:

$$\begin{array}{lcl} A \to B & := & [arg : A, val : B] \\ \lambda(x : A)b & := & [arg = \varsigma(self : A \to B)self.arg, val = \varsigma(self : A \to B)b\{x \backslash self.arg\}] \\ f(t) & := & (f.arg \Leftarrow t).val \end{array}$$

This encoding is adequate in the sense that it preserves β-reduction: for every terms t and b, $(\lambda(x : A)b)(t) \rightsquigarrow^* b\{x \backslash t\}$.

4.2 Translation of Types in Dedukti

Our translation of the ς-calculus to Dedukti aims at being as shallow as possible, that is, we want a translation preserving scoping, typing, and operational semantics. In particular, a ς-term of type A will be translated to a Dedukti term whose type depends on the translation of A so we start by the translation of ς-types to Dedukti. This is the topic of this section. In Section 4.3, we define a relation which plays an important role in our translation. In Section 4.4, we define a translation of ς-terms preserving scoping and typing and in Section 4.5 we slightly modify it so that it also preserves the operational semantics of the ς-calculus.

There is no predefined notion of records in Dedukti but lists are very easy to define, hence there are a few ways to define ς simple types in Dedukti:

1. Define them as lists (of pairs of labels and types) and rely on the translator to always print the labels in the same order (for example, in alphabetic order)

2. Use dependent types to add logical arguments to the constructor enforcing that the lists are sorted and duplicate-free by construction

3. Use rewriting to make lists given in different order convertible; this can be done either by declaring the list concatenation as an associative-commutative operation or by using rewrite rules to sort lists as follows:

```
def label : Type.
type : Type.
nil : type.
cons : label -> type -> type -> type.
[l1,A1,l2,A2,A3]
   cons l1 A1 (cons l2 A2 A3) --> cons l2 A2 (cons l1 A1 A3).
```

In order to keep a terminating rewrite system, we need to restrict the application of this rewrite rule to the case where `l1` $>$ `l2` by a side condition. Moreover, in order to guarantee uniqueness of labels, we need to break the symmetry and arbitrary choose one of the associated types.

The third solution does not seem appropriate for the current version of Dedukti, which features neither associative-commutative declarations nor side conditions but might be interesting in the future. The second solution is already doable but a bit complex: types are constructed from `nil` and `cons` at the same time than the inductive relation `minors` comparing labels and types: a label l minors a type A if l is strictly smaller than all labels occurring in A (or equivalently since A is sorted, if l is strictly smaller than the head label of A):

```
def label : Type.
def lt : label -> label -> Type.

type : Type.
def minors : label -> type -> Type.
nil : type.
cons : l : label -> A : type -> B : type -> minors l B -> type.
minors_nil : l : label -> minors l nil.
minors_cons : l : label -> l' : label -> A : type -> B : type ->
              H : minors l' B -> lt l l' -> minors l (cons l' A B H).
```

From the theorems stating that `lt` is a total ordering, we can build a function for inserting a label and the corresponding type in a type, without the trouble of manually finding its position and proving that the list is sorted:

```
def insert : label -> type -> type := ...
```

However, to ease the reading of normal forms of ς-types and ς-terms, we prefer the first solution. We simply encode types by association lists and the burden of sorting them is left to the translator. Here is the definition of types as association lists:

```
def label : Type.
type : Type.
tnil : type.
tcons : label -> type -> type -> type.
```

The translation function $[\![\bullet]\!]$ for types is given by

$$[\![[l_i : A_i]_{l_1<\ldots<l_n}]\!] := \texttt{tcons}\ l_1\ [\![A_1]\!]\ (\ldots(\texttt{tcons}\ l_n\ [\![A_n]\!]\ \texttt{tnil})\ldots)$$

Translated types are Dedukti terms of type `type`:

Theorem 7. *Let A be a ς-type and Γ be a well-formed context of the $\lambda\Pi$-calculus containing:*

- *the four previous declarations and*
- *for each label l occurring in A a declaration l : `label`,*

then the judgment $\Gamma \vdash [\![A]\!]$: `type` holds in the $\lambda\Pi$-calculus modulo.

Proof. A is a ς-type so it is of the form $[l_i : A_i]_{i=1\ldots n}$ for some n. Without loss of generality, we assume that the labels are sorted ($l_1 < \ldots < l_n$) and we proceed by induction on the size of A:

- if $n = 0$, then $[\![A]\!] = [\![[]]\!] =$ `tnil` and we conclude by the rule (Var) of Section 3.2
- if $n = m + 1$, then $[\![A]\!] = [\![[l_i : A_i]_{l_1<\ldots<l_{m+1}}]\!] =$ `tcons` l_1 $[\![A_1]\!]$ $[\![B]\!]$ where B is the ς-type $[l_i : A_i]_{l_2<\ldots<l_{m+1}}$.

  Both A_1 and B are smaller than A so we can apply the induction hypothesis to them to get $\Gamma \vdash [\![A_1]\!]$: `type` and $\Gamma \vdash [\![B]\!]$: `type`. Moreover, l_1 is a label occurring in A so the declaration l_1 : `label` is present in Γ hence $\Gamma \vdash l_1$: `label` by the rule (Var) of Section 3.2.

  By three applications of the (App) rule of Section 3.2, we conclude $\Gamma \vdash [\![A]\!]$: `type`.

□

This translation function is injective:

Theorem 8 (Injectivity of the translation of ς-types)**.** *Let A and B be two ς-types such that $[\![A]\!]$ and $[\![B]\!]$ are convertible, then A and B are identical.*

Proof. $[\![A]\!]$ and $[\![B]\!]$ have the size because they are convertible. Moreover a ς-type and its translation have the same size hence A and B have the same size. The proof is done by induction on this common size, both cases are trivial. □

A direct advantage of this solution is that we do not even need to compare labels in Dedukti.

4.3 Membership as an Inductive Relation

Since we gave up sorting, terms of type `type` in our translation may contain label duplicates. To distinguish them, we do not introduce membership as a function but as an inductive relation `mem`: if a label l appears several times associated to the type A in the type B, then the type `mem` l A B will have several distinct inhabitants, called the positions of $(l : A)$ in B. The two constructors of `mem` l A B are `mhead` indicating that $(l : A)$ is found at the head of the type B and `mtail` indicating that $(l : A)$ is found in the tail of B.

```
mem : label -> type -> type -> Type.
mhead : l : label -> A : type -> B : type -> mem l A (tcons l A B).
mtail : l1 : label -> l2 : label ->
        A1 : type -> A2 : type -> B : type ->
        mem l1 A1 B -> mem l1 A1 (tcons l2 A2 B).
```

This inductive relation plays an important role because it is very useful for defining recursive functions operating on types or objects without worrying about duplicates. In particular, we shall define selection and update this way in Section 4.4.2.

4.4 Terminating Translation of Terms

We start with an as-shallow-as-possible terminating translation of the simply typed ς-calculus. More precisely, we define a terminating Dedukti context Σ_ς and a translation function $[\![\bullet]\!]$ mapping well-typed ς-terms to Dedukti terms which are also well-typed in Σ_ς. We drop the termination requirement in the next section in order to get a fully shallow translation; that is, we define another Dedukti context Σ'_ς such that the translation function $[\![\bullet]\!]$ still preserves typing but also preserves the operational semantics.

Splitting our work this way has several advantages:

- The terminating translation is expressible in the Coq system, for which we are not required to prove termination nor confluence,
- Confluence and termination of Σ_ς will be obvious,

- Type-checking is decidable for Dedukti confluent and terminating rewrite systems. In practice, this guarantees that Dedukti terminates when asked to check this development, which is comfortable.

4.4.1 Objects, Methods, and Preobjects

In Section 4.2, we have translated ς-types to Dedukti terms of type `type`. To each of these Dedukti terms, we associate a Dedukti type to contain the translated terms of this type. Concretely, we need a function `Obj` interpreting the association lists of Section 4.2 as Dedukti types:

```
def Obj : type -> Type.
```

In order to type methods, we also introduce a function `Meth`; `Meth A B` is the type of the methods of objects of type A returning type B:

```
def Meth : type -> type -> Type.
```

We cannot define the type `Obj` as some dependent list of methods because sublists of objects are not themselves objects. For this reason, we introduce the notion of *preobject*: a preobject of an object of type A is given by a list of methods implementing part of A. More concretely, we introduce a new type constructor `Preobj` taking two parameters: one being the type of the object we are building and one beeing the type of the methods we have defined:

```
Preobj : type -> type -> Type.
```

The type `Preobj` A B reads as the type of preobjects of type A defined on part B. It is parametric in A and dependent in B. The only preobject of type A defined on the empty part is constructed by `pnil` A. A method labelled by l from A to B can be added to a preobject of type A defined on part C to form a preobject of type A defined on part `tcons` l B C.

```
pnil : A : type -> Preobj A tnil.
pcons : A : type -> l : label -> B : type -> C : type ->
        Meth A B -> Preobj A C -> Preobj A (tcons l B C).
```

`Obj` can now be defined, an object of type A is exactly a preobject of type A defined on part A:

```
[A] Obj A --> Preobj A A.
```

However, to construct objects, we need to construct methods; the simplest choice would be to define `Meth A B` as `Obj A -> Obj B` but this implies a negative occurence of `Preobj` in its recursive definition and leads to non-termination (as we will see in next section). We avoid this issue by axiomatizing the equivalence between `Meth A B` and `Obj A -> Obj B`:

```
def Eval_meth : A : type -> B : type -> Meth A B -> Obj A -> Obj B.
def Make_meth : A : type -> B : type -> (Obj A -> Obj B) -> Meth A B.
```

4.4.2 Method Selection and Update

Because objects are defined as a special case of preobjects, we first define selection and update on preobjects. Method selection and update functions on preobjects traverse the preobject structure following a path given as argument by a position. When the method is reached, it is returned by the selection function or replaced by the update function:

```
def preselect : A : type -> l : label -> B : type -> C : type ->
                mem l B C -> Preobj A C -> Meth A B.
def preupdate : A : type -> l : label -> B : type -> C : type ->
                mem l B C -> Preobj A C -> Meth A B -> Preobj A C.
```

These functions are defined by induction on position, the base case corresponds to a (label, type) pair found at the head of part C:

```
[m] preselect _ _ _ _ (mhead _ _ _) (pcons _ _ _ _ m _) --> m.
[A, B, C, l, o, m]
   preupdate _ _ _ _ (mhead _ _ _) (pcons A l B C _ o) m
     -->
   pcons A l B C m o.
```

These rules make extensive use of the capacities of Dedukti to recognize ill-typed linear patterns whose instances are all well-typed; they are equivalent to the more verbose

```
[A, l, B, C, l', B', C', A'', l'', B'', C'', m, o]
   preselect A l B C (mhead l' B' C') (pcons A'' l'' B'' C'' m o)
     --> m.
[A, B, C, l, o, m, A', l', B', C', A'', l'', B'', C'', m']
   preupdate A l B C (mhead l' B' C') (pcons A'' l'' B'' C'' m' o) m
     -->
   pcons A'' l'' B'' C'' m o.
```

from which the following constraints are inferred by Dedukti and used to type-check the right-hand side against the type of the left-hand side[1]:

- $A \equiv A''$,
- $l \equiv l' \equiv l''$,
- $B \equiv B' \equiv B''$,
- $C \equiv \texttt{tcons}\ l'\ B'\ C'$, and
- $C' \equiv C''$.

The inductive case corresponds to a (label, type) pair found in the tail of C and is defined similarly:

```
[A, l, B, C, p, o]
   preselect _ l B _ (mtail _ _ _ _ _ p) (pcons A _ _ C _ o)
     -->
   preselect A l B C p o.
[A, l, B, C, p, o, m, l', B', m']
   preupdate _ l B _ (mtail _ _ _ _ _ p) (pcons A l' B' C m' o) m
     -->
   pcons A l' B' C m' (preupdate A l B C p o m).
```

We can now define selection and update on objects by enforcing that the type A and the part C are identical. In the case of selection, moreover, we apply the returned method to the object itself:

```
def objselect : A : type -> l : label -> B : type -> mem l B A ->
                Obj A -> Obj B.
[A,l,B,p,a]
   objselect A l B p a
   --> Eval_meth A B (preselect A l B A p a) a.

def objupdate : A : type -> l : label -> B : type -> mem l B A ->
                Obj A -> Meth A B -> Obj A.
[A,l,B,p,a,m]
   objupdate A l B p a m
   --> preupdate A l B A p a m.
```

The Dedukti context containing all the declarations and rewrite rules that we have introduced so far in this chapter is noted Σ_ς. The underlying rewrite system is strongly normalizing and confluent:

[1] We denote Dedukti convertibility by $\equiv$ when the rewrite system is clear from context.

Theorem 9 (Strong Normalization for Σ_ς)**.** *For any Dedukti term t, if t is well-typed in context Σ_ς then t is strongly normalizing with respect to $\longrightarrow_{\beta\Sigma_\varsigma}$.*

Proof. All the functions defined in Σ_ς use single structural induction only. □

Theorem 10 (Confluence for Σ_ς)**.** *The relation $\longrightarrow_{\beta\Sigma_\varsigma}$ is confluent (on untyped Dedukti terms).*

Proof. This property has been checked by the confluence checker CSIˆHO. □

4.4.3 Translation Function for Terms

We can now translate well-typed ς-terms as well-typed Dedukti terms in Σ_ς. This translation actually maps a typing derivation of the simply-typed ς-calculus to a Dedukti term and it is defined by induction on the derivation. For simplicity, we omit the dependency to the typing derivation and present it as a function from ς-terms to Dedukti terms, this does not introduce ambiguity because the typing rules of Figure 4.2 are syntax-directed. The translation function $[\![\bullet]\!]$ is defined by:

$$
\begin{array}{lcl}
[\![x]\!] & := & x \\
[\![[l_i = \varsigma(x_i : A)a_i]_{l_1<\ldots<l_n}]\!] & := & \texttt{pcons}\ [\![A]\!]\ l_1\ [\![A_1]\!]\ [\![[l_i : A_i]_{1<i\leq n}]\!]\ [\![\varsigma(x_1 : A)a_1]\!](\ldots \\
 & & \quad (\texttt{pcons}\ [\![A]\!]\ l_n\ [\![A_n]\!]\ \texttt{tnil}\ [\![\varsigma(x_n : A)a_n]\!]\ (\texttt{pnil}\ [\![A]\!]))\ldots) \\
 & & \textit{when } A \textit{ is } [l_i : A_i]_{l_1<\ldots<l_n} \\
[\![a.l]\!] & := & \texttt{objselect}\ [\![A]\!]\ l\ [\![B]\!]\ p\ [\![a]\!] \\
 & & \textit{when } a : A,\ a.l : B \textit{ and } p \textit{ is the position of } (l : B) \textit{ in } A \\
[\![a.l \Leftarrow \varsigma(x : A)b]\!] & := & \texttt{objupdate}\ [\![A]\!]\ l\ [\![B]\!]\ p\ [\![\varsigma(x : A)b]\!]\ [\![a]\!] \\
 & & \textit{when } b : B \textit{ and } p \textit{ is the position of } (l, B) \textit{ in } A \\
[\![\varsigma(x : A)b]\!] & := & \texttt{Make\_meth}\ [\![A]\!]\ [\![B]\!]\ (x : \texttt{Obj}\ [\![A]\!]\ \texttt{=>}\ [\![b]\!]) \\
 & & \textit{when } b : B
\end{array}
$$

To translate labels, we extended the parser of Dedukti so that it could read strings delimited by double quotes. This parser extended with syntactic sugar is called Sukerujo. It is available from the following URL: `http://deducteam.gforge.inria.fr/sukerujo`. In Sukerujo, `string` is the primitive type of strings.

Example 9. *The encoding of simply-typed λ-calculus given in Section 4.1.4 can now be translated in Dedukti in the following set of definitions:*

```
[] label --> string.

def Arrow (A : type) (B : type) : type
  := tcons "arg" A (tcons "val" B tnil).

def p0 (A : type) (B : type) : mem "arg" A (Arrow A B)
  := mhead "arg" A (tcons "val" B tnil).

def p1 (A : type) (B : type) : mem "val" B (Arrow A B)
  := mtail "val" "arg" B A (tcons "val" B tnil) (mhead "val" B tnil).

def Lambda (A : type) (B : type) (b : Obj A -> Obj B)
      : Obj (Arrow A B)
  := pcons (Arrow A B) "arg" A (tcons "val" B tnil)
        (self : Obj (Arrow A B) =>
            objselect (Arrow A B) "arg" A (p0 A B) self)
        (pcons (Arrow A B) "val" B tnil
            (self : Obj (Arrow A B) =>
                b (objselect (Arrow A B) "arg" A (p0 A B) self))
            (pnil (Arrow A B))).

def App (A : type) (B : type) (f : Obj (Arrow A B)) (t : Obj A)
      : Obj B
  := objselect (Arrow A B) "val" B (p1 A B)
       (objupdate (Arrow A B) "arg" A (p0 A B) f
           (__ : Obj (Arrow A B) => t)).
```

The translation function for the encoding is defined by:

$$\begin{array}{lcl} [\![A \to B]\!] & = & \texttt{Arrow}\ [\![A]\!]\ [\![B]\!] \\ [\![\lambda(x:A)b]\!] & = & \texttt{Lambda}\ [\![A]\!]\ [\![B]\!]\ (\texttt{x} : \texttt{Obj}\ [\![A]\!]\ \texttt{=>}\ [\![b]\!]\ ) \\ [\![f(t)]\!] & = & \texttt{App}\ [\![A]\!]\ [\![B]\!]\ [\![f]\!]\ [\![t]\!] \end{array}$$

As it can already be seen on this very small example, the encoding is so verbose that the translated terms get quickly unreadable. We will see in Chapter 6 that there is room for much improvement in that matter.

4.4.4 Typing Preservation

Our encoding of ς-terms preserves typing:

Theorem 11 (Typing preservation for simply-typed ς-calculus)**.** *The translation of a typing derivation $\Delta \vdash a : A$ is a well-typed Dedukti term $[\![a]\!]$ of type $\texttt{Obj}\ [\![A]\!]$ in any context extending $[\![\Delta]\!]$ by the declarations of the labels occurring in $[\![A]\!]$.*

The translation function for contexts is defined in the expected manner:

- $[\![\emptyset]\!] := \Sigma_\varsigma$ and
- $[\![\Delta, x : A]\!] := [\![\Delta]\!], x : \texttt{Obj}\ [\![A]\!]$.

Proof. We proceed by induction on the typing derivation and case distinction on the last typing rule, there are four cases:

- Case $(TypeObj)$,

  A is of the form $[l_i : A_i]_{i=1\ldots n}$ and a is of the form $[l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n}$ and for all i, $\Delta, x_i : A \vdash a_i : A_i$. Without loss of generality, we assume the labels sorted. By induction hypothesis, for all i $[\![\Delta, x_i : A]\!] \vdash [\![a_i]\!] : \texttt{Obj}\ [\![A_i]\!]$ hence $[\![\Delta]\!] \vdash [\![\varsigma(x_i : A)a_i]\!] : \texttt{Meth}\ [\![A]\!]\ [\![A_i]\!]$. We show that $\texttt{pcons}[\![A]\!]l_k[\![A_k]\!][\![[l_i : A_i]_{k<i\leq n}]\!][\![\varsigma(x_k : A)a_k]\!](\ldots\texttt{pnil}[\![A]\!])$ has type $\texttt{Preobj}[\![A]\!][\![[l_i : A_i]_{k\leq i\leq n}]\!]$ in context $[\![\Delta]\!]$ for every k between 1 and n by a simple decreasing induction (that is, the base case is $k = n$ and hereditary consists in proving the case $k - 1$ assuming the case k holds). The case $k = 1$ gives us the expected result.

- Case $(TypeVar)$, $(TypeSelect)$ and $(TypeUpdate)$ are trivial.

□

The converse property does not hold: well-typed terms of type $\texttt{Obj}\ [\![A]\!]$ are not always translations of ς-terms because a term of type $\texttt{Obj}\ [\![A]\!]$ can contain subterms of type $\texttt{Obj}\ \beta$ where β is not the translation of a ς-type (it has duplicate or unsorted labels):

Example 10. *Let a be a term of type $\texttt{Obj}\ [\![A]\!]$ and b be a term of type $\texttt{Obj}\ \beta$, the term $\texttt{Eval\_meth}\ \beta\ [\![A]\!]\ (\texttt{Make\_meth}\ \beta\ [\![A]\!]\ (x : \texttt{Obj}\ \beta \texttt{=>} a))\ b$ has type $\texttt{Obj}\ [\![A]\!]$ but is the translation of no ς-term.*

Since Σ_ς terminates, the translation cannot preserve reduction so we cannot use it to perform proof by reflection of object-oriented programs.

4.5 Shallow Embedding

In order to recover reduction preservation (at the cost of termination), we identify `Meth A B` with `Obj A -> Obj B`:

```
[A,B] Meth A B --> (Obj A -> Obj B).
[f] Eval_meth _ _ f --> f.
[f] Make_meth _ _ f --> f.
```

To prove reduction preservation, we need a lemma for handling substitutions, this result is standard in HOAS [146].

Lemma 2 (Preservation of substitutions)**.** *Let x be a variable, A and B be ς-types, Δ and Δ' be ς-contexts and a and b be ς-terms such that $\Delta, x : B, \Delta' \vdash a : A$ and $\Delta \vdash b : B$, we have*

$$[\![a\{x\backslash b\}]\!] \equiv [\![a]\!]\{x\backslash[\![b]\!]\}$$

Proof. We proceed by induction on a.

- If a is the variable x, then $[\![a\{x\backslash b\}]\!] = [\![b]\!]$ and $[\![a]\!]\{x\backslash[\![b]\!]\} = [\![x]\!]\{x\backslash[\![b]\!]\} = x\{x\backslash[\![b]\!]\} = [\![b]\!]$.
- If a is another variable y, then $[\![a\{x\backslash b\}]\!] = [\![y\{x\backslash b\}]\!] = [\![y]\!] = y$ and $[\![a]\!]\{x\backslash[\![b]\!]\} = [\![y]\!]\{x\backslash[\![b]\!]\} = y\{x\backslash[\![b]\!]\} = y$
- If a is a ground object $[l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n}$, we assume without loss of generality that the labels are sorted. If one of the x_i is x, we can α-rename it thanks to the induction hypothesis on the corresponding a_i. The result then follows directly from induction.
- The two remaining cases (selection and update) are easy.

□

Theorem 12 (Reduction preservation)**.** *Let a and a' be two ς-terms of type A such that $a \rightsquigarrow a'$, we have $[\![a]\!] \longrightarrow^+ [\![a']\!]$.*

Proof. There are two cases, one per rule defining $\rightsquigarrow$:

- We consider a ς-type $A = [l_i : A_i]_{i=1\ldots n}$ and an object $a = [l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n}$ of type A, we have to prove that $[\![a.l_j]\!] \longrightarrow^+ [\![a_j\{x_j\backslash a\}]\!]$.

  By definition, $[\![a.l_j]\!] = \texttt{objselect}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ p\ [\![a]\!]$ where p is the position of $(l_j : A_j)$ in A.

$$\begin{array}{lcl} [\![a.l_j]\!] & = & \texttt{objselect}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ p\ [\![a]\!] \\ & \longrightarrow & \texttt{Eval\_meth}\ [\![A]\!]\ [\![A_j]\!]\ (\texttt{preselect}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ [\![A]\!]\ p\ [\![a]\!])\ [\![a]\!] \\ & \longrightarrow & \texttt{preselect}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ [\![A]\!]\ p\ [\![a]\!]\ [\![a]\!] \end{array}$$

  To conclude this case, we show by decreasing induction on $k \leq j$ that

$$\texttt{preselect}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ [\![[l_i : A_i]_{i=k\ldots n}]\!]\ p_k\ o_k\ [\![a]\!] \longrightarrow^+ [\![a_j\{x_j\backslash a\}]\!]$$

  where p_k is the position of $(l_j : A_j)$ in $[l_i : A_i]_{i=k\ldots n}$ and
  $o_k = \texttt{pcons}[\![A]\!]l_k[\![A_k]\!][\![[l_i : A_i]_{k<i\leq n}]\!][\![\varsigma(x_k : A)a_k]\!](\ldots\texttt{pnil}[\![A]\!])$:

  - Base case: $k = j$ and $p_k = \texttt{mhead}\ l_j\ [\![A_j]\!]\ [\![[l_i : A_i]_{j<i\leq n}]\!]$.

$$\begin{array}{l} \texttt{preselect}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ [\![[l_i : A_i]_{i=j\ldots n}]\!]\ p_j\ \alpha_j\ [\![a]\!] \\ \quad \longrightarrow [\![\varsigma(x_j : A)a_j]\!]\ [\![a]\!] \\ \quad \longrightarrow \texttt{Make\_meth}\ [\![A]\!]\ [\![A_j]\!]\ (x_j : [\![A]\!] \texttt{=>} [\![a_j]\!])\ [\![a]\!] \\ \quad \longrightarrow (x_j : [\![A]\!] \texttt{=>} [\![a_j]\!])\ [\![a]\!] \\ \quad \longrightarrow_\beta [\![a_j]\!]\{x_j\backslash[\![a]\!]\} \\ \quad \equiv [\![a_j\{x_j\backslash a\}]\!] \qquad \text{by Lemma 2} \end{array}$$

  - Inductive case: $k < j$ and $p_k = \texttt{mtail}\ l_j\ l_k\ [\![A_j]\!]\ [\![A_k]\!]\ [\![[l_i : A_i]_{k<i\leq n}]\!]\ p_{k+1}$.

$$\begin{array}{l} \texttt{preselect}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ [\![[l_i : A_i]_{i=k\ldots n}]\!]\ p_k\ \alpha_k\ [\![a]\!] \\ \quad \longrightarrow \texttt{preselect}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ [\![[l_i : A_i]_{k<i\leq n}]\!]\ p_{k+1}\ \alpha_{k+1}\ [\![a]\!] \\ \quad \longrightarrow^+ [\![a_j\{x_j\backslash a\}]\!] \qquad \text{by induction hypothesis} \end{array}$$

- We consider a variable x, a ς-type $A = [l_i : A_i]_{i=1\ldots n}$, an object $a = [l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n}$ of type A, and a term b of type A_j. We have to prove that $[\![a.l_j \Leftarrow \varsigma(x : A)b]\!] \longrightarrow^+ [\![[l_j = \varsigma(x : A)b, l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n, i\neq j}]\!]$.

  By definition, $[\![a.l_j \Leftarrow \varsigma(x : A)b]\!] = \texttt{objupdate}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ p\ [\![\varsigma(x : A)b]\!]\ [\![a]\!]$ where p is the position of $(l_j : A_j)$ in A.

$$\begin{array}{lcl} [\![a.l_j \Leftarrow \varsigma(x : A)b]\!] & = & \texttt{objupdate}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ p\ [\![\varsigma(x : A)b]\!]\ [\![a]\!] \\ & \longrightarrow & \texttt{preupdate}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ [\![A]\!]\ p\ [\![\varsigma(x : A)b]\!]\ [\![a]\!] \end{array}$$

To conclude this case, we show by decreasing induction on $k \leq j$ that

$$\texttt{preupdate}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ [\![[l_i : A_i]_{i=k\ldots n}]\!]\ p_k\ [\![\varsigma(x : A)b]\!]\ \alpha_k \longrightarrow^+ \alpha'_k$$

where p_k is the position of $(l_j : A_j)$ in $[l_i : A_i]_{i=k\ldots n}$,
$\alpha_k = \texttt{pcons}[\![A]\!]l_k[\![A_k]\!][\![[l_i : A_i]_{k<i\leq n}]\!][\![\varsigma(x_k : A)a_k]\!](\ldots\texttt{pnil}[\![A]\!])$, and α'_k is defined as α_k but replacing $[\![\varsigma(x_j : A)a_j]\!]$ by $[\![\varsigma(x : A)b]\!]$:

- Base case: $k = j$ and $p_k = \texttt{mhead}\ l_j\ [\![A_j]\!]\ [\![[l_i : A_i]_{j<i\leq n}]\!]$.

$$\texttt{preupdate}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ [\![[l_i : A_i]_{i=j\ldots n}]\!]\ p_k\ [\![\varsigma(x : A)b]\!]\ \alpha_j \longrightarrow \alpha'_j$$

- Inductive case: $k < j$ and $p_k = \texttt{mtail}\ l_j\ l_k\ [\![A_j]\!]\ [\![A_k]\!]\ [\![[l_i : A_i]_{k<i\leq n}]\!]\ p_{k+1}$.

$$\begin{aligned}
&\texttt{preupdate}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ [\![[l_i : A_i]_{i=k\ldots n}]\!]\ p_k\ [\![\varsigma(x : A)b]\!]\ \alpha_j \\
&\quad\longrightarrow \texttt{pcons}[\![A]\!]l_k[\![A_k]\!][\![[l_i : A_i]_{k<i\leq n}]\!][\![\varsigma(x_k : A)a_k]\!] \\
&\qquad\qquad (\texttt{preupdate}\ [\![A]\!]\ l_j\ [\![A_j]\!]\ [\![[l_i : A_i]_{k<i\leq n}]\!]\ p_{k+1}\ [\![\varsigma(x : A)b]\!]\ \alpha_j) \\
&\quad\longrightarrow^+ \texttt{pcons}[\![A]\!]l_k[\![A_k]\!][\![[l_i : A_i]_{k<i\leq n}]\!][\![\varsigma(x_k : A)a_k]\!]\alpha'_{k+1} \qquad \text{by induction hypothesis} \\
&\quad\equiv \alpha'_k
\end{aligned}$$

□

Chapter 5

Object Subtyping in Dedukti

Contrary to type systems for functional languages where it plays a minor role, subtyping is omnipresent in typed object-oriented languages. In ς-calculi, subtyping is used for implementing private methods (if an object a of type A is used as an object of type B, then all methods of A not present in B are private) and inheritance.

In Section 5.1, we present the extension of the simply-typed ς-calculus to subtyping proposed by Abadi and Cardeli. In Section 5.2, we give an example of use of subtyping to achieve privacy.

Our translation of the ς-calculus to Dedukti needs some adjustments to take subtyping into account. In Section 5.3 we translate the subtyping relation to Dedukti and in Section 5.4, we introduce explicit coercions on the Dedukti side to translate all the ς-terms that are well-typed in the extension of the ς-calculus with subtyping.

In Sections 5.5 and 5.6, we discuss two properties that our translation might enjoy in addition to the ones proved in the previous chapter. The first property is conservativity, it merely states that the translation is surjective and it is the topic of Section 5.5. The second property is canonicity, it merely states that the translation is injective and it is the topic of Section 5.6.

5.1 Simply-Typed ς-Calculus with Subtyping

The subtyping relation $<:$ is defined in the ς-calculus by the following inference rules

$$\frac{}{[l_i : A_i]_{i \in 1 \ldots n+m} <: [l_i : A_i]_{i \in 1 \ldots n}} \text{(subtype)} \qquad \frac{}{A <: A} \text{(refl)}$$

$$\frac{A <: B \qquad B <: C}{A <: C} \text{(trans)}$$

Since the order of labels is irrelevant, the (subtype) rule actually states that A is a subtype of B whenever every label of B is also in A, with the same type.

A term a of type A can be used with type B when $A <: B$ thanks to the subsumption typing rule:

$$\frac{\Delta \vdash a : A \qquad A <: B}{\Delta \vdash a : B} \text{(subsume)}$$

5.2 Example

The expressivity of the ς-calculus with subtyping can be illustrated by the following example from Abadi and Cardelli [2] assuming that we have a type Num for numbers and that the simply-typed λ-calculus has been encoded:

$$\begin{array}{lll}
RomCell & := & [\ get : Num\ ] \\
PromCell & := & [\ get : Num, set : Num \to RomCell\ ] \\
PrivateCell & := & [\ get : Num, contents : Num, set : Num \to RomCell\ ] \\
myCell : PromCell & := & [\ get = \varsigma(x : PrivateCell)x.contents, \\
 & & \quad contents = \varsigma(x : PrivateCell)0, \\
 & & \quad set = \varsigma(x : PrivateCell)\lambda(n : Num)x.contents \Leftarrow n\ ]
\end{array}$$

RomCell is the type of read-only memory cells; the only action that we can perform on a *RomCell* is to read it (*get* method).

A *PromCell* is a memory cell which can be written once (*set* method), we can either read it now or write it and get a *RomCell*.

PrivateCell is a type used for implementation; it extends *PromCell* with a *contents* field which should not be seen from the outside.

The object *myCell* implemented as an object of type *PrivateCell* can be given the type *PromCell* thanks to subsumption.

5.3 Translation of the Subtyping Relation

In Dedukti, the subtyping relation $A <: B$ can be simply defined by inclusion

```
def subtype (A : type) (B : type) :=
  l : label -> C : type -> mem l C B -> mem l C A.
```

Alternatively, we can also define it by induction on B:

```
subtype' : type -> type -> Type.
subtype_nil : A : type -> subtype' A tnil.
subtype_cons : A : type -> l : label -> B1 : type -> B2 : type ->
               subtype' A B2 -> mem l B1 A ->
               subtype' A (tcons l B1 B2).
```

These two definitions are equivalent, the first one is better for proving reflexivity and transitivity, the second one for proving decidability.

Lemma 3. *For all types* `A` *and* `B`*, the type* `subtype' A B -> subtype A B` *is inhabited.*

Proof. By induction on `B`. □

Lemma 4. *The relation* `subtype'` *is decidable.*

Proof. Trivial from decidability of membership. □

Lemma 5. *The relation* `subtype` *is reflexive and transitive.*

Proof. The proof terms are identity and composition respectively. □

5.4 Explicit Coercions

Considering two ς-types A and B such that $A <: B$, we do not achieve a correct encoding of subtyping by either rewriting $[\![A]\!]$ to $[\![B]\!]$ or $[\![B]\!]$ to $[\![A]\!]$ because that would make $[\![A]\!]$ and $[\![B]\!]$ convertible. We can however annotate terms by explicit coercions.

```
def coerce : A : type -> B : type -> subtype A B -> Obj A -> Obj B.
```

Unfortunately, these coercions get in the way of evaluation because `preselect` and `preupdate` only reduce when applied to `precons`. For example, a Dedukti expression of the form

`objselect _ _ _ _ (coerce _ _ _ _)` is stuck whereas the semantics of the ς-calculus requires to reduce it as if no coercion were present.

The symbol `coerce` can not be fully defined either but we can treat it as a smart constructor by adding the following rewrite rules:

```
def select : A : type -> l : label -> B : type -> mem l B A ->
             Obj A -> Obj B.
def update : A : type -> l : label -> B : type -> mem l B A ->
             Obj A -> Meth A B -> Obj A.

[l,B,p,l',B',C',m,o]
    select (tcons _ _ _) l B p (pcons _ l' B' C' m o)
      -->
    objselect (tcons l' B' C') l B p
              (pcons (tcons l' B' C') l' B' C' m o).

[l,B,p,l',B',C',m,o,m']
    update (tcons _ _ _) l B p (pcons _ l' B' C' m o) m'
      -->
    objupdate (tcons l' B' C') l B p
              (pcons (tcons l' B' C') l' B' C' m o) m'.

[A,B,C,l,p,st,a]
   select _ l C p (coerce A B st a)
     --> select A l C (st l C p) a.

[A,B,C,l,p,st,a,m]
   update _ l C p (coerce A B st a) m
     --> coerce A B st
            (update A l C (st l C p) a
               (self : Obj A => m (coerce A B st self))).
```

The functions `objselect` and `objupdate` are only defined on objects of the form `pcons _ _ _ _ _ _`, the functions `select` and `update` extend them by treating the case of the smart constructor `coerce`.

The translation of typing derivations given in Section 4.4.3 is adapted to subtyping by replacing `objselect` and `objupdate` by `select` and `update` respectively and the case of the typing rule (subsume) is given by `coerce`:

$$
\begin{array}{lcl}
[\![a.l]\!] & := & \texttt{select}\ [\![A]\!]\ l\ [\![B]\!]\ p\ [\![a]\!] \\
 & & \textit{when } a : A,\ a.l : B \textit{ and } p \textit{ is the position of } (l : B) \textit{ in } A \\
[\![a.l \Leftarrow \varsigma(x : A)b]\!] & := & \texttt{update}\ [\![A]\!]\ l\ [\![B]\!]\ p\ [\![\varsigma(x : A)b]\!]\ [\![a]\!] \\
 & & \textit{when } b : B \textit{ and } p \textit{ is the position of } (l, B) \textit{ in } A \\
[\![a]\!] & := & \texttt{coerce}\ [\![A]\!]\ [\![B]\!]\ st\ [\![a]\!] \\
 & & \textit{when the derivation ends with a subsume rule} \\
 & & \textit{between types } A \textit{ and } B
\end{array}
$$

5.5 Reverse Translation

An important property of embeddings in the λΠ-calculus modulo is their conservativity. The conservativity property is a weak converse of typing preservation: it states that the translation of empty types of the source language of the translation (here the ς-calculus) are empty types in the target language (here the λΠ-calculus modulo). In the light of the Curry-Howard correspondence, conservativity means that the translation does not introduce inconsistencies so reasoning in the target logic is as safe as in the source one.

The main conservativity result for embeddings in the λΠ-calculus modulo is a proof by Assaf [12, 11] that the embedding of Functional Pure Type Systems in the λΠ-calculus modulo defined by Cousineau and Dowek [55] is conservative.

Conservativity is too weak to be of interest in our context because all the ς-types are inhabited. However, the technique of Assaf conservativity proof is of interest in our setting.

Assaf devises partial functions φ and ψ translating back respectively the terms and the types of the target language to terms and types in (a conservative extension of) the source language. These functions are inverses of the translation functions for terms and types in the following sense: their domains contain the images of the translations for terms and types and $\varphi([\![t]\!])$ is equivalent to t and $\psi([\![A]\!])$ is equivalent to A. In our case, the notions of equivalence in the source language are given by reordering of labels for types and by the congruence induced by the operational semantics for terms.

Assaf then shows that, even for non-terminating encodings, all terms inhabiting translated types reduce to terms in the domain of the reverse translations.

Following Assaf, we would like to obtain the following strong conservativity result: Dedukti terms whose types are of the form `Obj` $[\![A]\!]$ reduce to translations of ς-terms.

As we have seen in Section 4.4.4, this strong conservativity statement fails in the case of our terminating encoding of simply-typed ς-calculus but the counterexample that we gave is not normal in the shallow system and it indeed reduces to $a : \texttt{Obj}\ [\![A]\!]$ which is not a counterexample anymore.

The reverse translations are defined in the natural way:

$$\begin{array}{lcl}
\varphi(x) & := & x \\
\varphi(\texttt{pnil}\ A) & := & [\,] \\
\varphi(\texttt{pcons}\ A\ l\ B\ C\ m\ o) & := & [l = \chi(m), \varphi(o)] \\
\varphi(\texttt{select}\ A\ l\ B\ p\ a) & := & \varphi(a).l \\
\varphi(\texttt{update}\ A\ l\ B\ p\ m\ a) & := & \varphi(a).l \Leftarrow \chi(m) \\
\varphi(\texttt{coerce}\ A\ B\ s\ a) & := & \varphi(a) \\
\chi(x : \texttt{Obj}\ A \Rightarrow o) & := & \varsigma(x : \psi(A))\varphi(o) \\
\psi(\texttt{tnil}) & := & [\,] \\
\psi(\texttt{tcons}\ \ l\ \ A\ B) & := & [l : A, \psi(B)]
\end{array}$$

Note that ψ is only defined for duplicate-free lists, χ is only defined for λ-abstractions of type $\texttt{Obj}\ A \to \texttt{Obj}\ B$ such that ψ is defined on A and φ is undefined on a lot of terms such as $(x : \texttt{Obj}\ A \Rightarrow b)\ a$.

The reverse translation ψ is extended to contexts by

$$\begin{array}{lcl}
\psi(\Sigma'_\varsigma) & := & \emptyset \\
\psi(\Gamma, x : \texttt{Obj}\ A) & := & \psi(\Gamma), x : \psi(A)
\end{array}$$

The following two results are easy to show:

Lemma 6 (reverse translations)**.** *For any ς-type A, $\psi([\![A]\!]) \equiv A$ and for any well-typed ς-term a, $\varphi([\![a]\!]) \equiv a$ where in both cases, $\equiv$ denotes syntactic equality modulo reordering of labels.*

Proof. By induction on the structure of the term, all cases are obvious. □

Lemma 7 (conservation of typing)**.** *If t is a term of the $\lambda\Pi$-calculus modulo of type $\texttt{Obj}\ A$ in context Γ such that ψ is defined on A and Γ and φ is defined on t then $\psi(\Gamma) \vdash \varphi(t) : \psi(A)$ in the ς-calculus with subtyping.*

Proof. By induction on the structure of the term t, each case corresponds to a different typing rule in the ς-calculus with subtyping. □

We conjecture that Assaf's proof can be adapted to obtain the strong conservativity lemma:

Conjecture 1 (Strong conservativity lemma)**.** *If t is a term of the $\lambda\Pi$-calculus modulo of type $\mathtt{Obj}\ A$ in context Γ such that ψ is defined on A and Γ, then t reduces to a term t' on which φ is defined.*

Contrary to the previous lemmata, this conjecture is far from trivial because the ς-calculus does not terminate so it is not sufficient to look at the normal forms.

A simple corollary of this conjecture can then be stated in terms of the direct translation functions only:

Corollary 1. *If t is a term of the $\lambda\Pi$-calculus modulo of type $\mathtt{Obj}\ [\![A]\!]$ in the context $[\![\Delta]\!]$ then t reduces to some $[\![a]\!]$ such that $\Delta \vdash a : A$ in the ς-calculus with subtyping.*

5.6 Canonicity

Thanks to transitivity of subtyping, we can optionally add the following rewrite rule to ensure that the size of annotations does not grow faster than the term:

```
def trans (A : type) (B : type) (C : type)
          (stAB : subtype A B)
          (stBC : subtype B C) : subtype A C :=
          l : label =>
          D : type =>
          p : mem l D C =>
          stAB l D (stBC l D p).

[A,B,C,stAB,stBC,a]
   coerce _ C stBC (coerce A B stAB a)
     -->
   coerce A C
     (trans A B C stAB stBC)
     a.
```

This is however hard to prove confluent. For example, the confluence of this rule alone relies on the associativity of `trans` so it is sensible to our definition of subtyping. The transitivity of the `subtype'` relation for example is also provable but not in an associative way. Moreover, this is a kind of rewrite rule that makes CSI^HO slow: after hours, CSI^HO finally replies `MAYBE`.

An argument in favor of this rule is that it eliminates a source of non-canonicity, the other source is dummy coercion from a type to itself which is eliminated by the following non-linear rule:

```
[A,a] coerce A A _ a --> a.
```

When both rules are present, the system is canonical:

Theorem 13 (Canonicity)**.** *Let a and a' be two well-typed terms of type* `Obj` $[\![A]\!]$ *such that* $\varphi(a) \equiv \varphi(a')$*, then* $a \equiv a'$*.*

Proof. We first remark that the rewrite system consisting of these two rewrite rules is strongly normalizing because the number of coercions decreases by one at each application of a rewrite rule.

Without loss of generality, we assume a and a' in normal form with respect to these rules.

Since φ is defined on a and a', they both have a shape among x, `pnil` A, `pcons` $A\ l\ B\ C\ m\ o$, `select` $A\ l\ B\ p\ o$, `update` $A\ l\ B\ p\ m\ o$, and `coerce` $A\ B\ s\ o$. All these shapes but the last are mapped to different syntactic constructs of the ς-calculus so the only interesting case occur when at least one of a and a' is a coercion. The cases where one of them is a coercion and the other one is a selection or an update are treated using the rules of Section 5.4, the cases where one is a coercion and the other one is a variable or an object violate normalization with respect to the rewrite rule `[A,a] coerce A A _ a --> a.`. The most interesting case is when both a and a' are coercions.

The term a is `coerce B A stBA b` and the term a' is `coerce B' A stB'A b'`. Our assumption is $\varphi(b) \equiv \varphi(b')$ but the types `B` and `B'` are not *a priori* related. The ς-calculus with subtyping admits minimal typings [1]. If we denote by `C` the translation of the minimal type of the ς-term $c := \varphi(b)$, then `C` is a subtype of both `B` and `B'` and both `B` and `B'` are strict subtypes of `A`.

Since `C` is a subtype of `B`, we can translate c at type `B` as $[\![c]\!] :=$ `coerce C B stCB c` and apply the induction hypothesis to get $b \equiv$ `coerce C B stCB c` hence $a \equiv$ `coerce B A stBA (coerce C B stCB c)`.

By symmetry, we get $a' \equiv$ `coerce B' A stB'A (coerce C B' stCB' c)`. Using the

rewrite rule for composition of coercions, we obtain $a \equiv \texttt{coerce C A stCA c} \equiv a'$. □

The translations that we have presented in this chapter and the previous one simplify the translations that we proposed in [42]. In [42], the inductive functions on the Dedukti and Coq sides such as `select` and `update` were defined by induction over positions of labels in types whereas here we consider only positions of (label, type) pairs in types. Contrary to [42], we do not need to put positions inside preobjects, which simplifies the definitions of selection and update: when positions are packed in preobjects, we need to ensure that the positions we find in the preobjects are the same as the one we used to access them. In Dedukti, this was done using non-linearity (which stops us from automatic verification of confluence) and in Coq it was done by adding an extra parameter to `pcons` asserting that the added label was not already present (hence we can prove that types of objects are duplicate-free so positions are unique). Another simplification in the presentation of our translation with respect to [42] concerns the role of minimal typing. In [42], two versions of the translation function for terms are mutually defined, the first one is annotated with the ς-type of the term, the second one is not annotated and translates the term according to its minimal type. The translations defined in [42] are really translations of well-typed **terms** whereas this chapter presents a translation of typing **derivations** so we end-up with several possible translations of the same well-typed term that can be related thanks to the canonicity result.

Chapter 6

The Implementation Sigmaid

We have implemented the translation functions presented in the previous chapters as a translator named Sigmaid (SIGMA-calculus In Dedukti) from ς-calculus to Coq and Dedukti. Our code is available at the following URL: `http://sigmaid.gforge.inria.fr`.

This chapter is devoted to the improvements that have been integrated in this implementation. These improvements in the translation do not affect the semantics of the translated terms but only their syntax.

The first improvement, described in Section 6.1, deals with the representation of concrete objects. We define additional Dedukti functions to shorten the translation of concrete objects and make it more readable. This alternative definition of the translation of concrete objects relies on the subtyping relation that we introduced in Chapter 5, this is the reason why we have chosen not to give it in Chapter 4.

The second improvement, described in Section 6.2, is the removal from the translation of the position arguments needed for the selection and update functions. Sigmaid does not need to justify precisely at which position the labels occur in types but relies on decidability lemmata.

In Section 6.3, we evaluate the time efficiency of Sigmaid once these two improvements are applied. The result heavily depends on the chosen representation for the labels. In Section 6.4, we make good use of Dedukti at the meta-level to perform label operations independently of the representation and speed-up Sigmaid.

6.1 Initiating Objects

All the types of the simply-typed ς-calculus are inhabited. If $A = [l_i : A_i]_{i=1\ldots n}$ is a ς-type, then we can define the following object $\mathbf{init}(A)$ of type A:

$$\mathbf{init}(A) := [l_i = \varsigma(x : A)x.l_i]_{i=1\ldots n}$$

All the methods of the object $\mathbf{init}(A)$ are *loop methods*, that is, methods directly calling themselves. We have already encountered loop methods in Section 4.1.4. They are commonly used in the ς-calculus as placeholders for methods whose body is irrelevant because the method is going to be updated.

The object $\mathbf{init}(A)$ can be used to avoid writing concrete objects. Let $a = [l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n}$ be a concrete object of type A, the object a is semantically equivalent to the following ς-term:

$$(\ldots((\mathbf{init}(A)).l_1 \Leftarrow \varsigma(x_1 : A)a_1)\ldots).l_n \Leftarrow \varsigma(x_n : A)a_n$$

Moreover, the order in which the methods are updated is not relevant. If σ is a permutation of $\{1, \ldots, n\}$, then both $(\ldots((\mathbf{init}(A)).l_1 \Leftarrow \varsigma(x_1 : A)a_1)\ldots).l_n \Leftarrow \varsigma(x_n : A)a_n$ and $(\ldots((\mathbf{init}(A)).l_{\sigma(1)} \Leftarrow \varsigma(x_{\sigma(1)} : A)a_{\sigma(1)})\ldots).l_{\sigma(n)} \Leftarrow \varsigma(x_{\sigma(n)} : A)a_{\sigma(n)}$ evaluate to a.

In this section, We use this idea to improve our translation of concrete objects in such a way that:

- Sigmaid does not need to manipulate preobjects. They are present in the Dedukti signature but all the objects printed by Sigmaid are full objects.
- Sigmaid is not asked to sort the labels of concrete objects anymore. It can print the methods in the same order than in the input file, which is more readable.
- Partially defined objects are accepted, the missing methods are defined as loop methods by default.

The new translation function is defined by:

```
(; Loop method ;)
def loop : A : type -> B : type -> l : label -> mem l B A ->
           Meth A B.
[A,B,l,H]
   loop A B l H
     -->
   Make_meth A B (self : Obj A => objselect A l B H self).

def preinit : A : type -> B : type -> subtype A B ->
              Preobj A B.
[A] preinit A tnil _ --> pnil A
[A,l,B,C,st]
    preinit A (tcons l B C) st
      -->
    pcons A l B C
          (loop A B l (st l B (mhead l B C)))
          (preinit A C (l' : label =>
                        B' : type =>
                        p : mem l' B' C =>
                        st l' B' (mtail l' l B' B C p))).

def init (A : type) : Obj A :=
  preinit A A (l : label => B : type => p : mem l B A => p).
```

Figure 6.1: Definition of the `init` function in Dedukti

$$[\![[l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n}]\!] := \texttt{update}\ [\![A]\!]\ [\![A_1]\!]\ l_1\ p_1\ [\![\varsigma(x_1 : A)a_1]\!]\ (\ldots \texttt{update}\ [\![A]\!]\ [\![A_n]\!]\ l_n\ p_n\ [\![\varsigma(x_n : A)a_n]\!]\ (\texttt{init}\ [\![A]\!]))\ \ldots)$$
when A *is* $[l_i : A_i]_{i=1\ldots n}$ *and* p_i *is the position of* $(l_i : A_i)$ *in* A

other cases are unchanged.

In this definition, `init` is a Dedukti function of type `A : type -> Obj A` playing the role of **init** on the Dedukti side. The definition of `init` is given in Figure 6.1.

The translation of the encoding of λ-calculus is now simplified to the following set of definitions:

```
[] label --> string.

def Arrow (A : type) (B : type) : type
  := tcons "arg" A (tcons "val" B tnil).

def p0 (A : type) (B : type) : mem "arg" A (Arrow A B)
  := mhead "arg" A (tcons "val" B tnil).
```

```
def p1 (A : type) (B : type) : mem "val" B (Arrow A B)
  := mtail "val" "arg" B A (tcons "val" B tnil) (mhead "val" B tnil).

def Lambda (A : type) (B : type) (b : Obj A -> Obj B)
       : Obj (Arrow A B)
  := update (Arrow A B) "val" B (p1 A B)
       (init (Arrow A B))
       (self : Obj (Arrow A B) =>
         b (select (Arrow A B) "arg" A (p0 A B) self)).

def App (A : type) (B : type) (f : Obj (Arrow A B)) (a : Obj A)
       : Obj B
  := select (Arrow A B) "val" B (p1 A B)
       (update (Arrow A B) "arg" A (p0 A B) f
               (__ : Obj (Arrow A B) => a)).
```

This is still heavy but already significantly more readable than what we got in Section 4.4.3. Only the definition of `Lambda` has changed. The simplification of the definitions of the positions `p1` and `p2` are investigated in the next subsection.

6.2 Decidability

To simplify the translator further, we can avoid providing arguments when they can be computed in the target systems. Type equality, membership and subtyping can all be proved decidable in the target systems. Formally, a *decidable relation* is a function returning a boolean. Assuming decidability of label equality, we can define the boolean versions of type equality `bteq`, membership `bmem`, and subtyping `bst`:

```
bool : Type.
true : bool.
false : bool.
def and : bool -> bool -> bool.
def or : bool -> bool -> bool.
[b] and true b --> b
[] and false _ --> false.
[] or true _ --> true
[b] or false b --> b.

def label_beq : label -> label -> bool.

def bteq : type -> type -> bool.
[] bteq tnil tnil --> true
[] bteq (tcons _ _ _) tnil --> false
[] bteq tnil (tcons _ _ _) --> false
```

```
[l, A, B, l', A', B']
    bteq (tcons l A B) (tcons l' A' B')
      -->
    and (and (label_beq l l') (bteq A A')) (bteq B B').

def bmem : label -> type -> type -> bool.
[] bmem _ _ tnil --> false
[l, A, l', A', B]
    bmem l A (tcons l' A' B)
      -->
    or (and (label_beq l l') (bteq A A')) (bmem l A B).

def bst : type -> type -> bool.
[] bst _ tnil --> true
[A, l, A', B] bst A (tcons l A' B) --> and (bmem l A' A) (bst A B).
```

Decidability of equality on strings is proved in Coq standard library. On the Dedukti side it requires some work but nothing very deep so we omit the definition of the symbol `label_beq` here.

With some efforts, we can prove that these decidable relations reflect equality, `mem`, and `subtype` respectively: the decidable relations return true if and only if the relations are inhabited. Actually, we only need the "only if" direction.

```
Istrue : bool -> Type.
Istrue_true : Istrue true.

def bmem_reflects_mem :
 l : label ->
 A : type ->
 B : type ->
 Istrue (bmem l A B) ->
 mem l A B.

def bst_reflects_subtype :
 A : type ->
 B : type ->
 Istrue (bst A B) ->
 subtype A B.
```

We do not show the proofs because they are quite long and not very interesting. The point of using the reflection technique is that we can compute with the proofs. For example, `bmem_reflects_mem "arg" A (Arrow A B)` normalizes to `mhead "arg" A (tcons "val" B tnil)` so the translator does not need to compute the positions and subtyping proofs itself, computing can be discharged to the target systems. The definitions of the positions `p1` and `p2`

in our running example of the translation of the λ-calculus become

```
def p0 (A : type) (B : type) : mem "arg" A (Arrow A B)
  := bmem_reflects_mem "arg" A (Arrow A B) Istrue_true.

def p1 (A : type) (B : type) : mem "val" B (Arrow A B)
  := bmem_reflects_mem "val" B (Arrow A B) Istrue_true.
```

6.3 Efficiency

To our knowledge, the only available implementation of a ς-calculus is Pericas-Geertsen's **Sigma** interpreter (`http://types.bu.edu/seminar-ool-mini/sigma.html`). Comparing Sigmaid with **Sigma** is not easy because **Sigma** features recursive types (which we did not consider) but not subtyping.

We tested Sigmaid on a few examples taken from [2]:

- The encoding of booleans: we check that `if true then t else e` is convertible to `t` and `if false then t else e` is convertible to `e`.
- The encoding of λ-calculus: we check that the β-redex $(\lambda(\mathtt{x} : \mathtt{A})\ \mathtt{f}\ \mathtt{x})\ \mathtt{a}$ is convertible to `f a`.
- We define types for points, colors, and colored points. We define an explicit cast operation $\mathtt{point\_of\_colorpoint} : \mathtt{ColorPoint} \to \mathtt{Point}$ by $\lambda(\mathtt{p} : \mathtt{ColorPoint})\ \mathtt{p}$ and check that we can select fields through it: $(\mathtt{point\_of\_colorpoint}[\mathtt{x} = 42; \mathtt{y} = 0; \mathtt{color} = \mathtt{red}]).\mathtt{x}$ is convertible to `42`.
- We can also write the example of memory cells of Section 5.2. We check that `myCell.get` is `0` and `myCell.set(42).get` is `42`.
- Finally, we check that adding a dummy private field does not affect late binding: let `XY` and `XYZ` be the types $[\mathtt{x} : \mathtt{Nat}, \mathtt{y} : \mathtt{Nat}]$ and $[\mathtt{x} : \mathtt{Nat}, \mathtt{y} : \mathtt{Nat}, \mathtt{z} : \mathtt{Nat}]$ respectively, if `a` is the object of type `XY` defined by $[\mathtt{x} = \varsigma(\mathtt{s} : \mathtt{XYZ})\ \mathtt{s.y}, \mathtt{y} = 0, \mathtt{z} = 0]$ then $(\mathtt{a.y} := 42).\mathtt{x}$ is `42`.

All these examples are included in the file `test.sigma` distributed with Sigmaid.

Sigmaid translates these examples almost instantaneously, Dedukti checks them in 38 seconds and Coq in 1.85 seconds. The huge difference in timing comes from the fact that characters are represented on the Dedukti side using a unary representation of natural numbers whereas Coq characters use a binary representation which is much more efficient.

6.4 Optimization at the Meta-Level

As we have seen, Sigmaid does not compute positions and subtyping proofs but relies on decidability lemmata instead. This simplifies Sigmaid but comes at a price on the Dedukti and Coq side:

- The decidability proofs are not very hard but they are a bit tedious and decidability of label equality is very specific to the implementation of labels.
- The type-checking time is dominated by these decidability checks so it also becomes very dependent on the implementation choices.

Instead of writing in Dedukti a **certified** membership and subtyping checker, we can solve these problems by writing a **certifying** checker in Meta-Dedukti. We use a non-linear and non-confluent rewrite system similar to the one of Section 3.4.5 for deciding membership, and subtyping:

```
def decide_mem : l : label -> A : type -> B : type -> mem l A B.
[l, A, B] decide_mem l A (tcons l A B) --> mhead l A B
[l, A, l', A', B]
   decide_mem l A (tcons l' A' B)
     -->
   mtail l l' A A' B (decide_mem l A B).

def decide_st' : A : type -> B : type -> subtype' A B.
[A] decide_st' A tnil --> subtype_nil A
[A, l, B, C]
   decide_st' A (tcons l B C)
     -->
   subtype_cons A l B C (decide_st' A C) (decide_mem l B A).
```

The type of `decide_mem` is a blatant lie but sometimes lying is very convenient!

We should not trust Dedukti when it uses this rewrite system to type-check an object-oriented program but we can ask Meta-Dedukti to normalize the Dedukti files produced

by Sigmaid. Meta-Dedukti is allowed to use this unsafe system but the Dedukti files produced by Meta-Dedukti are to be checked in a safe, confluent rewrite system which does not include `decide_mem` and `decide_st'`.

The computation which is done at the Meta-Level is purely symbolic, it does not depend on the chosen representation of labels so it is much more efficient. On the same set of examples, the cumulative time for Sigmaid, Meta-Dedukti, and Dedukti is less than 0.3 seconds.

Conclusion of Part II

We have translated the simply-typed ς-calculus to both Coq and Dedukti in Chapter 4. In the case of the Dedukti translation, we have taken benefit of rewriting to express the operational semantics of the ς-calculus using a shallow embedding. We have then extended this translation to handle subtyping in Chapter 5. Our first naive implementation in Chapter 6 was unreasonably slow. We have used Dedukti as a program transformer to optimize our implementation and we have obtained a considerable speedup: the optimized implementation is about 100 times faster than the naive one.

In order to use the object-oriented mechanisms of the ς-calculus for interoperability of proof systems, we need to extend it to logic. Following the Curry-Howard correspondence, rich specifications can be expressed by dependent type systems so it is natural to try to extend the ς-calculus to dependent typing.

The main obstacle to combine the ς-calculus with type theory is the lack of termination. In type theory, termination is usually seen as a key feature to ensure both decidability of type checking and consistency: in Martin-Löf Type Theory for example, the false proposition is identified with an inductive type with no constructor; in the empty context, a term of type false has to normalize to a value of type false but no such value exists.

In the ς-calculus however, non-terminating objects (that is, objects for which selecting a method would not terminate) are omnipresent because methods are initiated to loops. Even the very simple encoding of the simply-typed λ-calculus uses a non-terminating object: $(\lambda x : A.b).arg$ diverges.

The omnipresence of looping objects can easily be patched by considering `init` as a primitive undefined symbol and forbidding loops. This does not however forbid non-

terminating objects because more complex cycles would still be allowed. Typing in ς-calculus is not intended to enforce termination and we do not see how a terminating ς-calculus could be designed.

The only terminating object calculus that we are aware of is Castagna's λ&-calculus [40] [1]. This calculus differs greatly from the ς-calculus; in particular messages are first-class values and methods are not components of objects but overloaded functions.

With respect to consistency, the type of `init` already states that all object types are inhabited. The only way to accommodate it with consistency is to interpret the false proposition as a type which is not an object type. Such a type system distinguishing propositions from object types would however hardly pretend to the title of "object-oriented logic"; it would look a lot like a formalization of an object-oriented language in type theory and would not really empowers logic by object-oriented mechanisms.

Trying to accommodate type theory with non-termination is not easy either; if we restrict the interpretation to consider that only terminating terms are proofs we loose decidability of proof checking; if we restrict it further by considering that only normal forms are proofs then we obtain a criterion which is not stable by substitution.

Despite the apparent incompatibility between non-termination and dependent typing, it is noticeable that non-terminating dependently-typed programming languages actually exist. Cayenne [18] is a dependently-typed functional language; its type-checking is undecidable so Cayenne type-checker is incomplete. Cayenne is inconsistent in the sense that all Cayenne types are inhabited. In Ωmega [160], the programmer controls the interaction between static type checking and dynamic programming: the type-checker does not perform computation but when computation would be required for type-checking a reflection of the type system at the level of terms can be used. In ATS [174], indexing expressions form the terminating subclass of terms allowed to index type families.

To continue, we need to be more pragmatic. The object calculi are extremely dynamic languages, far more dynamic than object-oriented mechanisms implemented in real programming languages. Dynamic method update for example is rarely available in compiled

[1] Actually, the λ&-calculus is not terminating but admits slight variations which are strongly termination. See Chapter 3 of [40].

object-oriented languages[2]. If the programmers are happy with static mechanisms, we certainly should consider restricting to more static mechanisms if they allow to integrate logical features. The FoCaLiZe environment satisfies this requirement; FoCaLiZe features static object-oriented mechanisms which are eliminated by the FoCaLiZe compiler and logical methods which are implemented by proofs. The rest of this thesis puts emphases on FoCaLiZe as a tool for interoperability of proof systems. So in Part III, we propose a translation from FoCaLiZe to Dedukti, thus increasing the family of systems that have a Dedukti output. In Part IV we demonstrate the use of FoCaLiZe and its object-oriented mechanisms in the context of interoperability of proof systems.

[2]Compiled object-oriented language offer reflection to dynamically inspect objects and classes but this is considered an advanced feature that should be avoided when possible and it does not go as far as the dynamic nature of the ς-calculus.

Part III

From FoCaLiZe to Dedukti

FoCaLiZe [143] is an environment for certified programming. In FoCaLiZe, proofs do not need to be given in full details but high-level scripts can be written whose gaps are filled by Zenon [31]. Zenon is a complex program, which we fortunately do not need to trust because each proof can be independently checked by Coq.

FoCaLiZe proofs are usually composed of several calls to Zenon and the FoCaLiZe compiler combines all the Coq proofs produced by Zenon into bigger Coq proofs that Zenon was not able to find directly.

In order to benefit from the object-oriented mechanisms of FoCaLiZe for interoperability of proof systems, we developed a new backend for FoCaLiZe compiler[3] to the Dedukti [155] language called Focalide. Focalide itself is the topic of this part; its use as an interoperability platform will be described in Part IV. The work presented in this part will be presented at the 13th ICTAC conference [43].

Another motivation behind the development of Focalide is the integration of Deduction modulo. In order to make the task of writing proofs in FoCaLiZe as easy as possible for the programmer, we want to benefit from all the improvements that have been made recently in Zenon. One of these is the extension of Zenon to Deduction modulo [69], known as Zenon Modulo. In the context of program verification it is often useful to evaluate a program symbolically to prove its correctness, Deduction modulo is particularly adapted for this kind of proofs. Unfortunately, Deduction modulo proofs are in general not checkable by Coq so Zenon Modulo produces Dedukti proofs [44] instead. In order to take benefit from Zenon Modulo without blindly trusting it, this change of proof checker propagates to FoCaLiZe.

As a by-product, the overall performances of FoCaLiZe are enhanced when Dedukti is used to check the proofs because Dedukti is a lightweight proof checker. Coq is able to reconstruct a lot of missing information like type annotations whereas Dedukti is a mere type-checker which does very few inference. Replacing Coq by Dedukti forces us to provide theses pieces of type information instead of discarding them and asking Coq to infer them again, which is quite ineffective.

[3] This work is available online: `http://deducteam.gforge.inria.fr/focalide`.

Focalide is based on the existing backend to Coq which has been adapted to Dedukti syntax; the translation of types and formulae is straightforward but the translation of terms requires some work because Dedukti lacks some mechanisms of functional languages such as local pattern matching.

We present FoCaLiZe in detail in Chapter 7. In Chapter 8 we explain how FoCaLiZe terms are compiled to Dedukti and in Chapter 9, we describe how we integrated Zenon Modulo in Focalide.

More precisely, in Chapter 7, we present FoCaLiZe sublanguages. FoCaLiZe is both a programming language and a logical system. We present the programming part of FoCaLiZe, a variant of the functional language ML in Section 7.1 and the logical part of FoCaLiZe, a polymorphic first-order logic in Section 7.2. FoCaLiZe object-oriented mechanisms do not impact the translation of FoCaLiZe to Dedukti but will play an important role in Part IV, they are presented in Section 7.3. Our presentation of FoCaLiZe ends in Section 7.4 with a description of the architecture of FoCaLiZe compiler.

In Chapter 8, we describe our translation of FoCaLiZe computational language to Dedukti. Two features of ML are non trivial to translate to Dedukti: local pattern matching and recursive definitions. Pattern matching is the topic of Section 8.1 and recursion is the topic of Section 8.2.

In Chapter 9, we extend Zenon Modulo to interface it with FoCaLiZe and Focalide. The required improvements are typing, adapting the two Zenon extensions used by FoCaLiZe to Deduction modulo, and generalizing Deduction modulo to the kind of rewrite rules required by our translation of pattern matching and recursion.

Chapter 7

FoCaLiZe

In this chapter, we give a formal description of the FoCaLiZe language and its compiler.

The FoCaLiZe language can be decomposed into several simpler languages: FoCaLiZe computational language, a functional language very close to ML; FoCaLiZe specification language, a typed version of first-order logic; FoCaLiZe proof language; and FoCaLiZe object-oriented language, a static class-based object-oriented language used for code and proof sharing.

These languages are essentially independent and the two first are very close to well-known formalisms (ML and first-order logic) so the literature on FoCaLiZe mostly focus on the last language. However, this OO language does not affect FoCaLiZe code generation because OO mechanisms are statically resolved in a compiler pass occurring before code generation. Moreover, Dedukti lacks functional mechanisms which are present in other backend languages OCaml and Coq so we need a good description of FoCaLiZe computational language to define our compiler to Dedukti.

In Section 7.1, we give a detailed presentation of the variant of ML implemented in FoCaLiZe. In Section 7.2, we describe the specification and the proof languages. FoCaLiZe specifications are written in polymorphic first-order logic and FoCaLiZe proofs are written in a declarative style. In Section 7.3, we list FoCaLiZe static object-oriented mechanisms. Finally, Section 7.4 is devoted to the way FoCaLiZe developments are compiled.

Basic type	i	::=	**unit**	*Singleton type*
			bool	*Type of booleans*
			string	*Type of character strings*
			int	*Type of integers*
Type	τ	::=	α	*Type variable*
			i	*Basic type*
			$a(\tau_1, \ldots, \tau_n)$	*Constructed type*
			$(\tau_1, \ldots, \tau_n) \to \tau$	*Arrow type*
			$\tau_1 \times \ldots \times \tau_n$	*Cartesian product*
Type scheme	σ	::=	τ	*Type*
			$\Pi\alpha.\ \sigma$	*Universal scheme*

Figure 7.1: Syntax of FoCaLiZe types

7.1 FoCaLiZe Computational Language

FoCaLiZe computational language is a typed functional language featuring implicit ML-like polymorphism. The syntax of FoCaLiZe computational language can be split into types and expressions. As usual, we present types first in Section 7.1.1 and expressions next in Section 7.1.2.

7.1.1 Types

FoCaLiZe type system is very similar to Damas and Milner's Type System (see Section 2.3.1). FoCaLiZe type system includes a few simple basic types and can be extended by user-defined algebraic datatypes.

7.1.1.1 Syntax

The syntax of FoCaLiZe types is given in Figure 7.1. The types **unit**, **bool**, **string**, and **int** are basic types. Types are built from polymorphic type variables, type constructors, arrows, and Cartesian products. A prenex-quantified type is called a type scheme. For compatibility with typing of first-order terms, we consider n-ary versions of arrows and

polymorphism quantifiers.

7.1.1.2 User-Defined Types

User-defined algebraic datatypes can be introduced in FoCaLiZe using the following syntax:

$$\textbf{type}\ a\ (\alpha_1, \ldots, \alpha_k) = \mid C_1(\tau_{1,1}, \ldots, \tau_{1,k_1}) \mid \ldots \mid C_n(\tau_{n,1}, \ldots, \tau_{n,k_n})$$

where the α_i are type variables and the $\tau_{i,j}$ are types; k, and the k_i might be zero but $n \geq 1$.

This defines a new type-constructor a of arity k and its constructors $C_1, \ldots, C_n$. The type scheme associated with the constructor C_i is $\Pi\alpha_1. \ldots \Pi\alpha_k. (\tau_{i,1}, \ldots, \tau_{i,k_i}) \rightarrow a(\alpha_1, \ldots, \alpha_k)$.

7.1.2 Expressions

FoCaLiZe is a functional programming language. In FoCaLiZe, functions are first-class values as in the λ-calculus so we can define functions returning functions as values and functions taking functions as arguments. Unfortunately, Zenon is a first-order theorem prover so it cannot reason about such higher-order functions. Due to this mismatch, the first-order fragment of expressions plays an important role in FoCaLiZe.

In this section, we define the syntax of this first-order fragment of expressions, the typing rules for expressions and the operational semantics of the language.

7.1.2.1 First-Order Fragment

First-order terms play an especially important role in FoCaLiZe because they are the expressions allowed to appear in logical formulae. We describe their syntax in Figure 7.2 where x and f range over distinct enumerable sets. Because of the special treatment of **unit** and **bool** as basic types, their constructors (), **true**, and **false** are reserved keywords. The usual notations for string and integer literals is also assumed. A function symbol can be applied to a list of terms, in particular they cannot be passed as argument to other

Constant	c ::=	()	*Inhabitant of unit*
		true	*True boolean constant*
		false	*False boolean constant*
		"..."	*String literal*
		n	*Integer literal*
Pattern	p ::=	c	*Constant pattern*
		x	*Pattern variable*
		_	*Universal pattern*
		p **as** x	*Named pattern*
		$C(p_1, \ldots, p_n)$	*Constructed pattern*
Terms	t ::=	c	*Constant*
		x	*Variable*
		$f(t_1, \ldots, t_n)$	*Application*
		$t_1 = t_2$	*Equality test*
		let x := t_1 **in** t_2	*Local term definition*
		let $f(x_1, \ldots, x_n)$:= t_1 **in** t_2	*Local function definition*
		let rec $f(x_1, \ldots, x_n)$:= t_1 **in** t_2	*Local recursive definition*
		$C(t_1, \ldots, t_n)$	*Constructed term*
		$(t_1, \ldots, t_n)$	*Tuple*
		if t_1 **then** t_2 **else** t_3	*Conditional*
		match t **with** $\mid p_1 \rightarrow t_1 \ldots \mid p_n \rightarrow t_n$	*Pattern matching*

Figure 7.2: Syntax of FoCaLiZe first-order terms

functions. Anonymous functions are not part of the fragment. A polymorphic equality is assumed but it is rarely used because a user-defined equality is often used in FoCaLiZe programs. Definitions, recursive or not, can be introduced by the **let** keyword. Terms can be constructed by applying a constructor to arguments or by wrapping together terms in a tuple. Finally, terms can be inspected by pattern-matching and the usual **if then else** conditional. A pattern can either be a literal constant, which matches exactly that constant and nothing else, a variable, which is bound to the matched term, a wildcard, which matches any term without binding it to a variable, a named pattern, which when matched binds the variable to the matched term, or a constructor applied to sub-patterns, which matches terms constructed by this constructor applied to terms matching the sub-patterns. Moreover, patterns have to be linear: variables in patterns can occur at most once.

This presentation is not minimal. In particular, the conditional could be derived from pattern matching and the equality could be axiomatized. We include conditional and equality because they are simple to translate to Deduction modulo and Dedukti and will be used in Section 8.1.3 to compile pattern matching. This presentation is not complete either as it lacks two constructs which are seldom used in FoCaLiZe: mutual recursive definitions and records.

7.1.2.2 Full Syntax of Expressions

The syntax of expressions is given in Figure 7.3. Contrary to first-order terms, function symbols are not distinct from variables, the local definition **let** $x := e_1$ **in** e_2 is seen as the special case of the parametric definition **let** $x\ (x_1, \ldots, x_n) := e_1$ **in** e_2 with $n = 0$. Moreover, anonymous functions and recursive functions can be introduced respectively by the λ and μ binders (μ is actually not part of FoCaLiZe concrete syntax but it is convenient to define the semantics of recursive definitions). Heads of application are no longer limited to function symbols but constructors still need to be applied to arguments.

Expression $e ::=$	c	*Constant*
	x	*Variable*
	$\lambda(x_1, \ldots, x_n).e$	*Abstraction*
	$\mu x.e$	*Anonymous recursion*
	$e(e_1, \ldots, e_n)$	*Application*
	$e_1 = e_2$	*Equality test*
	let x $(x_1, \ldots, x_n)$ $:=$ e_1 **in** e_2	*Local definition*
	let rec x $(x_1, \ldots, x_n)$ $:=$ e_1 **in** e_2	*Local recursive definition*
	$C(e_1, \ldots, e_n)$	*Constructed expression*
	$(e_1, \ldots, e_n)$	*Tuple*
	if e_1 **then** e_2 **else** e_3	*Conditional*
	match e **with** $\mid p_1 \rightarrow e_1 \ \ldots \mid p_n \rightarrow e_n$	*Pattern matching*

Figure 7.3: Syntax of FoCaLiZe expressions

Typing Contex	Γ	$::=$	$\emptyset$ *Empty context*
			$\Gamma, x : \sigma$ *Variable declaration*

Figure 7.4: Syntax of FoCaLiZe typing contexts

Constants

$$\frac{}{\Gamma \vdash_c () : \mathbf{unit}} \text{(Type Unit)} \qquad \frac{}{\Gamma \vdash_c \mathbf{true} : \mathbf{bool}} \text{(Type True)}$$

$$\frac{}{\Gamma \vdash_c \mathbf{false} : \mathbf{bool}} \text{(Type False)} \qquad \frac{}{\Gamma \vdash_c \text{"} \ldots \text{"} : \mathbf{string}} \text{(Type String)}$$

$$\frac{}{\Gamma \vdash_c \text{n} : \mathbf{int}} \text{(Type Int)}$$

Patterns

$$\frac{\Gamma \vdash_c \text{c} : \tau}{\Gamma \vdash_p \text{c} : \tau, \emptyset} \text{(PType Constant)} \qquad \frac{}{\Gamma \vdash_p x : \tau, (x : \tau)} \text{(PType Var)}$$

$$\frac{}{\Gamma \vdash_p \_ : \tau, \emptyset} \text{(PType Wildcard)} \qquad \frac{\Gamma \vdash_p p : \tau, \Gamma'}{\Gamma \vdash_p p \ \mathbf{as}\ x : \tau, (\Gamma', x : \tau)} \text{(PType Named)}$$

$$\frac{(C : \Pi\alpha_1 \ldots \alpha_k.\ (\tau_1, \ldots, \tau_n) \to \tau) \in \Gamma \quad \Gamma \vdash_p p_i : \tau_i\rho, \Gamma_i \quad (i = 1 \ldots n)}{\Gamma \vdash_p C(p_1, \ldots, p_n) : \tau\rho, (\Gamma_1, \ldots, \Gamma_n)} \text{(PType Constr)}$$

Figure 7.5: Typing rules for FoCaLiZe constants and patterns

$$\frac{\Gamma \vdash_c \mathrm{c} : \tau}{\Gamma \vdash \mathrm{c} : \tau}\ \text{(Type Constant)} \qquad \frac{(x : \Pi\alpha_1.\ \ldots \Pi\alpha_k.\ \tau) \in \Gamma}{\Gamma \vdash x : \tau\{\alpha_1 \backslash \tau_1, \ldots, \alpha_n \backslash \tau_n\}}\ \text{(Type Var)}$$

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \lambda(x_1, \ldots, x_n).e : (\tau_1, \ldots, \tau_n) \to \tau}\ \text{(Type Abs)} \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mu x.e : \tau}\ \text{(Type Rec)}$$

$$\frac{\Gamma \vdash e : (\tau_1, \ldots, \tau_n) \to \tau \qquad \Gamma \vdash e_i : \tau_i \quad (i = 1 \ldots n)}{\Gamma \vdash e(e_1, \ldots, e_n) : \tau}\ \text{(Type App)}$$

$$\frac{\begin{array}{c}\{\alpha_1, \ldots, \alpha_k\} = (\mathsf{FV}(\tau_1) \cup \ldots \cup \mathsf{FV}(\tau_n) \cup \mathsf{FV}(\tau)) \setminus \mathsf{FV}(\Gamma) \\ \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau \quad \Gamma, x : \Pi\alpha_1 \ldots \alpha_k.\ (\tau_1, \ldots, \tau_n) \to \tau \vdash e' : \tau'\end{array}}{\Gamma \vdash \textbf{let}\ x\ (x_1, \ldots, x_n)\ :=\ e\ \textbf{in}\ e' : \tau'}\ \text{(Type Let)}$$

$$\frac{\begin{array}{l}\Gamma, x : (\tau_1, \ldots, \tau_n) \to \tau, \qquad \{\alpha_1, \ldots, \alpha_k\} = (\mathsf{FV}(\tau_1) \cup \ldots \cup \mathsf{FV}(\tau_n) \cup \mathsf{FV}(\tau)) \setminus \mathsf{FV}(\Gamma) \\ x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau \qquad \Gamma, x : \Pi\alpha_1.\ \ldots \Pi\alpha_k.\ (\tau_1, \ldots, \tau_n) \to \tau \vdash e' : \tau'\end{array}}{\Gamma \vdash \textbf{let rec}\ x\ (x_1, \ldots, x_n)\ :=\ e\ \textbf{in}\ e' : \tau'}\ \text{(Type LetRec)}$$

$$\frac{(C : \Pi\alpha_1.\ \ldots \Pi\alpha_k.\ (\tau_1, \ldots, \tau_n) \to \tau) \in \Gamma \qquad \Gamma \vdash e_i : \tau_i\rho \quad (i = 1 \ldots n)}{\Gamma \vdash C(e_1, \ldots, e_n) : \tau\rho}\ \text{(Type Constr)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \textbf{bool}}\ \text{(Type Eq)}$$

$$\frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau}\ \text{(Type If)}$$

$$\frac{\Gamma \vdash e_i : \tau_i \quad (i = 1 \ldots n)}{\Gamma \vdash (e_1, \ldots, e_n) : \tau_1 \times \ldots \times \tau_n}\ \text{(Type Tuple)}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash_p p_i : \tau_1, \Gamma_i \quad (i = 1 \ldots n) \quad \Gamma, \Gamma_i \vdash e_i : \tau_2 \quad (i = 1 \ldots n)}{\Gamma \vdash \textbf{match}\ e\ \textbf{with}\ \mid p_1\ \to\ e_1 \ldots \mid p_n\ \to\ e_n : \tau_2}\ \text{(Type Match)}$$

Figure 7.6: Typing rules for FoCaLiZe expressions

Value	v	::=	c
			$\lambda(x_1, \ldots, x_n).e$
			$\mu x.e$
			$C(v_1, \ldots, v_n)$
			$(v_1, \ldots, v_n)$
Evaluation Context	E	::=	$\square$
			$E(v_1, \ldots, v_n)$
			$v(v_1, \ldots, v_{i-1}, E, e_{i+1}, \ldots, e_n)$
			$v = E$
			$E = e$
			let x := E **in** e
			$C(v_1, \ldots, v_{i-1}, E, e_{i+1}, \ldots, e_n)$
			$(v_1, \ldots, v_{i-1}, E, e_{i+1}, \ldots, e_n)$
			if E **then** e **else** e
			match E **with** $\mid p_1 \rightarrow e_1 \ldots \mid p_n \rightarrow e_n$

Figure 7.7: Syntax of FoCaLiZe values and evaluation contexts

7.1.2.3 Typing

The typing rules for FoCaLiZe computational language are given in Figure 7.5 and Figure 7.6. They are defined with three inductive judgments:

- $\Gamma \vdash_c \mathrm{c} : \tau$ means that the constant c has type τ in context Γ.
- $\Gamma \vdash_p p : \tau, \Gamma'$ means that the pattern p has type τ in context Γ and binds variables according to Γ'.
- $\Gamma \vdash e : \tau$ means that the expression e has type τ in context Γ.

The typing context Γ associates type schemes to variables and constructors; its syntax is given in Figure 7.4. In these typing rules, ρ denotes substitutions.

7.1.2.4 Semantics

We give an operational call-by-value semantics to FoCaLiZe computational language. The syntax for values and evaluation contexts is given in Figure 7.7. Values form a subset

$$\frac{e \rightsquigarrow e'}{E[e] \rightsquigarrow E[e']} \text{ (Semantics Context)}$$

$$\frac{}{(\lambda(x_1, \ldots, x_n).e)(v_1, \ldots, v_n) \rightsquigarrow e\{x_1 \backslash v_1, \ldots, x_n \backslash v_n\}} \text{ (Semantics Beta)}$$

$$\frac{}{\textbf{let } x \ := \ v \textbf{ in } e \rightsquigarrow e\{x \backslash v\}} \text{ (Semantics Let)}$$

$$\frac{}{\textbf{let } x \ (x_1, \ldots, x_n) \ := \ e_1 \textbf{ in } e_2 \rightsquigarrow \textbf{let } x \ := \ \lambda(x_1, \ldots, x_n).e_1 \textbf{ in } e_2} \text{ (Semantics LetFun)}$$

$$\frac{}{\begin{array}{l}\textbf{let rec } x \ (x_1, \ldots, x_n) \ := \ e_1 \textbf{ in } e_2 \rightsquigarrow \\ \quad \textbf{let } x \ := \ \mu x.\lambda(x_1, \ldots, x_n).e_1 \textbf{ in } e_2\end{array}} \text{ (Semantics LetRec)}$$

$$\frac{}{(\mu x.e)(v_1, \ldots, v_n) \rightsquigarrow (e\{x \backslash \mu x.e\})(v_1, \ldots, v_n)} \text{ (Semantics Mu)}$$

$$\frac{}{\textbf{if true then } e_2 \textbf{ else } e_3 \rightsquigarrow e_2} \text{ (Semantics If True)}$$

$$\frac{}{\textbf{if false then } e_2 \textbf{ else } e_3 \rightsquigarrow e_3} \text{ (Semantics If False)}$$

$$\frac{\rho = mgu(v, p_1)}{\textbf{match } v \textbf{ with } \mid p_1 \ \rightarrow \ e_1 \ldots \mid p_n \ \rightarrow \ e_n \rightsquigarrow e_1\rho} \text{ (Semantics Match First)}$$

$$\frac{v \text{ does not unify with } p_1}{\begin{array}{l}\textbf{match } v \textbf{ with } \mid p_1 \ \rightarrow \ e_1 \ldots \mid p_n \ \rightarrow \ e_n \rightsquigarrow \\ \textbf{match } v \textbf{ with } \mid p_2 \ \rightarrow \ e_2 \ldots \mid p_n \ \rightarrow \ e_n\end{array}} \text{ (Semantics Match Next)}$$

$$\frac{}{\textbf{match } v \textbf{ with } \emptyset \rightsquigarrow \textbf{ERROR}} \text{ (Semantics Match Error)}$$

$$\frac{}{\mathrm{c} = \mathrm{c} \rightsquigarrow \textbf{true}} \text{ (Semantics Eq ConstTrue)} \qquad \frac{\mathrm{c}_1 \not\equiv \mathrm{c}_2}{\mathrm{c}_1 = \mathrm{c}_2 \rightsquigarrow \textbf{false}} \text{ (Semantics Eq ConstFalse)}$$

Figure 7.8: Operational semantics for FoCaLiZe expressions

Formula	φ	::=	t	*Atom*
			$\neg\varphi$	*Negation*
			$\varphi \wedge \varphi$	*Conjunction*
			$\varphi \vee \varphi$	*Disjunction*
			$\varphi \Rightarrow \varphi$	*Implication*
			$\varphi \Leftrightarrow \varphi$	*Equivalence*
			$\forall x : \tau.\varphi$	*Universal quantification*
			$\exists x : \tau.\varphi$	*Existential quantification*

Figure 7.9: Syntax of FoCaLiZe formulae

of expressions. An evaluation context E contains exactly one hole $\Box$, the substitution of the hole by an expression e is written $E[e]$.

The reduction relation $\rightsquigarrow$ is inductively defined in Figure 7.8. The constant **ERROR** represents a run-time error caused by a non-exhaustive pattern matching, it is not a value. We could add more reduction rules for equality on constructed values and tuples but they are not very useful so we prefer to omit them for simplicity.

7.2 Logical Language: FOL

For specification, FoCaLiZe uses as logical language a typed version of first-order logic very close to the TFF1 formalism [26] for typed first-order provers. The main difference between the formulae of this logic and the one that we presented in Section 1.1.3 consists in the choice of atoms: FoCaLiZe atoms are FoCaLiZe first-order terms of type **bool**. We describe this specification language in Section 7.2.1. In Section 7.2.2, we define FoCaLiZe declarative proof language based on Zenon.

7.2.1 Formulae

7.2.1.1 Syntax

The syntax of FoCaLiZe formulae is given in Figure 7.9. Atoms are first-order FoCaLiZe terms of type **bool**, the usual propositional connectives are available and quantification on well-typed terms is allowed.

$$\frac{\Gamma \vdash t : \mathbf{bool}}{\Gamma \vdash t\ \mathbf{prop}}\ \text{(Type Atom)} \qquad \frac{\Gamma \vdash \varphi\ \mathbf{prop}}{\Gamma \vdash \neg\varphi\ \mathbf{prop}}\ \text{(Type Not)}$$

$$\frac{\Gamma \vdash \varphi_1\ \mathbf{prop} \qquad \Gamma \vdash \varphi_2\ \mathbf{prop} \qquad * \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}}{\Gamma \vdash \varphi_1 * \varphi_2\ \mathbf{prop}}\ \text{(Type Connective)}$$

$$\frac{\Gamma, x : \tau \vdash \varphi\ \mathbf{prop} \qquad \mathcal{Q} \in \{\forall, \exists\}}{\Gamma \vdash \mathcal{Q}x : \tau.\varphi\ \mathbf{prop}}\ \text{(Type Quantifier)}$$

Figure 7.10: Typing rules for FoCaLiZe formulae

For example, $\forall x : \mathbf{int}.\exists y : \mathbf{int}.x = y$ is a formula.

7.2.1.2 Typing

In order to express well-typedness of formulae, we introduce a new typing judgment $\Gamma \vdash \varphi\ \mathbf{prop}$ defined in Figure 7.10.

The formula $\forall x : \mathbf{int}.\exists y : \mathbf{int}.x = y$ is well-typed.

7.2.2 Proofs

7.2.2.1 Syntax

FoCaLiZe uses a high-level proof language described in Figure 7.11 and discharges to Zenon the logical details. For this reason, there are no low-level tactics or proof objects in FoCaLiZe. However, the context passed to Zenon has to be given explicitly so that Zenon does not get lost by too many useless hypotheses and function definitions and so that dependencies can be finely controlled.

For example, $(\forall x : \mathbf{int}.x = x) \Rightarrow \forall x : \mathbf{int}.\exists y : \mathbf{int}.x = y$ can be proved by the following proof:

Fact	F	::=	**property** H
			definition of f
			type a
			declaration of x
Step	s	::=	**assume** $x : \tau$ s
			hypothesis $H : \varphi$ s
			prove φ π
Proof	π	::=	**Coq proof** "..."
			Dedukti proof "..."
			assumed
			by $F_1, \ldots, F_n$
			step H s π

Figure 7.11: Syntax of FoCaLiZe proofs

step H_1
  hypothesis $H_2 : \forall x : \textbf{int}.x = x$
  assume x : **int**
  prove $\exists y : \textbf{int}.x = y$
  by declaration of x, **property** H_2
by property H_1

The actual concrete syntax for FoCaLiZe proofs differs from the one we present here in two aspects:

- in FoCaLiZe, keywords introducing facts can take lists of facts (like "**property** H_1, ..., H_n") to avoid repeating the same keyword,
- FoCaLiZe distinguishes property facts by their provenance (it can either be a previous **step**, an **hypothesis** or a mere **property** proved somewhere else). However, introduction of a **declaration of** fact is also done, quite confusingly, by the **property** keyword.

Moreover, a very common pattern in FoCaLiZe proofs consists in a chain of steps of the form **step** H_1 s_1 **step** H_2 s_2 ... **step** H_n s_n **by property** $H_1, \ldots, H_n$. For this case, FoCaLiZe provides the keyword **conclude** to abbreviate this proof as **step** H_1 s_1 **step** H_2

Proof context	Σ	$::=$	$\emptyset$
			Σ, **prop** $H : \varphi$
			Σ, **def** $f\ (x_1, \ldots, x_n) := t$
			Σ, **type** $a(\alpha_1, \ldots, \alpha_k) := \ldots$
			Σ, **decl** $x : \tau$

Figure 7.12: Syntax of FoCaLiZe proof contexts

$s_2 \ldots$ **step** $H_n\ s_n$ **conclude**.

7.2.2.2 Proof Contexts

Proof contexts extend typing contexts by also containing named logical properties, function definitions and type definitions. Their syntax is described in Figure 7.12. We denote by $|\Sigma|$ the typing context extracted from the proof context Σ; this extraction operation can be defined by:

- $|\emptyset| := \emptyset$
- $|\Sigma, \textbf{prop}\ H : \varphi| := |\Sigma|$
- $|\Sigma, \textbf{def}\ f\ (x_1, \ldots, x_n) := t| := |\Sigma|$
- $\big|\Sigma, \textbf{type}\ a(\alpha_1, \ldots, \alpha_k) := |\ C_1(\tau_{1,1}, \ldots, \tau_{1,k_1}) \ldots |\ C_n(\tau_{n,1}, \ldots, \tau_{n,k_n})\big|$

  $$:= \quad |\Sigma|, \quad \begin{array}{l} C_1 : \Pi\alpha_1.\ \ldots \Pi\alpha_k.\ (\tau_{1,1}, \ldots, \tau_{1,k_1}) \to a(\alpha_1, \ldots, \alpha_k), \ldots, \\ C_n : \Pi\alpha_1.\ \ldots \Pi\alpha_k.\ (\tau_{n,1}, \ldots, \tau_{n,k_n}) \to a(\alpha_1, \ldots, \alpha_k) \end{array}$$

- $|\Sigma, \textbf{decl}\ x : \tau| := |\Sigma|, x : \tau$

If $F_1, \ldots, F_n$ are facts present in Σ, we denote by $\Sigma(F_1, \ldots, F_n)$ the sub-context from Σ containing only $F_1, \ldots, F_n$. We write $\mathit{Zenon}(\Sigma, \varphi)$ when Zenon is able to find a proof of φ in context Σ.

7.2.2.3 Statement Associated with a Proof Step

The statement $\mathsf{st}(s)$ associated with a proof step s is the formula defined by:

$$\frac{|\Sigma| \vdash \varphi\ \mathbf{prop}}{\Sigma \vdash \mathbf{Coq\ proof}\ \text{"}\ldots\text{"} : \varphi}\ \text{(Check Coq)}$$

$$\frac{|\Sigma| \vdash \varphi\ \mathbf{prop}}{\Sigma \vdash \mathbf{Dedukti\ proof}\ \text{"}\ldots\text{"} : \varphi}\ \text{(Check Dedukti)}$$

$$\frac{|\Sigma| \vdash \varphi\ \mathbf{prop}}{\Sigma \vdash \mathbf{assumed} : \varphi}\ \text{(Check Assumed)}$$

$$\frac{\Sigma \vdash s \qquad \Sigma, \mathbf{prop}\ H : \mathsf{st}(s) \vdash \pi : \varphi}{\Sigma \vdash \mathbf{step}\ H\ s\ \pi : \varphi}\ \text{(Check Step)}$$

$$\frac{Zenon(\Sigma(F_1, \ldots, F_n), \varphi) \qquad |\Sigma| \vdash \varphi\ \mathbf{prop}}{\Sigma \vdash \mathbf{by}\ F_1, \ldots, F_n : \varphi}\ \text{(Check Zenon)}$$

$$\frac{\Sigma, \mathbf{decl}\ x : \tau \vdash s}{\Sigma \vdash \mathbf{assume}\ x : \tau\ s}\ \text{(Check Assume)} \qquad \frac{\Sigma, \mathbf{prop}\ H : \varphi \vdash s}{\Sigma \vdash \mathbf{hypothesis}\ H : \varphi\ s}\ \text{(Check Hypothesis)}$$

$$\frac{\Sigma \vdash \pi : \varphi}{\Sigma \vdash \mathbf{prove}\ \varphi\ \pi}\ \text{(Check Prove)}$$

Figure 7.13: Proof checking

- $\mathsf{st}(\mathbf{assume}\ x : \tau\ s) := \forall x : \tau.\mathsf{st}(s)$
- $\mathsf{st}(\mathbf{hypothesis}\ H : \varphi\ s) := \varphi \Rightarrow \mathsf{st}(s)$
- $\mathsf{st}(\mathbf{prove}\ \varphi\ \pi) := \varphi$

7.2.2.4 Valid Proofs

We can now define the proof checking relation; the relation $\Sigma \vdash \pi : \varphi$ means that π is a valid proof of the formula φ in context Σ. It is defined in Figure 7.13 mutually with the relation $\Sigma \vdash s$ meaning that s is a valid proof step in context Σ. The proof that we gave for the formula $(\forall x : \mathbf{int}.x = x) \Rightarrow \forall x : \mathbf{int}.\exists y : \mathbf{int}.x = y$ is valid.

7.3 Object-Oriented Mechanisms

We have seen in Section 7.1.1.2 that algebraic datatypes can be defined at toplevel. FoCaLiZe programs can also contain global definitions of expressions and global theorems. Similarly to the possibility to prove directly a theorem in one of the target logical languages (Coq and Dedukti), we can define global symbols by external expressions of the target languages (OCaml, Coq, and Dedukti). For example, addition of integers is defined in FoCaLiZe standard library as follows:

```
let ( + ) =
  internal int -> int -> int
  external
  | caml -> {* Ml_builtins.bi__int_plus *}
  | coq -> {* coq_builtins.bi__int_plus *}
  | dedukti -> {* dk_int.plus *}
;;
```

Each branch of this definition corresponds to a function written manually in the corresponding target language.

In this section, we briefly describe the other toplevel constructions of the FoCaLiZe languages: object-oriented mechanisms. These mechanisms are statically resolved by the FoCaLiZe compiler before code generation so they are unseen by Focalide but they are very useful in practice, in particular when using FoCaLiZe as an interoperability framework.

7.3.1 Species

Species are the main building blocks of FoCaLiZe developments. They are used to group together a type, functions operating on it and specifications of these functions. Species are very similar to abstract classes in OO languages; distinctions arise from the presence of logical methods in FoCaLiZe.

7.3.2 Methods

There are three kinds of methods in FoCaLiZe:

- Computational methods whose bodies are expressions (from Figure 7.3); they correspond to the usual notion of methods in OO languages; they are introduced by the

`let` keyword

- Logical abbreviations whose bodies are formulae; they are introduced by the `logical let` keyword
- Logical methods whose bodies are proofs (from Figure 7.11); they are introduced by the `theorem` keyword
- The representation method whose body is a type (from Figure 7.1); each species contains exactly one representation; it is introduced by the `representation` keyword.

Here is an example of a species definition where all the proofs are assumed to shorten the example:

```
open "basics";;

species PlusInteger =
  representation = int;
  let plus (x : Self, y : Self) : Self = x + y;
  theorem plus_associative :
    all x : Self, all y : Self, all z : Self,
      plus(x, plus(y, z)) = plus(plus(x, y), z)
  proof = assumed;
  theorem plus_commute :
    all x : Self, all y : Self, plus(x, y) = plus(y, x)
  proof = assumed;
end;;
```

7.3.3 Inheritance

To avoid code duplication, FoCaLiZe offers the possibility to define species by inheriting from one or several other species.

For example, we can extend the `PlusInteger` species by a neutral element:

```
species ZeroPlusInteger =
  inherit PlusInteger;
  let zero : Self = 0;
  theorem plus_zero : all x : Self, plus(x, zero) = x
  proof = assumed;
end;;
```

Multiple inheritance is allowed, when a defined method is present in several parent species, the ambiguity is syntactically solved by taking the definition of the rightmost

defining parent in the inherit clause.

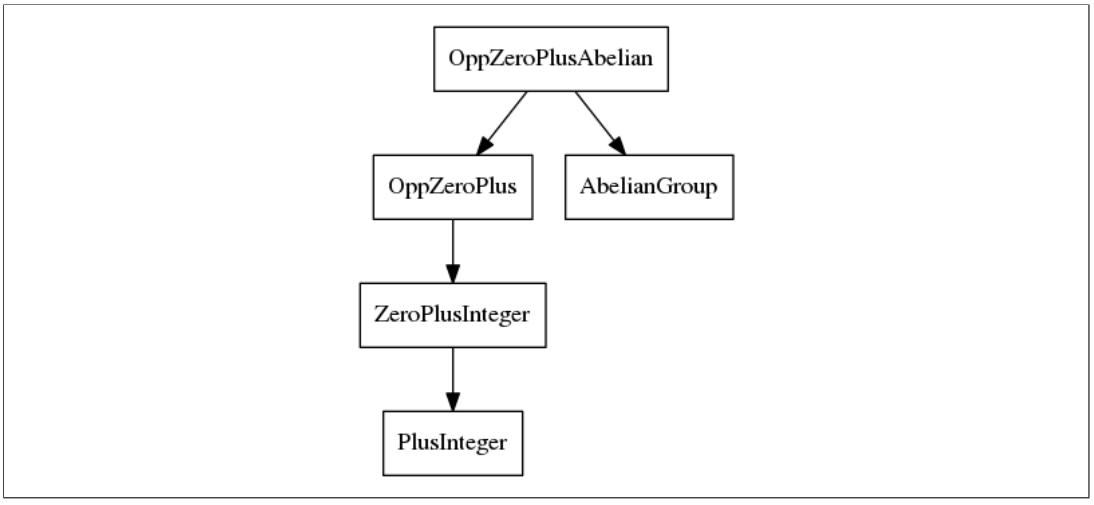
7.3.4 Undefined methods

Providing definitions for methods is not mandatory. An undefined computational method can be introduced by the keyword `signature` and an undefined logical method can be introduced by the keyword `property`; undefined in this case means that the logical method has no proof yet. As there must be exactly one representation (defined or not) per species, no keyword is required for an undefined representation.

Undefined methods can be used to define species representing algebraic structures. For example, we can prove the unicity of the neutral element of any abelian group and instantiate this theorem in the special case of `ZeroPlusInteger`:

```
species AbelianGroup =
  signature plus : Self -> Self -> Self;
  signature zero : Self;
  signature opp : Self -> Self;
  property plus_associative :
    all x : Self, all y : Self, all z : Self,
      plus(x, plus(y, z)) = plus(plus(x, y), z);
  property plus_commute :
    all x : Self, all y : Self, plus(x, y) = plus(y, x);
  property plus_zero : all x : Self, plus(x, zero) = x;
  property opp_plus : all x : Self, plus(x, opp(x)) = zero;

  theorem zero_uniq :
     all z : Self, (all x : Self, plus(x, z) = x) -> z = zero
  proof = assumed;
end;;

species OppZeroPlus =
  inherit ZeroPlusInteger;
  let opp (x : Self) : Self = 0 - x;
  theorem opp_plus : all x : Self, plus(x, opp(x)) = zero
  proof = assumed;
end;;

species OppZeroPlusAbelian =
  inherit OppZeroPlus, AbelianGroup;
end;;
```

All the methods of species `OppZeroPlusAbelian` are defined (we say that `OppZeroPlusAbelian` is a *complete* species) and the proof of `zero_uniq` does not depend on the choice of the type for the representation and can be used in any abelian group.



In this figure, frames represent species and there is an arrow from a species A to a species B if A directly inherits from B.

Figure 7.14: Inheritance Hierarchy

In the course of a typical FoCaLiZe development, the first species are totally abstract and the latter species refine them until all the methods are defined. The inheritance hierarchy of this example is pictured in Figure 7.14.

Species look a lot like typeclasses as found in Haskell, Coq and Isabelle. The main difference which gives a real OO flavor to FoCaLiZe species is redefinition.

7.3.5 Redefinition

When a defined computational method is inherited, it is possible to give it a new definition overriding the inherited one. As usual in object-oriented languages, the semantics of redefinition is given by *early binding*: when a method gets several definitions it is bound to the last one, if a method m_1 refers to method m_2 and m_2 get redefined, the meaning of m_1 silently changes to refer to the new definition of m_2.

When a method is redefined, proofs of logical methods depending on the previous definition via **definition of** facts are then not valid anymore and are removed in the child species, turning these logical methods to undefined.

A typical example of the use of redefinition is to provide default definitions that can then be replaced by more efficient ones when more properties are assumed. From an ordering relation, equality can be defined by antisymmetry:

```
species Setoid =
  signature eq : Self -> Self -> bool;
  property eq_refl : all x : Self, eq(x, x);
  property eq_symm : all x y : Self, eq(x, y) -> eq(y, x);
  property eq_trans :
    all x y z : Self, eq(x, y) -> eq(y, z) -> eq(x, z);
end;;

species Ordering =
  inherit Setoid;

  signature leq : Self -> Self -> bool;
  property leq_refl : all x : Self, leq(x, x);
  property leq_trans :
    all x y z : Self, leq(x, y) -> leq(y, z) -> leq(x, z);

  let eq(x, y) = leq(x, y) && leq(y, x);
  proof of eq_refl = by definition of eq property leq_refl;
  proof of eq_symm = by definition of eq;
  proof of eq_trans = by definition of eq property leq_trans;
end;;

type nat = | O | S(nat);;

species OrderedNat =
  inherit Ordering;

  representation = nat;
  let rec leq(x, y) =
    match x with
    | O -> true
    | S(x) ->
    (match y with
     | O -> false
     | S(y) -> leq(x, y));

end;;

species EfficientOrderedNat =
  inherit OrderedNat;

  let rec eq(x, y) =
    match (x, y) with
    | (O, O) -> true
    | (O, S(_)) -> false
    | (S(_), O) -> false
    | (S(x), S(y)) -> eq(x, y);
```

```
end;;
```

In this example, the theorems on equality `eq_refl`, `eq_symm`, and `eq_trans` are proved in species `OrderedNat` but their proofs are erased in species `EfficientOrderedNat` because they depend on the definition of `eq` found in species `OrderedNat` which is unavailable in species `EfficientOrderedNat` and its descendents due to early binding.

7.3.6 Collections

Complete species can be transformed into *collections* on which methods can be called from the outside.

```
collection C =
  implement OppZeroPlusAbelian;
end;;
```

The construction operation also abstracts the representation so `C!zero_uniq` proves the formula `all z : C, (all x : C, plus(x, z) = x) -> z = zero` and the type `C` is abstract, it can only be manipulated by the methods of the collection.

7.3.7 Parameters

Species can be parameterized. Two kinds of parameters are allowed:

- **is**-parameters: if `S` is a previously defined species, parameterization over collections instantiating species inheriting from `S` is allowed with the syntax `P is S`
- **in**-parameters: if `C` is a collection, paramerization over elements of `C` is allowed with the syntax `p in C`

For example, given two abelian groups `G1` and `G2`, we can build the product of `G1` and `G2` which is also an abelian group:

```
species AbelianProduct (G1 is AbelianGroup, G2 is AbelianGroup) =
  inherit AbelianGroup;

  representation = G1 * G2;

  let zero = (G1!zero, G2!zero);
```

```
  let opp (g) = (G1!opp(fst(g)), G2!opp(snd(g)));

  let plus (g, h) =
     (G1!plus(fst(g), fst(h)), G2!plus(snd(g), snd(h)));

  proof of plus_associative =
    by definition of plus
       property G1!plus_associative, G2!plus_associative;
  proof of plus_commute =
    by definition of plus
       property G1!plus_commute, G2!plus_commute;
  proof of plus_zero =
    by definition of plus, zero
       property G1!plus_zero, G2!plus_zero;
  proof of opp_plus =
    by definition of opp, plus, zero
       property G1!opp_plus, G2!opp_plus;
end;;
```

We can also translate a group by one of its elements:

```
species TranslatedAbelianGroup (G is AbelianGroup, g in G) =
  inherit AbelianGroup;

  representation = G;

  let zero = g;
  let plus(g1, g2) = G!plus(G!plus(g1, g2), G!opp(g));
  let opp(g1) = G!plus(G!opp(g1), g);

  proof of plus_associative =
    by definition of plus
       property G!plus_associative;
  proof of plus_commute =
    by definition of plus
       property G!plus_commute;
  proof of plus_zero =
    by definition of plus, zero
       property G!opp_plus, G!plus_associative;
  proof of opp_plus =
    by definition of opp, plus, zero
       property G!opp_plus, G!plus_zero;
end;;
```

7.4 Compilation

We now briefly describe the architecture of `focalizec`, the FoCaLiZe compiler. The compilation process is composed of several passes, some of them are specific to FoCaLiZe

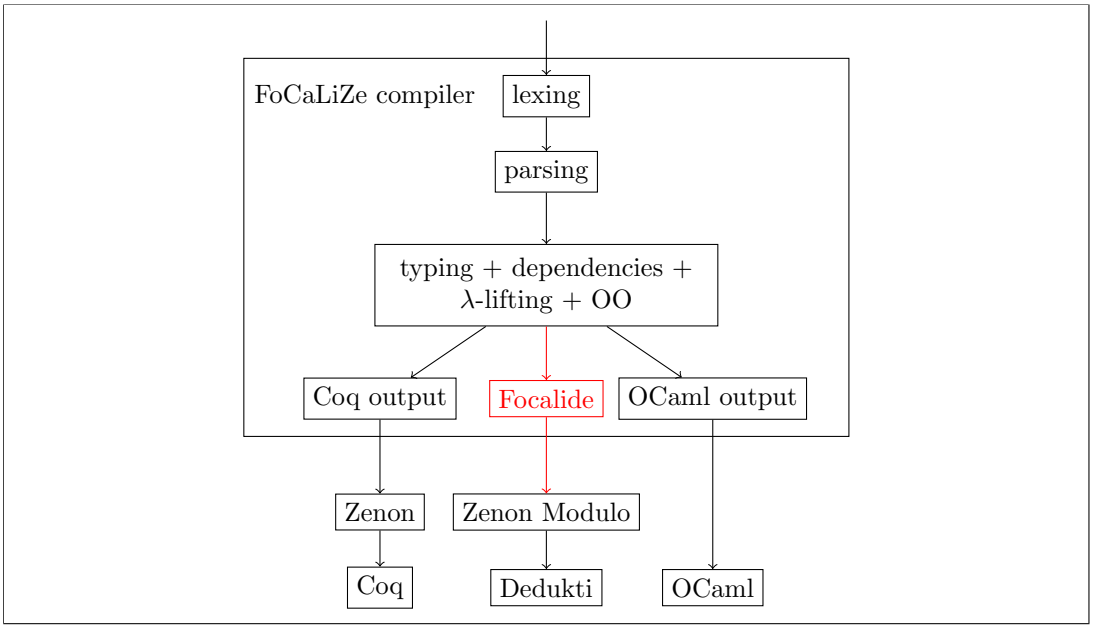


Figure 7.15: FoCaLiZe Compilation Scheme

and need a bit of explanations.

7.4.1 Compilation Passes

In Figure 7.15, we show the pipeline of FoCaLiZe compilation passes. The first two ones, lexing and parsing transform the input file into a parsed abstract syntax tree. In the third pass all relevant nodes are annotated by typing information in order to detect type errors as early as possible. In this typing pass, since most binders in our syntax do not require type annotations, Algorithm W from Damas and Milner [59] is used to infer them. During this third pass, FoCaLiZe static object-oriented mechanisms are also resolved and dependencies between methods and parameters are computed.

7.4.2 Lifting and Dependency Calculus

Because of early binding, a defined method cannot be translated directly as a toplevel definition because its body may contain calls to declared (not yet defined) methods (of the same species, of inherited species, or from parameters) or calls to other methods which can

be redefined later. The solution to delay the definition of auxiliary functions in functional languages consists in adding parameters for these auxiliary functions so that their chosen implementation can be provided as late as the time of calling the function. This technique is known as λ-lifting [101] and is easy to implement in a compiler. λ-lifting is also performed in the third pass of the compiler.

Things get a bit harder when methods start to **inspect** the code of other methods. This is for example the case when the proof of a logical method m_1 is based on the definition of a computational method m_2, a feature which we absolutely need in a system such as FoCaLiZe where the primary goal of proofs is to certify that the definitions of functions satisfy their specifications. In this case, it is not possible anymore to λ-lift m_2 in the definition of m_1. Moreover, redefining m_2 should invalidate the proof of m_1 since it relied on an obsolete implementation of m_2. This leads to the distinction of two kinds of dependencies between methods:

- m_1 **decl-depends** on m_2 if m_1 does not inspect the definition of m_2, m_2 can even be undefined. The only information that m_1 needs about m_2 is its type (or its statement when m_2 is a logical method). This kind of dependency is handled by λ-lifting.

- m_1 **def-depends** on m_2 if it does inspect the definition of m_2. m_1 knows both the type and the definition of m_2. This kind of dependency is not λ-lifted, but the definition of m_1 is erased as soon as m_2 is redefined.

In the case of the dependencies on the representation, we still have both cases however:

- decl-dependence on the representation brings no special information, λ-lifting the representation corresponds exactly to defining a polymorphic function

- def-dependence on the representation is so common that redefining the representation would usually erase all the interesting part of species. To avoid accidental erasure of methods, redefinition of the representation is forbidden by the compiler.

The concrete way to compute the dependencies of a method is described in [143, 150]. This is the purpose of the compilation pass labeled "typing + OO" in Figure 7.15.

7.4.3 Backend Input Language

The input of the different backends (Coq output, OCaml output, and Focalide) is composed of the input abstract syntax tree with two extra pieces of information: each node is annotated with its type and each method is annotated with its decl-dependencies, its def-dependencies and, for defined methods, the name of the species containing its current definition. It is then easy to replace each species with a namespace and each definition in a species with a λ-lifted definition. Namespaces corresponding to complete species also contain shortcuts to unlifted definitions of their methods; these shortcuts are wrapped in a record.

Collections can also be implemented by namespaces, each method of a collection is a definition in this namespace pointing to the record field in the implemented complete species.

So the interesting part of the backend, which really depends on the language toward which we are compiling, is the translation from the computational and logical languages. The backend input language is the system presented in Sections 7.1 and 7.2 for which files consist of definitions of datatypes, higher-order functions and proved theorems (whose atoms are first-order terms of type **bool**); moreover, these definitions can appear at toplevel or be grouped together in namespaces.

7.4.4 Compilation of Proofs to Coq

FoCaLiZe computational language has been designed with Coq and OCaml backends in mind; the features available in this language (such as local definitions, pattern-matching, and tuples) are available in both OCaml and Coq and are trivially translated. The proof language however is very different because it has been designed to be convenient for Zenon.

Zenon output to the Coq language actually gives a Coq toplevel theorem, from its statement with the **Theorem** keyword to the end of the proof with the **Qed** keyword. For example, a Zenon proof of a property `a : A` depending on a previously known property `b : B` will be a toplevel Coq theorem `Theorem a : A. Proof. ... b ... Qed.`

In order to integrate this in a FoCaLiZe development compiled to Coq, we need to add

arguments for λ-lifting. The **Section** mechanism of Coq is a way to perform implicit λ-lifting: inside a section, **Variables** can be declared and used in the definitions, statements, and proofs of the section; from outside the section, each symbol gets λ-lifted with respect to all the variables it uses. The FoCaLiZe compiler wraps all the toplevel theorems generated by Zenon into sections so on the previous example, the compiled Coq file shall be

```
Section Proof_of_a.
  Variable b : B.
  Theorem a : A.
  Proof.
    ... b ...
  Qed.
End Proof_of_a.
```

which is synonym of

```
Theorem a (b : B) : A.
Proof.
  ... b ...
Qed.
```

so from outside the section, `a` has type `B -> A` as expected.

Coq sections seem very convenient in this case because it seems that we only need to wrap the output of Zenon in a section to get the expected λ-lifted theorem. There is a bad corner-case, however, which introduces some complexity: if Zenon finds a proof which does not use one of the hypotheses, then this hypothesis does not get lifted by the section mechanism and then the set of lifted methods does not correspond to the set of decl-dependencies computed by the dependency calculus. For example, if Zenon is able to prove `A` without using the hypothesis `b : B`, the obtained section

```
Section Proof_of_a.
  Variable b : B.
  Theorem a : A.
  Proof.
    ...
  Qed.
End Proof_of_a.
```

is now a synonym of

```
Theorem a : A.
Proof.
  ...
Qed.
```

and `a` has type `A` instead of the expected type `B -> A`.

The compiler cannot foresee this problem because it has no access to the actual definition of the logical method, which will be computed by Zenon after the compiler has exited, but only to the proof script which might contain useless dependencies. Fortunately, this issue can be solved by doing a second proof on the same theorem in which all the expected decl-dependencies are enforced before exploiting the proof provided by Zenon. The actual generated code is

```
Section Proof_of_A.
  Variable b : B.
  Theorem for_zenon_a : A.
  Proof.
    ...
  Qed.

  Theorem a : A.
  Proof.
    assert (__force_use_b := b).
    apply for_zenon_a; auto.
  Qed.
End Proof_of_same_is_not_different.
```

This issue is very specific to Coq and its interaction with Zenon. In the case of Focalide, no section mechanism is available in Dedukti but Zenon Modulo is able to output a Dedukti **term** (and nothing more) which can be plugged in any context. The other distinctions between Zenon and Zenon Modulo will be discussed in Chapter 9.

Chapter 8

Computational Part: Compiling ML to Dedukti

As we have seen, the interesting part of the Focalide backend is the translation of the computational and logical languages to Dedukti. This chapter describes the two difficult points in the translation of the computational language, the next chapter will show how Deduction modulo and its implementation in Zenon Modulo can be extended to accept the rewrite system that we are going to introduce now. Most features of the computational language can be translated in a very simple, shallow way: variables are translated by variables, applications by applications, and abstractions by abstractions. The interesting part of the translation of the computational language is the translation of pattern matching and recursion. We encode pattern matching in Dedukti in Section 8.1 and recursion in Section 8.2. Section 8.3 is devoted to a comparison with the techniques of the literature.

8.1 Pattern Matching

Dedukti's pattern matching differs from the one of functional languages such as FoCaLiZe in two important ways:

- Locality: in Dedukti, pattern matching is only allowed at toplevel, introduction of new rewrite rules is a global process whereas the **match ... with** construct is local.
- Overlapping: rewrite rules are requested to be confluent whereas overlapping of patterns is solved by their ordering.

For these reasons, we need to compile FoCaLiZe pattern matching to simpler constructs which are expressible in Dedukti. In our context, the efficiency of the produced Dedukti code is not fundamental because definitions using pattern matching in FoCaLiZe are usually simple. The main reasons for this is that complex pattern matchings are hard to specify and that Zenon support for pattern matching is limited. Hence we want to avoid complex compilation techniques such as decision trees [121] because the generated code would be hard to reason about automatically.

We start by solving the locality issue in Section 8.1.1. To solve the overlapping issue, we define two program transformation functions. The first one, called serialization, compiles pattern matchings with arbitrary number of branches to a simpler form with exactly two branches. The second one, called flattening, simplifies the nesting of patterns. Serialization is defined in Section 8.1.2 and flattening is defined in Section 8.1.3. After these two transformations, the programs can be expressed in Dedukti, this is the topic of Section 8.1.4.

8.1.1 Lifting of Pattern Matchings

The locality issue is not hard because it is always possible to λ-lift the expression

$$E[\textbf{match } a \textbf{ with } \mid p_1 \ \rightarrow \ e_1 \ldots \mid p_n \ \rightarrow \ e_n]$$

(where E is an evaluation context) as

$$\textbf{let } f(x_0, x_1, \ldots, x_n) \ := \ \textbf{match } x_0 \textbf{ with } \mid p_1 \ \rightarrow \ x_1(\mathsf{FV}(p_1)) \ldots \mid p_n \ \rightarrow \ x_n(\mathsf{FV}(p_n)) \textbf{ in } E[f(a, \lambda(\mathsf{FV}(p_1)).e_1, \ldots, \lambda(\mathsf{FV}(p_n)).e_n)]$$

where $f, x_0, x_1, \ldots, x_n$ are fresh and $\mathsf{FV}(p)$ denotes the set of all the variables occurring in the pattern p.

However, even at toplevel, we cannot directly translate FoCaLiZe pattern matching by rewriting without breaking confluence as demonstrated by the following example.

Consider the following top-level definition

$$\begin{array}{l}\textbf{let } f(x) := \textbf{match } x \textbf{ with}\\ \mid 0 \to 1\\ \mid \_ \to 0\end{array}$$

Using the definition of integers from Section 3.4, the direct translation in Dedukti would be

```
def f : int -> int.
[] f (Diff 0 0) --> Diff 1 0
[] f _ --> Diff 0 0.
```

which is not confluent and from which it is easy to obtain a Dedukti proof of $0 = 1$:

```
int_equal : int -> int -> Type.
int_refl : n : int -> int_equal n n.
def fn_is_0 (n : int) : int_equal (Diff 0 0) (f n)
    := int_refl (Diff 0 0).
def 0_is_1 : int_equal (Diff 0 0) (Diff 1 0) := fn_is_0 (Diff 0 0).
```

8.1.2 Serialization

We say a pattern matching *simple* if it has two branches and the second pattern is _.

Serialization is the program transformation process by which each pattern matching with an arbitrary number of branches becomes a sequence of simple pattern matchings. We formally describe this program transformation by a function $\mathcal{S}$ which is defined in Figure 8.1.

The function $\mathcal{S}$ recursively descends the expression until it finds a pattern matching, it then replaces a pattern matching with n branches by a sequence of n simple pattern matchings. To avoid useless duplication of the matched expression, it is first factorized using a let binding. This transformation is linear and preserves the operational semantics defined in Figure 7.8.

Once serialized, the priority of patterns is made apparent but since the pattern matching in our counterexample in Section 8.1.1 used a simple pattern matching, it is clear that we still cannot simply lift pattern matchings and translate them directly. The difficulty comes from the complexity of the patterns, in the next section, we use another program transformation to simplify them until they become easy to define by confluent rewrite rules.

$$
\begin{array}{l}
\mathcal{S}(x) := x \\
\mathcal{S}(f(a_1, \ldots, a_n)) := f(\mathcal{S}(a_1), \ldots, \mathcal{S}(a_n)) \\
\ldots \\
\mathcal{S}(\textbf{match } a \textbf{ with } \mid p_1 \ \rightarrow \ e_1 \ldots \mid p_n \ \rightarrow \ e_n) \ := \\
\quad \textbf{let } x \ := \ a \textbf{ in} \\
\quad \textbf{match } x \textbf{ with} \\
\quad \mid p_1 \ \rightarrow \ \mathcal{S}(e_1) \\
\quad \mid \_ \ \rightarrow \\
\qquad \textbf{match } x \textbf{ with} \\
\qquad \mid p_2 \ \rightarrow \ \mathcal{S}(e_2) \\
\qquad \mid \_ \ \rightarrow \ \ldots \\
\qquad\quad \textbf{match } x \textbf{ with} \\
\qquad\quad \mid p_n \ \rightarrow \ \mathcal{S}(e_n) \\
\qquad\quad \mid \_ \ \rightarrow \ \textbf{ERROR} \\
\textit{where } x \textit{ is a fresh variable}
\end{array}
$$

Figure 8.1: Definition of the function $\mathcal{S}$ for serialization of pattern matching

8.1.3 Compiling Patterns to Destructors

Instead of translating each matching to a rewrite system, we introduce symbols called *destructors* associated with constructors. These destructors will be defined by orthogonal (and hence confluent) rewrite systems.

If C is a constructor of arity n for some datatype, we call the expression

$$
\begin{array}{rl}
D_C := \lambda a, b, c. \ \textbf{match } a \textbf{ with} & \mid C(x_1, \ldots, x_n) \ \rightarrow \ b(x_1, \ldots, x_n) \\
& \mid \_ \ \rightarrow \ c
\end{array}
$$

the *destructor* associated with C. We say that a pattern matching has *the shape of a destructor* if it is a fully applied destructor. In other words, a pattern matching has the shape of a destructor if it is simple and its first pattern is a constructor applied to variables.

In this section, we show how to translate FoCaLiZe expressions to the fragment of FoCaLiZe where each pattern matching has the shape of a destructor, that is we want to restrict our grammar of expressions to the following:

$$
e' ::= \ldots \ \Big| \ \textbf{match } e' \textbf{ with } \mid C(x_1, \ldots, x_n) \ \rightarrow \ e' \mid \_ \ \rightarrow \ e'
$$

We introduce another program transformation $\mathcal{F}$ called flattening defined in Figure 8.2. It also descends down the expression looking for a pattern matching. The input for $\mathcal{F}$ is the output of $\mathcal{S}$ so it must be a simple pattern matching whose matching expression is a variable. Function $\mathcal{F}$ then inspects the first pattern; constant patterns are transformed using equality tests and alternatives; for named patterns, the function recursively calls itself after let binding of the pattern name to the matched variable; variable patterns are directly transformed to let bindings; and wildcard patterns are transformed to their bodies. The most interesting case is the case of constructed patterns where each subpattern is tried from left to right; if all succeed, we proceed with the branch body, otherwise we continue with the default case. Similarly to function $\mathcal{S}$, we avoid code duplication of the default case by factorizing it using a let binding but we also do not want it to be evaluated when all pattern match so we use the usual trick of wrapping it in a λ-abstraction expecting an argument of type **unit** whose purpose is to delay evaluation until it is applied to the expression ().

To prove the termination of $\mathcal{F}$, we define the notion of first-pattern size for an expression e as follows:

- if e is **match** a **with** $\mid p_1 \ \rightarrow \ e_1 \ldots$ then its first-pattern size is the size of p_1: $\mathsf{size}_{1^{\mathsf{st}}\mathsf{pat}}(e) = \mathsf{size}(p_1)$,
- otherwise it is 0: $\mathsf{size}_{1^{\mathsf{st}}\mathsf{pat}}(e) = 0$.

The lexical ordering $e_1 \leq e_2$ defined as $(\mathsf{size}_{1^{\mathsf{st}}\mathsf{pat}}(e_1), \mathsf{size}(e_1)) \leq_{\mathsf{lex}} (\mathsf{size}_{1^{\mathsf{st}}\mathsf{pat}}(e_2), \mathsf{size}(e_2))$ is well-founded and strictly decreasing at each recursive call of $\mathcal{F}$ so $\mathcal{F}$ terminates.

Flattening $\mathcal{F}$ preserves the semantics of pattern matching:

Theorem 14. *For any expression e and any substitution θ from variables to values, the expressions $\mathcal{F}(e)\theta$ and $e\theta$ are semantically equivalent.*

Proof. The proof is done by induction on the structure of the function $\mathcal{F}$. The only non-trivial case is the case of nested patterns: we want to prove $\mathcal{F}(\textbf{match}\ x\ \textbf{with} \mid C(p_1, \ldots, p_n) \ \rightarrow \ e_1 \mid \_ \ \rightarrow \ d)$ $\equiv \textbf{match}\ x\theta\ \textbf{with} \mid C(p_1, \ldots, p_n) \ \rightarrow \ e_1\theta \mid \_ \ \rightarrow \ d\theta$.

$$
\begin{array}{l}
\mathcal{F}(x) := x \\
\mathcal{F}(f(a_1, \ldots, a_n)) := f(\mathcal{F}(a_1), \ldots, \mathcal{F}(a_n)) \\
\ldots \\
\mathcal{F}(\textbf{match } x \textbf{ with } \mid c \ \rightarrow \ e \mid \_ \ \rightarrow \ d) \ := \\
\quad \textbf{if } x = c \textbf{ then } \mathcal{F}(e) \textbf{ else } \mathcal{F}(d) \\
\mathcal{F}(\textbf{match } x \textbf{ with } \mid p \textbf{ as } y \ \rightarrow \ e \mid \_ \ \rightarrow \ d) \ := \\
\quad \mathcal{F}(\textbf{match } x \textbf{ with } \mid p \ \rightarrow \ \textbf{let } y \ := \ x \textbf{ in } e \mid \_ \ \rightarrow \ d) \\
\mathcal{F}(\textbf{match } x \textbf{ with } \mid y \ \rightarrow \ e \mid \_ \ \rightarrow \ d) \ := \\
\quad \textbf{let } y \ := \ x \textbf{ in } \mathcal{F}(e) \\
\mathcal{F}(\textbf{match } x \textbf{ with } \mid \_ \ \rightarrow \ e \mid \_ \ \rightarrow \ d) \ := \\
\quad \mathcal{F}(e) \\
\mathcal{F}(\textbf{match } x \textbf{ with } \mid C(p_1, p_2, \ldots, p_n) \ \rightarrow \ e \mid \_ \ \rightarrow \ d) \ := \\
\quad \textbf{let } f(x_0 : \textbf{unit}) \ := \ \mathcal{F}(d) \textbf{ in} \\
\quad \textbf{match } x \textbf{ with} \\
\quad \mid C(x_1, \ldots, x_n) \ \rightarrow \\
\qquad \mathcal{F}(\textbf{match } x_1 \textbf{ with} \\
\qquad\quad \mid p_1 \ \rightarrow \ \mathcal{F}(\textbf{match } x_2 \textbf{ with} \\
\qquad\qquad \mid p_2 \ \rightarrow \ \ldots \mathcal{F}(\textbf{match } x_n \textbf{ with} \\
\qquad\qquad\quad \mid p_n \ \rightarrow \ \mathcal{F}(e) \\
\qquad\qquad\quad \mid \_ \ \rightarrow \ f()) \ldots \\
\qquad\qquad \mid \_ \ \rightarrow \ f()) \\
\qquad\quad \mid \_ \ \rightarrow \ f()) \\
\quad \mid \_ \ \rightarrow \ f() \\
\textit{where the variables } x_i \textit{ and the function symbol } f \textit{ are fresh}
\end{array}
$$

Figure 8.2: Definition of the function $\mathcal{F}$ for flattening of pattern matching

After unfolding of definitions, we need to prove that

$$
\begin{array}{l}
\textbf{let } f(x_0 : \textbf{unit}) \ := \ d' \textbf{ in} \\
\textbf{match } v \textbf{ with} \\
\mid C(x_1, \ldots, x_n) \ \rightarrow \\
\quad (\mathcal{F}(\textbf{match } x_1 \textbf{ with} \\
\qquad \mid p_1 \ \rightarrow \ \mathcal{F}(\textbf{match } x_2 \textbf{ with} \\
\qquad\quad \mid p_2 \ \rightarrow \ \ldots \mathcal{F}(\textbf{match } x_n \textbf{ with} \\
\qquad\qquad \mid p_n \ \rightarrow \ \mathcal{F}(e'_1) \\
\qquad\qquad \mid \_ \ \rightarrow \ f()) \ldots \\
\qquad\quad \mid \_ \ \rightarrow \ f()) \\
\qquad \mid \_ \ \rightarrow \ f()) \\
\mid \_ \ \rightarrow \ f()) \\
\equiv \textbf{match } v \textbf{ with} \mid C(p_1, \ldots, p_n) \ \rightarrow \ e'_1 \mid \_ \ \rightarrow \ d'
\end{array}
$$

where the variables x_i and the function symbol f are fresh and d', v, and e'_1 are respective abbreviations for $\mathcal{F}(d)\theta$, $x\theta$, and $e_1\theta$.

– If v does not unify with $C(x_1, \ldots, x_n)$, then it does not unify with the more specific pattern $C(p_1, \ldots, p_n)$ either and both sides are equivalent to d'.

– Otherwise, $v = C(v_1, \ldots, v_n)$ and the left-hand side can be reduced:

$$
\begin{array}{l}
\textbf{let } f(x_0 : \textbf{unit}) \ := \ d' \textbf{ in} \\
\textbf{match } C \textbf{ with}(v_1, \ldots, v_n) \\
\mid C(x_1, \ldots, x_n) \ \rightarrow \\
\quad (\mathcal{F}(\textbf{match } x_1 \textbf{ with} \\
\qquad \mid p_1 \ \rightarrow \ \mathcal{F}(\textbf{match } x_2 \textbf{ with} \\
\qquad\quad \mid p_2 \ \rightarrow \ \ldots \mathcal{F}(\textbf{match } x_n \textbf{ with} \\
\qquad\qquad \mid p_n \ \rightarrow \ \mathcal{F}(e'_1) \\
\qquad\qquad \mid \_ \ \rightarrow \ f()) \ldots \\
\qquad\quad \mid \_ \ \rightarrow \ f()) \\
\qquad \mid \_ \ \rightarrow \ f()) \\
\mid \_ \ \rightarrow \ f())
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{l}
\textbf{match } C(v_1, \ldots, v_n) \textbf{ with} \\
\mid C(x_1, \ldots, x_n) \ \rightarrow \\
\quad \mathcal{F}(\textbf{match } x_1 \textbf{ with} \\
\qquad \mid p_1 \ \rightarrow \ \mathcal{F}(\textbf{match } x_2 \textbf{ with} \\
\qquad\quad \mid p_2 \ \rightarrow \ \ldots \mathcal{F}(\textbf{match } x_n \textbf{ with} \\
\qquad\qquad \mid p_n \ \rightarrow \ \mathcal{F}(e'_1) \\
\qquad\qquad \mid \_ \ \rightarrow \ d'') \ldots \\
\qquad\quad \mid \_ \ \rightarrow \ d'') \\
\qquad \mid \_ \ \rightarrow \ d'') \\
\mid \_ \ \rightarrow \ d''
\end{array}
$$

$$
\begin{array}{l}
\rho_0(\mathcal{F}(\textbf{match } x_1 \textbf{ with} \\
\quad \mid p_1 \ \rightarrow \ \mathcal{F}(\textbf{match } x_2 \textbf{ with} \\
\quad\quad \mid p_2 \ \rightarrow \ \ldots \mathcal{F}(\textbf{match } x_n \textbf{ with} \\
\quad\quad\quad \mid p_n \ \rightarrow \ \mathcal{F}(e'_1) \\
\quad\quad\quad \mid \_ \ \rightarrow \ d'') \ldots \\
\quad\quad \mid \_ \ \rightarrow \ d'') \\
\quad \mid \_ \ \rightarrow \ d''))
\end{array}
\quad \rightsquigarrow
$$

where ρ_0 is the substitution mapping each x_i to the corresponding v_i and d'' is the redex $(\lambda(x_0 : \textbf{unit}).d')()$.

By iterated induction hypothesis on all the arguments of $\mathcal{F}$, this term is equivalent to

$$\begin{array}{l}
\textbf{match } v_1 \textbf{ with} \\
\quad | \; p_1 \;\to\; (\textbf{match } v_2 \textbf{ with} \\
\qquad | \; p_2 \;\to\; (\ldots \textbf{match } v_n \textbf{ with} \\
\qquad\quad | \; p_n \;\to\; \rho_0(e'_1) \\
\qquad\quad | \; _ \;\to\; \rho_0(d''))\ldots \\
\qquad | \; _ \;\to\; \rho_0(d'')) \\
\quad | \; _ \;\to\; \rho_0(d'')
\end{array}$$

Since the x_i do not appear in e'_1 nor d'', this term is simply

$$\begin{array}{l}
\textbf{match } v_1 \textbf{ with} \\
\quad | \; p_1 \;\to\; (\textbf{match } v_2 \textbf{ with} \\
\qquad | \; p_2 \;\to\; (\ldots \textbf{match } v_n \textbf{ with} \\
\qquad\quad | \; p_n \;\to\; e'_1) \\
\qquad\quad | \; _ \;\to\; d'')\ldots \\
\qquad | \; _ \;\to\; d'') \\
\quad | \; _ \;\to\; d''
\end{array}$$

To prove that this term is semantically equivalent to the expected one **match** v **with** $\mid C(p_1, \ldots, p_n) \;\to\; e'_1 \mid \_ \;\to\; d'$, we distinguish two subcases:

– If $v = C(v_1, \ldots, v_n)$ unifies with $C(p_1, \ldots, p_n)$, then let $\rho = mgu(v, C(p_1, \ldots, p_n)) = mgu(v_1, p_1) \circ \ldots \circ mgu(v_n, p_n)$ (the last equality comes from pattern linearity) and both sides reduce to $e'_1\rho$.

– Otherwise, one of the v_i does not unify with its corresponding p_i. Let k be the smallest index such that v_k does not unify with p_k and ρ be the substitution $mgu(v_1, p_1) \circ \ldots \circ mgu(v_{k-1}, p_{k-1})$. The left-hand side reduces to $d'\rho$ and the right-hand side reduces to d'' which β-reduces to d'. Fortunately, by pattern linearity, the patterns p_i can not capture a variable occurring in d' so $d'\rho = d'$. □

8.1.4 Destructors in Dedukti

Destructors are easy to write in Dedukti. Each destructor of a datatype with n constructors is defined by n rewrite rules, one rule for each possible application of the destructor to a constructor of the datatype. This rewrite system is however a bit tedious to formalize in the general case. In order to understand the essence of the definition of destructors in Dedukti, we start with a few special cases.

8.1.4.1 Peano Natural Numbers

We start with the very simple datatype of Peano natural numbers which can be written in FoCaLiZe as

$$\textbf{type}\ \text{nat} = |\ \text{O}\ |\ \text{S}\ (\text{nat}).$$

So we only have two constructors and no polymorphic variable. The produced Dedukti code for this type nat, its constructors O and S, and its destructors D_O and D_S is:

```
nat : Type.
O : nat.
S : nat -> nat.

def D_O : R : Type -> nat -> R -> R -> R.
[b] D_O _ O b _ --> b
[c] D_O _ (S _) _ c --> c.

def D_S : R : Type -> nat -> (nat -> R) -> R -> R.
[c] D_S _ O _ c --> c
[n,b] D_S _ (S n) b _ --> b n.
```

$$\text{nat} : \textbf{Type}.$$

$$\text{O} : \text{nat}.$$
$$\text{S} : \text{nat} \to \text{nat}.$$

$$\textbf{def}\ D_O : \Pi R : \textbf{Type}.\ \text{nat} \to R \to R \to R.$$
$$[R, b, c] \quad D_O\ R\ \text{O} \quad b\ c \longrightarrow b$$
$$[R, n, b, c] \quad D_O\ R\ (\text{S}\ n) \quad b\ c \longrightarrow c.$$

$$\textbf{def}\ D_S : \Pi R : \textbf{Type}.\ \text{nat} \to (\text{nat} \to R) \to R \to R.$$
$$[R, b, c] \quad D_S\ R\ \text{O} \quad b\ c \longrightarrow c$$
$$[R, n, b, c] \quad D_S\ R\ (\text{S}\ n) \quad b\ c \longrightarrow b\ n.$$

As expected, the rewrite system is orthogonal hence confluent.

8.1.4.2 Linear Expressions

To illustrate the case where the datatype has more than two constructors, we now consider linear arithmetic expressions defined by

$$\begin{aligned}\textbf{type}\ \text{lexpr} = \ & |\ \text{Variable}(\textbf{string}) \\ & |\ \text{Constant}(\textbf{int}) \\ & |\ \text{Times}(\textbf{int}, \text{lexpr}) \\ & |\ \text{Plus}(\text{lexpr}, \text{lexpr})\end{aligned}$$

We have four constructors and still no polymorphic variable. Assuming the type of strings as been defined, the generated Dedukti code is the following:

```
lexpr : Type.
Variable : string -> lexpr.
Constant : int -> lexpr.
Times : int -> lexpr -> lexpr.
Plus : lexpr -> lexpr -> lexpr.

def D_Variable : R : Type -> lexpr -> (string -> R) -> R -> R.
[s,b] D_Variable _ (Variable s) b _ --> b s
[c] D_Variable _ (Constant _) _ c --> c
[c] D_Variable _ (Times _ _) _ c --> c
[c] D_Variable _ (Plus _ _) _ c --> c.

def D_Constant : R : Type -> lexpr -> (int -> R) -> R.
[c] D_Constant _ (Variable _) _ c --> c
[n,b] D_Constant _ (Constant n) b _ --> b n
[c] D_Constant _ (Times _ _) _ c --> c
[c] D_Constant _ (Plus _ _) _ c --> c.

def D_Times : R : Type -> lexpr -> (int -> lexpr -> R) -> R.
[c] D_Times _ (Variable _) _ c --> c
[c] D_Times _ (Constant _) _ c --> c
[n,e,b] D_Times _ (Times n e) b _ --> b n e
[c] D_Times _ (Plus _ _) _ c --> c.

def D_Plus : R : Type -> lexpr -> (lexpr -> lexpr -> R) -> R.
[c] D_Plus _ (Variable _) _ c --> c
[c] D_Plus _ (Constant _) _ c --> c
[c] D_Plus _ (Times _ _) _ c --> c
[e1,e2,b] D_Plus _ (Plus e1 e2) b _ --> b e1 e2.
```

For the more general case of a monomorphic datatype defined by

$$\textbf{type}\ \text{a} = |\ C_1(\tau_{1,1}, \ldots, \tau_{1,k_1})\ |\ \ldots\ |\ C_n(\tau_{n,1}, \ldots, \tau_{n,k_n}),$$

each destructor is defined by an orthogonal set of rules, one rule per constructor; the term $D_{C_i}\ R\ (C_j\ e_1\ \ldots\ e_{k_j})\ b\ c$ reduces to $b\ e_1\ \ldots\ e_{k_j}$ if $i \equiv j$ and to c otherwise.

8.1.4.3 Polymorphic Lists

To illustrate the impact of polymorphism to the definition of destructors, we consider the datatype of polymorphic lists defined by

type list(α) = | Nil | Cons (α, list(α))

The generated Dedukti code is as follows:

```
list : Type -> Type.

Nil : a : Type -> list a.
Cons : a : Type -> a -> list a -> list a.

def D_Nil : R : Type -> a : Type -> list a -> R -> R -> R.
[b] D_Nil _ _ (Nil _) b _ --> b
[c] D_Nil _ _ (Cons _ _ _) _ c --> c.

def D_Cons :
   R : Type -> a : Type -> list a -> (a -> list a -> R) -> R -> R.
[c] D_Cons _ _ (Nil _) _ c --> c
[e,l,b] D_Cons _ _ (Cons _ e l) b _ --> b e l.
```

8.1.4.4 General Case

Finally, in the general case

$$\textbf{type}\ \mathrm{a}(\alpha_1, \ldots, \alpha_k) = \mid C_1(\tau_{1,1}, \ldots, \tau_{1,k_1}) \mid \ldots \mid C_n(\tau_{n,1}, \ldots, \tau_{n,k_n}),$$

we get the following rewrite system:

a : **Type** $\to \ldots \to$ **Type** $\to$ **Type**.

$C_1 : \Pi\alpha_1, \ldots, \alpha_k :$ **Type**. $\tau_{1,1} \to \ldots \to \tau_{1,k_1} \to$ a $\alpha_1 \ \ldots \ \alpha_k$.
$\ldots$
$C_n : \Pi\alpha_1, \ldots, \alpha_k :$ **Type**. $\tau_{n,1} \to \ldots \to \tau_{n,k_n} \to$ a $\alpha_1 \ \ldots \ \alpha_k$.

def $D_{C_1} : \Pi R, \alpha_1, \ldots, \alpha_k :$ **Type**. a $\alpha_1 \ \ldots \ \alpha_k \to (\tau_{1,1} \to \ldots \to \tau_{1,k_1} \to R) \to R \to R$.
$[e_1, \ldots, e_{k_1}, b]$ D_{C_1} _ _ $\ldots$ _ (C_1 _ $\ldots$ _ $e_1 \ \ldots \ e_{k_1}$) b _ $\longrightarrow b \ e_1 \ \ldots \ e_{k_1}$
$[c]$ D_{C_1} _ _ $\ldots$ _ (C_2 _ $\ldots$ _) _ $c \longrightarrow c$
$\ldots$
$[c]$ D_{C_1} _ _ $\ldots$ _ (C_n _ $\ldots$ _) _ $c \longrightarrow c$.
$\ldots$
def $D_{C_n} : \Pi R, \alpha_1, \ldots, \alpha_k :$ **Type**. a $\alpha_1 \ \ldots \ \alpha_k \to (\tau_{n,1} \to \ldots \to \tau_{n,k_n} \to R) \to R \to R$.
$[c]$ D_{C_n} _ _ $\ldots$ _ (C_1 _ $\ldots$ _) _ $c \longrightarrow c$
$\ldots$
$[c]$ D_{C_n} _ _ $\ldots$ _ (C_{n-1} _ $\ldots$ _) _ $c \longrightarrow c$
$[e_1, \ldots, e_{k_n}, b]$ D_{C_n} _ _ $\ldots$ _ (C_n _ $\ldots$ _ $e_1 \ \ldots \ e_{k_n}$) b _ $\longrightarrow b \ e_1 \ \ldots \ e_{k_n}$.

8.2 Recursive Functions

Recursion is a powerful but subtle feature in FoCaLiZe; it raises a number of issues among which:

- Termination of Zenon:

  Zenon might need to unfold function definitions to complete a proof. In the case of recursive functions, even terminating ones, this unfolding process has to be used with parsimony otherwise Zenon could diverge. FoCaLiZe activates a Zenon extension called **induct** which performs a special treatment for recursive definitions but it has not been ported to Zenon Modulo; Zenon Modulo expects its rewrite system to be strongly terminating and normalizes terms eagerly.

- Termination proofs:

  Coq is a strongly terminating system, it offers two mechanisms for defining recursive functions:

  – Structural recursion with the keywords **fix** and **Fixpoint** and

  - General recursion with the keywords **Function** and **Program**. In this case, the user is asked to provide a Coq proof of the termination of the function.

  FoCaLiZe uses both structural definition by **Fixpoint** and general definitions by **Function**. In the case of **Function**, the user can prove the termination of the function in FoCaLiZe with the assistance of Zenon [71].

  In Dedukti, there are two reasons for which termination of the $\beta\Gamma$ relation is expected:

  - *completeness* of the type-checking algorithm heavily depends on it;
  - confluence is easier to prove when the rewrite system is known to be strongly terminating, confluence is used to prove *correctness* of the type-checking algorithm;

  Since we are using Dedukti to recheck proofs, not developing new proofs directly in Dedukti, completeness of type-checking is not vital. Correctness however is wanted so we would like to preserve termination of FoCaLiZe terminating programs but, contrary to the Coq backend, it is not mandatory to support termination proofs coming from the system; these termination proofs are simply dropped by Focalide.

- Induction

  Certifying recursive functions usually requires induction principles which are difficult to integrate in a first-order theorem prover such as Zenon because induction principles are second-order formulae. The **induct** extension supports a bit of higher-order reasoning but we cannot rely on it in Zenon Modulo because is very hard to adapt to typing (see Section 9.3).

This work is based on the encoding of recursion in Dedukti defined in the context of Coqine [11, 28]. The situation compared to Coqine is simplified by the absence of dependent types in input but also made more complex by the generality of the recursion: in Coq kernel, termination of recursive functions is guaranteed by very restrictive syntactic side conditions which are not imposed in FoCaLiZe. In the next section, we will look at some idiomatic examples of recursive definitions in FoCaLiZe. We will then show why the translation of recursive definitions cannot be handled as easily as the translation of

non-recursive definitions. We will then present the translation of recursive definitions implemented in Focalide and finally discuss its behaviour in terms of termination, size and efficiency of the generated Dedukti code.

8.2.1 Examples

We start with a few examples of recursive definitions in FoCaLiZe, aiming at illustrating the diversity of styles allowed by FoCaLiZe. For simplicity, we want to treat all these examples in a uniform way.

8.2.1.1 Factorial

We start with the usual example of the factorial function. In FoCaLiZe, like in many functional languages, functions on natural numbers are usually implemented using the built-in type **int** which gets translated to OCaml's machine integers, implicitly assuming no overflow. The type **nat** of natural numbers would have a better inductive structure but leads to exponentially larger values.

On negative values, the behaviour of the factorial function is not important; we arbitrarily choose to fix it to the value 1.

```
let rec fact (n) = if n < 2 then 1 else n * fact (n - 1)
```

Obviously, the recursive call of `fact` is not performed on a subterm of the argument because there is no notion of subterm for the built-in type **int**. So we can not restrict our attention to structural recursion. Moreover, since the only argument to the factorial function is typed by a built-in type, we can not inspect its definition (actually, the type **int** has several definitions, one for each backend) and look at the behaviour of the function on applied constructors. For the Dedukti backend, the definition of **int** is the definition we gave in Section 3.4.1:

```
int : Type.
def Diff : nat -> nat -> int.
[m,n] Diff (S m) (S n) --> Diff m n.
```

```
type list ('a) =
| Nil | Cons ('a, list ('a));;

let rec equal (l1, l2) =
  match l1 with
  | Nil -> (match l2 with
             | Nil -> true
             | Cons (_, _) -> false)
  | Cons (h1, t1) -> (match l2 with
                       | Nil -> false
                       | Cons (h2, t2) ->
                           (h1 = h2) && equal (t1, t2));;
```

Figure 8.3: Constructor-based equality of lists

8.2.1.2 Equality of Lists

Structural recursion on a known datatype is not enough but it is fortunately also possible. Our second example illustrates structural recursion on the usual datatype of lists. In Figure 8.3, we define equality on lists using pattern-matching.

In FoCaLiZe however, we also have the opportunity to define list equality in a more general setting by abstracting over the concrete representation of lists and requiring only the necessary functions used to check equality. This definition of equality of lists replaces pattern matching by calls to the projections `head` and `tail` so we refer to it as the projection-based definition of equality of lists. This approach is illustrated in Figure 8.4.

Similarly to the case of the factorial function, the arguments of the recursive function are now inhabitants of an unknown (or not-yet known) type: **Self**; the recursive calls are not performed on syntactic subterms but on more complex terms (`n-1` in the case of the factorial function, `tail(l)` here). This style is encouraged in FoCaLiZe because it makes sharing of definitions and proofs easier. We can for example prove that `equal` is an equivalence relation and this fact will be available for each possible implementation of lists (as long as we do not redefine equality).

However, this generality comes at a price: the `equal` function can not be proved ter-

```
species List (A is Setoid) =
  inherit Setoid;
  signature nil : Self;
  signature cons : A -> Self -> Self;
  signature is_nil : Self -> bool;
  signature head : Self -> A;
  signature tail : Self -> Self;

  property surjective_pairing :
    all l : Self, ~ (is_nil(l)) <-> l = cons(head(l), tail(l));

  property head_proj :
    all a : A, all l : Self, head (cons (a, l)) = a;

  property tail_proj :
    all a : A, all l : Self, tail (cons (a, l)) = l;

  property is_nil_nil : is_nil(nil);

  let rec equal (l1, l2) =
    (is_nil (l1) && is_nil (l2)) ||
    (~~ is_nil(l1) && ~~ is_nil(l2) &&
     A!equal(head(l1), head(l2)) && equal(tail(l1), tail(l2)));
 end;;
```

Figure 8.4: Projection-based equality of lists

minating because we can not prove that the tail of a list is smaller than the list. Actually, there is an implementation of the `List` species in which this recursive definition of `equal` diverges: our definition of lists does not rule out possibly infinite streams. The type of possibly infinite streams can be defined in FoCaLiZe (see Figure 8.5).

8.2.2 Naive Translation

Let f be a recursive function, we assume that it is defined as

```
let rec f (x) = g(f(h(x)), x)
```

If we just ignore the **rec** keyword, we have two simple ways of translating the definition of f in Dedukti:

```
[ ] f --> x : A => g (f (h x)) x.
```

or

```
[x] f x --> g (f (h x)) x.
```

In the second case, the term f itself does not reduce until it is applied to an argument. Unfortunately, in both cases the term $f\ a$ where a is a term of type A diverges. Even if the type A is empty, since reduction is allowed under λ-abstraction, the term $\lambda x.\ f\ x$ diverges.

To recover termination, we need to restrict the shape of arguments allowed to unfold the recursive definition of f.

8.2.3 Call-by-Value Application Combinator

Infinite unfolding of recursive definitions is avoided in FoCaLiZe computational language by a syntactic condition: unfolding of a recursive function is performed only after all its arguments have been reduced to values. We want to mimick part of this behaviour to avoid useless divergence but we can not hope to fully reflect the **call-by-value** semantics in our shallow embedding for several reasons:

- checking that a term is a value costs linear time; performing such a test at each unfolding would make the generated code very inefficient;

```
type stream ('a) =
  | Finite (list ('a)) (* finite streams are lists *)
  | Infinite (int -> 'a)
     (* Infinite(f) represents the infinite stream
        f(0), f(1), f(2), ... *);;

species Stream (A is Setoid) =
  inherit List(A);

  representation = stream (A);

  let nil = Finite (Nil);
  let cons (a, l) =
    match l with
      | Finite (tl) -> Finite (Cons(a, tl))
      | Infinite (f) ->
          Infinite
             (function i -> if i = 0 then a else f(i - 1));

  let is_nil (l) =
    match l with
      | Finite (Nil) -> true
      | _ -> false;

  let head (l) =
    match l with
      | Finite (Nil) -> A!element
      | Finite (Cons(a, _)) -> a
      | Infinite (f) -> f(0);

  let tail (l) =
    match l with
      | Finite (Nil) -> l
      | Finite (Cons(_, tl)) -> Finite (tl)
      | Infinite (f) -> Infinite (function i -> f(i + 1));
 end;;
```

Figure 8.5: Definition of possibly infinite streams in FoCaLiZe

- beeing a value is a property which is not stable by substitution so defining a predicate for testing it in a confluent manner is not compatible with the preservation of substitution by the embedding;

- to reason about a recursive definition requires unfolding of the definition on open terms, for example if we define the concatenation of lists by

```
let rec append (l1, l2) = match l1 with
  | Nil => l2
  | Cons(a, l) => Cons(a, append(l,l2))
```

  then proving the theorem $\forall l, \texttt{append}(\texttt{Nil}, l) = l$ requires an unfolding of the definition on the open term $\texttt{append}(\texttt{Nil}, l)$.

Since we can not inspect the definition of types to decide wether or not we should unfold the recursive definition, we delegate this to the rewrite system defining the type. This takes the form of a combinator `CBV` of type `A : Type -> B : Type -> (A -> B) -> A -> B` which is defined by ad-hoc polymorphism on type `A`. The definition of `CBV` is extended each time a new datatype is defined.

This combinator should behave like application on values, should not reduce on most non-values (especially on variables) and should imply only a constant-time overhead.

Here is the definition of `CBV` for type `int`:

```
def CBV : A : Type -> B : Type -> (A -> B) -> A -> B.

[m,n,f] CBV int _ f (Diff m n) --> f (Diff m n).
```

For algebraic datatypes, `CBV` is defined by giving a rewrite rule for each constructor. Here is the definition for the algebraic type `nat`:

```
[f]   CBV nat _ f O     --> f O.
[n,f] CBV nat _ f (S n) --> f (S n).
```

In the general case

$$\textbf{type}\ \mathrm{a}(\alpha_1, \ldots, \alpha_k) = \mid C_1(\tau_{1,1}, \ldots, \tau_{1,k_1}) \mid \ldots \mid C_n(\tau_{n,1}, \ldots, \tau_{n,k_n}),$$

`CBV` is defined by the rewrite system

$[f, \alpha_1, \ldots, \alpha_k, e_1, \ldots, e_{k_1}]$
`CBV` $(a\ \_\ \ldots\ \_)\ \_\ f\ (C_1\ \alpha_1\ \ldots\ \alpha_k\ e_1\ \ldots\ e_{k_1}) \longrightarrow f\ (C_1\ \alpha_1\ \ldots\ \alpha_k\ e_1\ \ldots\ e_{k_1})$
$[f, \alpha_1, \ldots, \alpha_k, e_1, \ldots, e_{k_n}]$
`CBV` $(a\ \_\ \ldots\ \_)\ \_\ f\ (C_n\ \alpha_1\ \ldots\ \alpha_k\ e_1\ \ldots\ e_{k_n}) \longrightarrow f\ (C_n\ \alpha_1\ \ldots\ \alpha_k\ e_1\ \ldots\ e_{k_n})$

Now that we dispose of a combinator for freezing evaluation until the arguments of the recursive functions start with a constructor, we can resume to our definition of the recursive function f:

```
[x] f x --> g (f' (h x)) x.
[x] f' x --> CBV A B f x.
```

As expected, the symbol f alone does not reduce, it is unfolded exactly once when it is applied to a variable, it is fully unfolded if f is applied to a value and is partially unfolded if f is applied to a non-value starting with a constructor (just unfolded enough to reason about recursive unfoldings in an abstract way).

For example, remember the FoCaLiZe recursive definition of the factorial function:

```
let rec fact (n) = if n < 2 then 1 else n * fact (n - 1)
```

This recursive definition gets translated to the following rewrite system in Dedukti:

```
def fact : int -> int.
def fact' : int -> int.
[n] fact n -->
   if (lt n (Diff 2 0))
      (Diff 1 0)
      (mult n (fact' (minus n (Diff 1 0)))).
[n] fact' n --> CBV int int fact n.
```

The term `fact n` normalizes to
`if (lt n (Diff 2 0)) (Diff 1 0) (mult n (CBV int int fact (minus n (Diff 1 0))))`. To continue the evaluation of the term, we have to substitute the variable `n` by a term starting with the smart constructor of integers `Diff`. The term `fact (Diff 2 0)` is evaluated as follows:

```
                        if (lt (Diff 2 0) (Diff 2 0))
                          (Diff 1 0)
fact (Diff 2 0) ⟶*        (mult (Diff 2 0)                    ⟶*
                            (CBV int int fact
                              (minus (Diff 2 0) (Diff 1 0))))

mult (Diff 2 0)
  (CBV int int fact  ⟶  mult (Diff 2 0)      ⟶*
    (Diff 1 0))           (fact (Diff 1 0))

mult (Diff 2 0)
  (Diff 1 0)      ⟶* Diff 2 0
```

8.2.4 Local Recursion

We have seen in previous section how toplevel recursion is translated. Local recursion is not harder, we define a constant `Fix` for translating the μ binder:

```
Fix : A : Type -> B : Type -> ((A -> B) -> A -> B) -> A -> B.
[A, B, F, a] Fix A B F a --> CBV A B (F (Fix A B F)) a.
```

8.2.5 Termination

The `CBV` combinator is only an approximation of the call-by-value strategy which is intentionally incomplete for efficiency reasons. In the pathological case where the function `h` reduces to a term starting with a constructor, we still obtain a diverging rewrite system, even if the original code was terminating with respect to the call-by-value semantics. At the expense of loosing complexity preservation, we can really check that arguments are variables.

To encode value checking, we send a signal through terms which is only allowed to come back if the term is actually a value. We use three new constants for this: one for sending the signal down the value, one for waiting for the signal echo and one for sending the signal back toward the root of the term:

```
def ping : A : Type -> A -> A.
def wait : A : Type -> A -> A.
def pong : A : Type -> A -> A.
```

For each constructor C of type $(\tau_1, \ldots, \tau_n) \to \tau$[1], we add the following rules where n is the arity of C:

```
[x_1, ..., x_n] ping tau (C x_1 ... x_n) -->
                wait tau (C (ping tau_1 x_1) ... (ping tau_n x_n)).
[x_1, ..., x_n] wait tau (C (pong tau_1 x_1) ... (pong tau_n x_n))
                  -->
                pong tau (C x_1 ... x_n).
```

In particular for constructors of arity 0, we get the rewrite rules

```
[] ping tau C --> wait tau C.
[] wait tau C --> pong tau C.
```

so the signal comes back when it reaches constructors of arity 0 such as `0`: `ping nat 0` $\longrightarrow$ `wait nat 0` $\longrightarrow$ `pong nat 0`.

The same idea can be used for values of built-in types `int`, `char`, and `string`:

```
[m] ping int (Diff m 0) --> wait int (Diff (ping nat m) 0).
[n] ping int (Diff 0 n) --> wait int (Diff 0 (ping nat n)).
[m] wait int (Diff (pong nat m) 0) --> pong int (Diff m 0).
[n] wait int (Diff 0 (pong nat n)) --> pong int (Diff 0 n).

[n] ping char (char_make n) --> wait char (char_make (ping nat n)).
[n] wait char (char_make (pong nat n)) --> pong char (char_make n).

[] ping string "" --> wait string "".
[c, s] ping string (string_cons c s) -->
    wait string (string_cons (ping char c) (ping string s)).
[] wait string "" --> pong string "".
[c, s] wait string (string_cons (pong char c) (pong string s)) -->
    pong string (string_cons c s).
```

For abstractions, which are values too, checking that they are values is even simpler as we do not need to go through the `wait` state:

```
[A, B, f] ping (A -> B) (x => f x) --> pong (A -> B) (x => f x).
```

Thanks to value checking, we can recover termination by changing the definition of `CBV` as follows:

```
def CBV : A : Type -> B : Type -> (A -> B) -> A -> B.
def CBV_wait : A : Type -> B : Type -> (A -> B) -> A -> B.
[A, B, f, x] CBV A B f x --> CBV_wait A B f (ping A x).
[A, B, f, x] CBV_wait A B f (pong A x) --> f x.
```

[1] we only consider the monomorphic case for simplicity, adding polymorphism does not bring any difficulty

In Focalide, we never encountered an interesting terminating FoCaLiZe function which got translated to a diverging rewrite system so we did not implement value checking.

8.2.6 Efficiency and Limitations

The size of the code produced by Focalide is linear with respect to the input, the operational semantics of FoCaLiZe is preserved and each reduction step in the input language corresponds to a bounded number of rewriting steps in Dedukti, so the execution time for the translated program is linear.

Our treatment of recursive definitions generalizes directly to mutual recursion. However, only toplevel recursive definitions and recursive methods are accepted; local recursive definitions are not handled because in Dedukti the rewrite system is only defined at toplevel. This limitation does not reduce the expressive power of the language because local recursive definitions can always be λ-lifted to toplevel but this lifting has not yet been implemented in Focalide.

Moreover, the understanding of datatypes by Zenon Modulo is still incomplete; it is able to perform computation using the rewrite rules defining destructors but it is not yet able to reason about datatypes by induction or even case distinction; nor is it able to prove injectivity and distinctness of constructors. These properties still need to be proved directly in Dedukti until Focalide is able to automatically generate them from the datatype definition.

8.3 Related Work

- Coqine

  In the context of Coqine, a translator from a fragment of Coq to Dedukti, Assaf [11] has proposed several techniques to compile recursive functions and pattern matching in Dedukti.

  Pattern matching is limited in Coq kernel to flat patterns so it is possible, in the context of Coqine, to define a single `match` symbol for each inductive type, which simplifies greatly the compilation of pattern matching to Dedukti and avoids the use

of dynamic error handling.

However, it does not seem possible to define a single `fix` symbol without breaking strong normalization of the rewrite system so, as in our work, each fixpoint has to be named and recursive unfolding has to be limited to expressions starting with a constructor. Assaf distinguishes two ways to achieve this; we can either wrap each constructor as proposed in [28] or use a combinator similar to `CBV` (called a filter function in [11]). Because of dependent typing, function arguments have to be duplicated when using the latter solution so it is unclear which solution (wrapping constructors or duplicating arguments) is the best in the context of Coqine. In our case, the input type system does not feature dependent types so this duplication of argument is unnecessary.

- Termination of programs using rewrite systems

  A lot of work has been done to compile programs (especially functional recursive definitions [79, 76, 117]) to rewrite systems. The focus has often been on termination preserving translations to prove termination of recursive functions using termination checkers for TRSs. However, these translations do not try to preserve the semantics of the programs so they can hardly be adapted for handling translations of correctness proofs.

- Compilation of pattern matching to λ-calculus

  The semantics of functional languages often rely on λ-calculus. Pattern matching is a common feature in these languages so proving the correction of a compiler for a functional language usually require to define a translation function from pattern matching to λ-calculus. This has been achieved by enriching the λ-calculus with simple forms of pattern matching. These enriched λ-calculi are then used as intermediate compilation languages between the rich functional language and the low-level λ-calculus.

  - In [145], Peyton-Jones and Walder extend the λ-calculus with an abstraction over pattern and internalize the list of patterns using a [] operator. Matching failure is represented by the constant **FAIL** which is left-neutral for [] and

non-exhaustiveness is represented by the constant **ERROR**. We avoid the introduction of constants **FAIL** and [] for tracking matching failure and so we avoid the appearance of some alien terms such as **FAIL** + 2. In our work, failure is replaced by the default behaviour of destructors. However, we still rely on a dynamic error mechanism to test exhaustiveness of pattern coverage whereas this property can be checked statically and even reduced to type-checking [105].

- In [106], Oostrom, Klop, and Vrijer generalize the enriched λ-calculus of Peyton-Jones and Walder; they define an other extension of the λ-calculus, the λ-calculus with patterns, generalizing the shape of λ-terms allowed to build abstractions from variables to terms verifying the *Rigid Pattern Condition*. However, they restrict their attention to uniform patterns, in the sense that the order of the branches of pattern matching should not matter, which we find too restrictive in the context of the compilation of functional languages in general and FoCaLiZe in particular.
- More recently, Wolfram Kahl introduced [102] the Pattern Matching Calculus, focusing on the notion of matchings (patterns, possibly fed with arguments) constituting a grammatical class distinct from terms. Like Peyton-Jones and Walder, [] and **FAIL** are part of the calculus but matching success is easier to detect and alien terms are harder to produce.

Following [145], we could add optimization steps to replace destructors by eliminators (called case-expressions in [145]) which are considered more efficient and would limit the use of dynamic errors, in particular in the common case where, like in our first example of equality over lists, the only pattern matchings used in the source file are eliminators. However, we believe that keeping destructors is the best choice when the last pattern of the matching is universal (a variable or a wildcard), in which case we do not emit any **ERROR**.

Contrary to advanced techniques targeting at the efficiency of the produced code such as [121], we obtain a light translation, close to the compilation to Coq, predictable by the programmer, and simple enough to be supported by Zenon.

Chapter 9

Logical Part: Interfacing FoCaLiZe with Zenon Modulo

As already mentioned, Zenon Modulo is an extension of the first-order theorem prover Zenon to Deduction modulo (hence the name). Checking Deduction modulo proofs requires to express rewriting in the proof checker which naturally leads to replace Coq by Dedutki as the backend proof checker. This however also requires to add typing in Zenon, we present this work in Section 9.1. The built-in first-order theory of FoCaLiZe is integrated into Zenon in the form of two Zenon extensions, the FoCaLiZe extension and the induction extension. We present these extensions and the work needed to adapt them to our context in Sections 9.2 and 9.3. Finally, the translation of pattern matching and recursion that we have presented in the previous chapter escapes slightly the framework of Deduction modulo because it relies on higher-order functions. In Section 9.4, we discuss our implementation of higher-order right-hand sides in Zenon Modulo rewrite rules.

9.1 Extending Zenon to Typing

Zenon is a prover for the classical monosorted first-order logic. It might seem surprising to plug an untyped prover in the FoCaLiZe backend to Coq since both FoCaLiZe and Coq are typed systems. This is not an issue in practice for the following reasons:

- Zenon ignores but preserves type annotations on quantifiers,
- Zenon is not expected to be trusted; if it produces an ill-typed (from Coq point of

view) proof, Coq fails and the theorem is not considered proven,

- missing type information in a well-typed Zenon proof can in most cases be inferred by Coq.

The two first points are equally valid in the context of Zenon Modulo and Dedukti but the third one is not because Dedukti performs almost no type inference.

FoCaLiZe is not the only typed system discharging proofs to Zenon or Zenon Modulo. Zenon Modulo has been developed in the context of the BWare project [62] in which it is used to provide checkable proofs for industrially produced proof obligations in Atelier B, whose logical foundations are a typed version of set theory. In this context, Zenon Modulo uses heuristics to turn most axioms of set theory into rewrite rules [61, 38]. As we have seen in Section 1.3, only universally quantified equalities and equivalences can be turned into rewrite rules so translating typed formulae to untyped ones is not an option because the translation does not preserve these shapes (translated formulae are universally quantified implications).

Lack of typing in Zenon also affected the ability to extend Zenon to arithmetic [38], even in the context of Coq-checked proofs, because the meaning of the ordering relation greatly depends on the type of the compared elements: $n < m$ means $n+1 \leq m$ for natural numbers and integers but not for rational numbers so we need ad-hoc polymorphism for understanding arithmetic formulae.

The extension of Zenon and Zenon Modulo to typing is a joint work with Halmagrand, the developer of Zenon Modulo, and Bury, the developer of Zenon Arith, an extension of Zenon to arithmetic. We have implemented the polymorphic extension of first-order logic of Section 1.1.3. The implementation details have been published in [37].

The parts of this work which most directly affect Focalide are the following:

- Parsing

  Contrary to the TFF1 format used for BWare and arithmetic problems, there is no syntactic distinction in our input format (the fragment of Dedukti produced by Focalide) between variables and constants. We cannot force Focalide to make a

syntactic difference because what is a variable in a problem can become a constant in another problem (if x is a constant in the statement of a proof step s, then it is a variable in the statement of the proof step **assume** $x : \tau\ s$). For this reason, we need to distinguish a parsing, a scoping, and a typing pass in the case of the Dedukti input.

- Extensions

  Zenon is an extensible prover and two extensions are used by FoCaLiZe: the so-called FoCaLiZe extension and the induction extension. These extensions are not used by Zenon Arith and Zenon Modulo in the context of BWare. To be useable with a typed version of Zenon Modulo, these extensions need to be typed (types have to be declared for the symbols introduced in these extensions). This was easy for the FoCaLiZe extension but too hard for the induction extension.

We now detail the other aspects of these two extensions.

9.2 The FoCaLiZe Extension

The FoCaLiZe extension aims at efficiency of the behaviour of Zenon on FoCaLiZe built-in type **bool**. Contrary to other classical systems such as HOL and PVS, **bool** is not identified with the type **Prop** of formulae (which is not part of FoCaLiZe syntax but is the type of formulae in both Coq and Dedukti). FoCaLiZe atoms are injected into the syntax of formulae using the unary predicate symbol **istrue** of type **bool** $\rightarrow$ **Prop**. For efficiency reasons, the FoCaLiZe extension duplicates every **bool**-valued function symbol f as a fresh predicate symbol **istrue**f** of the same arity. The extension also integrates a lot of lemmata on the behaviour of **istrue** with common operations on **bool** (negation, conjunction, disjunction, exclusive disjunction, equivalence, alternative, equality).

Apart from adding all the necessary type information, we also described the computational behaviour of projectors of Cartesian product by rewrite rules.

9.3 The Induction Extension

The induction extension adds the treatment of datatypes, computation behaviour of pattern matching, discrimination and injectivity of constructors, and the ability of instantiating induction principles. This extension is a higher-order extension in two respects:

- instantiation of induction principles requires second-order reasoning,
- pattern matching contains binding and computing with pattern matching requires to perform substitutions.

The first point is forbidden by the scoping policy: while there is *a priori* no reason to forbid quantification over **bool**-valued functions, we do not allow to apply such quantified functions to arguments because variables cannot be applied.

The second point is also very problematic because the encoding of pattern-matching is untyped and hard to type.

For these reasons, Focalide does not call the induction extension but replace (part of) its features with Deduction modulo.

9.4 Higher-Order Right-Hand Sides

Zenon is a first-order theorem prover so it will refuse rewrite rules such as the ones defining destructors and `CBV` because they are of the form $[\ldots]F(\ldots, f, x) \longrightarrow f(x)$ which does not fit in the scope of first-order Deduction modulo because the left-hand-side parameter f is used as a function symbol in the right-hand-side.

We can limit the problem to only one rewrite rule by introducing an explicit higher-order polymorphic application symbol @ of type $\Pi\alpha_1, \alpha_2.\ (\alpha_1 \to \alpha_2, \alpha_1) \to \alpha_2$ and replacing the above rule by $[\ldots]F(\ldots, f, x) \longrightarrow @(\ldots, f, x)$.

In order for this rule to have the expected behaviour, we need to force some reasoning modulo β-reduction, that is, we want the terms $@\ (\alpha_1, \alpha_2; \lambda x : \alpha_1.t, a)$ and $t\{x \backslash a\}$ to be convertible from Zenon Modulo's point of view. There are (at least) two ways to do so:

- Add a deduction rule for converting between $@\ (\alpha_1, \alpha_2; \lambda x : \alpha_1.t, a)$ and $t\{x \backslash a\}$

- Maintain the following invariant on the terms manipulated by Zenon Modulo: the first term argument of @ is a variable. The only place where this invariant is susceptible to be broken is in the substitution function, which has to be modified in order to perform β-reduction when the first term argument of @ is substituted by an abstraction.

The first option introduces many proof steps which correspond to nothing in the back-end checker but the second option might slow down the substitution function and is less modular so it can impact the performances of Zenon Modulo, even on pure first-order problems. We have chosen and implemented the second option.

Conclusion of Part III

We have extended the FoCaLiZe compiler to a new output language: Dedukti. Contrary to previously existing FoCaLiZe outputs OCaml and Coq, Dedukti is not a functional programming language but an extension of a dependently-typed λ-calculus with rewriting so pattern matching and recursion are not trivial to compile to Dedukti.

However, we have shown that ML pattern matching can easily and efficiently be translated to Dedukti using destructors. The compilation of pattern matching can be further optimized, in particular to limit the use of dynamic error handling. For recursion, however, efficiency comes at a cost in term of normalization because we can not fully enforce the use of the call-by-value strategy without loosing preservation of the complexity of the source code.

Our approach is general enough to be adapted to other functional languages because FoCaLiZe language for implementing functions is an ML language without specific features. FoCaLiZe originality comes from its object-oriented mechanisms which are invisible to Focalide because they are statically resolved in an earlier compilation pass. Moreover, it can also easily be adapted to other rewriting formalisms, especially untyped and polymorphic rewrite engines because features specific to Dedukti (such as higher-order rewriting or dependent typing) are not used.

In the introduction of this part, we claimed that Focalide would have better performances than the Coq backend because Dedukti is a lightweight proof checker compared to Coq and because Zenon Modulo is more efficient at deductive program verification than Zenon.

We can split this claim in two parts. First we want to compare Focalide to the Coq

backend on existing FoCaLiZe developments to evaluate the possible gain in performances that the FoCaLiZe users can expect by replacing the Coq backend by Focalide. Second we give an example of a problem which becomes solvable by using Focalide. This problem is almost instantly solved by Focalide but extremely slow on the Coq side.

We evaluate Focalide by running it on different available FoCaLiZe developments. When proofs required features which are not yet implemented in Focalide, we commented the problematic lines and ran both backends on the same input files; the coverage column of Figure 9.1 indicates the percentage of remaining lines.

FoCaLiZe ships with three libraries: the standard library (stdlib) which defines a hierarchy of species for setoids, Cartesian products, disjoint unions, orderings and lattices, the external library (extlib) which defines mathematical structures (algebraic structures and polynomials) and the user contributions (contribs) which are a set of concrete applications. Unfortunately, none of these library uses pattern matching and recursion extensively so the fact that Focalide gives comparable or better results than the old backend is reassuring but does not tell much about the validity of our approach.

The other developments are more interesting in this respect; they consist of a test suite for termination proofs of recursive functions (term-proof), a pedagogical example of FoCaLiZe features with several examples of functions defined by pattern matching (ejcp) and a specification of Java-like iterators together with an implementation by lists using both recursion and pattern matching (iterators).

The results, shown in Figure 9.1 and Figure 9.2, show that on FoCaLiZe problems Zenon Modulo is about twice slower than Zenon, which is not very bad considering that Zenon has been optimized for FoCaLiZe since its beginning. The produced Dedukti code (column "Dedukti" in 9.1) is about twice bigger than its Coq counterpart (column "Coq" in 9.1), which is not very surprising because Coq is a proof assistant encouraging to omit information when it can be inferred; Dedukti however is a mere checker and provides almost no inference so its input is more verbose but the Focalide user gets a huge speedup in proof-checking time. Moreover, each time Coq checks a file coming from FoCaLiZe, it has to load a significant part of its standard library which often takes the majority of the checking time (about a second per file). In the end, finding a proof and checking it is

Library	FoCaLiZe	Coverage	Coq	Dedukti
stdlib	163335	99.42%	1314934	4814011
extlib	158697	100%	162499	283939
contribs	126803	99.54%	966197	2557024
term-proof	24958	99.62%	227136	247559
ejcp	13979	95.16%	28095	239881
iterators	80312	88.33%	414282	972051

Figure 9.1: Size (in bytes) comparison of Focalide with the old backend on available FoCaLiZe developments

Library	Zenon	ZMod	Coq	Dedukti	Zenon + Coq	ZMod + Dedukti
stdlib	11.73	32.87	17.41	1.46	29.14	34.33
extlib	9.48	26.50	19.45	1.64	28.93	28.14
contribs	5.38	9.96	26.92	1.17	32.30	11.13
term-proof	1.10	0.55	24.54	0.02	25.64	0.57
ejcp	0.44	0.86	11.13	0.06	11.57	0.92
iterators	2.58	3.85	6.59	0.27	9.17	4.12

Figure 9.2: Time (in seconds) comparison of Focalide with the old backend on available FoCaLiZe developments

usually faster when using Focalide.

These files have been developed prior to Focalide so they do not yet benefit from Deduction modulo as much as they could. The Coq backend going through Zenon is not very efficient on proofs requiring computation because all reduction steps are registered as proof steps in Zenon leading to huge proofs which take a lot of time for Zenon to find and for Coq to check. For example, if we define a polymorphic datatype `type wrap ('a) = | Wrap ('a)`, we can define the isomorphism `f : 'a -> wrap('a)` by `let f (x) = Wrap(x)` and its inverse `g : wrap('a) -> 'a` by `let g(y) = match y with | Wrap (x) -> x`. The time taken for our tools to deal with the proof of $(\mathtt{g} \circ \mathtt{f})^n(x) = x$ for n from 10 to 19 is given in Figure 9.3; as we can see, the Coq backend becomes quickly unusable whereas deduction modulo is so fast that it is even hard to measure it.

We also claimed at several occasions that FoCaLiZe and Focalide could be used as an

Value of n	Zenon	Coq	Zenon Modulo	Dedukti
10	31.48	4.63	0.04	0.00
11	63.05	11.04	0.04	0.00
12	99.55	7.55	0.05	0.00
13	197.80	10.97	0.04	0.00
14	348.87	1020.67	0.04	0.00
15	492.72	1087.13	0.04	0.00
16	724.46	$>$ 2h	0.04	0.00
17	1111.10	1433.76	0.04	0.00
18	1589.10	$>$2h	0.07	0.00
19	2310.48	$>$2h	0.04	0.00

Figure 9.3: Time comparison (in seconds) for computation-based proofs

interoperability platform for the exchange of proofs between different logical systems. In the next part of this dissertation, we will make this claim more precise by proposing an interoperability methodology based on Focalide.

Part IV

Object-Oriented Interoperability between Logical Systems

Formalization of mathematics is a very expensive task. The successes of the field – the four-color theorem [84], Feit-Thomson theorem [85] and Kepler conjecture [89] – required entire teams to work for respectively 5, 6, and 8 years. These considerable developments are unfortunately only available for users of a single logical system or even worse to users of a specific version of a logical system.

Proof systems implement various logics. Coq, Agda, and NuPRL are constructive but PVS, HOL, and Mizar are classical. Zenon, FoCaLiZe and Mizar are first-order but most interactive provers are higher-order. Coq and NuPRL are proof relevant but Matita and HOL are proof irrelevant.

Moreover, they are getting more and more specialized. Isabelle for example has very good integration of countermodel and counterexample finding to avoid loosing time at trying to prove a false theorem and the Sledgehammer tool can be used to automatize a lot of tedious proofs by calling, without trusting them, external automated theorem provers and SMT solvers [25]. The Coq proof assistant features a very expressive tactic language [60] and has a very good library focusing on the proof by reflection technique [86]. Theorem provers are also specialized to specific theories such as linear arithmetic, set theory, arrays, and bitvectors.

Unfortunately, all these features can hardly be used in combination, even between tools implementing the same logic. The Flyspeck project has experimented this interoperability issue between Isabelle/HOL and HOL Light in which significant parts of the proof of Kepler conjecture have been developed but could not be imported into the other proof assistant and required the development of a new exchange format [103].

For proof systems agreeing on a common logic, proof exchange formats can be defined. The TPTP Format for Derivation [164] used in the library of solutions for TPTP problems TSTP [162] is used by a few theorem provers such as E [157], Vampire [109], and Zipperposition [56]. By outputting proofs in these formats, these automatic theorem provers can be integrated in interactive proof assistants such as Mizar [6] and Isabelle/HOL [25] to automate first-order reasoning. Similarly, as we mentioned in Section 2.3.2, the OpenTheory format can be used to exchange proofs between the proof assistants of the HOL family.

Several tools have been developed in order to solve the problem of interoperability between proof systems for different logics:

- The ProofCert project [126, 127] aims at defining a universal format of *proof certificates* to be checked by an independent checker called CHECKERS based on sequent calculus and focusing. It currently accepts certificates for classical, intuitionistic, and modal logic [47, 129]. The originality of ProofCert resides in the flexibility of the notion of certificates; whereas most approaches to interoperability require extremely detailed proof objects, various levels of details are allowed in certificates so that the checker can compensate for the imprecision of the system producing the certificate. This point is of high importance for fully automated tools which very rarely provide detailed proof objects but more often only proof *traces*. This flexibility comes from the backtracking ability offered by the language λ-prolog in which CHECKERS is implemented.

- The Logosphere project [158] aims at translating big formal libraries from proof systems and relate them.

- The LATIN project [97] aims at representing formally the connections between logics, proof assistants, theorem provers, SAT solvers, model checkers, and even programming languages.

- The MetaPRL system [95] is a logical framework built to relate NuPRL with HOL, Isabelle and PVS. It integrates JProver, an intuitionistic theorem prover.

All these projects have a lot of common points. At their very base lies a **logical framework** in which the different logics can be represented, then a **module system** is used to relate different logics and theories and finally **proof search** is used to automatically solve most of the proof obligations required to finish the development. For Logosphere and LATIN, the underlying logical framework is Twelf, an implementation of the $\lambda\Pi$-calculus in which many logics can be encoded using deep encodings in the sense of Section 3.5.1. MetaPRL is a logical framework defined as an extension of the programming language OCaml. The logical framework underlying ProofCert is λ-prolog. Once the logical framework is chosen, logics and theories are encoded in a modular fashion: Logosphere uses the

language of category theories, LATIN uses institutions and institution morphisms from model theory, MetaPRL and ProofCert rely on modular programming languages. Finally, interoperability tends in practice to generate a high number of simple proof holes. In ProofCert, Logosphere, and LATIN, the underlying logical framework is an higher-order extension of prolog so it is able to perform proof search to fill these holes. In MetaPRL, a first-order theorem prover is integrated for this task.

Our approach follows a similar pattern. Our logical framework is Dedukti, our module system is FoCaLiZe system of species. In the case study that we shall present in Chapter 11, only the object-oriented mechanisms are taken from FoCaLiZe; we define a hierarchy of species relying on inheritance, paramerization, and early binding but we do not write FoCaLiZe programs in the FoCaLiZe programming language. We use FoCaLiZe modularity independently of its programming language as favored by Leroy in the modular module system [115] originally developed for ML but easy to adapt to very different languages such as Atelier B [144]. Moreover, similarly to the use of the first-order theorem JProver in MetaPRL, we automate tedious proofs thanks to a first-order theorem prover, Zenon Modulo. We see two main advantages of using Dedukti compared to the other alternatives:

- Rewriting makes Dedukti very expressive so complex proof systems such as HOL and CIC can be embedded by **shallow** encodings. Using shallow encodings, we loose the possibility to express and prove meta-properties of the logical systems but we increase the scalability of the approach.

- Meta-programming is available for transforming proofs during the exchange. This can be used to simplify the translators when some parts of the proofs are easy to infer similarly to our simplification of Sigmaid in Section 6.4 and it can also be used to automatically eliminate unnecessary axioms to avoid strengthening the final logical system too much such as the Law of Excluded Middle, extensionality axioms, and the univalence axiom. It does however not natively support backtracking since Dedukti is intended to be used with confluent rewrite systems.

In the rest of this part, we focus on interoperability between Coq and HOL. We are going to use Dedukti as a common formalism in which HOL and Coq proofs can be translated

and combined. We start by merging Coq and HOL logics manually in Chapter 10, then we use Focalide to automate this process in Chapter 11. Finally, in Chapter 12 we use Dedukti as a meta-language for automatic elimination of classical axioms.

Chapter 10

Manual Interoperability between Coq and HOL

This chapter is the result of a common work with Assaf [15]. The aim of this chapter is to provide a first proof of concept of interoperability between Coq and HOL in order to discover the difficulties that we are going to face in practice.

We first need to relate Coq and HOL type systems and logics in Dedukti. This mixing of Coq and HOL logics is the topic of Section 10.1. We can then proceed to a toy example of interoperability in Section 10.2: an implementation of the insertion sort algorithm in Coq instantiated with the standard HOL definition of natural numbers. We conclude this first experiment by a discussion on the limitations of this approach in Section 10.3.

10.1 Mixing Coq and HOL Logics

As we discussed in Section 3.5, Coq and HOL use very different logics. Translators for both systems to Dedukti exist but they use unrelated signatures `coq.dk` and `hol.dk`. We examine the differences that set these two systems apart and show how we were able to bridge these gaps.

10.1.1 Type Inhabitation

The notion of types is different between HOL and Coq. In HOL, types are those of the simply-typed λ-calculus where every type is inhabited. In contrast, Coq allows

the definition of empty types, which in fact play an important role as they are used to represent falsehood. A naive union of the two theories would therefore be inconsistent: the formula $\exists x : \alpha, \top$, where α is a free type variable, is provable in HOL but its negation $\neg\forall\alpha : \mathbf{Type}, \exists x : \alpha, \top$ is provable in Coq.

Instead, we match the notion of HOL types with that of Coq's **inhabited** types, as done by Keller and Werner [104]. We define inhabited types in the Coq module `holtypes`:

```
Inductive type : Type := inhabited : forall (A : Type), A -> type.
```

It is then easy to prove in Coq that given inhabited types A and B, the arrow type $A \to B$ is also inhabited:

```
Definition carrier (A : type) : Type :=
  match A with inhabited B b => B end.
Definition witness (A : type) : carrier A :=
  match A with inhabited B b => b end.
Definition arrow (A : type) (B : type) : type :=
  inhabited (carrier A -> carrier B) (fun _ => witness B).
```

This is all that we need to interpret `hol.type`, `hol.term`, and `hol.arr` using rewrite rules. The symbol `hol.type` is a Dedukti type whereas `holtypes.type` represents a Coq type (in the universe $\mathbf{Type}_1$) so it is a Dedukti term of type `Coq.U (Coq.s Coq.z)` (see Section 3.5.2) the Dedukti type of its inhabitants is `Coq.T (Coq.type (Coq.s Coq.z)) holtypes.type`. We map `hol.type` to it using the following rewrite rule:

```
[] hol.type --> Coq.T (Coq.type (Coq.s Coq.z)) holtypes.type.
```

Thanks to this rewrite rule, a term `A` of type `hol.type` is identified with a Coq inhabited type but we have to distinct ways to state that a term inhabits this type, that is we still have distinct notions of belonging to a type in the sense of HOL and belonging to the carrier of an inhabited type in the sense of Coq. We identify these two notions by the following rewrite rule:

```
[A] hol.term A --> Coq.T (Coq.type Coq.z) (holtypes.carrier A).
```

Finally, we identify the notions of arrow types:

```
[] hol.arr --> holtypes.arrow.
```

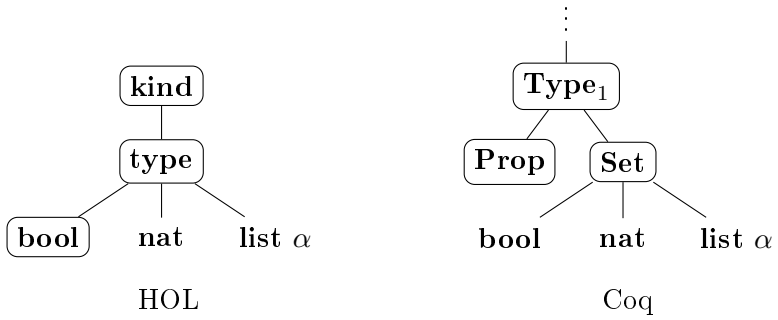


Figure 10.1: Booleans and propositions in HOL and Coq. Boxes represent universes.

10.1.2 Booleans and Propositions

In Coq, there is a clear distinction between booleans and propositions. Booleans are defined as an inductive type **bool** with two constructors **true** and **false**. The type **bool** lives in the universe **Set** (which is another name for the universe $\mathbf{Type}_0$). In contrast, following the Curry-Howard correspondence, propositions are represented as types with proofs as their inhabitants. These types live in the universe **Prop**. Both **Set** and **Prop** live in the universe $\mathbf{Type}_1$. As a consequence, **Prop** is not on the same level as other types such as **bool** or **nat** (the type of natural numbers), a notorious feature of the calculus of constructions. Moreover, since Coq is an intuitionistic system, there is no bijection between booleans and propositions. The excluded middle does not hold, though it can be assumed as an axiom.

In HOL, there is no distinction between booleans and propositions and they are both represented as a single type **bool**. Because the system is classical, it can be proved that there are only two inhabitants $\top$ and $\bot$, hence the name. Moreover, the type **bool** is just another simple type and lives on the same level as other types such as **nat**.

To combine the two theories, one must therefore reconcile the two pictures in Figure 10.1, which show how the types of HOL and Coq are organized.[1] One solution is to interpret

[1] Since **bool** is the type of propositions, and propositions are the types of proofs in the Curry-Howard correspondence, **bool** can be viewed as a universe [23, 78].

the types of HOL as types in **Set**. To do this, we must rely on a reflection mechanism that interprets booleans as propositions, so that we can retrieve the theorems of HOL and interpret them as theorems in Coq. In our case, it consists of a function **istrue** of type **hol.bool** → **coq.prop**, which we use to define **hol.proof**:

```
def Is_true : b : hol.term hol.bool -> Coq.U Coq.prop.
[b] hol.proof b --> Coq.T Coq.prop (Is_true b).
```

Another solution is to translate **hol.bool** as **coq.prop**. To do this, we must therefore translate the types of HOL as types in $\mathbf{Type}_1$ instead of $\mathbf{Type}_0$. In particular, if we want to identify **hol.nat** and **coq.nat**, we must have **coq.nat** in $\mathbf{Type}_1$. Fortunately, we have this for free with cumulativity since any element of $\mathbf{Type}_0$ is also an element of $\mathbf{Type}_1$.

We choose the first approach as it is more flexible and places less restrictions (e.g. regarding **Prop** elimination in Coq) on what we can do with booleans. In particular, it allows us to build lists by case analysis on booleans, which is needed in our case study.

10.2 Case Study: Sorting Coq Lists of HOL Numbers

We prove in Coq the correctness of the insertion sort algorithm on polymorphic lists and we instantiate it with the canonical order of natural numbers defined in HOL. More precisely, on the Coq side, we define polymorphic lists, the insertion sort function, the **sorted** predicate, and the **permutation** relation. We then prove the following two theorems:

```
Theorem sorted_insertion_sort: forall l, sorted (insertion_sort l).
Theorem perm_insertion_sort: forall l, permutation l (insertion_sort l).
```

with respect to a given (partial) order:

```
Variable A : Set.
Variable compare : A -> A -> bool.
Variable leq : A -> A -> Prop.
Hypothesis leq_trans : forall a b c, leq a b -> leq b c -> leq a c.
Hypothesis leq_total : forall a b,
   if compare a b then leq a b else leq b a.
```

The order comes in two flavors: a relation **leq** used for proofs, and a decidable version **compare** which we can destruct for building lists. The totality assumptions relates **leq** and **compare** and can be seen as a specification of **compare**.

On the HOL side, we use booleans, natural numbers and the order relation on natural numbers as defined in the OpenTheory packages `bool.art` and `natural.art`. By composing the results, we obtain two Dedukti theorems:

```
def insertion_sort_sorted : l : Natlist ->
                            coq_proof (sorted (insertion_sort l))
                          :=
                            sort.insertion__sort__sorted
                               hol_nat
                               hol_compare
                               hol_Leq
                               hol_leq_trans
                               hol_leq_total.

def insertion_is_permutation : l : Natlist ->
                               coq_proof
                                  (permutation l (insertion_sort l))
                             :=
                               sort.perm__insertion__sort
                                 hol_nat
                                 hol_compare.
```

The composition takes place in a Dedukti file named `interop.dk`. This file takes care of matching the interfaces of the proofs coming from Coq with the proofs coming from HOL. Most of the work consists in proving that HOL's comparison is indeed a total order in Coq:

```
def leq_total (m : Nat)
              (n : Nat)
              : cproof
                (cif_prop (compare m n)
                          (leq m n)
                          (leq n m)).
```

We prove it using the following theorems from OpenTheory:

$$\forall m\ n : \mathbf{hol\_nat}.\ m < n \Rightarrow m \leq n$$
$$\forall m\ n : \mathbf{hol\_nat}.\ m \not\leq n \Leftrightarrow n < m$$

and some additional lemmata on **if**...**then**...**else**. We chose this example because the interaction between Coq and HOL types is very limited thanks to polymorphism: there is no need to reason about HOL natural numbers on the Coq side and no need to reason about lists on the HOL side so the only interaction takes place at the level of booleans which we wanted to study. Our implementation is illustrated in Figure 10.2. All the components are successfully verified by Dedukti.

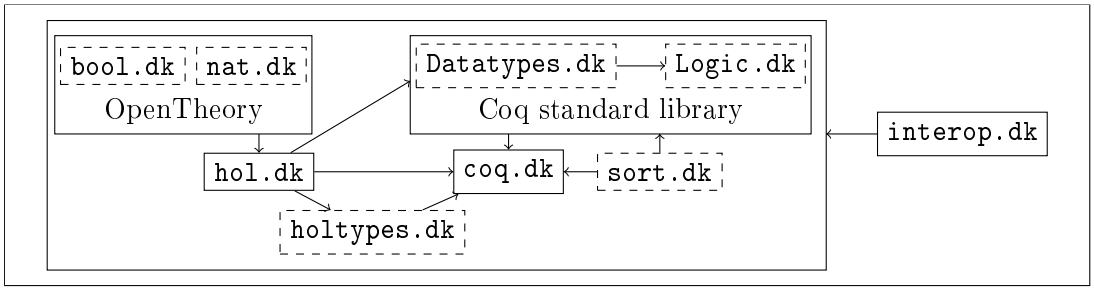


Figure 10.2: Components of the implementation. Solid frames represent source files. Dashed frames represent automatically generated files. Arrows represent dependencies.

Because of the verbosity of Dedukti and small style differences between HOL and Coq, this proof is long (several hundreds of lines) for such a simple fact. However, most of it is first-order reasoning and we will see in next section how we can integrate Zenon to automate it. This is needed to scale to harder problems requiring for example to translate and link theorems about natural numbers in HOL and theorems about natural numbers in Coq.

10.3 Limitations

We successfully translated a small Coq development to Dedukti and instanciated it with the HOL definition of natural numbers. The results have been validated by Dedukti. Mixing the underlying theories of Coq and HOL raised interesting questions but did not require a lot of human work: the file `hol.dk` is very close to the version included with Holide and the file `holtypes.v` is very small. In retrospect, the result looks a lot like an embedding of HOL in Coq but performed in Dedukti. This is not surprising, as the theory of HOL is fairly simple compared to Coq and is in fact a subset of the logic of Coq [23, 78, 104].

The interoperability layer `interop.dk` which is specific to our case study required a lot of work which could be automated by Zenon as it is mostly simple first-order reasoning.

Interoperability raises more issues than mere proof rechecking and our translators to Dedukti need to be improved. The translations produce code intended for machines that is

not very usable by humans. In particular, the OpenTheory article format lacks names for theorems; they are simply numbered by Holide and the numbering changes when we include more article files. Lack of name is not a major issue for other uses of OpenTheory because good notations make output such as the OpenTheory webpages at `http://opentheory.gilith.com/packages/` readable but finding a lemma in a Dedukti file generated by Holide takes a lot of time.

Another limitation of this example of interoperability is the lack of executability. Even though we have constructed a sorting "algorithm" on lists of HOL natural numbers and we have proved it correct, there is no way to actually execute this algorithm. Indeed, there is no notion of computation in HOL, so when the sorting algorithm asks **compare** for a comparison between two numbers, it will not return something which will unblock the computation. Therefore, **insertion_sort** $[4, 1, 3, 2]$ is not *computationally* equal to $[1, 2, 3, 4]$. However, the result is still *provably* equal to what is expected: we can show that **insertion_sort** $[4, 1, 3, 2]$ is equal to $[1, 2, 3, 4]$. Solving this issue requires to change the presentation of HOL in order to get one that is both constructive and computational. By constructive, we mean that it should present an intuitionistic version of Higher-Order Logic and by computational we mean that it should contain a notion of reduction. We could then define a shallow, reduction-preserving encoding to Dedukti. The pure type system presentation of HOL [23, 78] is a reasonable candidate for that but the proofs of OpenTheory will need to be adapted. Holala [171] is an encouraging step in this direction, as it essentially takes the union of $\mathcal{Q}_0$ and the PTS presentation.

Finally, the correctness proof of the sorting algorithm is performed in a logic which has more axioms than both HOL and Coq. We avoided some choices when we realized that they would lead to inconsistencies but we did not prove that the logic obtained by combining Coq and HOL is consistent. Even if this logic is consistent, and we believe it is, we might want to limit the dependencies to axioms on a per-development basis so that they could be exchanged further with other logics. In this particular case, we expect the correctness proof to depend on no controversial axiom such as the Law of Excluded Middle or the Axiom of Choice so we would like to eliminate them from this development.

Chapter 11

Automation using FoCaLiZe and Zenon Modulo

We now consider a more complicated example: a proved version of the Sieve of Eratosthenes. We have chosen this example because contrary to the previous one, HOL and Coq have to agree on the type of natural numbers despite having slightly different definitions for it:

- in Coq, the type of natural numbers is defined as an inductive type;
- in HOL, inductive types are not primitive and natural numbers are encoded.

We program the sieve in Coq in order to get the good reduction behaviour, we also prove most of the correctness theorem in Coq but we want to use the arithmetic lemmata of OpenTheory standard library. Moreover, we use FoCaLiZe, Focalide, and Zenon Modulo to automate most of the interoperability layer. This example is intended as a proof of concept of interoperability to demonstrate the role that FoCaLiZe, Zenon Modulo, Dedukti, and Focalide can play for making logical systems communicate. The formal development presented in this chapter is available at the following URL: `http://dedukti-interop.gforge.inria.fr`.

Despite being significantly bigger than the previous example, this new proof of concept is still too small to exploit a combination of the strengths of Coq and HOL. It is simpler to formally prove the Sieve of Eratosthenes in either Coq or HOL than to setup the interoperability framework that we are about to describe.

In Section 11.1, we give the Coq implementation of the sieve of Eratosthenes and highlight the missing theorems that we want to import from OpenTheory. In Section 11.2, we connect the logic of FoCaLiZe to the mixed logic of Coq and HOL that we defined in Section 10.1. In Section 11.3, we explain how we improved readability of Holide in order to find the required lemmata. Section 11.4 is devoted to our use of FoCaLiZe as an interoperability framework in this proof of concept, it presents a hierarchy of FoCaLiZe species totally independent of Coq and HOL which then gets instantiated to bridge arithmetic definitions and theorems between HOL and Coq. We conclude this chapter in Section 11.5 where we discuss the limitations of our approach and what features we felt missing in our tools for future interoperability developments.

11.1 An Implementation of the Sieve of Eratosthenes in Coq

The Sieve of Eratosthenes is an algorithmic method for listing all the prime numbers smaller than a given bound. Its implementation in a functional programming language such as OCaml looks like:

```
(* interval a b is the sorted list of all numbers between a and b *)
let rec interval a b = if a > b then [] else a :: interval (a+1) b

(* The core of the Sieve of Eratosthenes *)
let rec sieve = function
  | [] -> []
  | a :: l -> a :: sieve (List.filter (fun b -> b mod a > 0) l)

(* The Sieve of Eratosthenes, eratosthenes n is the sorted list of all
   primes smaller or equal to n. *)
let eratosthenes n = sieve (interval 2 n)
```

In this section, we propose a certified implementation of this program in the Coq proof assistant. We decompose this task in three: we have to program the sieve in Coq, to specify its correctness, and to prove it. In Section 11.1.1, we program the sieve in Coq and in Section 11.1.2 we specify it. In Section 11.1.3, we write an informal but rigorous proof of the correctness of the algorithm to discover the mathematical theorems on which this correctness proof relies. In order to experiment with interoperability, we will not prove these mathematical results in Coq but import them from OpenTheory.

11.1.1 Programming the Sieve of Eratosthenes in Coq

Since only positive integers are used in this algorithm, we use the Coq type of natural numbers because they have an inductive structure which is easy to reason about.

```
Inductive nat : Set := O | S of nat -> nat.
```

This definition is available in Coq standard library but as we discussed in Section 3.5.2, Coqine is not able to translate a significant part of Coq standard library so instead of including the library, we just copy the definitions that we need from it.

Since we only need lists of natural numbers, we consider a monomorphic type of lists of natural numbers:

```
Inductive list : Set :=
| Nil : list
| Cons : nat -> list -> list.
```

In Coq, we avoid the manipulation of empty intervals by a slight change in the definition: the Coq version intervals takes two natural numbers `a` and `b` and returns the sorted list of natural numbers between `a` and `a + b`. Because we use positive numbers only, we are guaranteed that the upper bound of the interval is greater than the lower bound.

```
Fixpoint interval a b : list :=
  match b with
    | O => Cons a Nil
    | S b => Cons a (interval (S a) b)
  end.
```

The code of the filtering function for lists is not surprising:

```
Fixpoint list_filter (p : nat -> bool) l :=
  match l with
    | Nil => Nil
    | Cons a l =>
      let l' := list_filter p l in
      if p a then Cons a l' else l'
  end.
```

Divisibility is a bit harder to get right. Divisibility plays two purposes in our development: we need a divisibility test inside the filter (corresponding to `b mod a > 0` in our OCaml implementation) and we also need divisibility to define primality and specify the algorithm. In order to get a simple definition of primality, we introduce *strict* divisibility:

we say that a is a strict divisor of b if a divides b, $a > 1$, and $a < b$. A natural number $p > 1$ is then called a *prime* number if and only if it has no strict divisor.

Strict divisibility is characterized as follows:

$$\forall a > 1.\ \forall b > 0.\ a \text{ strictly divides } b \Leftrightarrow \exists q > 1.\ aq = b$$

It is hence sufficient for this work to consider euclidean divisions in the case where the dividend, the divisor, and the quotient are all positive. This restriction simplifies a bit the definition of the auxiliary function `modaux` computing the euclidean division. `modaux a b` returns a pair `(q, r)` such that `q+1` is the quotient of `a+1` by `b+1` and `r` is the complement of the remainder of this euclidean division.

```
(* modaux a b = (q, r) <-> (q+1)(b+1) = a + 1 + r *)
Fixpoint modaux a b :=
  match a with
    | O => (O, b)                (* 1*(b+1) = 0 + 1 + b *)
    | S a' =>
      let (q, r) := modaux a' b in (* (q+1)(b+1) = a' + 1 + r *)
      match r with
        | O => (S q, b)          (* (q+1+1)(b+1) = (q+1)(b+1) + b + 1
                                      = a' + 1 + r + b + 1 = a + 1 + b *)
        | S r' => (q, r')        (* (q+1)(b+1) = a' + 1 + r = a + r *)
      end
  end.
```

From `modaux`, it is easy to define *strict divisibility*: for all a, b, and q,

$$\begin{array}{lcl} \texttt{modaux}\ a\ b = (q, 0) & \Leftrightarrow & (q+1)(b+1) = a+1 \\ b+1 \text{ is a strict divisor of } a+1 & \Leftrightarrow & b > 0 \wedge \exists q > 0, \texttt{modaux}\ a\ b = (q, 0) \end{array}$$

hence the definition of strict divisibility (`sd`):

```
Definition sd b' a' :=
  match a', b' with
    | S a, S (S b) =>
      match modaux a (S b) with
        | (S _, 0) => true
        | _ => false
      end
    | _, _ => false
  end.
```

The regular notion of divisibility would equally be appropriate for filtering in sieve's core but strict divisibility gives a simpler definition of primality.

Since the sieve's core filters the **non**-multiples of some number, we also need negation on booleans:

```
Definition negb b := match b with true => false | false => true end.
```

We now have all the prerequisites for defining the sieve's core function. The simple Coq translation of the OCaml function

```
Fixpoint Sieve (l : list) : list :=
  match l with
    | Nil => Nil
    | Cons a l =>
      Cons a (Sieve (list_filter (fun b => negb (sd a b)) l))
  end.
```

is rejected by Coq because `list_filter (fun b => negb (sd a b)) l` is not a strict subterm of `Cons a l`. This can be fixed by adding a dummy parameter `(fuel : nat)` on which the function `Sieve` recurses:

```
Fixpoint Sieve (l : list) (fuel : nat) {struct fuel} : list :=
  match fuel with
    | O => Nil
    | S fuel =>
      match l with
        | Nil => Nil
        | Cons a l =>
            Cons a
                 (Sieve (list_filter (fun b => negb (sd a b)) l) fuel)
      end
  end.
```

When `fuel` is bigger than the length of `l`, the Coq version `Sieve l fuel` behaves like the OCaml version `sieve l` so the length of `l` is an interesting default value for `fuel`:

```
Fixpoint length l :=
  match l with
    | Nil => O
    | Cons _ l => S (length l)
  end.

Definition sieve_len l := Sieve l (length l).
```

Finally, the prime numbers smaller than $2 + n$[1] can be computed by

```
Definition eratosthenes n := sieve_len (interval 2 n).
```

[1] We recall that in our Coq implementation {interval a b} is the list of natural numbers between `a` and `a + b`.

11.1.2 Specification

The specification of the sieve of Eratosthenes is quite simple: a number `p` is a member of the list returned by `eratosthenes n` if and only if `p` is a prime number smaller than `2 + n`.

We need a few straightforward definitions in order to state this specification:

```
Inductive le (n :  nat) : nat -> Prop :=
| le_n : le n n
| le_S m : le n m -> le n (S m).
Infix "<=" := le.

Fixpoint In n l :=
  match l with
    | Nil => False
    | Cons a l => n = a \/ In n l
  end.

Inductive Istrue : bool -> Prop := ITT : Istrue true.

Definition prime p := 2 <= p /\ forall d, Istrue (negb (sd d p)).
```

We state the specification of the sieve of Eratosthenes as three lemmata:

```
Lemma eratosthenes_sound_1 p n : In p (eratosthenes n) -> p <= 2 + n.
Lemma eratosthenes_sound_2 p n : In p (eratosthenes n) -> prime p.
Lemma eratosthenes_complete p n :
  prime p ->
  p <= 2 + n ->
  In p (eratosthenes n).
```

11.1.3 Correctness proof

We start by a rather informal proof of the three lemmata forming the specification of the sieve of Eratosthenes in order to highlight the arithmetic results needed to complete the proof.

We start by completeness, that is, given a prime number p smaller than $2+n$, we want to prove that p appears in the list returned by the function `eratosthenes`. For this, it is enough to prove that the `Sieve` function preserves prime numbers (assuming it received enough fuel), which is obvious because this function only removes a number when it found a strict divisor and by definition of primality, p has no strict divisor.

The first soundness lemma also relies on an invariant of the `Sieve` function, namely that the members of `Sieve l fuel` are all members of `l`. The proof is then concluded by a simple soundness property of intervals : if p is a member of `interval a b` then $p \leq a + b$.

The second soundness lemma is where arithmetic is required. Let p be a member of `eratosthenes n`, we can easily prove that $2 \leq p$ by an argument similar to the proof of the first soundness lemma. To prove that p has no strict divisor, we use the following standard arithmetic result:

Theorem 15 (Smallest Prime Divisor)**.** *Let n be a natural number greater than 2, the smallest divisor of n is prime.*

Actually, the following corollary is enough for our proof:

Corollary 2. *Let n be a natural number greater than 2, n has a prime divisor.*

To conclude the proof, we remark the following facts:

- the `Sieve` function preserves and conserves the ordering: if a and b are two members of `Sieve` l `fuel`, then a appears before b in `Sieve` l `fuel` if and only if a appears before b in l
- a appears before b in an interval $[c, d]$ if and only if $c \leq a < b \leq d$
- if a appears before b in `Sieve` l `fuel`, then a is not a strict divisor of b
- let d be a prime divisor of p, if $d = p$ we are done, otherwise d is a strict divisor of p and d is prime so by the completeness lemma, d is a member of the list returned by the function but since d is a strict divisor of p, $d < p$ so d appears before p in the returned list.

This concludes our informal certification of the Sieve of Eratoshenes. The required ingredient from arithmetic is the existence of a prime divisor. For the sake of proof of concept, we shall not prove this result in Coq but import it from OpenTheory.

We proved the correctness of the Sieve of Eratosthenes in Coq when Corollary 2 is considered as a parameter. This development can be split into three parts of approximately the same size:

- straightforward arithmetic results such as commutativity of addition and multiplication, these results are proved in both Coq standard library and Holide but they are so straightforward that it was easier to reprove them than to import them and we wanted to limit the dependency of this work to Coq standard library because Coqine lacks some features needed for it,
- correctness of auxiliary functions which could be reused in other developments (`modaux`, strict divisibility and functions manipulating lists), and
- correctness of the functions `Sieve` and `eratosthenes` which are specific to this problem.

As in Chapter 10, the results that we want to import from HOL are hypotheses of the final theorem that have to be provided in Dedukti.

The biggest part of the development is written in FoCaLiZe in which the arithmetic library of OpenTheory is related to Coq natural numbers.

11.2 Relating FoCaLiZe Logic with Coq and HOL

As we bound the signatures of Coq and HOL in Section 10.1, we bind the FoCaLiZe built-in types and constants that we used in our specification of arithmetic operations to those available in Coq and HOL.

Actually, FoCaLiZe logic can easily be injected into HOL:

- FoCaLiZe types `bool` and `prop` are mapped to `hol.bool`,
- FoCaLiZe logical connectives are mapped to their HOL definitions; in particular logical equivalence is mapped to `hol.eq hol.bool`,
- FoCaLiZe equality is mapped to `hol.eq`.

Thanks to FoCaLiZe support for external languages, this mapping can almost completely be done in FoCaLiZe itself: FoCaLiZe definitions of `bool` and `eq` are provided by the first file in FoCaLiZe standard library `basics.fcl`, their default definitions can thus be overridden by writing another `basics.fcl` file:

```
type bool =
  internal
  external
  | dedukti -> {*
     hol.bool.
     def true := hol.true.
     def false := hol.false
     *}
;;

let ( = ) =
  internal 'a -> 'a -> bool
  external | dedukti -> {* hol.eq __var_a *}
;;
```

Because they affect the way Zenon Modulo proofs should be read, the mappings for logical connectives have however to be overridden in Dedukti:

```
#NAME dk_logic.

def Prop := cc.eT hol.bool.
def eP : Prop -> Type := hol.proof.

def true := hol.true.
def false := hol.false.
def not := hol.not.
def and := hol.and.
def or := hol.or.
def imp := hol.imp.
def forall := hol.forall.
def exists := hol.exists.
def eqv := hol.eq hol.bool.
```

The lemmata corresponding to Zenon Modulo proof rules have to be reproved for these definitions, this raises no major difficulty and takes approximately 200 lines of Dedukti code.

11.3 FoCaLiZe as a User Interface to HOL

As we have seen in the previous chapter, a limitation of the OpenTheory format is the lack of names which makes hard to find lemmata. While this is not much of a problem when looking at the content of an OpenTheory package online on `http://opentheory.gilith.com/packages/`, it becomes very unpractical when we need to find the theorem number in the Dedukti file generated by Holide.

To alleviate this burden, we patched Holide so that it could produce a FoCaLiZe file alongside the usual Dedukti file. Only the statements of the theorems are translated in FoCaLiZe, the proofs are pointers to the Dedukti file. For this reason, the FoCaLiZe file is small enough to be readable.

For real HOL developments such as OpenTheory standard library, the generated FoCaLiZe file will usually be ill-typed because of name conflicts for example. This is not a problem for this work because we are only interested in very few lemmata, which happen not to break typing.

For example, if we want to import the property of injectivity of the successor operation, we can look for the theorems containing the symbol `suc` in the FoCaLiZe file produced by Holide and we quickly find the one we need:

```
theorem thm_3523 : all m n : natural, (suc(m) = suc(n)) <-> (m = n)
  proof = dedukti proof {* natural__div__full.thm_3523. *};;
```

Since Zenon Modulo is a first-order theorem prover, we do not translate all statements of Higher-Order Logic to FoCaLiZe but filter the statements to print only those who are first-order.

This is done by traversing the formula from its root to its leaves using two translation modes called formula mode and term mode. In formula mode, boolean connectives are interpreted as logical connectives and quantification is allowed. Equality is interpreted as logical equivalence if its arguments have type **bool** and as a binary predicate symbol otherwise. When a predicate symbol is traversed, term mode is activated and boolean connectives become interpreted as boolean functions; quantification is no more allowed. Variables are not allowed to be applied to argument in either mode.

The portion of OpenTheory standard library that we translated for this study contains 114 theorems among which only 10 are not first-order theorems.

11.4 Specifying Arithmetic as a FoCaLiZe Hierarchy of Species

The FoCaLiZe part of the development can be divided in five parts:

- an abstract specification of arithmetic structures designed as a hierarchy of species

starting at Peano axioms and culminating at the prime divisor theorem, this part is presented in Section 11.4.1,

- a corresponding hierarchy of species isomorphisms, this part is presented in Section 11.4.2,
- a partial instantiation of the hierarchy of arithmetic structures by the definitions of Coq, this part is presented in Section 11.4.3,
- a full instantiation of the hierarchy of arithmetic structures by the definitions and theorems of OpenTheory, this part is presented in Section 11.4.4,
- a full instantiation of the hierarchy of isomorphisms relating the Coq and the HOL definitions, this provides a version of the prime divisor theorem talking about Coq numbers and operations, this part is presented in Section 11.4.5.

11.4.1 Abstract arithmetic structures

On the FoCaLiZe side, we define a hierarchy of species which axiomatize natural numbers and arithmetic to various extents. Thanks to the object-oriented features of FoCaLiZe, we can provide default implementations for arithmetic operations and redefine methods to map them to external definitions coming from Coq, HOL, or any proof system featuring a Dedukti output.

The first building block of our hierarchy of species specifies what it means to be a representation of natural numbers; there should be a constant `zero` and a function `succ` such that the axioms of Peano hold:

```
(* The basic block: Peano axiomatization of natural numbers *)

species NatDecl =
  signature zero : Self;
  signature succ : Self -> Self;
  property zero_succ : all n : Self, ~(zero = succ(n));
  property succ_inj : all m n : Self, succ(m) = succ(n) -> m = n;
  property ind : all p : Self -> prop,
    p(zero) ->
    (all n : Self, p(n) -> p(succ(n))) ->
    all n : Self, p(n);
end;;
```

The logical method `ind` is out of the scope of the presentation of FoCaLiZe that we gave in Chapter 7 because it uses higher-order quantification over `p`. Actually, FoCaLiZe does not enforce formulae to stay in first-order but higher-order properties are discouraged because Zenon and Zenon Modulo are first-order theorem provers. The only place where we can use `ind` is in an external Dedukti proof.

For example, we can use induction to derive the particular case where the induction hypothesis is not used. This particular case has the advantage of being expressible as a first-order formula $\forall n.\ n = 0 \vee \exists m.\ n = 1 + m$:

```
species NatCase =
  inherit NatDecl;
  (* When the induction hypothesis is not needed, we can use
     reasoning by case which is better supported by Zenon *)
  logical let casep (n : Self) = n = zero \/ ex m : Self, n = succ(m);
  theorem case : all n : Self, n = zero \/ ex m : Self, n = succ(m)
  proof =
  <1>1 assume n : Self,
       prove casep(n) <-> (n = zero \/ ex m : Self, n = succ(m))
       by definition of casep
  <1>2 prove casep(zero) ->
              (all n : Self, casep(n) -> casep(succ(n))) ->
              all n : Self, casep(n)
       dedukti proof property ind {* abst_ind abst_casep *}
  <1>3 prove casep(zero) by step <1>1
  <1>4 assume n : Self, prove casep(succ(n)) by step <1>1
  <1>f conclude;
end;;
```

In this proof, Dedukti is only used to instantiate the induction principle with the predicate `casep`, all the rest of the proof is first-order reasoning and is handled by Zenon Modulo.

We further extend the species `NatCase` by introducing an iteration operation; `iter(f, a, n)` is our notation for $f(\ldots(f(a)))$ where f is iterated n times:

```
let ( @ ) (f, x) = f(x);;

species NatIter =
  inherit NatCase;
  signature iter : (Self -> Self) -> Self -> Self -> Self;
  property iter_zero : all f : (Self -> Self), all z : Self,
    iter(f, z, zero) = z;
  property iter_succ : all f : (Self -> Self), all z n : Self,
    iter(f, z, succ(n)) = f @ iter(f, z, n);
end;;
```

To formulate the specification of `iter` at first-order, we use an explicit infix symbol `@` for application.

Iteration is used to provide default definitions for addition and multiplication. Large ordering is defined from addition ($(a \leq b) := (\exists c.\ a+c = b)$) and divisibility is defined from multiplication ($(a|b) := (\exists c.\ a*c = b)$). Strict ordering is defined by $(a < b) := (1+a \leq b)$ and strict divisibility by (a strictly divides b) $:= (1 < a < b \wedge a|b)$. Finally, a number p is prime if $1 < p$ and p has no strict divisor. Each definition is added in a different species as pictured in Figure 11.1.

The last block of the hierarchy of abstract arithmetic structures states the prime divisor theorem:

```
species NatPrimeDiv =
  inherit NatPrime;

  signature primediv : Self -> Self;
  property primediv_prime : all n : Self,
     (~ n = succ(zero)) -> prime(primediv(n));
  property primediv_divides : all n : Self,
     (~ n = succ(zero)) -> divides(primediv(n), n);
end;;
```

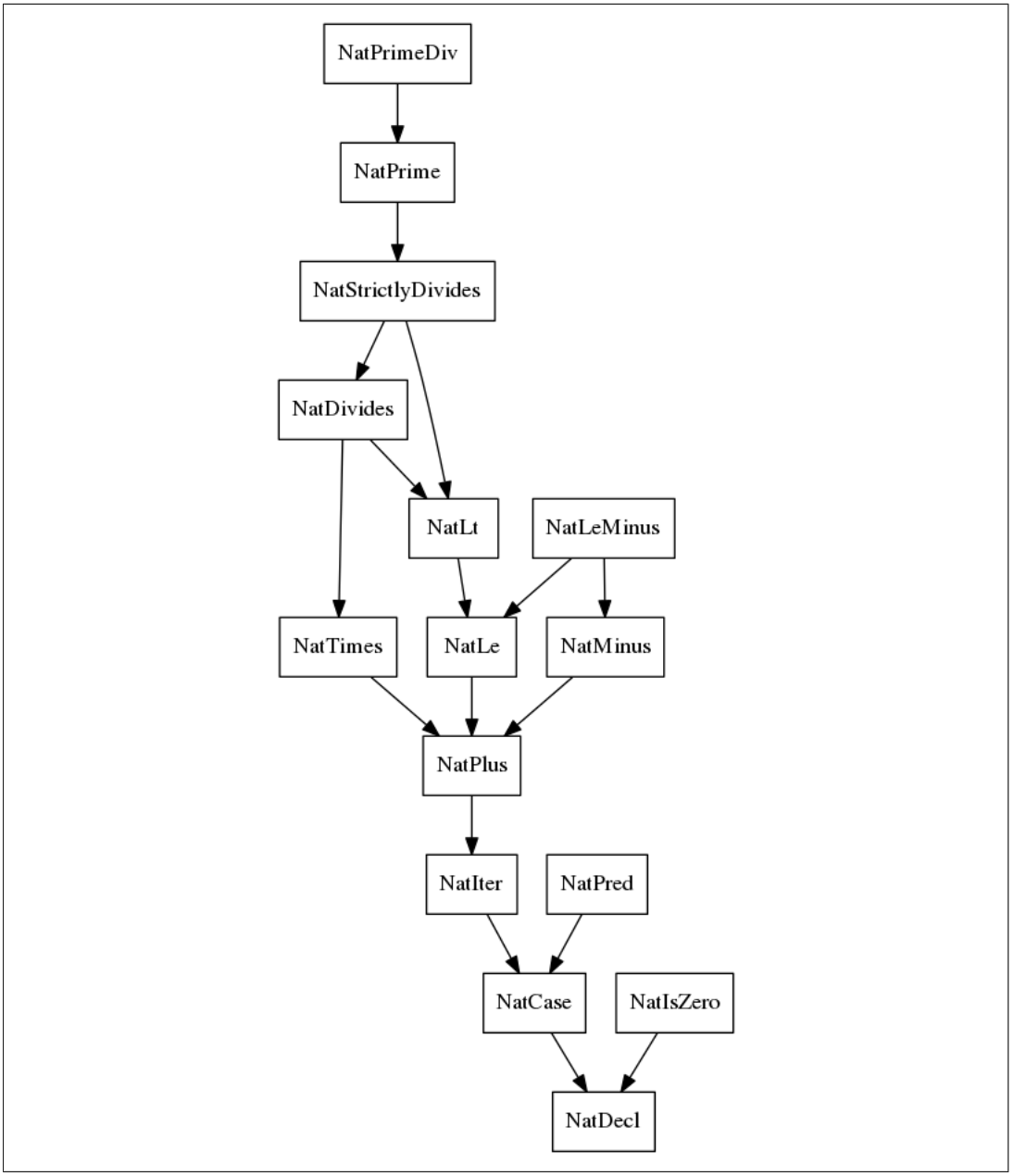
11.4.2 Morphisms Between Representations

The hierarchy of abstract arithmetic structures can be instantiated with Coq and HOL arithmetic libraries as we will see in Sections 11.4.3 and 11.4.4 but to relate these two instantiations, we need to study morphisms between abstract arithmetic structures.

A morphism from a `NatDecl` to another `NatDecl` B is defined by a function `morph` of type `Self -> B` preserving zero and successors.

```
species NatMorph (B is NatDecl) =
  inherit NatDecl;

  signature morph : Self -> B;
  property morph_zero : morph(zero) = B!zero;
  property morph_succ : all n : Self, morph(succ(n)) = B!succ(morph(n));
end;;
```

From the axioms of Peano assumed in both the current species and in the parameter `B`, we can prove that `morph` is bijective.



In this figure, frames represent species and there is an arrow from a species A to a species B if A directly inherits from B.

Figure 11.1: A hierarchy of FoCaLiZe species for arithmetic properties

We follow the hierarchy of arithmetic structures to produce a hierarchy of morphisms. We prove that all operations are preserved by the morphisms, Zenon Modulo is extensively used for this task. This hierarchy is depicted on Figure 11.2, it culminates with the following species:

```
species NatPrimeDivMorph (B is NatPrimeDiv) =
  inherit NatPrimeDiv, NatPrimeMorph(B);

  let primediv (n) = ...;
  proof of primediv_prime = ...;
  proof of primediv_divides : ...;
end;;
```

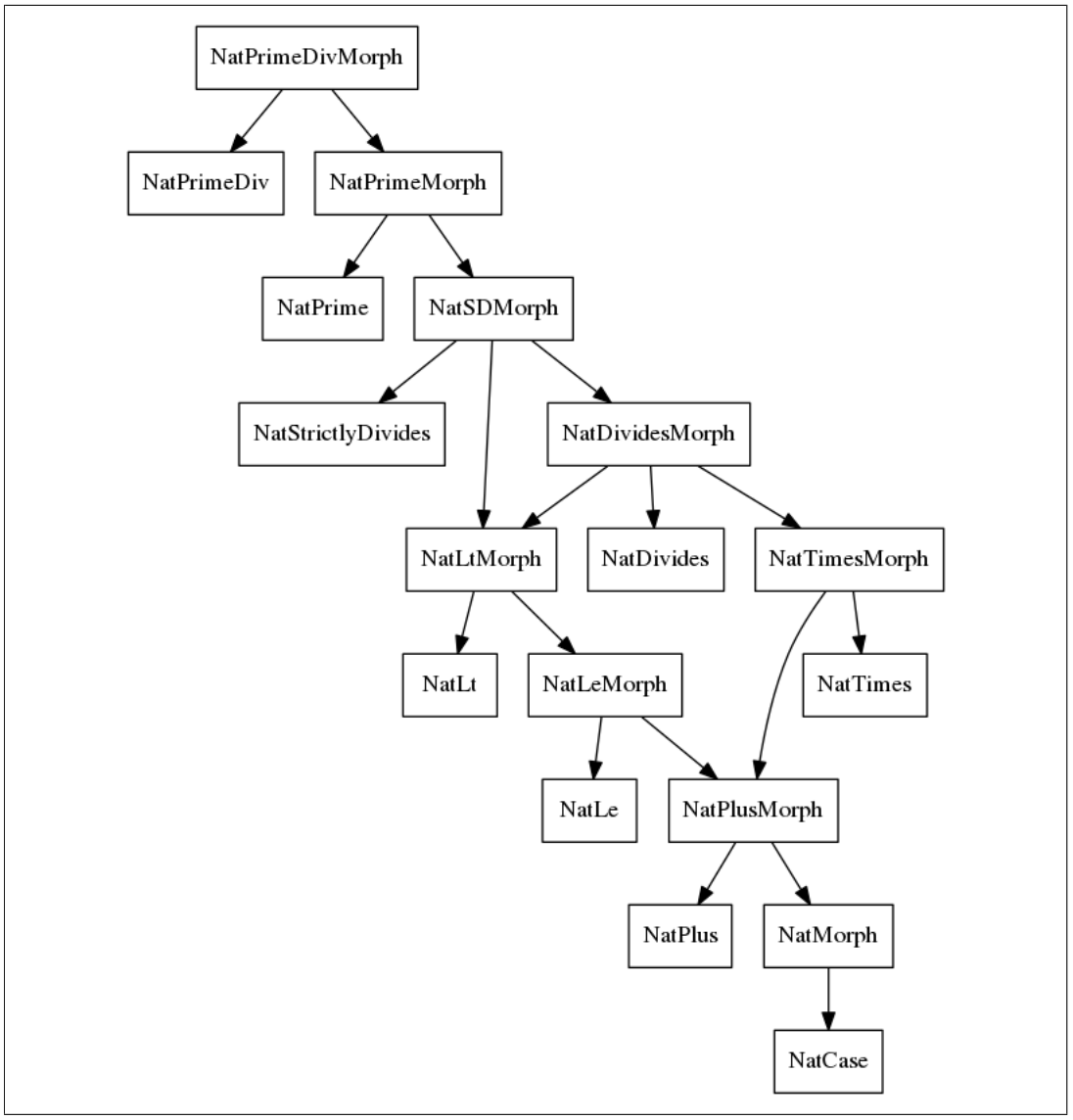
where the dots stand for relatively long definitions and proofs.

11.4.3 Instantiation of Coq Natural Numbers

We can instantiate the hierarchy of species on the Coq side using FoCaLiZe external Dedukti definitions mapping directly the symbols to their Coqine translation in Dedukti.

For example, the first species is (partially) instantiated by

```
type coq_nat =
  internal
  external
  | dedukti -> {* holtypes.inhabited
                        Coq__Init__Datatypes.nat
                        Coq__Init__Datatypes.O *};;

species CoqNat =
  inherit NatDecl;

  representation = coq_nat;
  let zero = internal coq_nat
    external
    | dedukti -> {* Coq__Init__Datatypes.O *};

  let succ(n : coq_nat) = internal coq_nat
    external
    | dedukti -> {* Coq__Init__Datatypes.S n *};

  proof of ind =
    dedukti proof definition of zero, succ
    {* (p : (hol.term abst_T -> hol.term hol.bool) =>
        Coq__Init__Datatypes.nat__ind
           (n : hol.term abst_T => hol_to_coq.Is_true (p n))). *};
end;;
```



In this figure, frames represent species and there is an arrow from a species A to a species B if A directly inherits from B. Inheritance arrows of Figure 11.1 are ommited.

Figure 11.2: A hierarchy of FoCaLiZe species for arithmetic properties

Our need for the symbols `holtypes.inhabited` and `hol_to_coq.Is_true` has been discussed in previous chapter.

To prove an equality, we can use conversion at the level of Dedukti by using a Dedukti proof of reflexivity. An example is given by the predecessor function, which is introduced to prove injectivity of the successor:

```
species CoqPred =
  inherit NatPred, CoqNat;

  let pred(n : coq_nat) = internal coq_nat
    external
    | dedukti -> {* Coq__Init__Peano.pred n *};

  theorem pred_succ : all n : Self, pred(succ(n)) = n
  proof = dedukti proof definition of pred, succ
    {* (n : hol.term abst_T => hol.REFL abst_T n). *};
end;;
```

Harder theorems such as the prime divisor theorems are not proven directly on the Coq side but are imported by a morphism from HOL.

11.4.4 Instantiation of HOL Natural Numbers

Thanks to FoCaLiZe external definitions and thanks to our FoCaLiZe-generating version of Holide, we can import in FoCaLiZe the HOL definition of natural numbers coming from OpenTheory:

```
type hol_natural =
  internal
  external
  | dedukti -> {* natural__div__full.Number_2ENatural_2Enatural *};;

let hol_zero =
  internal hol_natural
  external
  | dedukti -> {* natural__div__full.Number_2ENatural_2Ezero *};;

let hol_succ =
  internal hol_natural -> hol_natural
  external
  | dedukti -> {* natural__div__full.Number_2ENatural_2Esuc *};;

theorem hol_induction :
  all p : hol_natural -> bool,
      (p(hol_zero) /\ (all n : hol_natural, p(n) -> p(hol_succ(n)))) ->
```

```
      (all n : hol_natural, p(n))
proof = dedukti proof {* natural__div__full.thm_3723. *};;
```

This almost gives us the required interface for implementing `NatDecl` using HOL natural numbers. The only difference is a matter of curryfication in the statement of `hol_induction`. Unfortunately, Zenon Modulo cannot deal with this simple curryfication because it refuses higher-order problems so we prove the property directly in Dedukti. This is the most complicated theorem that we had to prove in Dedukti. However it is considerably simpler than the theorems we had to prove in the previous chapter.

```
species HolNat =
  inherit NatDecl;
  representation = hol_natural;
  let zero = hol_zero;
  let succ = hol_succ;
  proof of ind =
  dedukti proof
    definition of zero, succ
    {* (p : (cc.eT abst_T -> cc.eT hol.bool) =>
        H0 : hol.proof (p abst_zero) =>
        HS : (n : cc.eT abst_T ->
              hol.proof (p n) ->
              hol.proof (p (abst_succ n))) =>
        hol_induction p
          (hol.and_intro
             (p abst_zero)
             (hol.forall abst_T (n : hol.term abst_T =>
                hol.imp (p n) (p (abst_succ n))))
             H0 HS)). *};
end;;
```

The hierarchy is fully implemented and can be turned in a collection:

```
species HolPrimeDiv =
  inherit NatPrimeDiv, HolPrime;
  ...
end;;

collection HolPrimeDivColl = implement HolPrimeDiv; end;;
```

11.4.5 Instantiation of the Morphism

Since the iteration operator that we imported from OpenTheory and Coq is polymorphic, we can also use it for defining morphisms:

```
let coq_iter =
```

```
  internal coq_nat -> 'a -> ('a -> 'a) -> 'a
  external
  | dedukti -> {* ... *};;

species CoqMorph (B is NatPrimeDiv) =
  inherit NatPrimeDivMorph(B), CoqLe, CoqTimes;

  let morph(n) = coq_iter(n, B!zero, B!succ);
  proof of morph_zero =
  dedukti proof
    definition of morph, zero
    {* hol.REFL _p_B_T _p_B_zero. *};
  proof of morph_succ =
  dedukti proof
    definition of morph, succ
    {* (n : (hol.term coq_nat__t) =>
        hol.REFL _p_B_T (abst_morph (abst_succ n))). *};
end;;
```

At this point, we observe a small duplication of proof work due to the lack of polymorphic methods in FoCaLiZe. If such methods were allowed we could have derived `morph` from `iter` before instantiation with HOL and Coq naturals. Polymorphic methods are forbidden in FoCaLiZe because, as shown by Prevosto [150], specializing the type of a polymorphic method through inheritance can break the type system.

11.5 Discussion

We achieved our goal of certifying a Coq implementation of the sieve of Eratosthenes using the arithmetic results from OpenTheory. Doing so, the object-oriented mechanisms of FoCaLiZe allowed us to devise a hierarchy of arithmetic species with default definitions for arithmetic operations. Zenon Modulo was of great help during this formalization since a lot of small steps of equational reasoning were needed and proving them in Dedukti would have been very painful. The whole power of Zenon Modulo was however clearly not used in this development since we did not take profit of Deduction modulo. For this use case, we believe that Zenon Modulo could be replaced by less powerful provers such as Waldmeister [9], which happens to output constructive proofs.

In the course of this development, we had to work around some limitations of FoCaLiZe and Zenon Modulo. We have already discussed the discouraged use of higher-order in

FoCaLiZe. The other limitations of FoCaLiZe come from its treatment of polymorphism:

- As we have seen, Prevosto showed in [150] that instantiating the type of a polymorphic method during inheritance is inconsistent and it is the reason why polymorphic methods are forbidden in FoCaLiZe. It is however possible to extend FoCaLiZe by polymorphic methods without allowing their type to change during inheritance at all. We believe this extension to be consistent because it does not seem to violate the translation scheme to Coq and Dedukti.

- The handling of polymorphism inside Zenon Modulo should be improved. During the instantiation of HOL and Coq representations of natural numbers, we used Dedukti proofs to instantiate polymorphic theorems because bugs in Zenon Modulo made it fail to find these proofs. These bugs should be fixed, they are critical for using FoCaLiZe in combination with HOL.

We tried to do as much work as possible in a system independent way. Our hierarchy of species and morphisms is totally independent of HOL and Coq and could be reused in similar situations. The fact that most of the development is totally symmetric between Coq and HOL is also very encouraging with respect to the generality of this approach. At the base layer of merging of logics, we however strongly commit to merge FoCaLiZe logic with HOL, thus breaking the symmetry. We map FoCaLiZe equality to HOL equality because we only consider terms which are convertible on the Coq side. Since Coqine is a shallow encoding, two convertible Coq terms t_1 and t_2 of type A are translated to convertible Dedukti terms t_1' and t_2' of type A' and `hol.REFL` A' t_1' is of type `hol.eq` A' t_1' t_2'. If we wanted to study interoperability of two systems using different axiomatizations of equality, we would rather specify it on the FoCaLiZe side. Such an axiomatization of equality is very common in FoCaLiZe.

Contrary to the previous example from Chapter 10, the sieve function that we defined in pure Coq in Section 11.1 is a certified program which can be run in Coq, Dedukti or, through Coq extraction, in OCaml. The lack of reduction behaviour on the HOL side does not impact the runnability of the function because HOL functions are only used for certifying the sieve function, they are called by the sieve function as was the case of the

sorting function in Chapter 10. FoCaLiZe compilation to OCaml is however not usable because we did not express the sieve function in FoCaLiZe. Our abstract treatment of natural numbers could in principle be used to replace the representation of numbers to a more effective one such as binary or machine integers if the efficiency of the run code becomes an issue.

Chapter 12

Proof Constructivization

When merging two theories T_1 and T_2 as we did for Coq and HOL, we end up with a theory T_3 which is stronger than both of T_1 and T_2. As we have seen when dealing with type inhabitation (see Section 10.1.1), this common theory can quickly become inconsistent if we are not careful enough.

For a given existing formal development, all the axioms of the theory might not be useful. For example, a recent index of the OpenTheory standard library[1] by the Proof Cloud search engine [172] revealed that 44.75% of the theorems (541 / 1209) do not depend on the law of excluded middle.

If an axiom is especially problematic or if its removal brings good properties, we can try to transform proofs depending on it in order to remove the dependency to the axiom. In this chapter, we focus on the classical axioms because their elimination is important for both the integration of classical proof assistants into intuitionistic proof assistants (in particular into proof assistants based in intuitionistic type theory such as Coq) and for interoperability between classical and intuitionistic proof assistants (such as interoperability between Coq and HOL).

Intuitionistic logic is usually presented as the fragment of classical logic obtained by removing the Law of Excluded Middle (or equivalent principles such as the Law of Double Negation) from the primitive axioms. Interestingly, it can also be seen as a supersystem of classical logic in the sense that classical formulae and proofs can be translated in

[1] See `https://airobert.github.io/proofcloud/hol_stdlib.html`

intuitionistic logic thanks to double-negation translations [107, 83, 77, 112, 110, 33, 68, 81].

Unfortunately, neither point of view is very practical when we want to use a classical theorem prover together with a constructive proof assistant. In the first interpretation, the classical prover is only usable if the Law of Excluded Middle is added as an axiom in the proof assistant thus limiting the interpretation of the proof as an algorithm. In the second interpretation, the classical prover is seen as only able to produce proofs for formulae belonging to a fragment of the syntax where double-negations are mandatory at certain positions.

In practice, classical provers often use the refutation method which consists in adding the negation of the goal as hypothesis and trying to prove the inconsistency of the set of hypotheses. The justification for this simplification is exactly the Law of Double Negation, hence every proof coming from a refutation-based theorem prover contains at least one occurrence of a classical principle. However, a lot of automatically generated classical proofs are believed to be only accidentally classical in the sense that they use classical principles at non-critical places so constructive proofs can be extracted from them; we call this *proof constructivization*. One goal of this section is to give an experimental lower-bound on the number of proofs which can be constructivized.

Proof constructivization is an inherently incomplete activity. It obviously has to fail when the classically proved formula is not constructively provable but also when intuitionistic proofs of the formula require ingredients which are not present in the classical proof.

Type theory usually attaches no computational behaviour to axioms. We propose however to interpret axioms such as the Law of Excluded Middle as partial functions defined by a set of meta-level rewrite rules in Dedukti. Normalizing a proof relying on some axiom with respect to this rewrite system may (or not) lead to an axiom-free proof of the same theorem.

In this section, we focus on the case of constructivization in first-order logic because it is the standard framework for automatic theorem provers such as Zenon and benchmarks such as the TPTP database are available for validating our approach. First-order logic is

represented by the deep encoding of Section 3.5.1.

The work described in this section has been presented at the LFMTP workshop [41].

In Section 12.1, we propose rewrite systems interpreting two classical axioms as partial functions. Normalizing classical proofs with respect to these rewrite systems might lead to constructive proofs. To increase the success rate, we propose additional rewrite rules in Section 12.2 that use higher-order rewriting to inspect the shape of the proof. We explain in Section 12.3 how we define strategies by choosing between different combinations of our rewrite systems. We then detail the constructivization process on the example of the proof of $A \implies A$ automatically produced by Zenon in Section 12.4. We evaluate in Section 12.5 our tool on the TPTP benchmark and discuss related work in Section 8.3.

12.1 Partial Definitions of Classical Axioms

There are a lot of possible axiom schemes for turning intuitionistic logic into classical logic, we will focus on two of them:

- the Law of Excluded Middle: $\varphi \vee \neg\varphi$
- the Law of Double Negation: $\neg\neg\varphi \Rightarrow \varphi$

Contrary to other schemes such as Pierce's law $((\varphi_1 \Rightarrow \varphi_2) \Rightarrow \varphi_1) \Rightarrow \varphi_1$, instantiating these schemes is done by providing just one formula. These schemes are equivalent but their instances are not: for a given formula φ, $\varphi \vee \neg\varphi$ constructively implies $\neg\neg\varphi \Rightarrow \varphi$ but the converse is false. Because of this, both schemes do not have the same computational behaviour.

12.1.1 A Rewrite System for the Law of Excluded Middle

Let us abbreviate by $\mathsf{LEM}(\varphi)$ the formula $\varphi \vee \neg\varphi$. The following are easy constructive theorems, their proofs are not very interesting but we give them in Figure 12.1 for the sake of completeness:

- $l_0 : \mathsf{LEM}(\top)$

- $l_1 : \mathsf{LEM}(\bot)$
- $l_2 : (\mathsf{LEM}(A) \wedge \mathsf{LEM}(B)) \Rightarrow \mathsf{LEM}(A \wedge B)$
- $l_3 : (\mathsf{LEM}(A) \wedge \mathsf{LEM}(B)) \Rightarrow \mathsf{LEM}(A \vee B)$
- $l_4 : (\mathsf{LEM}(A) \wedge \mathsf{LEM}(B)) \Rightarrow \mathsf{LEM}(A \Rightarrow B)$

Thanks to these theorems, we can define a first rewrite system R_{lem} pushing the classical axiom through the propositional connectives:

```
def lem : A : prop -> PLEM A.
[] lem true --> l0
[] lem false --> l1
[A,B] lem (and A B) --> l2 A B (lem A) (lem B)
[A,B] lem (or A B) --> l3 A B (lem A) (lem B)
[A,B] lem (imp A B) --> l4 A B (lem A) (lem B).
```

12.1.2 A Rewrite System for the Law of Double Negation

We can do the same job for other classical axioms such as the Law of Double Negation. Let $\mathsf{DN}(A)$ abbreviate $\neg\neg A \Rightarrow A$, the following are constructive theorems proved in Figure 12.2:

- $d_0 : \mathsf{DN}(\top)$
- $d_1 : \mathsf{DN}(\bot)$
- $d_2 : (\mathsf{DN}(A) \wedge \mathsf{DN}(B)) \Rightarrow \mathsf{DN}(A \wedge B)$
- $d_3 : \mathsf{DN}(B) \Rightarrow \mathsf{DN}(A \Rightarrow B)$
- $d_4 : (\forall x.\ \mathsf{DN}(P(x))) \Rightarrow \mathsf{DN}(\forall x.\ P(x))$

This leads to the following rewrite system R_{dn}:

```
def dn : A : prop -> DN A.
[p] dn true p --> d0 p
[p] dn false p --> d1 p
[A,B,p] dn (and A B) p --> d2 A B (dn A) (dn B) p
[A,B,p] dn (imp A B) p --> d3 A B (dn B) p
[a,A,p] dn (all a A) p --> d4 a A (x : term a => dn (A x)) p.
```

```
def LEM (A : prop) := or A (not A).
def PLEM (A : prop) := proof (LEM A).

def left (A : prop) : proof A -> PLEM A := or_intro_1 A (not A).
def right (A : prop) (p : proof A -> proof false) : PLEM A
:= or_intro_2 A (not A) (imp_intro A false p).

def l0 : PLEM true := left true true_intro.
def l1 : PLEM false := right false (p => p).
def l2_nA (A : prop) (B : prop) (p : proof (not A)) : PLEM (and A B)
:= right (and A B) (q => imp_elim A false p (and_elim_1 A B q)).
def l2_nB (A : prop) (B : prop) (p : proof (not B)) : PLEM (and A B)
:= right (and A B) (q => imp_elim B false p (and_elim_2 A B q)).
def l2 (A : prop) (B : prop) (p : PLEM A) (q : PLEM B)
    : PLEM (and A B)
:= or_elim A (not A) (LEM (and A B))
     (r => or_elim B (not B) (LEM (and A B))
         (s => left (and A B) (and_intro A B r s))
         (s => l2_nB A B s)
         q)
     (r => l2_nA A B r)
     p.

def l3_nAnB (A : prop) (B : prop)
            (p : proof (not A)) (q : proof (not B)) : PLEM (or A B)
:= right (or A B)
     (or_elim A B false (imp_elim A false p) (imp_elim B false q)).
def l3 (A : prop) (B : prop) (p : PLEM A) (q : PLEM B) : PLEM (or A B)
:= or_elim A (not A) (LEM (or A B))
     (r => left (or A B) (or_intro_1 A B r))
     (r => or_elim B (not B) (LEM (or A B))
         (s => left (or A B) (or_intro_2 A B s))
         (s => l3_nAnB A B r s)
         q)
     p.

def l4_B (A : prop) (B : prop) (p : proof B) : PLEM (imp A B)
:= left (imp A B) (imp_intro A B (q => p)).
def l4_AnB (A : prop) (B : prop)
           (p : proof A) (q : proof (not B)) : PLEM (imp A B)
:= right (imp A B) (r => imp_elim B false q (imp_elim A B r p)).
def l4_nA (A : prop) (B : prop) (p : proof (not A)) : PLEM (imp A B)
:= left (imp A B)
     (imp_intro A B (q => false_elim B (imp_elim A false p q))).
def l4 (A : prop) (B : prop) (p : PLEM A) (q : PLEM B)
    : PLEM (imp A B)
:= or_elim A (not A) (LEM (imp A B))
     (r => or_elim B (not B) (LEM (imp A B))
         (s => l4_B A B s)
         (s => l4_AnB A B r s)
         q)
     (r => l4_nA A B r)
     p.
```

Figure 12.1: Constructive instances of the Law of Excluded Middle

```
def DN (A : prop)
:= ((proof A -> proof false) -> proof false) -> proof A.

def d0 : DN true := p => true_intro.

def d1 : DN false := p => p (q => q).

def d2_A (A : prop) (B : prop) (p : DN A)
         (q : (proof (and A B) -> proof false) -> proof false)
         : proof A
:= p (r => q (s => r (and_elim_1 A B s))).
def d2_B (A : prop) (B : prop) (p : DN B)
         (q : (proof (and A B) -> proof false) -> proof false)
         : proof B
:= p (r => q (s => r (and_elim_2 A B s))).
def d2 (A : prop) (B : prop) (p : DN A) (q : DN B)
   : DN (and A B)
:= r : ((proof (and A B) -> proof false) -> proof false) =>
   and_intro A B (d2_A A B p r) (d2_B A B q r).

def d3 (A : prop) (B : prop) (p : DN B) : DN (imp A B)
:= q : ((proof (imp A B) -> proof false) -> proof false) =>
   imp_intro A B (r => p (s => q (t => s (imp_elim A B t r)))).

def d4 (a : type) (A : term a -> prop)
       (p : x : term a -> DN (A x))
       : DN (all a A)
:= q : ((proof (all a A) -> proof false) -> proof false) =>
   all_intro a A
             (x => p x (s => q (t => s (all_elim a A t x)))).
```

Figure 12.2: Constructive instances of the Law of Double Negation

These two rewrite systems are not very efficient at constructivizing proofs because they can do nothing smart on atoms. The rewrite system R_{lem} is only able to constructivize proofs for formulae without atoms or quantifiers; it simply computes boolean values. The rewrite system R_{dn} performs a bit better because the rewrite rule for implication $A \Rightarrow B$ works for any A; in particular $\mathsf{dn}(\neg A)$ reduces to a constructive proof so the rewrite system constructivizes proofs of double-negated formulae. Fortunately, we can go further by inspecting the proof term.

12.2 Inspecting the Proof

Seen as a function symbol in type theory, the symbol dn is a function of two parameters; the first one is a formula A, the second one is a proof of the formula $\neg\neg A$. The rewrite system that we have just presented only inspects the first argument A and acts independently of the second one. Thanks to higher-order rewriting, it is also possible to inspect the second one.

12.2.1 Two Trivial Special Cases

Regardless of the shape of A, there are two trivial ways in which a proof $\pi_{\neg\neg A}$ of $\neg\neg A$ can be constructivized into a proof of A:

- seen as a function from $\neg A$ to $\bot$, $\pi_{\neg\neg A}$ does not use its argument, hence the current context is inconsistent so we can build a proof of A
- $\pi_{\neg\neg A}$ is an instance of the canonical constructive proof of $A \Rightarrow \neg\neg A$ (which is $\lambda p : A.\ \lambda q : \neg A.\ q\ p$)

These two special cases can be written in Dedukti as higher-order rewrite rules R_1 and R_2:

```
[A,p] dn A (q => p) --> false_elim A p
[A,p] dn A (q => q p) --> p.
```

These rules restrict the positions in which the assumption q of $\neg A$ is allowed to appear; in order to favor their application, we consider proof transformations which make some proofs of $\neg A$ disappear.

12.2.2 Eliminating Negation Proofs

The typical case where a proof of $\neg A$ is useless is when it is eliminated to build a proof of A using the following rewrite rule R_3:

```
[A,p] false_elim A (imp_elim A false _ p) --> p.
```

In turn, to favor the application of this rewrite rule, we can give `false_elim` some freedom by adding the usual rewrite rules $R_{\mathsf{abort}-@}$ and $R_{\mathsf{abort}-\lambda}$ which interpret elimination of falsehood as error propagation:

```
[A,B,p] imp_elim A B (false_elim _ p) _ --> false_elim B p.
[A,B,p] imp_intro A B (x => false_elim _ p) -->
            false_elim (imp A B) p.
```

Similar rewrite rules for all introduction and elimination rules can be added this way.

Another option for eliminating dn is to make it progress toward the leaves in the hope that R_1 will be applicable in some branches and R_2 in others; this is the topic of next subsection.

12.2.3 Exchanging Elimination Rules

We further inspect the proof of $\bot$ that missed to be captured by the pattern `p` in R_1 and the pattern `q p` in R_2 by looking at where it does use the hypothesis `p`. $\bot$ has no introduction rule so it can only be proved by an elimination rule. Elimination rules for disjunction and existential can be traversed by dn if the required proof of $B \vee C$ and $\exists x : \tau.\ \varphi$ respectively do not use the assumption p:

```
[A,B,C,q,r,s]
    dn A (p => or_elim B C _ (q p) (r p) s)
      -->
    or_elim B C A (t => dn A (p => q p t)) (t => dn A (p => r p t)) s
[A,B,C,q,r,s]
    dn A (p => ex_elim a B _ (q p) r)
      -->
    ex_elim a B A (x => s => dn A (p => q p x s)) r.
```

To ease triggering of these new rules, we want to push elimination rules for disjunction and existential toward the root of the formula in the hope that they will meet the dn symbol and help it progress toward the leaves of the proof.

We avoid commuting with introduction rules because it goes a lot against cut-elimination and does not seem useful in practice for normal forms with respect to the rewrite system R_{dn}. Commuting with other elimination rules is however achieved easily:

```
[A,B,C,p,q,r]
    false_elim C (or_elim A B _ p q r)
      -->
    or_elim A B C
            (s => false_elim C (p s))
            (s => false_elim C (q s))
            r.
[A,B,C,D,p,q,r]
    and_elim_1 C D (or_elim A B _ p q r)
      -->
    or_elim A B C
            (s => and_elim_1 C D (p s))
            (s => and_elim_1 C D (q s))
            r.
[A,B,C,D,p,q,r]
    and_elim_2 C D (or_elim A B _ p q r)
      -->
    or_elim A B D
            (s => and_elim_2 C D (p s))
            (s => and_elim_2 C D (q s))
            r.
[A,B,C,D,p,q,r,s]
    imp_elim C D (or_elim A B _ p q r) s
      -->
    or_elim A B D
            (t => imp_elim C D (p t) s)
            (t => imp_elim C D (q t) s)
            r.
[A,B,a,C,p,q,r,x]
    all_elim a C (or_elim A B _ p q r) x
      -->
    or_elim A B (C x)
            (t => all_elim a C (p t) x)
            (t => all_elim a C (q t) x)
            r.

[a,A,B,p,q]
    false_elim B (ex_elim a A _ p q)
      -->
    ex_elim a A B (x => r => false_elim B (p x r)) q.
[a,A,B,C,p,q]
    and_elim_1 B C (ex_elim a A _ p q)
      -->
    ex_elim a A B (x => r => and_elim_1 B C (p x r)) q.
[a,A,B,C,p,q]
    and_elim_2 B C (ex_elim a A _ p q)
      -->
```

```
    ex_elim a A C (x => r => and_elim_2 B C (p x r)) q.
[a,A,B,C,p,q,r]
    imp_elim B C (ex_elim a A _ p q) r
      -->
    ex_elim a A C (x => s => imp_elim B C (p x s) r) q.
[a,A,b,B,p,q,y]
    all_elim b B (ex_elim a A _ p q) y
      -->
    ex_elim a A (B y) (x => s => all_elim b B (p x s) y) q.
```

12.2.4 Confluence

The rules R_1 and R_2 are not confluent with the rewrite system R_{dn} of Section 12.1. For example, the term `p : proof false => dn true (q => p)` reduces to `p => true_intro` with respect to R_{dn} and to `p => false_elim true p` with respect to R_1. Even worse, the rules of Section 12.2.2 are to be used together but they form a non-confluent rewrite system: the term `imp_elim A B (false_elim (imp A B) (imp_elim (imp A B) false p q)) r` reduces to both `imp_elim q r` (using R_3) and `false_elim B (imp_elim (imp A B) false p q)` (using $R_{\mathsf{abort}-@}$).

In order to obtain the best behaviour out of our rewrite systems, we need to give them priorities.

12.3 Combining Rewrite Systems

As we have seen in Section 3.3, Dedukti is intended to be used with confluent rewrite systems so it does not provide a way for controlling the strategy. It does however provide a command for printing a normal form of a term with respect to a rewrite system; this gives us two ways of combining two rewrite systems R_A and R_B:

- union: we can ask Dedukti to compute $\longrightarrow^*_{R_A \cup R_B}$ by writing both systems in the same file
- sequence: by calling Dedukti twice, we can reduce terms using the relation $\longrightarrow^*_{R_A} \times \longrightarrow^*_{R_B}$, that is we can ask for normal forms with respect to R_B of normal forms with respect to R_A.

Moreover, the order in which the rewrite rules are given in a non-confluent Dedukti file is relevant: the earlier a rule is declared the higher its priority. For giving the rule R_3 priority over $R_{\mathsf{abort}-@}$ and $R_{\mathsf{abort}-\lambda}$, we just have to declare it first.

In which way to combine the rewrite systems of previous sections is a matter of heuristic choice; in practice, a good strategy consists in trying first the rules which remove axioms ($R_1 \cup R_2'$), then rules reducing the formula (R_{dn}), then rules pushing the axioms toward the leaves at the expense of exchanging the order of the elimination rules (the rewrite system of Section 12.2.3) and finally the union of all the rewrite systems for dn presented in this paper together with cut-elimination rules.

12.4 Example: Zenon Classical Proof of $A \Rightarrow A$

Once translated to natural deduction, the classical proof of $A \Rightarrow A$ that comes out of Zenon is the following term:

```
A : prop.
def example_1 : proof (imp A A)
:= dn (imp A A)
      (p =>
       p (imp_intro A A (q =>
         false_elim A
            ((r : proof A => p (imp_intro A A (s => r))) q)))).
```

Two rules apply here, the rule for implication from system R_{dn} and the rule R_3 for elimination of $\bot_E$. Following the heuristic strategy of Section 12.3, the first step ($R_1 \cup R_2$) is skipped and we apply the rule in R_{dn} leading to the following term:

```
def example_2 : proof (imp A A)
:= d3 A A (dn A)
      (p =>
       p (imp_intro A A (q =>
         false_elim A
            ((r : proof A => p (imp_intro A A (s => r))) q)))).
```

we now need to unfold the definition of d_3:

```
def d3 (A : prop) (B : prop) (p : DN B) : DN (imp A B)
:= q : ((proof (imp A B) -> proof false) -> proof false) =>
   imp_intro A B (r => p (s => q (t => s (imp_elim A B t r)))).
```

so our proof of $A \Rightarrow A$ is now

```
def example_3 : proof (imp A A)
:= imp_intro A A (t => dn A (u : (proof A -> proof false) =>
      (p : (proof (imp A A) -> proof false) =>
       p (imp_intro A A (q =>
         false_elim A
            ((r : proof A => p (imp_intro A A (s => r))) q))))
      (v : proof (imp A A) => u (imp_elim A A v t)))).
```

we now perform elimination of `false_elim` (R_3), which has priority over cut elimination for implication:

```
def example_4 : proof (imp A A)
:= imp_intro A A (t => dn A (u : (proof A -> proof false) =>
      (p : (proof (imp A A) -> proof false) =>
       p (imp_intro A A (q => q)))
      (v : proof (imp A A) => u (imp_elim A A v t)))).
```

we now perform cut elimination:

```
def example_5 : proof (imp A A)
:= imp_intro A A (t => dn A (u : (proof A -> proof false) => u t)).
```

and finally apply R_2, getting rid of the classical axiom:

```
def example_6 : proof (imp A A)
:= imp_intro A A (t => t).
```

As we can see, the translation to natural deduction has introduced a fortunate cut. Reducing this cut would forbid to fire R_3 but we need cut elimination to simplify the resulting proof so that R_2 can in turn be fired.

12.5 Experimental Results

We have performed tests on the latest version (v6.3.0) of the reference library for first-order problems: TPTP. This library contains 6528 problems for first-order logic (TPTP FOF format). We filtered these problems by running Zenon with a short timeout[2]. Zenon claimed to have proved 1371 problems which form our starting benchmark. For every problems in this benchmark but two, Zenon provided a proof in classical sequent calculus that we type-checked in Dedukti. Among these 1369 proofs, 1258 (91.9%) were translated

[2]The choice of this timeout does not affect much the results because the number of proofs found by Zenon in more than a few seconds is very low. It has however a direct impact on the time needed for running the benchmark since this timeout is reached on most TPTP problems.

to classical natural deduction. Among these natural deduction proofs, 1240 (98.6%) were normalized by the combination of rewrite systems presented in Section 12.3. All these normalized natural deduction proofs were rechecked in Dedukti and 856 (69.0% of the normalized classical proofs) were checked in intuitionistic natural deduction.

As a constructivization tool for natural deduction, our approach succeeded for 68.0% of the classical proofs. We can distinguish four sources of failure:

1. normalization reaches memory or time limits because matching of higher-order patterns can be costly;
2. some TPTP problems are classical theorems but have no constructive proofs;
3. some problems have constructive proofs but these proofs require ingredients that are not present in the classical proof provided by Zenon, a typical example would be a formula of the form $\varphi \vee P \vee \neg P$ where φ has a complex intuitionistic proof, finding such proofs would require intuitionistic proof search and is out of the scope of our approach;
4. because our approach is heuristic, it is fundamentally incomplete so other proofs are missed, these problems are a good source of inspiration for further improving our heuristics.

The first source of failure affects only 21 proofs (1.7% of the proofs in classical natural deduction). The second source is very hard to count: one goal of the ILTP library [151] was to associate a constructive status to TPTP problems but the majority of them (69.7% for ILTP v1.1) remains unsolved or open. Finally, when an intuitionistically valid problem fails to be constructivized by our approach, it is not always clear whether the failure comes from the third or the fourth source because we did not formalize the notion of ingredient present in a classical proof; for example, the formula $P \Rightarrow (P \vee \neg P)$ has two classical proofs, a constructive one and a non-constructive one, our technique fails to constructivize the non-constructive one $\lambda H_A.\, \mathsf{lem}(A)$ as it requires to query the proof context, an operation which can be seen as a very limited form of proof search.

The details for each TPTP category of problem are summarized in Figure 12.3.

The experimental conditions for this study were the following:

- Processor: Intel Core i5-4310M @ 2.70GHz
- Timeouts: 10 seconds for Zenon filtering phase, 10 minutes for each Dedukti call
- Tools versions: We used development versions of the tools built from their respective git repositories (`git://scm.gforge.inria.fr/dedukti/dedukti.git` for Dedukti and `git://scm.gforge.inria.fr/zenon/zenon.git` for Zenon). More precisely, Dedukti was built from branch `develop` (latest commit: April 11th 2016), Zenon was built from branch `modulo_intuit` (latest commit: February 5th 2016).

12.5.1 B Proof Obligations

Our approach is very easy to adapt to Deduction modulo and typing so we can also benchmark it on the proofs produced by Zenon Modulo. The main benchmark for Zenon Modulo is generated from the proof obligations of Atelier B [62]. In the same conditions as before, Zenon Modulo is able to prove 8687 problems among which 6823 can be translated in natural deduction in the required amount of time and memory. The word "problem" is here to be understood as we defined it in Chapter 1.1, that is a first-order theory in Deduction modulo consisting of axioms and rewrite rules together with a formula called the goal of the problem which Zenon Modulo is asked to prove in the given theory. Our constructivization procedure is then able to normalize 5398 of them and only 20 of the final proofs depend on classical axioms.

As a constructivization procedure for natural deduction, we get a constructivization rate of 78.8%.

12.5.2 FoCaLiZe Standard Library

In FoCaLiZe, the user is encouraged to write decidable predicates as functions to the primitive type **bool** because such predicates can be exported to OCaml. The type **bool** is injected into the syntax of formulae by an implicit predicate `Is_true`

```
bool : type.
btrue : term bool.
```

	LK	**NK**	**Normalized**	**NJ**
AGT	17	17	17	13
ALG	23	13	11	7
CAT	2	2	2	2
COM	11	11	11	11
CSR	91	91	87	57
GEO	213	210	210	203
GRA	3	2	2	2
GRP	4	4	4	3
HWV	3	3	3	0
KLE	6	6	6	2
KRS	62	62	62	12
LAT	9	9	8	7
LCL	28	8	8	6
MED	4	4	4	3
MGT	39	38	35	23
MSC	5	5	5	3
NLP	11	11	11	6
NUM	101	92	92	78
PUZ	11	11	10	6
RNG	24	24	24	21
SCT	7	7	7	6
SET	135	135	135	105
SEU	84	80	80	66
SWB	21	21	21	20
SWC	43	43	43	1
SWV	132	132	131	111
SWW	11	11	11	10
SYN	264	201	195	67
SYO	2	2	2	2
TOP	3	3	3	3
Total	1369	1258	1240	856

Each line is a TPTP category of problems;

- the **LK** column contains the number of proofs found by Zenon in classical sequent calculus
- the **NK** column contains the number of proofs translated in classical natural deduction
- the **Normalized** column contains the number of classical proofs which have been normalized with respect to our rewrite systems
- the **NJ** column contains the number of proofs for which constructivization has succeeded in a proof in intuitionistic natural deduction

Figure 12.3: Per-category results

```
bfalse : term bool.
def Is_true : term bool -> prop.
[] Is_true btrue --> true
[] Is_true bfalse --> false.
```

For this reason, we get a high constructivization rate on FoCaLiZe problems by simply adding the following rewrite rule in R_{lem}:

```
def lem_b : b : term bool -> PLEM (Is_true b).
[] lem_b btrue --> l0
[] lem_b bfalse --> l1.
[b] lem (Is_true b) --> lem_b b.
```

Note that the lemma `lem_b` is perfectly valid in intuitionistic logic.

FoCaLiZe standard library contains 437 first-order problems in Deduction modulo which are all proved by Zenon Modulo. Among the produced proofs, 422 are translated in natural deduction. We were able to normalize 387 of them and only 3 proofs still required classical axioms.

As a constructivization procedure for natural deduction, we get a constructivization rate of 91.0%.

12.6 Related Work

The differences between classical and intuitionistic logic have been deeply studied since the early days of intuitionistic logic leading to the discovery of double-negation translations and extensions of the Curry-Howard correspondence to classical logic. Concretely, a few automated theorem provers, iLeanCoP in particular, can be used for intuitionistic logic but their integration in intuitionistic proof assistants is far from easy because they do not yet provide proof certificates in a checkable format. We know only one exception to this rule: a constructivization module for Zenon called Zenonide which is able to produce proofs in Dedukti format.

12.6.1 Double-Negation Translations

It is usually easy to test whether a formula is in the image of a given double-negation translation. Since for such formulae intuitionistic provability corresponds to classical prov-

ability, this provides a simple criterion for proof constructivization which does not depend on the classical proof but only on the proven formula. This criterion is not very powerful but it is very efficient: typically in linear time and finite memory.

Our first rewrite system for the Law of Double Negation R_{dn} is related to the way one can replace a double-negation translation by another one. Because the right-hand side of the rule for $\mathsf{dn}(A \Rightarrow B)$ uses $\mathsf{dn}(B)$ but not $\mathsf{dn}(A)$, the correct way to look at R_{dn} is as a transformer for polarized double-negation translations [33].

In the particular case of Zenon proofs in classical sequent calculus and their translation to classical natural deduction, the Law of Double Negation is used at the head of the proof and after introduction of universal quantification only. Because Zenon finds cut-free proofs in sequent calculus, the subformula property guarantees that all double negations corresponding to these classical axioms appear in positions where the polarized version of Gödel-Gentzen double-negation translation would also have added a double-negation. After normalization by our rewrite system R_{dn}, they are placed at positions where a lighter translation, Gilbert's double-negation translation [81], would also put double-negations.

12.6.2 Intuitionistic Provers

A few automated theorem provers for intuitionistic logic have been developed. The ILTP library [151] is a benchmark constructed from the TPTP problems in FOF format (non-clausal first-order formulae) to evaluate intuitionistic provers. The most performant intuitionistic first-order prover on this benchmark is by far the iLeanCoP prover. It is noticeable that iLeanCoP is built as a constructivization extension of a classical prover, LeanCoP.

The main difference between our work and an intuitionistic prover such as iLeanCoP is that, in case of failure, iLeanCoP can ask LeanCoP to provide another classical proof. For example, our technique fails to constructivize the following proof of $A \Rightarrow (A \vee \neg A)$: $\lambda H_A.\, \mathsf{lem}(A)$. Backtracking makes however iLeanCoP complete for first-order intuitionistic logic. According to [138], this backtracking feature is rarely used because the first classical proof is usually constructive.

Unfortunately, intuitionistic provers such as iLeanCoP do not produce certificates so we can not easily integrate them in intuitionistic proof assistants.

12.6.3 Zenonide

Zenonide is a constructivization module for Zenon developed by Frédéric Gilbert. Because Zenonide has access to the internal representation of proofs in Zenon, it has access to the proof context for each proof node so the constructivization of Zenon proof of $A \Rightarrow A$ is trivial for Zenonide whereas we have seen in Section 12.4 that it required some work in our case.

Zenonide is however not able to backtrack to another classical proof as iLeanCoP so it is an interesting middle point between our approach to constructivization and intuitionistic proof search.

Zenonide does not need to translate the classical proof to natural deduction, it tries to transform a proof in classical sequent calculus to a proof in intuitionistic sequent calculus so it avoids the combinatorial explosion appearing with some problems of the syntactic category of TPTP which have been especially designed to have no small proof in natural deduction. The price to pay for using sequent calculus is that more commutations of deduction rules have to be taken into account.

Zenon, as many theorem provers, searches for cut-free proofs; this is a very good point for Zenonide since the cut rule in sequent calculus behaves very badly with proof constructivization. Our input proofs do however use natural deduction cuts and these cuts are, as we have seen, sometimes welcomed.

For all these reasons, Zenonide globally performs better than our rewrite systems but also fails on some proofs that we manage to constructivize: on the set of 1371 problems proved by Zenon on TPTP, Zenonide proves 915 problems constructively but a very fair amount of proofs (113) was constructivized by our rewrite systems despite Zenonide lacks to prove it. If we use our rewrite systems together with Zenonide, we obtain a total constructivization rate of 81.7% of the tableau proofs found by Zenon on TPTP.

12.6.4 Extensions of the Curry-Howard Correspondence for Classical Logic

The Curry-Howard correspondence, which is at the heart of the use of logical frameworks such as Dedukti for checking proofs, has been extended to classical reasoning in several ways.

Minimal logic can be extended to a classical logic of implication by Pierce Law $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ which is a possible type for the call-cc control operator found in the Scheme programming language for example. This remark led to Parigot's $\lambda\mu$-calculus which corresponds to a classical extension of minimal natural deduction where several formulae are allowed at the right side of sequents [140].

Classical sequent calculus has also been the subject of interpretations through the Curry-Howard correspondence leading to Curien and Herbelin's $\bar{\lambda}\mu\tilde{\mu}$-calculus for minimal classical sequent calculus [57].

An interpretation of classical logic in terms of stack manipulations is also investigated in the context of classical realizability [111].

All these systems suffer, as we do, from a lack of confluence but this is directly connected with non-confluence of classical cut elimination whereas we do not even need to consider cut-elimination to loose confluence.

As extensions of typed λ-calculus, it is possible to ask in these systems whether a given term (that is, a classical proof) reduces to a pure λ-term (that is, a constructive proof). Conversely, the rewrite systems that we have proposed can be seen as alternative semantics or program transformations in these systems.

We believe that Dedukti is a good framework for studying extensions of these systems to first-order logic.

Conclusion of Part IV

In Chapter 10, we have achieved a simple interoperability proof of concept consisting in an instantiation of a certified Coq sorting program by the datatype of natural numbers imported from the OpenTheory library of HOL proofs. Despite the very restricted interaction between Coq and HOL in this first example, gluing the developments in Dedukti to obtain our final theorem was tedious. Dedukti is a mere proof checker, not an interactive proof assistant; it features almost no automation.

In order to scale to a more reasonable example where non-logical operations such as arithmetic operations are defined on both sides of the development and need to be related in Dedukti, we added FoCaLiZe and Zenon to our toolbox in Chapter 11 in order to exploit the automation they offer. The main contribution of this second proof of concept of interoperability in Dedukti is a methodology for proof exchange based on FoCaLiZe object-oriented structures.

When combining logical systems, we should pay particular attention to avoid bringing inconsistencies in the combined logic otherwise the logical validity of the resulting theorems is compromised. To facilitate this task, we propose Meta-Dedukti as a general axiom eliminator which can be used to remove dependencies on unwanted axioms in existing formal developments. We have focused on classical axioms in Chapter 12 for two reasons. First, they are important for interoperability between type theoretical proof assistants on one side and both automatic and interactive classical theorem provers on the other side. Second, constructivization is easy to evaluate because large libraries of classical proofs are available in Dedukti. Now that automatic constructivization through Meta-Dedukti normalization has been validated on various benchmarks for first-order theorem provers, the method is ready to be adapted to higher-order logic and to other axioms. According to

Proof Cloud [172], the HOL proofs that we imported from OpenTheory in Chapters 10 and 11 are classical. We believe that a large part of the OpenTheory library still remains to be constructivized but the rewrite system of Chapter 12 will probably require adaptations. Automatic elimination of other axioms such as functional extensionality and univalence is the topic of ongoing research [45].

Conclusion

In this thesis, we have embedded an object-calculus in Coq and Dedukti in Part II. We have taken advantage of the ability of Dedukti to model computation by rewriting to translate the operational semantics of the ς-calculus by a non-terminating rewrite system. Doing so, we can use Dedukti as an evaluator for object-oriented programs. We have extended the encoding to handle object subtyping using explicit coercions defined by partial functions in Dedukti. Our first naive implementation was quite inefficient; it spent about 99% of its time doing string comparisons. Using Dedukti as a meta-programming tool, we have achieved a considerable speed-up.

In Part III, we have proposed a translation to Dedukti for one of the few object-oriented logical systems, FoCaLiZe. Compiling a polymorphic λ-calculus such as the one of Section 2.3.1 in Dedukti is an easy exercise but real implementations of functional languages feature pattern matching and recursion for which some work is needed. We have integrated this translation as an extension of the FoCaLiZe compiler which improves its performances and opens the way to new applications of FoCaLiZe such as highly computational program verification.

The development of Dedukti and translation tools to the Dedukti language are motivated by interoperability of proof systems. As such, this thesis is a direct continuation of those of Saillard [155] and Assaf [11]. Saillard made Dedukti a lot more efficient so that it became possible to recheck in Dedukti large libraries of proofs such as OpenTheory standard library, FoCaLiZe standard library and various libraries generated by the automatic theorem provers Zenon Modulo and iProver Modulo. He also added higher-order rewriting in Dedukti, which we intensively used in Chapter 12 and bridged Dedukti with higher-order confluence checkers such as CSI^HO. Assaf developed Holide and Coqine, the translators

that we used in case studies described in Chapters 10 and 11. Thanks to Saillard and Assaf, Dedukti, Holide, and Coqine were mature enough to start working on interoperability.

In Part IV, we have conducted a first experiment with Assaf presented in Chapter 10. While successful, this experiment has shown that modularity and automation are required for scalability. We have then combined more existing tools producing Dedukti code: Coqine, Holide, Zenon Modulo, and Focalide. We have managed to develop a non-trivial proof in this combination and we have drawn a methodology for interoperability of proof systems out of this experimentation. FoCaLiZe has many interesting aspects which make it a good interoperability framework. FoCaLiZe is a compiled language which already featured a shallow translation to Coq. External definitions and proofs facilitate to link to the code of the target languages, the usability of FoCaLiZe for our interoperability experiment would have significantly decreased if it did not implement this feature. The integration of Zenon (Modulo) is a nice bonus which certainly saved us a lot of time compared to writing the interoperability proofs directly in Dedukti but we felt it harder to predict than the tactic-based interaction of Coq. We appreciate however the simplicity of FoCaLiZe proof language. FoCaLiZe object-oriented structuration mechanisms help to fulfill the requirements and we believe it has made an important part of our development directly reusable for other similar proof exchanges, even between other proof systems. Since FoCaLiZe has not been developed with this particular application in mind we faced some limitations related to polymorphism and higher-order programming and reasoning but these were quite easy to work around in our use case. If these issues slow down future bigger interoperability developments in FoCaLiZe, we believe that it will be possible without too much work to extend FoCaLiZe toward polymorphism and higher-order reasoning. For example, FoCaLiZe could be able to handle the instantiation of second-order induction principles and discharge to Zenon (Modulo) the required proofs.

Combining logical systems quickly leads to inconsistency. In Chapter 10, we have avoided some relatively simple inconsistencies but we did not formally prove that the combined logic is consistent. In general, when combining two logics, consistency is the most important property of the combination to prove because it allows not to translate back the proofs obtained by interoperability into one of the original systems. To simplify such

consistency proofs, we propose to first minimize the dependencies to the axioms which are present in only one of the systems. In the particular case of the classical axioms, Wang showed that about half of OpenTheory standard library does not depend on classical axioms and we obtained in Chapter 12 constructivization rates ranging from 68 to 91% depending on the benchmarks. These good results show that it is reasonable to try to eliminate the classical axioms from classical developments before transferring these developments to a constructive system so that these classical axioms are not needed in the combined logic. We believe that the approach to axiom elimination that we adopted in Chapter 12 can be generalized to other common axioms such as extensionality, choice, and univalence axioms. Important Dedukti libraries of proofs using these axioms are however first needed because in our experience the best way to discover which rewrite rules are helpful is to inspect the normal forms resulting from axiom elimination failures. The Dedukti translation of OpenTheory standard library given by Holide is a good starting point for elimination of functional extensionality and choice axioms. Future work in this direction would consist in defining rewrite systems similar to the one that we have proposed for proof constructivization but specialized to these two axioms.

In this thesis, we often took an important deviation from the orthodox use of Dedukti consisting of first defining a confluent and terminating rewrite system and then devising translations for terms and proofs that preserve typing modulo this rewrite system. The main advantage of Dedukti compared to other logical frameworks is the possibility to devise very shallow encodings preserving the notion of reduction of the object languages. In the case of programming languages, no terminating rewrite system can support a shallow encoding so we have to choose between keeping the translation shallow at the price of decidability of type-checking or returning to deeper encodings. Actually, the question of the termination of a term in a rewrite system is only worth asking when the rewrite system is not terminating. Our shallow encodings of programming languages in Dedukti can be used to reduce termination of object-oriented and functional programs to termination of terms in non-terminating higher-order rewrite systems.

The meta-theory of the $\lambda\Pi$-calculus modulo gets greatly simplified if we restrict the syntax of terms to the λ-free fragment. In this fragment, functions are still definable

by rewriting, β-reduction is vacuous so the conversion relation is simply defined by the congruence generated by the rewrite system. Confluence of the untyped system becomes decidable when the rewrite system terminates. If higher-order rewrite rules are not used, all the first-order termination and confluence checkers are available.

In the case of FoCaLiZe, we can hope for using Focalide together with automated termination tools to automatically prove the termination of functions. These termination proofs should then be communicated back to FoCaLiZe so that they could be used in the Coq backend for which termination proofs are mandatory. Automation of termination proofs in the Coq backend of FoCaLiZe has been considerably improved recently [71] with the possibility of proving termination of functions using measures and well ordering. Zenon can be used to solve some generated proof obligations but it does not guess the required definition of the well-founded ordering or the measure that currently needs to be provided by the user. Certifying termination checkers such as AProVE [80] could hopefully be used for providing the definitions of the required measures and well orderings. Focalide could be used to express FoCaLiZe recursive functions as rewrite systems that termination checkers understand.

We often hacked what could be considered deficiencies of Dedukti and turned them into useful features. We defined rewrite systems transgressing all the usual and reasonable requirements on purpose: termination, confluence, totality of functions, and even consistency. All these badly behaved rewrite systems have their utility, notably in term of efficiency.

We also pioneered the use of Dedukti as a meta-language. We have shown in Chapter 6 that Meta-Dedukti can help deciding properties in Sigmaid to increase the efficiency and the readability of generated code and we presented in Chapter 12 rewrite systems for eliminating axioms at the meta-level. These applications are very different in nature and we hope that many other applications of Dedukti as a meta-language will be discovered. In particular, we expect meta-level rewrite rules to be helpful for the interactive development of Dedukti terms.

This thesis opens many perspectives. The study of Dedukti encodings of other programming paradigms such as imperative programming and other program verification logics

such as Hoare logic, separation logic, and temporal logic would support Dedukti's claim to universality, contribute to more interoperability in program verification, and facilitate the development of more complex Dedukti programs. We believe that Meta-Dedukti and encodings of programming languages in Dedukti can benefit each other: we have seen in Chapter 6 that Meta-Dedukti can simplify and improve the efficiency of a translator and the issues related to termination of the shallow encodings are less relevant at the meta-level where consistency is not required. Meta-Dedukti itself can be improved in many ways. To deal with non-confluent rewrite systems such as our rewrite system for proof constructivization, backtracking abilities would be appreciated. With respect to interoperability of proof systems, and more particularly interoperability of interactive proof assistants, the generality of our approach should be stressed by trying different combinations of proof systems in particular Matita and PVS for which formal libraries are being translated to Dedukti and Agda, LEAN, and Isabelle whose logical foundations are understood as fragments of CIC and HOL for which Dedukti translators are available. Finally Dedukti should become more liberal in its input in order to ease the development of Dedukti backends for automatic tools which often provide incomplete traces. We believe this objective can be achieved by establishing a connection between Dedukti and the certificate checker Checkers of the ProofCert project.

Part V

Résumé en Français

Dans ce chapitre, nous reprenons en français les résultats de ce manuscrit.

Le vingtième siècle a vu le développement de nombreuses logiques et de nombreux logiciels permettant de démontrer des théorèmes de ces logiques sur ordinateur, soit de manière totalement automatique dans le cas des démonstrateurs automatiques de théorèmes soit en interaction avec l'utilisateur dans le cas des assistants de preuve.

Un obstacle actuel à l'utilisation des systèmes de preuve à plus grande échelle est l'absence d'intéreopérabilité. Contrairement aux langages de programmation, qui sont fréquemment utilisés de manière combinée pour bénéficier de leurs différents avantages dans de nombreux projets logiciels, même les développements de preuves formelles les plus importants comme la preuve du théorème de Feit-Thomson [85] sont réalisés à l'aide d'un unique assistant de preuve.

Les systèmes de preuve sont pourtant très spécialisés et proposent donc des fonctionnalités complémentaires qui pourraient grandement simplifier le travail de leurs utilisateurs si ces fonctionnalités pouvaient être utilisées simultanément. La raison pour laquelle l'interopérabilité est actuellement absente des processus de développement de preuves formelles est que l'interopérabilité entre systèmes de preuves est un problème difficile. Les systèmes de preuves implantent des logiques différentes et combiner des preuves provenant de logiques différentes nécessite en général de les exprimer dans des logiques très puissantes dont la cohérence n'est pas évidente. Par ailleurs, même pour échanger des preuves entre deux systèmes implantant exactement la même logique, un travail conséquent peut être nécessaire si la preuve est de taille importante ; cette situation est par exemple apparue dans le cadre du projet Flyspeck [89] lorsque la preuve obtenue dans le système HOL Light a été importée dans le système Isabelle/HOL [103].

Afin d'étudier la combinaison de preuves provenant de systèmes différents, l'équipe Deducteam au sein de laquelle cette thèse a été effectuée propose un format de preuve à vocation universelle qui s'appelle Dedukti [155]. Afin de comprendre si Dedukti peut effectivement servir de format d'échange entre les systèmes de preuve, il est important de traduire un maximum de systèmes de preuve différents dans Dedukti mais ce n'est pas suffisant pour résoudre le problème d'intéropérabilité puisqu'il faut également être capable de lier des preuves formelles en Dedukti provenant de systèmes différents.

Au commencement de cette thèse, seuls quelques traducteurs vers Dedukti existaient et la priorité étaient donc de traduire de nouveaux systèmes avant de se pencher sur le problème de la liaison de preuves Dedukti. La contribution principale de cette thèse est le développement d'un nouveau traducteur pour un système logique. Ce système logique est l'environnement de développement de programmes certifiés FoCaLiZe. FoCaLiZe est à la fois un système de preuve et un langage de programmation compilé. Traduire FoCaLiZe vers Dedukti a donc entre autre nécessité la compilation d'un langage de programmation vers Dedukti. Grâce à cette traduction de FoCaLiZe en Dedukti, les mécanismes orientés-objet statiques de FoCaLiZe peuvent être utilisés pour faciliter l'échange de preuves Dedukti entre différents systèmes.

Nous proposons dans cette thèse des traductions pour FoCaLiZe ainsi que pour deux paradigmes de programmation populaires, la programmation orientée-objet et la programmation fonctionnelle. De plus, afin de mieux comprendre comment les preuves formelles peuvent être mises en relation une fois traduites en Dedukti, nous avons expérimenté l'utilisation des mécanismes orientés-objet de FoCaLiZe pour l'échange de preuve entre les assistants de preuve Coq et HOL. Dans la suite de cet aperçu en français de nos travaux de thèse, nous allons préciser un peu ces contributions. Le chapitre 13 présente notre traduction d'un calcul orienté-objet avec sous-typage en Dedukti. Le chapitre 14 présente notre traduction du paradigme fonctionnel et de l'environnement FoCaLiZe dans Dedukti. Le chapitre 15 traite de notre principale expérience d'interopérabilité reposant sur FoCaLiZe et Dedukti.

Chapter 13

Un Calcul Orienté-Objet et sa Traduction en Dedukti

Le ς-calcul [1] a été proposé par Abadi et Cardeli comme calcul pour formaliser le paradigme de programmation orienté-objet. Il joue donc pour ce paradigme un rôle similaire à celui que joue pour la programmation fonctionnelle le λ-calcul.

Nous nous intéressons ici au ς-calcul simplement typé parce que c'est le calcul le plus simple permettant d'étudier le sous-typage des objets.

La syntaxe de ce calcul est donnée dans la figure 13.1 ; il y a deux classes syntaxiques distinctes, les types et les termes. Les types sont des enregistrements possiblement vides de types ; chaque étiquette ne peut apparaître qu'une fois et l'ordre des éléments n'est pas pris en compte tant que chaque étiquette l_i reste associée au même type A_i. Les termes sont soit des variables, soit des objets concrets, soit des sélections de méthode soit des mises-à-jour de méthode. Un objet concret est un enregistrement possiblement vide de méthodes, chaque méthode étant un terme (le corps de la méthode) lié à l'objet concret lui-même par le lieur ς. Lorsque ce lieur n'est pas utilisé, nous l'omettrons pour simplifier l'écriture ; nous écrirons par conséquent respectivement $[l = []]$ et $[l = \varsigma(x : A)x.l].l \Leftarrow []$ au lieu de $[l = \varsigma(x : A)[]]$ et $[l = \varsigma(x : A)x.l].l \Leftarrow \varsigma(x : A)[]$ où A est le type $[l : []]$.

Les règles de typage du ς-calcul simplement typé sont listées dans la figure 13.2, la plupart sont sans surprise. Les variables sont typées grâce au contexte de typage Δ. Un objet concret $[l_i = \varsigma(x_i : A)a_i]$ est de type $A = [l_i : A_i]$ si chaque a_i est de type A_i. Si a est de type $[l_i : A_i]$ alors $a.l_i$ est de type A_i. La mise-à-jour de méthode renvoie un nouvel

Type	$A, B, \ldots$	$::=$	$[l_i : A_i]_{i=1\ldots n}$	*Type d'objets*
Terme	$a, b, \ldots$	$::=$	x	*Variable*
			$[l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n}$	*Objet concret*
			$a.l$	*Sélection de méthode*
			$a.l \Leftarrow \varsigma(x : A)b$	*Mise-à-jour de méthode*

Figure 13.1: Syntax of the simply-typed ς-calculus

objet du même type.

Un type est un sous-type d'un autre s'il contient plus de méthodes. La notion de sous-typage est utilisée dans la règle de subsumption qui permet d'oublier en partie l'information de typage que l'on a sur un terme : si A est un sous-type de B alors tout objet de type A peut être vu comme un objet de type B. En particulier, il n'y a pas d'unicité du type d'un ς-terme bien typé.

Afin d'être utilisé comme calcul orienté-objet, le ς-calcul est enfin équipé d'une sémantique opérationnelle qui décrit le comportement des opérations de sélection et de mise-à-jour grâce aux deux règles de réduction suivantes :

$$\begin{array}{lcl} a.l_j & \rightsquigarrow & a_j\{x_j \backslash a\} \\ a.l_j \Leftarrow \varsigma(x : A')b & \rightsquigarrow & [l_j = \varsigma(x : A)b, l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n, i\neq j} \end{array}$$

où A et a sont des abréviations pour respectivement $[l_i : A_i]_{i=1\ldots n}$ et $[l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n}$.

Dans la seconde règle, le type A' peut être n'importe quel type. Les règles de typage de la figure 13.2 assurent que si le membre gauche est bien typé alors A est un sous-type de A'.

La relation de réduction $\rightsquigarrow$ du ς-calcul préserve le typage : si $a \rightsquigarrow a'$ et $\Delta \vdash a : A$ alors $\Delta \vdash a' : A$. Tous les types de a sont des types de a'. Notons cependant que a' peut admettre plus de types que a.

Nous proposons une traduction superficielle du ς-calcul dans Dedukti, c'est-à-dire une traduction qui préserve à la fois le typage et la réduction. La plupart des constructions du ς-calcul sont faciles à traduire dans Dedukti : les types et les objets concrets sont représentés

Notation: Dans cette figure, A est une abréviation pour $[l_i : A_i]_{i=1\ldots n}$

$$\frac{(x : A) \in \Delta}{\Delta \vdash x : A}\ \text{(Type Var)} \qquad \frac{\Delta, x_i : A \vdash a_i : A_i \qquad \forall i \in 1 \ldots n}{\Delta \vdash [l_i = \varsigma(x_i : A)a_i]_{i=1\ldots n} : A}\ \text{(Type Obj)}$$

$$\frac{\Delta \vdash a : A \qquad j \in 1 \ldots n}{\Delta \vdash a.l_j : A_j}\ \text{(Type Select)}$$

$$\frac{\Delta \vdash a : A \qquad \Delta, x : A \vdash b : A_j \qquad j \in 1 \ldots n}{\Delta \vdash a.l_j \Leftarrow \varsigma(x : A)b : A}\ \text{(Type Update)}$$

$$\frac{}{[l_i : A_i]_{i \in 1\ldots n+m} <: [l_i : A_i]_{i \in 1\ldots n}}\ \text{(Subtype)}$$

$$\frac{\Delta \vdash a : A \qquad A <: B}{\Delta \vdash a : B}\ \text{(Type Subsume)}$$

Figure 13.2: Règles de typage pour le ς-calcul simplement typé

comme des listes d'association, les variables sont représentées par des variables, le lieur ς est représenté par le lieur λ et la sélection et la mise-à-jour de méthode sont des fonctions.

La difficulté vient de la traduction du sous-typage. Dans Dedukti, le type d'un terme est unique (modulo réduction) donc il est nécessaire de distinguer côté Dedukti les ς-termes traduits en fonction du type qui leur est associé. Nous utilisons pour ce faire des coercions explicites : si A est un sous-type de B alors on dispose en Dedukti d'une fonction de coercion de type $A \to B$.

$$\texttt{coerce}\ A\ B : A <: B \to A \to B$$

Cependant, rajouter ce symbole `coerce` ne suffit pas à représenter fidèlement le ς-calcul en Dedukti parce que l'apparition du symbole `coerce` peut bloquer la réduction. Si l'on tente de sélectionner une méthode d'un objet concret coercé, aucune règle de réduction ne s'applique. La situation est similaire dans le cas de la mise-à-jour d'une méthode d'un objet coercé. Pour remédier à ce problème, on ajoute dans Dedukti des règles de réécriture faisant commuter le symbole `coerce` avec la sélection et la mise-à-jour.

Nous avons implantée cette traduction du ς-calcul simplement typé avec sous-typage

en Dedukti. Notre outil s'appelle Sigmaid et est disponible à l'adresse suivante : `http://sigmaid.gforge.inria.fr/`

Chapter 14

De FoCaLiZe à Dedukti

La Déduction modulo [67] est une technique de recherche de preuve pour la logique du premier ordre. Elle repose sur le principe de Poincaré, c'est-à-dire la distinction entre le raisonnement et le calcul dans les preuves. En Déduction Modulo, le calcul est modélisé par de la réécriture et la part de calcul des preuves est laissée implicite. À l'inverse, la part de raisonnement des preuves reste explicite et consiste en un système d'inférence (par exemple la déduction naturelle, le calcul des séquents ou le calcul de résolution) adapté pour remplacer l'égalité syntaxique entre formules logiques par la congruence générée par le système de réécriture qui modélise le calcul.

Lorsque le système de réécriture utilisé est à la fois fortement normalisant et confluent, la vérification de preuves de Déduction modulo est décidable. Grâce à l'orientation des règles de calcul, la recherche de preuve en Déduction modulo est généralement plus efficace.

Quelques prouveurs automatiques comme Zenon [31] sont capables de produire des preuves vérifiables par des outils indépendants. Ainsi, pour se convaincre de la justesse du résultat de Zenon, il n'est pas nécessaire de faire confiance à Zenon lui-même mais il est suffisant de faire confiance à Coq, le vérificateur de preuve utilisé pour certifier les preuves de Zenon.

Cependant, pour vérifier les preuves de la Déduction modulo, le vérificateur de preuve a besoin d'être capable de raisonner modulo un système de réécriture arbitraire ce qui exclu la plupart des vérificateurs de preuves, notamment Coq. Dedukti est un vérificateur de preuve qui a été développé spécifiquement pour vérifier les preuves de la Déduction modulo.

Pendant ma thèse, mon collègue Halmagrand a adapté le prouveur automatique Zenon [31] à la Déduction modulo. Cette version, Zenon Modulo, produit des preuves formelles en Dedukti à la place des preuves Coq.

Zenon est utilisé pour automatiser la recherche de preuve dans le cadre de l'environnement de développement de logiciels sûrs FoCaLiZe [143]. Cet environnement se compose d'un langage de programmation, d'un langage de spécification, d'un langage de preuve et d'un langage permettant de structurer les développements de manière modulaire.

Le langage de programmation de FoCaLiZe est un dialecte du langage de programmation fonctionnelle ML. Ce dialecte propose les éléments communs à la plupart des implantations de ML. En particulier, c'est un langage typé à l'aide d'un système de types polymorphes prénex, l'utilisateur peut définir des fonctions récursives et des types algébriques dont les valeurs sont inspectées par filtrage de motif.

Le langage de spécification de FoCaLiZe est une version typée de la logique du premier ordre. Le système de type utilisé est essentiellement le même que pour le langage de programmation (polymorpisme prénex). Les atomes de la logique sont les termes ML du premier ordre de type booléen.

Le langage de preuves de FoCaLiZe est un langage déclaratif de haut niveau. Une preuve FoCaLiZe contient généralement uniquement les grandes lignes du raisonnement, les détails étant déchargés à Zenon.

Enfin, FoCaLiZe propose des mécanismes orientés objet statiques permettant de modulariser les développements en regroupant des méthodes à l'intérieur d'espèces.

Les développements FoCaLiZe sont compilés vers le langage de programmation OCaml d'une part et vers le vérificateur de preves Coq d'autre part. La sortie OCaml permet d'exécuter les programmes FoCaLiZe et la sortie Coq contient à la fois les programmes et leurs preuves de correction. Ces preuves sont principalement générées par Zenon. Afin de bénéficier de l'adaptation de Zenon à la Déduction modulo, il est nécessaire de développer une sortie Dedukti à FoCaLiZe.

Cette sortie Dedukti pour FoCaLiZe, Focalide, est le second outil que nous avons développé au cours de cette thèse. La difficulté provient de la compilation en Dedukti

du langage ML et en particulier de deux fonctionnalités de ce langage également présentes dans OCaml et dans Coq mais pas dans Dedukti : le filtrage de motif et la récursivité.

Le filtrage de motif est une fonctionnalité des langages fonctionnels permettant d'effectuer des actions différentes en fonction de la forme du valeur. Les différentes formes sont représentées par des motifs qui peuvent lier des variables. Les motifs autorisés dans FoCaLiZe-ML sont les suivants :

- les motifs constants : ces motifs sont des valeurs concrètes, le motif constant c filtre exactement c et ne lie pas de variable,
- les motifs variables : ces motifs sont des variables, le motif variable x filtre toute valeur et la lie à la variable x,
- le motif universel : ce motif filtre toute valeur et ne lie pas de variable,
- les motifs construits : si p_1, ..., p_n sont des motifs et C un constructeur de donnée d'arité n alors $C(p_1, \ldots, p_n)$ filtre les valeurs de la forme $C(v_1, \ldots, v_n)$ telles que chaque motif p_i filtre la valeur v_i correspondante.

Par exemples, la définition de fonction suivante filtre l'argument `x` suivant deux motifs : le premier est le motif constant `0` et le second est le motif variable `n`.

```
let f x =
  match x with
  | 0 -> 0
  | n -> n-1;
```

Cette fonction calcule le prédécesseur d'un entier non nul et retourne 0 sur l'entrée 0.

La traduction naïve de cette fonction en Dedukti consiste en une déclaration de fonction et deux règles de réécriture :

```
f : int -> int.
[] f 0 --> 0
[n] f n --> - n 1.
```

Cette définition est rejetée par le vérificateur de confluence de Dedukti. En effet le terme `f 0` réduit à la fois vers 0 et -1. À partir de cet échec de confluence, il serait facile de prouver 0 = -1.

Le soucis vient d'une différence de sémantique entre le filtrage de motif de ML et la réécriture de Dedukti : en cas de conflit entre plusieurs motifs, la sémantique du filtrage précise que le premier motif filtrant la valeur est choisie alors que du coté de la réécriture, on réclame la confluence c'est-à-dire qu'en cas de conflit le choix du motif ne change rien au résultat.

Pour traduire le filtrage de motif, nous introduisons une transformation de programme qui restreint les filtrages de motifs autorisés à une forme facile à traduire en Dedukti : la forme de destructeur.

Le destructeur associé à un constructeur C d'arité n est le filtrage de motif suivant :

```
match a with
| C(x_1, \ldots, x_n) -> b
| _ -> c
```

Notre transformation de programme procède en deux temps, le premier temps traite le nombre de motifs et le second temps traite leur imbrication.

À l'issue de la première étape, les seuls filtrages de motifs restants sont de la forme suivante (où `p` est un motif) :

```
match a with
| p -> b
| _ -> c
```

On appelle les filtrages de cette forme des filtrages *simples*. Les destructeurs sont des exemples de filtrages simples.

Pour atteindre cette forme, on transforme chaque filtrage de motif

```
match a with
| p_1 -> b_1
| p_2 -> b_2
| ...
| p_n -> b_n
```

en le terme

```
match a with
| p_1 -> b_1
| _ -> (match a with
        | p_2 -> b_2
        | _ -> ...
               match a with
               | p_n -> b_n
```

```
                | _ -> ERROR)
```

Dans ce terme, le symbole `ERROR` représente une erreur d'exécution qui signifie que le filtrage de motif initial n'était pas exhaustif.

Pour éviter de dupliquer le terme `a`, on peut le lier à une variable fraîche `x` de la manière suivante :

```
let x = a in
match x with
| p_1 -> b_1
| _ -> (match x with
         | p_2 -> b_2
         | _ -> ...
              match x with
              | p_n -> b_n
              | _ -> ERROR)
```

La deuxième étape de notre transformation de programme permet de transformer les filtrages simples en destructeur. Pour ce faire, on regarde plus en détail le motif que l'on veut transformer.

Pour tous les cas sauf les motifs construits, on peut éliminer directement le filtrage : les motifs constants deviennent des alternatives, les motifs variables deviennent des définitions locales.

Enfin, le fitrage suivant :

```
match a with
| C(p_1, ..., p_n) -> b
| _ -> c
```

est transformé en

```
match a with
| C(x_1, ..., x_n) ->
   (match x_1 with
    | p_1 ->
        ...
        (match x_n with
         | p_n -> b
         | _ -> c)
        ...
    | _ -> c)
| _ -> c
```

où les variables `x_i` sont fraîches. Ce terme comporte le destructeur associé à `C` et `n` filtrages simples plus petits que le filtrage simple de départ.

À titre d'exemple, notre transformation de programme transforme le filtrage

```
match x with
| 0 -> 0
| n -> n-1
```

en

```
if x = 0 then 0 else (let n = x in n-1)
```

et le filtrage définissant le prédécesseur sur les entiers de Piano

```
match x with
| 0 -> 0
| S(n) -> n
```

en

```
match x with
| 0 -> 0
| _ -> (match x with
         | S(n) -> n
         | _ -> ERROR)
```

Nous pouvons donc transformer tous les filtrages de motif de ML de manière à n'utiliser que des destructeurs. Les destructeurs sont eux-même facile à exprimer par la réécriture de Dedukti.

L'autre fonctionnalité de ML qui a une sémantique légèrement différente en Dedukti est la récursivité. Dans FoCaLiZe, il est possible de définir des fonctions récursives comme par exemple la fonction factoriel :

```
let rec fact x =
  if x <= 1
   then 1
   else x * fact (x - 1)
```

Une traduction naïve de cette fonction en Dedukti produit un système de réécriture non terminant :

```
fact : nat -> nat.
[x] fact x --> if (<= x 1) 1 (* x (fact (- x 1))).
```

Le problème provient du fait que la sémantique en appel par valeur de ML permet de contraindre le déroulement des fonctions récursives au cas où elles sont appliquées à des

valeurs tandis que la sémantique de Dedukti ne distingue pas d'ensemble de valeurs, la règle de réécriture précédente est utilisable pour tous les appels de la fonction `fact`.

Nous contournons ce problème de manière similaire au travail d'Assaf dans Coqine [11]. Nous dupliquons la fonction `fact` et nous avons donc une fonction `fact` et une fonctions `fact2`. Ces deux fonctions sont identiques lorsqu'elles sont appelées sur des valeurs mais la fonction `fact2` ne réduit pas lorsqu'elle est appelée sur un terme qui n'est pas une valeur comme `- x 1`. La fonction `fact` est définie comme précédemment mais en utilisant la fonction `fact2` dans son appel récursif :

```
fact : nat -> nat.
fact2 : nat -> nat.

[] fact2 0 --> fact 0
[n] fact2 (S n) --> fact (S n).

[x] fact x --> if (<= x 1) 1 (* x (fact2 (- x 1))).
```

Ce système de réécriture termine et préserve la sémantique des appels récursifs de la sémantique en appel par valeur de ML.

Notre sortie de FoCaLiZe vers Dedukti a été comparée expérimentalement à la sortie vers Coq sur six bibliothèques FoCaLiZe. Notre traducteur a permis d'exprimer la quasi-totalité (97.9%) de ces bibliothèques en Dedukti. Le temps nécessaire à Zenon Modulo pour trouver les preuves est comparable bien que supérieur à celui nécessaire à Zenon, cette différence provient du fait que notre traduction utilise des fonctions d'ordre supérieur (comme les destructeurs) pour lesquelles nous avons eu besoin d'étendre Zenon Modulo au delà du premier ordre et la manière dont nous avons effectué ce travail est sans doute sous-optimale. En revance, pour vérifier les preuves et le typage des programmes, Dedukti est sur nos exemples bien plus rapide que Coq (jusqu'à deux ordres de grandeur), cela provient du fait que dans le cadre de FoCaLiZe, Coq vérifie une partie importante de sa bibliothèque standard à chaque fichier tandis que les développements Dedukti produits ont peu de dépendances. Finalement, sur les bibliothèques existantes qui ont été développées à l'aide de la sortie Coq de FoCaLiZe, l'utilisateur obtient généralement de meilleures performances en utilisant la sortie Dedukti. Mais surtout, cette nouvelle sortie permet d'utiliser FoCaLiZe et Zenon Modulo pour vérifier des preuves très calculatoires pour lesquelles Zenon et Coq

sont actuellement inutilisables en pratique.

Chapter 15

Interopérabilité entre Systèmes de Preuves

Nous nous sommes attaqués au problème de l'intéropérabilité entre systèmes de preuves à travers deux études de cas. La première est un travail que nous avons mené en collaboration avec Assaf, l'auteur des traducteurs vers Dedukti des systèmes Coq et HOL. L'objectif de cette première étude de cas était de développer une logique dans Dedukti permettant de d'exprimer à la fois les logiques de Coq et d'HOL et de se servir des traducteurs pour construire une preuve Dedukti dont certaines parties ont été écrites en Coq et d'autres en HOL. La seconde étude de cas visait à proposer une méthodologie d'interopérabilité passant mieux à l'échelle grâce à l'automatisation fournie par FoCaLiZe et Zenon Modulo.

Le problème concret résolu au cours de cette seconde étude de cas était celui de la correction du crible d'Ératosthenes, un algorithme permettant de lister les nombres premiers.

Une implantation possible de cet algorithme dans un langage fonctionnel comme OCaml est la suivante :

```
let rec interval a b =
  if a > b then [] else a :: interval (a+1) b

let rec sieve = function
  | [] -> []
  | a :: l ->
     a :: sieve (List.filter (fun b -> b mod a > 0) l)

let eratosthenes n = sieve (interval 2 n)
```

La fonction `interval` calcule la liste des nombres entiers entre `a` et `b` rangés dans

l'ordre croissant. La fonction `sieve` est le coeur de l'implantation, elle transforme une liste de nombres en une autre liste de nombres de la manière suivante : si la liste est vide, elle retourne la liste vide tandis que si la liste commence par le nombre `a` et se poursuit par la liste de nombre `l` alors la fonction retire de `l` tous les multiples de `a` et s'appelle récursivement sur le résultat. Enfin, le fonction `eratosthenes` appelle la fonction `sieve` sur l'intervalle des nombres compris entre `2` et `n`.

La spécification de ce programme est la suivante : un nombre p apparaît dans la liste `eratosthenes n` si et seulement si p est un nombre premier plus petit que `n`. Pour prouver ce résultat, on le décompose en trois propriétés :

- Complétude : si p est un nombre premier inférieur à `n` alors p apparaît effectivement dans la liste `eratosthenes n`.

  Cette propriété découle d'un invariant de la fonction `sieve` : si un nombre premier apparaît dans une liste `l` alors il apparaît aussi dans la liste `sieve l`.

- Premier résultat de correction : si un nombre p apparaît dans la liste `eratosthenes n` alors il est compris entre `2` et `n`.

  Cette propriété découle également d'un invariant de la fonction `sieve` : si un nombre p apparaît dans la liste `sieve l` alors il apparaît déjà dans `l`.

- Second résultat de correction : si on nombre p apparaît dans la lise `eratosthenes n` alors il est premier.

  Ce résultat est un peu plus difficile à montrer que les deux précédents. Il repose aussi sur un invariant de la fonction `sieve` : si la liste `l` est triée alors la liste `sieve l` l'est aussi.

  Soit p un nombre apparaissant dans `eratosthenes n`, par le premier résultat de correction, on sait que $2 \leq p \leq$ `n`. Comme tous les nombres supérieurs à 2, le plus petit diviseur de p est un nombre premier d. Comme d est un diviseur de p, il est plus petit que p et *a fortiori* plus petit que `n`. Les hypothèses du résultat de complétude sont satisfaites donc d apparaît également dans la liste `eratosthenes n`. De deux choses l'une, soit $d = p$ soit $d < p$.

- Dans le premier cas, $p = d$ est un nombre premier comme on le souhaitait.
- Dans le second cas, d apparaît avant p dans la liste `eratosthenes n` (parce que cette liste est triée puisque `interval 2 n` est une liste triée) ce qui est absurde parce que p aurait du être retiré de la liste par la fonction `sieve` au tour de d.

Le crible d'Ératosthenes est intéressant pour l'interopérabilité des systèmes de preuve parce que la preuve que nous venons de voir combine des propriétés arithmétiques (pour tout nombre supérieur à 2, le plus petit diviseur est un nombre premier) à des propriétés très spécifiques à l'implantation. Nous proposons une preuve formelle de la correction du crible d'Ératosthenes dont la partie arithmétique provient d'une bibliothèque HOL tandis que l'implantation de l'algorithme et les lemmes spécifiques à cette implantation sont écrits en Coq.

Contrairement à notre première étude de cas qui limitait autant que possible les notions partagées entre les deux systèmes, il est ici nécessaire de faire correspondre toutes les notions d'arithmétiques des deux systèmes qui sont mises en jeu dans cette preuve telles que l'ordre, la divisibilité et la primalité. La répartition du travail entre les différents systèmes est donc la suivante : l'existence d'un diviseur premier pour tout nombre supérieur à 2 est importée du système HOL, l'implantation, la spécification et la correction du crible sont écrits en Coq, la liaison entre les logiques de Coq et de HOL est écrite en Dedukti (elle est reprise de notre première étude de cas) mais la liaison entre les notions mathématiques des deux systèmes est établie en FoCaLiZe afin de bénéficier d'un langage de plus haut niveau et de l'automatisation des preuves par Zenon Modulo.

La situation est schématisée Figure 15.1. Dans ce schéma, sont représentés en bleu les opérations arithmétiques traduites de HOL et en rouge celles traduites de Coq. Le rôle du développement FoCaLiZe est de montrer l'équivalence entre ces opérations différentes.

Ce développent est structuré en une hiérarchie d'espèces FoCaLiZe. L'espèce la plus primitive de notre hiérarchie axiomatise les entiers par les axiomes de Péano. Les différentes opérations arithmétiques (addition, multiplication, ordre large, ordre strict, divisibilité, divisibilité stricte et primalité) sont tour-à-tour ajoutées de manière axiomatique sans prendre parti ni pour la définition traduite de Coq ni pour celle traduite de HOL.

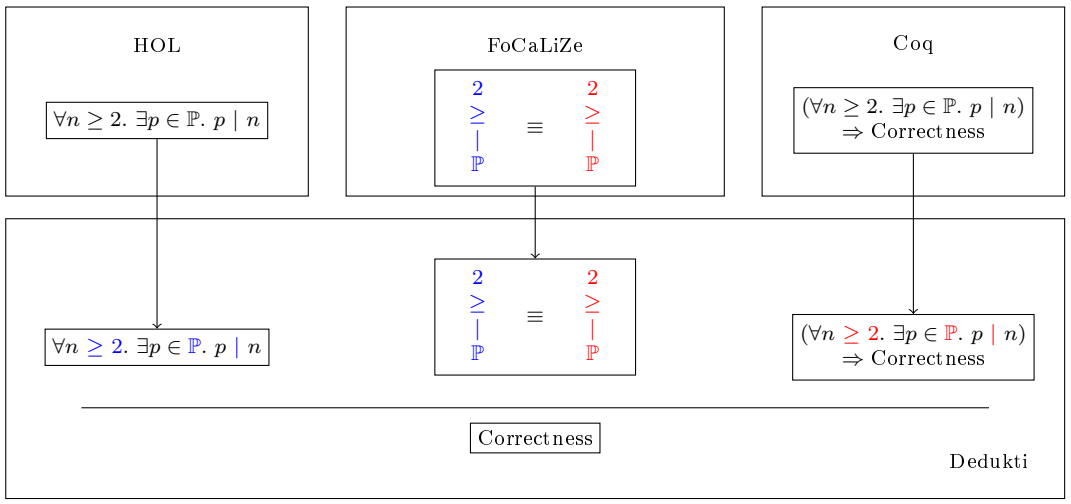


Figure 15.1: Interopérabilité sur le cas d'étude du crible d'Ératosthenes

Finalement, la hiérarchie est achevée par le théorème d'existence du diviseur premier.

En parallèle de cette hiérarchie de structures arithmétiques, nous développons une hiérarchie d' (iso-)morphismes entre ces structures, cette seconde hiérarchie permet de prouver qu'une propriété est vraie dans un modèle d'une structure algébrique si et seulement si elle est vraie dans un modèle isomorphe. Cette hiérarchie culmine avec la preuve que l'énoncé du théorème d'existence du diviseur premier dans un modèle est vrai si et seulement si il est vrai dans un modèle isomorphe.

Enfin, nous montrons que les définitions des opérations arithmétiques de Coq et HOL définissent des modèles isomorphes des structures arithmétiques et nous en déduisons l'équivalence entre les énoncés Coq et HOL du théorème d'existence du diviseur premier. Ce résultat permet de boucler la démonstration en Dedukti.

Finalement, nous obtenons une preuve majoritairement écrite en FoCaLiZe (61 Ko de FoCaLiZe contre 31Ko de Coq et 9Ko de Dedukti), qui exploite bien l'automatisation offerte par Zenon Modulo (environ 2/3 du code Dedukti provenant de la partie FoCaLiZe est généré par Zenon Modulo) et dont l'essentiel est réutilisable pour des travaux similaires d'interopérabilité entre systèmes de preuves nécessitant de partager des notions

d'arithmétique.

Chapter 16

Conclusion

Nous avons développé des traductions superficielles pour deux paradigmes de programmation : la programmation orientée objet du ς-calcul et la programmation fonctionnelle de FoCaLiZe-ML. Ces traductions ont été implantées dans des outils logiciels performants.

La traduction de FoCaLiZe en Dedukti permet d'étendre FoCaLiZe à la Déduction modulo pour résoudre de nouveaux problèmes pour lesquelles la sortie historique de FoCaLiZe vers Coq ne passe pas à l'échelle.

Nous avons aussi utilisé ce traducteur de FoCaLiZe vers Dedukti pour résoudre un problème d'intéropérabilité entre les systèmes de preuve Coq et HOL : la preuve de correction du crible d'Ératosthenes. Cette étude de cas a permis le développement d'une méthodologie basée sur FoCaLiZe et ses mécanismes orientés objet.

Au cours de ces travaux de thèse, nous avons souvent détourné les logiciels de leur usage habituel : FoCaLiZe a été utilisé non comme environnement de développement de programmes sûrs mais comme plateforme d'interopérabilité et Dedukti a été utilisé comme normalisateur de preuve pour supprimer de manière heuristique des axiomes classiques [41]. Nous sommes convaincus que continuer à détourner ces outils sera fructueux dans l'avenir, par exemple en éliminant d'autres axiomes.

L'interopérabilité entre systèmes de preuve est encore balbutiante, la prochaine étape consistera à tester les possibilités d'interopérabilité dans le cas d'une collaboration entre des systèmes différents de ce que nous avons essayés et également plus nombreux.

Bibliography

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *TACS'94, Theoretical Aspects of Computing Sofware*, pages 296–320, 1994.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer New York, 1996.

[3] Azriel Levy Abraham A. Fraenkel, Yehoshua Bar-Hillel. *Foundations of Set Theory*. Studies in Logic and the Foundations of Mathematics. Elsevier, Academic Press, 2 edition, 1973.

[4] Jean-Raymond Abrial. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, 1996.

[5] Stephen Adams. Functional Pearls: Efficient sets–a balancing act. *Journal of functional programming*, 3(4):553–562, 1993.

[6] Jesse Alama. Escape to Mizar from ATPs. In Pascal Fontaine, Renate A. Schmidt, and Stephan Schulz, editors, *Third Workshop on Practical Aspects of Automated Reasoning, PAAR-2012, Manchester, UK, June 30 - July 1, 2012*, volume 21 of *EPiC Series in Computing*, pages 3–11. EasyChair, 2012.

[7] Peter D. Andrews. An Introduction to Mathematical Logic and Type Theory: To Truth through Proof. In *Computer Science and Applied Mathematics Series*. Academic Press, 1986.

[8] Jesús Aransay-Azofra, Jose Divasón, Jónathan Heras, Laureano Lambán, María Vico Pascual, Angel Luis Rubio, and Julio Rubio. Obtaining an ACL2 specification

from an isabelle/hol theory. In Gonzalo A. Aranda-Corral, Jacques Calmet, and Francisco J. Martín-Mateos, editors, *Artificial Intelligence and Symbolic Computation - 12th International Conference, AISC 2014, Seville, Spain, December 11-13, 2014. Proceedings*, volume 8884 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2014.

[9] Alasdair Armstrong, Simon Foster, and Georg Struth. Dependently Typed Programming based on Automated Theorem Proving. *CoRR*, abs/1112.3833, 2011.

[10] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita Interactive Theorem Prover. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69. Springer, 2011.

[11] Ali Assaf. *A Framework for Defining Computational Higher-Order Logics.* PhD thesis, École Polytechnique, 2015.

[12] Ali Assaf. Conservativity of Embeddings in the lambda Pi Calculus Modulo Rewriting. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 31–44. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[13] Ali Assaf and Guillaume Burel. Translating HOL to Dedukti. In Cezary Kaliszyk and Andrei Paskevich, editors, Proceedings Fourth Workshop on *Proof eXchange for Theorem Proving,* Berlin, Germany, August 2-3, 2015, volume 186 of *Electronic Proceedings in Theoretical Computer Science*, pages 74–88, Berlin, Germany, August 2015. Open Publishing Association.

[14] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Expressing Theories in the λΠ-Calculus Modulo Theory and in the Dedukti

System. Draft available online at `http://www.lsv.ens-cachan.fr/~dowek/Publi/expressing.pdf`, 2016.

[15] Ali Assaf and Raphaël Cauderlier. Mixing HOL and Coq in Dedukti. In Kaliszyk, Cezary and Paskevich, Andrei, editor, Proceedings 4th Workshop on *Proof eXchange for Theorem Proving,* Berlin, Germany, August 2-3, 2015, volume 186 of *Electronic Proceedings in Theoretical Computer Science*, pages 89–96, Berlin, Germany, August 2015. Open Publishing Association.

[16] Ali Assaf, Gilles Dowek, Jean-Pierre Jouannaud, and Jiaxiang Liu. Encoding Proofs in Dedukti: the case of Coq proofs. In *Proceedings Hammers for Type Theories*, Proc. Higher-Order rewriting Workshop, Coimbra, Portugal, July 2016. Easy Chair.

[17] Ali Assaf, Gilles Dowek, Jean-Pierre Jouannaud, and Jiaxiang Liu. Untyped Confluence in Dependent Type Theories. In *Proceedings Higher-Order Rewriting Workshop*, Proc. Higher-Order rewriting Workshop, Porto, Portugal, June 2016. Easy-Chair.

[18] Lennart Augustsson. Cayenne - a Language with Dependent Types. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998.*, pages 239–250. ACM, 1998.

[19] Brian E. Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal Reasoning Techniques in Coq: (Extended Abstract). *Electr. Notes Theor. Comput. Sci.*, 174(5):69–77, 2007.

[20] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[21] Stefan Banach and Alfred Tarski. Sur la décomposition des ensembles de points en parties respectivement congruentes. *Fundamenta Mathematicae*, 6:244 – 277, 1924.

[22] Henk Barendregt. Introduction to Generalized Type Systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[23] Henk Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabbay, and Thomas S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[24] Henk Barendregt and Erik Barendsen. Autarkic Computations in Formal Proofs. *Journal of Automated Reasoning (JAR)*, 28, 2002.

[25] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic Proof and Disproof in Isabelle/HOL. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 2011.

[26] Jasmin Christian Blanchette and Andrei Paskevich. TFF1: the TPTP typed first-order form with rank-1 polymorphism. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 414–420. Springer, 2013.

[27] François Bobot, Sylvain Conchon, Évelyne Contejean, and Stéphane Lescuyer. Implementing Polymorphism in SMT solvers. In *SMT 2008: 6th International Workshop on Satisfiability Modulo*, 2008.

[28] Mathieu Boespflug and Guillaume Burel. CoqInE : Translating the calculus of inductive constructions into the λΠ-calculus modulo. In *Proceedings of the Second International Workshop on Proof Exchange for Theorem Proving*, page 44, 2012.

[29] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus Modulo as a Universal Proof Language. In Tjark Weber David Pichardie, editor, *the Second International Workshop on Proof Exchange for Theorem Proving (PxTP 2012)*, volume Vol. 878, pages pp. 28–43, Manchester, United Kingdom, June 2012.

[30] Bernard Bolzano. *Paradoxien des Unendlichen.* C.H. Reclam sen., 1851.

[31] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Logic for Programming,*

Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, volume 4790 of *LNCS/LNAI*, pages 151–165. Springer, 2007.

[32] Viviana Bono, Michele Bugliesi, and Luigi Liquori. A Lambda Calculus of Incomplete Objects. In *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science*, MFCS '96, pages 218–229, London, UK, UK, 1996. Springer-Verlag.

[33] Mélanie Boudard and Olivier Hermant. Polarizing Double Negation Translations. *CoRR*, abs/1312.5420, 2013.

[34] Luitzen E. J. Brouwer. On the Significance of Principle of Excluded Middle in Mathematics, Especialy in Function Theory. In *in van Heijenoort 1967*, pages 302–334, 1923.

[35] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing Object Encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

[36] Guillaume Burel. A Shallow Embedding of Resolution and Superposition Proofs into the λΠ-Calculus Modulo. In Jasmin Christian Blanchette and Josef Urban, editors, *PxTP 2013. 3rd International Workshop on Proof Exchange for Theorem Proving*, volume 14 of *EasyChair Proceedings in Computing*, pages 43–57, Lake Placid, USA, June 2013. EasyChair.

[37] Guillaume Bury, Raphaël Caudelier, and Pierre Halmagrand. Implementing Polymorphism in Zenon. In *11th International Workshop on the Implementation of Logics (IWIL)*, Suva, Fiji, November 2015.

[38] Guillaume Bury, David Delahaye, Damien Doligez, Pierre Halmagrand, and Olivier Hermant. Automated Deduction in the B Set Theory using Typed Proof Search and Deduction Modulo. In *LPAR 20 : 20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning* , Suva, Fiji, November 2015.

[39] Georg Cantor. Über eine Eigenschaft des Inbegriffs aller reellen algebraischen Zahlen. *Journal für die reine und angewandte Mathematik (Crelle's Journal)*, pages 258–262, 1874.

[40] G. Castagna. *Surcharge, sous-typage et liaison tardive : fondements fonctionnels de la programmation orientée objets.* Phd. thesis, Université Paris 7, jan 1994. (in English).

[41] Raphaël Cauderlier. A rewrite system for proof constructivization. In *Proceedings of the 2016 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice*. ACM, 2016. To appear.

[42] Raphaël Cauderlier and Catherine Dubois. Objects and subtyping in the λΠ-calculus modulo. In *Post-proceedings of the 20th International Conference on Types for Proofs and Programs (TYPES 2014)*, Leibniz International Proceedings in Informatics (LIPIcs), Paris, 2014. Schloss Dagstuhl.

[43] Raphaël Cauderlier and Catherine Dubois. ML pattern-matching, recursion, and rewriting: from FoCaLiZe to Dedukti. In *Theoretical Aspects of Computing - ICTAC 2016*, LNCS. Springer Berlin Heidelberg, 2016. To appear.

[44] Raphaël Cauderlier and Pierre Halmagrand. Checking Zenon Modulo Proofs in Dedukti. In Kaliszyk, Cezary and Paskevich, Andrei, editor, Proceedings 4th Workshop on *Proof eXchange for Theorem Proving,* Berlin, Germany, August 2-3, 2015, volume 186 of *Electronic Proceedings in Theoretical Computer Science*, pages 57–73, Berlin, Germany, August 2015. Open Publishing Association.

[45] Raphaël Cauderlier. A rewrite system for elimination of the functional extensionality and univalence axioms. Draft available online at `https://who.rocq.inria.fr/Raphael.Cauderlier/`, 2016.

[46] James Cheney and Christian Urban. alpha-Prolog: A Logic Programming Language with Names, Binding and a-Equivalence. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2004.

[47] Zakaria Chihani. *Certification of First-order proofs in classical and intuitionistic logics.* PhD thesis, EDX École Polytechnique, 2015.

[48] Alonzo Church and John B. Rosser. Some properties of conversion. In *Transactions of the American Mathematical Society*, volume 39, pages 472–482, 1936.

[49] Alberto Ciaffaglione, Luigi Liquori, and Marino Miculan. Reasoning About Object-based Calculi in (Co)Inductive Type Theory and the Theory of Contexts. *J. Autom. Reason.*, 39(1):1–47, July 2007.

[50] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Matching Power. In *Proceedings of RTA'2001*, Lecture Notes in Computer Science. Springer-Verlag, May 2001.

[51] Horatiu Cirstea, Luigi Liquori, and Benjamin Wack. Rewriting Calculus with Fixpoints: Untyped and First-order Systems. In *TYPES*, volume 3085. Springer, 2003.

[52] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 System. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[53] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. Research Report RR-0529, INRIA, 1986.

[54] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall, 1986.

[55] Denis Cousineau and Gilles Dowek. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

[56] Simon Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond.* PhD thesis, Ecole Polytechnique, 2015.

[57] Pierre-Louis Curien and Hugo Herbelin. The Duality of Computation. *SIGPLAN Not.*, 35(9):233–243, September 2000.

[58] Haskell B. Curry. Functionality in Combinatory Logic. In *In Proceedings of the National Academy of Sciences of the United States of America*, pages 584–590, 1934.

[59] Luís Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In Richard A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, pages 207–212. ACM Press, 1982.

[60] David Delahaye. A Tactic Language for the System Coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science (LNCS)/Lecture Notes in Artificial Intelligence (LNAI)*, pages 85–95, Reunion Island (France), November 2000. Springer.

[61] David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand, and Olivier Hermant. Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 8312 of *LNCS/ARCoSS*, pages 274–290. Springer Berlin Heidelberg, dec 2013.

[62] David Delahaye, Catherine Dubois, Claude Marché, and David Mentré. The BWare Project: Building a Proof Platform for the Automated Verification of B Proof Obligations. In *Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*, LNCS. Springer, 2014.

[63] The Coq development team. *The Coq Reference Manual, version 8.4*, August 2012.

[64] Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. A Lambda Calculus of Objects with Self-inflicted Extension. *SIGPLAN Not.*, 33(10):166–178, October 1998.

[65] Radu Diaconescu. Axiom of choice and complementation. In *Proceedings of the American Mathematical Society*, volume 51, pages 176–178, August 1975.

[66] Gilles Dowek. *Proofs and Algorithms - An Introduction to Logic and Computability.* Undergraduate Topics in Computer Science. Springer, 2011.

[67] Gilles Dowek. Deduction modulo theory. In *All about proofs, proofs for all*, 2014.

[68] Gilles Dowek. On the definition of the classical connectives and quantifiers. In *Why is this a Proof? Festschrift for Luiz Carlos Pereira*. College Publication, 2015.

[69] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning (JAR)*, 31, 2003.

[70] Gilles Dowek and Benjamin Werner. Proof normalization modulo. *The Journal of Symbolic Logic*, 68(4):1289–1316, 2003.

[71] Catherine Dubois and François Pessaux. Termination proofs for recursive functions in focalize. In Manuel Serrano and Jurriaan Hage, editors, *Trends in Functional Programming - 16th International Symposium, TFP 2015, Sophia Antipolis, France, June 3-5, 2015. Revised Selected Papers*, volume 9547 of *Lecture Notes in Computer Science*, pages 136–156. Springer, 2016.

[72] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[73] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1:3–37, 1994.

[74] Kathleen Fisher and John C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *In Proc. of FCT*, pages 42–61. Springer-Verlag, 1995.

[75] J. Nathan Foster and Dimitrios Vytiniotis. A Theory of Featherweight Java in Isabelle/HOL. *Archive of Formal Proofs*, March 2006. `http://afp.sf.net/entries/FeatherweightJava.shtml`, Formal proof development.

[76] Thomas Genet, Barbara Kordy, and Amaury Vansyngel. Vers un outil de vérification formelle légère pour OCaml. In Frédéric Dadeau and Pascale Le Gall, editors, *AFADL 2015*, pages 28–33, Bordeaux, France, May 2015.

[77] Gerhard Gentzen. Über das verhältnis zwischen intuitionistischer und klassischer arithmetik. *Archiv für mathematische Logik und Grundlagenforschung*, 16(3):119–132, 1974.

[78] Herman Geuvers. *Logics and type systems.* PhD thesis, University of Nijmegen, 1993.

[79] Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. Automated Termination Proofs for Haskell by Term Rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7:1–7:39, February 2011.

[80] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with aprove. In *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer, 2004.

[81] Frederic Gilbert. A Lightweight Double-negation Translation. In Ansgar Fehnker, Annabelle McIver, Geoff Sutcliffe, and Andrei Voronkov, editors, *LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations*, volume 35 of *EPiC Series in Computing*, pages 81–93. EasyChair, 2015.

[82] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris 7, 1972.

[83] Kurt Gödel. Zur intuitionistischen Arithmetik und Zahlentheorie. *Ergebnisse eines mathematischen Kolloquiums*, 4(1933):34–38, 1933.

[84] Georges Gonthier. The Four Colour Theorem: Engineering of a Formal Proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[85] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Sandrine Blazy,

Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013.

[86] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2015.

[87] Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000.

[88] Albert Gräf. *Pure Quick Reference*, 2013.

[89] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *CoRR*, abs/1501.02155, 2015.

[90] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[91] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, 2007.

[92] Felix Hausdorff. Bemerkung über den inhalt von punktmengen. *Mathematische Annalen*, 75:428 – 434, 1914.

[93] Ludovic Henrio, Florian Kammüller, Bianca Lutz, and Henry Sudhof. Locally Nameless Sigma Calculus. *Archive of Formal Proofs*, April 2010. `http://afp.sf.net/entries/Locally-Nameless-Sigma.shtml`, Formal proof development.

[94] Arend Heyting. Die formalen Regeln der intuitionistischen Logik. *I, II, III. Sitzungsberichte Akad. Berlin*, pages 42–56, 1930.

[95] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL - A Modular Logical Environment. In David A. Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2003.

[96] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[97] Fulya Horozal and Florian Rabe. Representing model theory in a type-theoretical logical framework. *Theoretical Computer Science*, 412:4919–4945, 2011.

[98] William A. Howard. The Formulae-as-types Notion of Construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Reprint of a manuscript written in 1969.

[99] Joe Hurd. The OpenTheory Standard Theory Library. In *NASA Formal Methods*, pages 177–191, 2011.

[100] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java - A Minimal Core Calculus for Java and GJ. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[101] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[102] Wolfram Kahl. Basic Pattern Matching Calculi: A Fresh View on Matching Failure. In Yukiyoshi Kameyama and Peter Stuckey, editors, *Functional and Logic Programming, Proceedings of FLOPS 2004*, volume 2998 of *LNCS*, pages 276–290. Springer, 2004.

[103] Cezary Kaliszyk and Alexander Krauss. Scalable LCF-style proof translation. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, number 7998 in LNCS, pages 51–66. Springer Berlin Heidelberg, 2013.

[104] Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP*, number 6172 in LNCS, pages 307–322. Springer Berlin Heidelberg, 2010.

[105] Delia Kesner, Laurence Puel, and Val Tannen. A Typed Pattern Calculus. *Information and Computation*, 124(1):32–61, 1996.

[106] Jan Willem Klop, Vincent van Oostrom, and Roel de Vrijer. Lambda calculus with patterns. *Theoretical Computer Science*, 398(1–3):16–31, 2008. Calculi, Types and Applications: Essays in honour of M. Coppo, M. Dezani-Ciancaglini and S. Ronchi Della Rocca.

[107] Andrey N. Kolmogorov. On the principle of the excluded middle. In Jean van Heijenoort, editor, *A Source Book in Mathematical Logic, 1879–1931*, pages 414–437. Harvard Univ., 1925.

[108] Andrey N. Kolmogorov. Zur deutung der intuitionistischen logik. mathematische zeitschrift, 35:5865. In *English translation in Selected works of A.N. Kolmogorov. Volume I: Mathematics and Mechanics*, 1932.

[109] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.

[110] Jean-Louis Krivine. Opérateurs de mise en mémoire et traduction de gödel. *Archive for Mathematical Logic*, 30(4):241–267, 1990.

[111] Jean-Louis Krivine. Typed lambda-calculus in classical Zermelo-Frænkel set theory. *Archive for Mathematical Logic*, 40(3):189–205, 2001.

[112] Sigekatu Kuroda. Intuitionistische untersuchungen der formalistischen logik. *Nagoya Math. J.*, 2:35–47, 1951.

[113] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ML. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 173–184. ACM, 2007.

[114] Xavier Leroy. Introduction to types in compilation. In Xavier Leroy and Atsushi Ohori, editors, *Types in Compilation, Second International Workshop, TIC '98, Kyoto, Japan, March 25-27, 1998, Proceedings*, volume 1473 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 1998.

[115] Xavier Leroy. A modular module system. *J. Funct. Program.*, 10(3):269–303, 2000.

[116] Luigi Liquori and Giuseppe Castagna. A typed Lambda Calculus of Objects. In *Concurrency and Parallelism, Programming, Networking, and Security: Second Asian Computing Science Conference*, volume ASIAN'96 Singapore, pages 129–141, Berlin, Heidelberg, December 1996. Springer Berlin Heidelberg.

[117] Salvador Lucas and Ricardo Peña. Rewriting Techniques for Analysing Termination and Complexity Bounds of SAFE Programs. In *LOPSTR'08*, pages 43–57, Valencia, Spain, July 2008.

[118] Zhaohui Luo. *Logical Foundations of Computer Science — Tver '92: Second International Symposium Tver, Russia, July 20–24, 1992 Proceedings*, chapter A unifying theory of dependent types: the schematic approach, pages 293–304. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992.

[119] Zhaohui Luo. PAL^{+}: a lambda-free logical framework. *J. Funct. Program.*, 13(2):317–338, 2003.

[120] Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron. Encoding Featherweight Java with Assignment and Immutability Using the Coq Proof Assistant. In *Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 11–19, New York, NY, USA, 2012. ACM.

[121] Luc Maranget. Compiling Pattern Matching to Good Decision Trees. In *Workshop on the Language ML*. ACM Press, September 2008.

[122] Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

[123] Per Martin-Löf. An Intuitionistic Theory of Types. *Twenty-five years of constructive type theory*, 36:127–172, 1998.

[124] Richard Mayr and Tobias Nipkow. Higher-Order Rewrite Systems and their Confluence. *Theoretical Computer Science*, 192:3–29, 1998.

[125] Dale Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation*, 1:253–281, 1991.

[126] Dale Miller. A proposal for broad spectrum proof certificates. In *Certified Programs and Proofs (CPP)*. Springer, 2011.

[127] Dale Miller. Foundational proof certificates: making proof universal and permanent. In Alberto Momigliano, Brigitte Pientka, and Randy Pollack, editors, *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks & Meta-languages: Theory & Practice, LFMTP 2013, Boston, Massachusetts, USA, September 23, 2013*, pages 1–2. ACM, 2013.

[128] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

[129] Dale Miller and Marco Volpe. Focused labeled proof systems for modal logic. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2015.

[130] Julian Nagele. CoCo 2015 Participant: CSIˆho 0.1 . In Ashish Tiwari and Takahito Aoto, editors, *IWC 2015, 4th International Workshop on Confluence, Proceedings*, page 47, 2015.

[131] Pavel Naumov, Mark-Oliver Stehr, and José Meseguer. The HOL/NuPRL proof translator. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics*, number 2152 in LNCS, pages 329–345. Springer Berlin Heidelberg, 2001.

[132] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected papers on Automath*. Elsevier, Amsterdam, 1994.

[133] Bengt Nordstrom, Kent Petersson, and Jan M Smith. Programming in Martin-Lof's Type Theory. An Introduction. In *Number 7 in International series of monographs on computer science*. Oxford University Press, 1989.

[134] Ulf Norell. Dependently Typed Programming in Agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009.

[135] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, number 4130 in LNCS, pages 298–302. Springer Berlin Heidelberg, 2006.

[136] Hans Jürgen Ohlbach. Extensions of first-order logic, maria manzano. *Journal of Logic, Language and Information*, 7(3):389–391, 1998.

[137] Kouta Onozawa, Kentaro Kikuchi, Takahito Aoto, and Yoshihito Toyama. ACPH: System Description. In Ashish Tiwari and Takahito Aoto, editors, *IWC 2015, 4th International Workshop on Confluence, Proceedings*, page 39, 2015.

[138] Jens Otten. Clausal Connection-Based Theorem Proving in Intuitionistic First-Order Logic. In Bernhard Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2005, Koblenz, Germany, September 14-17, 2005, Proceedings*, volume 3702 of *Lecture Notes in Computer Science*, pages 245–261. Springer, 2005.

[139] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[140] Michel Parigot. Lambda-Mu-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning,International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992.

[141] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.

[142] Francis Jeffry Pelletier, Geoff Sutcliffe, and Christian B. Suttner. The development of CASC. *AI Commun.*, 15(2-3):79–90, 2002.

[143] François Pessaux. FoCaLiZe: Inside an F-IDE. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.*, volume 149 of *EPTCS*, pages 64–78, 2014.

[144] Dorian Petit, Vincent Poirriez, and Georges Mariano. Reuse of SML module system for the B language. In *Forum on specification and Design Languages, FDL 2004, September 14-17, 2004, Lille, France, Proceedings*, pages 637–649. ECSI, 2004.

[145] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[146] Frank Pfenning and Conal Elliott. Higher-Order Abstract Syntax. In *Programming Language Design and Implementation*, 1988.

[147] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *International Conference on Automated Deduction (CADE-16)*, number 1632 in LNCS, pages 202–206. Springer Berlin Heidelberg, 1999.

[148] Benjamin C. Pierce and David N. Turner. Simple Type-Theoretic Foundations for Object-Oriented Programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[149] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

[150] Virgile Prevosto. *Conception et Implantation du langage FoC pour le développement de logiciels certifiés*. Thèse de doctorat, Université Paris 6, September 2003.

[151] Thomas Raths, Jens Otten, and Christoph Kreitz. The ILTP library: Benchmarking automated theorem provers for intuitionistic logic. In Bernhard Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2005, Koblenz, Germany, September 14-17, 2005, Proceedings*, volume 3702 of *Lecture Notes in Computer Science*, pages 333–337. Springer, 2005.

[152] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, January 1965.

[153] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30, 1908.

[154] Ronan Saillard. Towards explicit rewrite rules in the λΠ-calculus modulo. In *IWIL - 10th International Workshop on the Implementation of Logics*, 2013.

[155] Ronan Saillard. *Type Checking in the Lambda-Pi-Calculus Modulo: Theory and Practice*. PhD thesis, MINES Paritech, 2015.

[156] Ronan Saillard. *User Manual for Dedukti v2.5*, 2015.

[157] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[158] Carsten Schürmann and Mark-Oliver Stehr. An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf. In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, number 4246 in LNCS, pages 150–166. Springer Berlin Heidelberg, 2006.

[159] Dana Scott. *Constructive Validity*, pages 237–275. Springer Berlin Heidelberg, Berlin, Heidelberg, 1970.

[160] Tim Sheard. Languages of the future. *SIGPLAN Notices*, 39(12):119–132, 2004.

[161] Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.

[162] Geoff Sutcliffe. TPTP, TSTP, CASC, etc. In V. Diekert, M. Volkov, and A. Voronkov, editors, *Proceedings of the 2nd International Computer Science Symposium in Russia*, number 4649 in Lecture Notes in Computer Science, pages 7–23. Springer-Verlag, 2007.

[163] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[164] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Allen Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In Ulrich Fur-

bach and Natarajan Shankar, editors, *Automated Reasoning*, volume 4130 of *LNCS*, pages 67–81. Springer Berlin Heidelberg, 2006.

[165] Geoff Sutcliffe and Christian B. Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.

[166] Christian Urban and Christine Tasson. Nominal Techniques in Isabelle/HOL. In Robert Nieuwenhuis, editor, *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings*, volume 3632 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 2005.

[167] Vincent van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159 – 181, 1997.

[168] Vincent van Oostrom and Femke van Raamsdonk. Weak Orthogonality Implies Confluence: The Higher Order Case. In Anil Nerode and Yuri Matiyasevich, editors, *Logical Foundations of Computer Science, Third International Symposium, LFCS'94, St. Petersburg, Russia, July 11-14, 1994, Proceedings*, volume 813 of *Lecture Notes in Computer Science*, pages 379–392. Springer, 1994.

[169] Jeffrey A. Vaughan. A Review of Three Techniques for Formally Representing Variable Binding. Technical report, University of Pennsylvania, December 2006.

[170] Daniel Wand. Polymorphic+Typeclass Superposition. In Boris Konev, Leonardo de Moura, and Stephan Schulz, editors, *4th Workshop on Practical Aspects of Automated Reasoning (PAAR 2014)*, Vienna, Austria, 2014.

[171] Shuai Wang. A Quantitative Analysis of Kernel Extension for Higher Order Proof Checking. Draft available online at `https://airobert.github.io/holala.pdf`.

[172] Shuai Wang. Higher Order Proof Engineering: Proof Collaboration, Transformation, Checking and Retrieval. In *AITP 2016 - Conference on Artificial Intelligence and Theorem Proving*, Obergurgl, Austria, April 2016.

[173] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*, volume Volume 1. Cambridge University Press, 1st edition, 1910.

[174] Hongwei Xi. Applied Type System: Extended Abstract. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 2003.

[175] Ernst Zermelo. Untersuchungen über die grundlagen der mengenlehre. i. *Mathematische Annalen*, 65(2):261–281, 1908.

[176] Jan Zwanenburg. *Object-Oriented Concepts and Proof Rules: Formalization in Type Theory and Implementation in Yarrow*. PhD thesis, Eindhoven University of Technology, 1999.

le cnam

Raphaël CAUDERLIER

Object-Oriented Mechanisms for Interoperability between Proof Systems

le cnam

Résumé :
Dedukti est un cadre logique résultant de la combinaison du typage dépendant et de la réécriture. Il permet d'encoder de nombreux systèmes logiques au moyen de plongements superficiels qui préservent la notion de réduction.
Ces traductions de systèmes logiques dans un format commun sont une première étape nécessaire à l'échange de preuves entre ces systèmes. Cet objectif d'interopérabilité des systèmes de preuve est la motivation principale de cette thèse.
Pour y parvenir, nous nous inspirons du monde des langages de programmation et plus particulièrement des langages orientés-objet parce qu'ils mettent en œuvre des mécanismes avancés d'encapsulation, de modularité et de définitions par défaut. Pour cette raison, nous commençons par une traduction superficielle d'un calcul orienté-objet en Dedukti. L'aspect le plus intéressant de cette traduction est le traitement du sous-typage.
Malheureusement, ce calcul orienté-objet ne semble pas adapté à l'incorporation de traits logiques. Afin de continuer, nous devons restreindre les mécanismes orientés-objet à des mécanismes statiques, plus faciles à combiner avec la logique et apparemment suffisant pour notre objectif d'interopérabilité. Une telle combinaison de mécanismes orientés-objet et de logique est présente dans l'environnement FoCaLiZe donc nous proposons un encodage superficiel de FoCaLiZe dans Dedukti. Les difficultés principales proviennent de l'intégration de Zenon, le prouveur automatique de théorèmes sur lequel FoCaLiZe repose, et de la traduction du langage d'implantation fonctionnel de FoCaLiZe qui présente deux constructions qui n'ont pas de correspondance simple en Dedukti : le filtrage de motif local et la récursivité.
Nous démontrons finalement comment notre encodage de FoCaLiZe dans Dedukti peut servir en pratique à l'interopérabilité entre des systèmes de preuve à l'aide de FoCaLiZe, Zenon et Dedukti. Pour éviter de trop renforcer la théorie dans laquelle la preuve finale est obtenue, nous proposons d'utiliser Dedukti en tant que méta-langage pour éliminer des axiomes superflus.

Abstract :
Dedukti is a Logical Framework resulting from the combination of dependent typing and rewriting. It can be used to encode many logical systems using shallow embeddings preserving their notion of reduction.
These translations of logical systems in a common format are a necessary first step for exchanging proofs between systems. This objective of interoperability of proof systems is the main motivation of this thesis.
To achieve it, we take inspiration from the world of programming languages and more specifically from object-oriented languages because they feature advanced mechanisms for encapsulation, modularity, and default definitions. For this reason we start by a shallow translation of an object calculus to Dedukti. The most interesting point in this translation is the treatment of subtyping.
Unfortunately, it seems very hard to incorporate logic in this object calculus. To proceed, object-oriented mechanisms should be restricted to static ones which seem enough for interoperability. Such a combination of static object-oriented mechanisms and logic is already present in the FoCaLiZe environment so we propose a shallow embedding of FoCaLiZe in Dedukti. The main difficulties arise from the integration of FoCaLiZe automatic theorem prover Zenon and from the translation of FoCaLiZe functional implementation language featuring two constructs which have no simple counterparts in Dedukti: local pattern matching and recursion.
We then demonstrate how this embedding of FoCaLiZe to Dedukti can be used in practice for achieving interoperability of proof systems through FoCaLiZe, Zenon, and Dedukti. In order to avoid strengthening to much the theory in which the final proof is expressed, we use Dedukti as a meta-language for eliminating unnecessary axioms.