Interoperable Proof Systems

by

Florent Kirchner

Interoperable Proof Systems

by

Florent Kirchner

Submitted in Requirement for the
Degree of Doctor in Philosophy

Laboratoire d'Informatique
École Polytechnique
2007

© 2007 Florent Kirchner
All rights reserved

Imprimé à l'École Polytechnique, Palaiseau.

Parcourir sa route et rencontrer des merveilles, voilà le grand thème.

Cesare Pavese
Il mestiere di vivere

Contents

List of Figures

List of Tables

Introduction

An interoperable representation of mathematical proofs can be achieved by combining a simple logical framework with a formal definition of the concept of proof language.

⋆

Formal methods is a discipline at the intersection of mathematics and computer science. It advocates the use of software programs called *formal tools* to discover, specify and verify the properties of mathematical objects. And since mathematics are the preferred language of experimental sciences such as physics, chemistry or biology, and their engineering derivatives, the range of applications for formal methods is very broad — from verification of spacecraft flight software (Gluck and Holzmann, 2002) to modeling of protein interaction in biological networks (Danos and Laneve, 2004; Eker et al., 2002).

In formal methods, a significant status is given to the objects representing evidence that a given property is verified. For instance, this kind of evidential information is mandatory to pass standardized certification requirements, such as the assurance level 7 of the Common Criteria (ISO, 1998), recently adopted as the international standard 15408. The form of the evidence varies, and can be very different from a mathematical proof of the property: some tools just return a `Yes` when the verification is achieved, while others output a data structure equivalent to a detailed mathematical proof.

Proof assistants, or *theorem provers*[1], belong to the second category: they help the formal methods developer walk through the details of the mathematical proof of a given property, in order to verify it. Modern-day examples of such tools are ACL2, Coq, HOL, Isabelle, Lego, Mizar, NuPrl, PVS, *etc.* A proof assistant provides a set of commands, or *proof language*, manipulated by the user to produce a proof, under the supervision of a *proof engine* ensuring that the user's operations are mathematically sound. For instance, the proof engine verifies that the specifications input by the user are well-formed, or that a language construct is indeed a description of a mathematically correct proof. Because they already allow their users to deal closely with the proofs during the verification process, usually these tools produce evidence which comes close to complete mathematical proofs.

Still, depending on the proof assistant, the practical form of evidential data is very different. For instance, proofs in PVS are represented by constructs of the proof language, while in Coq they are objects of the proof term language. What is more, the specification and proof languages also vary greatly between provers. These discrepancies are serious enough to im-

[1] In this manuscript, we consider "theorem prover" as a synonym for "proof assistant". Historically, theorem provers were fully-automated tools while proof assistants proceeded much more interactively; however the current trend (Boulton, 1992; Aagaard et al., 1993; Lowe and Duncan, 1997) seems to be the convergence of these kinds of systems.

peach the interchangeability, or interoperability, of proofs between tools. As a consequence, if a proof in, *e.g.* Coq cannot be reused in another prover, then it needs to be re-coded and maintained independently, which is, at best, inefficient. This also limits the opportunities for collaborative work between proof developers. Furthermore, as the domain matures, the size of developments grows, amplifying these problems.

As stated earlier, the two main components of a theorem prover are its proof language and its proof engine. In this manuscript, we make the case that interoperability in the representation of proofs can be obtained by enforcing a few conditions on the design of these two components. In particular, we present a simple logical framework, *i.e.* a base for a proof engine, which checks proofs that are generic enough to fit other proof assistant's proof engines. We also propose a formalization of the definition of proof languages, which includes the presentation of semantical and typing frameworks, and allow us to better comprehend the existing languages, identify interoperable features, and eventually suggest ways to enhance them.

⋆

Let us spare some words on the topic of proof languages. The concept of proof language was introduced in the 1970s, with the emergence of the first proof assistants: AUTOMATH (de Bruijn, 1970), Mizar (Trybulec, 1978) and Nqthm (Boyer and Moore, 1979, 1988). Most of these used *declarative* languages, that consist in stating intermediary lemmas until the proof engine manages to combine them into the final proof. This approach is quite close to the way proofs are developed in natural language, but does not allow much interaction between the user and the proof engine. Tools using *procedural* languages, that provide users with commands to direct the proof engine through the proof construction, appeared almost simultaneously, and blossomed in the following decade: LCF (Gordon et al., 1978, 1979) which can be seen as the ancestor of procedural theorem provers, soon followed by Coq (Coquand and Huet, 1985), NuPrl (Constable et al., 1986), Isabelle 1988 (Paulson, 1988), PVS (Owre et al., 1992), HOL (Gordon and Melham, 1993), Lego (Pollack, 1994)), ... A detailed comparison of these tools, their logical frameworks and their proof languages, can be found in (Delahaye, 2001; Wiedijk, 2006).

In all these tools, the proof language, be it declarative or procedural, is only a fraction of the complete interaction language of the proof assistant. In addition, there are instructions to specify mathematical objects, a necessary step before starting proving properties on these objects. Depending on the tool reviewed, one can also find instructions enabling modular developments, undo/redo facilities, proof display triggers, *etc.* A proof language is used solely in *proof-editing mode*, *i.e.* the mode entered by the prover when a proof is started, and that the prover leaves when the proof is completed. On the contrary, the rest of the interaction language can be found in any part of the formal developments. In this work, we are mainly interested in

the proof language part of the interaction, and more precisely in procedural proof languages.

Traditionally, elements of proof languages have been divided between *tactics* and *strategies* (also called *tacticals* in some tools), with tactics being used to modify the state of the proof, and strategies being considered as tactics combinators. With time, strategies evolved into an abstraction of the prover's implementation language, borrowing more and more features from the programming world, and eventually leading to the ML family of programming languages. As a result of this evolution, the exact definition of tactics and strategies is quite vague, and subject to recurring debates. Since these were named after military terms, to examine their meaning in this context does not seem an inappropriate place to start.

The Dictionary of Military and Associated Terms (Uni, 2001) defines tactics as:

> "The level of war at which battles and engagements are planned and executed to achieve military objectives."

And strategies are:

> "A prudent idea or set of ideas for employing the instruments of national power in a synchronized and integrated fashion to achieve theater, national, and/or multinational objectives."

In other terms, tactics are used as a mean to achieve a given local objective: say, prove a particular case of a property. An emphasis is made on the execution of tactics, and they stand close to the action (the construction of the proof) itself. On the contrary, strategies are viewed as a mean to attain a global objective — for example, prove a non-trivial lemma — by devising a way to "synchronize and integrate" tactics.

An atypical definition of the notions of tactics and strategies is given, in a less belligerent context, by theologian and philosopher Michel de Certeau (de Certeau, 1948). De Certeau was interested in the link between human beings and the space they occupy: he links strategies to the design and manipulation of the urban landscape on a high level by institutions and structure of power, and defines tactics as the means employed by individuals to create space for themselves in environments defined by strategies:

> "[A tactic is deployed] on and with a terrain imposed on it and organized by the law of a foreign power."

In this setting, the same idea surfaces: tactics are viewed as fine-grained means to meet an objective in a framework defined by strategies.

In this dissertation, we follow the aforementioned definitions, and we define tactics as procedures that directly modify the proof: the application of a tactic on a proof generates an extension of this proof. The part of the proof that the tactic applies onto is called the *goal*, and the extensions generated

are called *subgoals*. Tactics, as transformers of goals into subgoals, thus have a definition that is both operational (*i.e.* tactics are about execution of the transformation) and local (*i.e.* tactics are only concerned with one extension of the proof). On the other hand, we view strategies as constructs that take other tactics and strategies as parameters to build a proof. Instantiated strategies, also called *proof scripts*, derive subgoals from a given goal: they behave just as tactics; however, strategies in their uninstantiated form are inapplicable to proofs, and appear as a way to combine, integrate tactics together. In other terms, we consider strategies as higher-order tactics, *i.e.* functions on tactics.

Therefore, if it is just a difference between higher and first-order constructs, is there a need to separate tactics from strategies in a proof language? Or is it time to reunite the two notions under the unified terminology of *proof commands*? In general, we leave this question up for grabs; but for the work exposed here we believe that the distinction between tactics and strategies, if not relevant from an analytical point of view, can still have its utility in the semantical realm. This can be compared to the distinction made in analysis between functions and constants: the first are just a higher-level version of the second, and having different labels for them helps cope with the mental manipulation of these objects. Hence in the rest of this work, we will try as much as possible to state general properties on proof commands, but we will resort to the tactic / strategies paradigm whenever we feel it aids comprehension. We will also demonstrate how a different take on the classification of proof language constructs can shed a different light on the structure of proof languages.

⋆

Two postulates, originally formulated by de Bruijn as advices to developers of formal tools, guide this work. They can be paraphrased as follows:

> **Postulate 1** (Simplicity)**.** *Strive to keep the underlying formalism of provers as simple as possible.*

> **Postulate 2** (Choice)**.** *People will never agree on the logical framework: they need to be given a choice.*

Postulate 1 is motivated by both the need to ascertain that at least the core of formal tools is bug-free, supporting the long-standing concept of correctness-by-minimality and the concern that if formal methods are to be popularized, one cannot afford to put off potential users with intricate frameworks and involved theories. Postulate 2 was enacted as a response to the multitude of logical frameworks (in particular with duplication engendered by the classical / intuitionistic schism) that are eligible as a basis for formal tools, and the seemingly never-ending controversy over their respective merits and limitations.

It is easy to see that the two postulates fit tightly the topic of this dissertation: interoperability is, if anything, about choice; and the thesis of this work relies on the elegant simplicity of a logical framework and of a formalism for proof languages. The stake here is in whether both features can be implemented without interference (because in particular, choice hardly entails simplicity), and if the compromises generated with respect to the other parts of the system in order to satisfy postulates 1 and 2 are reasonable.

⋆

The dissertation follows the didactic pattern used in the this introduction. Chapter 1, after a brief recall on predicate languages and sequent calculus, exposes the current state of the art in terms of proof representation. Chapter 2 presents the logical frameworks that are used throughout the rest of the manuscript, and highlights their relations to one another. In chapter 3 to 5 the formalization of the concept of proof language is tackled: chapter 3 draws a parallel between imperative programming languages and procedural proof languages, and sketches a semantical framework for these types of languages; chapter 4 uses a bit of category theory to characterize a pivotal element of the semantics of strategies; chapter 5 provides insights on a typing system for proof languages, and uses the semantical formalism developed in the previous chapters to provide type-safety results.

While first five chapters deal with proof languages in general, and are illustrated using toy examples, the second part of this manuscript instantiates these frameworks with concrete proof languages. The chapter 6 builds upon the previous parts to propose a series of methods to achieve interoperability between proofs in various logical settings, and for various formal tools. Chapter 7 describes an implementation of these ideas on the form of a prototype system for developing interoperable proofs, *i.e.* an interoperable proof assistant called Fellowship. Finally chapter 8 justifies the viability of the approach taken, by demonstrating a way for first-order logic to finitely express the full power of axiom schemes.

Parts of this dissertation have been published in international conferences and workshops (Kirchner, 2005a,b, 2006; Kirchner and Muñoz, 2006; Kirchner and Sinot, 2006).

1 Sequent Calculus

This chapter recalls the definition of sequent calculus, exposes some of the means of representing proofs in this formalism, and proposes a classification of proof language constructs.

⋆

First-order logic is a well understood, very widespread formalism. Moreover it is quite a capable framework: in practice a lot of real-world specifications and proofs are first-order, and one could argue that *in fine* the representation of any problem as bits and registers in a computer's memory is a first-order one. In this manuscript we are interested in a particular logical formalism, where formulas are written in a predicate language and proofs are build using sequent calculus: this chapter presents that formalism. It continues by addressing the topic of the representation of proofs, and finally following up on the discussion of the introduction and proposing a taxonomy for proof language elements.

1.1 Syntax of predicate languages

Definition 1.1.1 ($\mathcal{L}^1$)**.** Let $\mathcal{L}^1$ be a language parametrized by the following possibly infinite, but countable, sets of symbols:

- the set of predicate symbols $\mathcal{P} = \{\mathtt{p}, \mathtt{q}, \ldots\}$, along with their arities;
- the set of function and constant symbols $\mathcal{F} = \{\mathtt{f}, \mathtt{g}, \mathtt{h} \ldots\}$, along with their arities;
- the set of variable symbols $\mathcal{V} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots\}$;

The syntax of well-formed terms and formulas of the language follows, in Backus-Naur form:

$$\begin{aligned} \mathtt{t}, \mathtt{u} &::= \mathtt{x} \mid \mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n) \\ A, B &::= \top \mid \bot \mid \mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n) \\ &\quad \mid \neg A \mid A \Rightarrow B \mid A \wedge B \mid A \vee B \mid \forall \mathtt{x}.A \mid \exists \mathtt{x}.A \end{aligned}$$

where $\mathtt{f}$ and $\mathtt{p}$ are respectively function and predicate symbols of arity $\mathtt{n}$, and $\mathtt{x}$ is a variable.

Note. The reunion of the sets $\mathcal{P}$ and $\mathcal{F}$ constitute the *signature* of $\mathcal{L}^1$.

We define informally the main concepts and operations that take place over the language $\mathcal{L}^1$ — for a formal definition of these mainstream notions, see for instance (Krivine, 1993).

Bound variables of a formula A are variables appearing in A under the scope of a quantifier on the same variable. Free variables are unbound

variables, and fresh variables *wrt.* a formula A appear neither bound nor free in a A. For instance, in $\forall x.p(x, y)$, x is bound (by the quantifier $\forall x$), y is free and z is fresh.

Substitution is the replacement of a variable by a term in a term or formula, *modulo renaming of variables* to avoid variable catching. Substitution is denoted $[x \leftarrow t]$. For instance, $(\forall x.p(x, y))[y \leftarrow f(x)]$ is $\forall z.p(z, f(x))$.

Sub-terms and sub-formulas are parts of a term or formula, with free variables possibly substituted. For instance, $p(f(x))$ is a sub-formula of $\forall x.p(x)$.

The above definition of a predicate language can be adapted to deal with multiple sorts, thus giving rise to many-sorted predicate languages.

Definition 1.1.2 ($\mathcal{L}^1_m$)**.** Let $\mathcal{L}^1_m$ be a language that contains:

- a set of sorts $\mathcal{S} = \{s, r, \ldots\}$ including a sort bool;
- a set of predicate symbols $\mathcal{P} = \{p, q, \ldots\}$, each with their arity;
- a set of function and constant symbols $\mathcal{F} = \{f, g, h, \ldots\}$, each with their arity;
- for each sort s, a countable set of variables $\mathcal{V}_s = \{x, y, z, \ldots\}$.

Moreover,

- to each function symbol of arity n is associated a rank $s_1 \to \ldots \to s_n \to s_{n+1}$ where $s_1, \ldots, s_n$ are the sorts of its arguments and s_{n+1} is the sort of its result;
- to each predicate symbol p is associated a rank $s_1 \to \ldots \to s_n \to \text{bool}$ where $s_1, \ldots, s_n$ are the sorts of its arguments;
- for any function or predicate symbol z, we name z^* its associated sort.

The syntax of well-formed terms and formulas is derived from the syntax of $\mathcal{L}^1$, with additional constraints enforced by sorts. This syntax uses Church notation, where abstractions are decorated with the sorts of their variables. The well-formedness constraints are given using inference rules to reflect the conditional nature of their formulation: for instance, a term $f(t_1, \ldots, t_n)$ is well-formed if f is a function symbol of rank $s_1 \to \ldots \to s_n \to s_{n+1}$, and each t_i has sort s_i. Using the notation $\rhd a : s$ for the statement "a is a well-formed term or formula of sort s", figure 1.1 presents the inference rules for well-formed terms and formulas.

Note. This is a slight variation in notation from the common presentation of many-sorted first-order languages, where ranks are written using tuples instead of arrows, and for predicates the final sort bool is omitted.

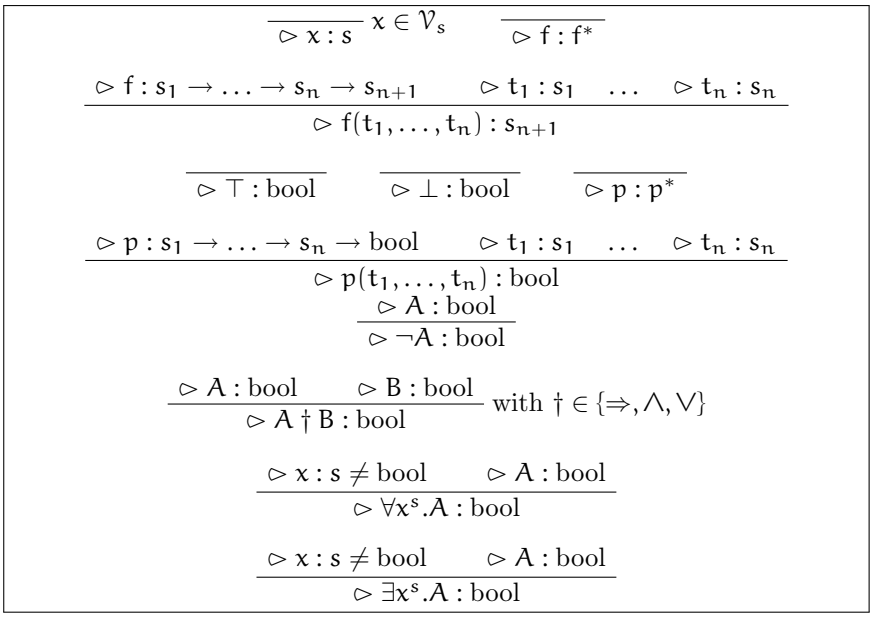


Figure 1.1: Syntax of well-formed terms and formulas in $\mathcal{L}^1_m$

Notation. Contiguous universal and existential quantifications over similarly-sorted variables are collapsed. For instance, the formula $\forall x^s_1.(\ldots(\forall x^s_n.A))$ is written $\forall x_1, \ldots, x^s_n.A$.

All other concepts, such as bound and free variables, replacement, substitution, sub-terms and sub-formulas, are extended similarly to deal with many-sorted terms and formulas.

1.2 The sequent calculus

Sequent calculus is a logical formalism pioneered by Gentzen in 1935, originally as a tool to study natural deduction (Gentzen, 1935). As with any calculus, it has two fundamental components: objects called sequents, and some sort of computation called deduction.

Sequents assert that from a finite multiset of formulas Γ, one can prove at least a formula of another multiset Δ. This assertion is noted $\Gamma \vdash \Delta$, and Γ and Δ are called respectively the *antecedent* and the *succedent*. Unlike in the Gentzen's presentation of sequents, we make it possible to distinguish a particular formula amongst the members of Γ and Δ, called *active formula*. Depending if an active formula A belongs to the antecedent or the succedent of a sequent, we note $\Gamma; A \vdash \Delta$ or $\Gamma \vdash A; \Delta$: we say that such a sequent is *polarized*.

Deduction consists in a series of *rules*, that transform an input sequent into zero, one or more resulting sequents. These transformations are called *inferences*, and denoted by the vertical stacking of input and output sequents, separated by a horizontal bar and annotated with the name of the rule. For instance, is S_0 is the input sequent and $S_1, \ldots, S_n$ are the output sequents of a rule r, we write:

$$r\,\frac{S_1 \quad \ldots \quad S_n}{S_0}$$

In particular, if the input sequent contains an active formula, only this formula is subject to a computation, *i.e.* its non-active parts will be found unchanged by the inference in the output sequents. Rules that deal with an active formula are earmarked with either a left label $_{\mathcal{L}}$ or a right label $_{\mathcal{R}}$, depending on the polarization of the sequent.

When a sequent calculus is defined based on a first-order predicate language such as the ones of definitions 1.1.1 and 1.1.2, it forms a *logical framework* called *first-order logic*.

Definition 1.2.1 ($\mathrm{LK}_{\mu\tilde{\mu}}$)**.** Figure 1.2 presents a set of inference rules that define a classical sequent calculus called $\mathrm{LK}_{\mu\tilde{\mu}}$ (Herbelin, 2005), based on the single-sorted predicate language $\mathcal{L}^1$.

Remark that because we use a definition of sequents where the antecedent and the succedent are multisets, there is no need in this system for formula-swapping rules. Also, contraction rules are missing from the figure 1.2. One can chose either to add them directly, resulting in two new inference rules. The alternative is to encode them using the cut rule with a notable drawback: the cut-elimination property no longer holds, simply because contraction cannot be eliminated.

1.3 Proofs and traces

A proof consists of a series of computations over an initial input sequent, by successive application of rules: when no output sequent remains, the initial sequent is proved. The trace of the computation is often stored for future reference, in the form of a tree of sequents where edges are labelled by rules, often called *proof tree*. Figure 1.3 illustrates the form of a trace.

Notice however that there is some redundancy in this representation, because inference rules are deterministic. For instance in $\mathrm{LK}_{\mu\tilde{\mu}}$, if an input sequent $\Gamma; A \Rightarrow B \vdash \Delta$ produces two sequents $\Gamma \vdash A; \Delta$ and $\Gamma; B \vdash \Delta$, then we know that only the $\Rightarrow_{\mathcal{L}}$ rule can have resulted in that transformation. Hence the information contained in the tree of sequents is enough to build a complete representation of a proof.

$$\mathrm{ax}_{\mathcal{L}}\ \frac{}{\Gamma; A \vdash A, \Delta} \qquad \mathrm{ax}_{\mathcal{R}}\ \frac{}{\Gamma, A \vdash A; \Delta}$$

$$\bot_{\mathcal{L}}\ \frac{}{\Gamma; \bot \vdash \Delta} \qquad \top_{\mathcal{R}}\ \frac{}{\Gamma \vdash \top; \Delta}$$

$$\mathrm{activate}_{\mathcal{L}}\ \frac{\Gamma, A \vdash \Delta}{\Gamma; A \vdash \Delta} \qquad \mathrm{activate}_{\mathcal{R}}\ \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A; \Delta}$$

$$\mathrm{cut}\ \frac{\Gamma \vdash A; \Delta \qquad \Gamma; A \vdash \Delta}{\Gamma \vdash \Delta}$$

$$\Rightarrow_{\mathcal{L}}\ \frac{\Gamma \vdash A; \Delta \qquad \Gamma; B \vdash \Delta}{\Gamma; A \Rightarrow B \vdash \Delta} \qquad \Rightarrow_{\mathcal{R}}\ \frac{\Gamma, A \vdash B; \Delta}{\Gamma \vdash A \Rightarrow B; \Delta}$$

$$\wedge_{\mathcal{L}}\ \frac{\Gamma, x : A, y : B \vdash \Delta}{\Gamma; A \wedge B \vdash \Delta} \qquad \wedge_{\mathcal{R}}\ \frac{\Gamma \vdash A; \Delta \qquad \Gamma \vdash B; \Delta}{\Gamma \vdash A \wedge B; \Delta}$$

$$\vee_{\mathcal{L}}\ \frac{\Gamma; A \vdash \Delta \qquad \Gamma; B \vdash \Delta}{\Gamma; A \vee B \vdash \Delta}$$

$$\vee_{1\mathcal{R}}\ \frac{\Gamma \vdash A; \Delta}{\Gamma \vdash A \vee B; \Delta} \qquad \vee_{2\mathcal{R}}\ \frac{\Gamma \vdash B; \Delta}{\Gamma \vdash A \vee B; \Delta}$$

$$\neg_{\mathcal{L}}\ \frac{\Gamma \vdash A; \Delta}{\Gamma; \neg A \vdash \Delta} \qquad \neg_{\mathcal{R}}\ \frac{\Gamma; A \vdash \Delta}{\Gamma \vdash \neg A; \Delta}$$

$$\forall_{\mathcal{L}}\ \frac{\Gamma; B[x \leftarrow t] \vdash \Delta}{\Gamma; \forall x^{A}.B \vdash \Delta} \qquad \forall_{\mathcal{R}}\ \frac{\Gamma \vdash B; \Delta}{\Gamma \vdash \forall x^{A}.B; \Delta}$$

$$\exists_{\mathcal{L}}\ \frac{\Gamma; B \vdash \Delta}{\Gamma; \exists \alpha^{A}.B \vdash \Delta} \qquad \exists_{\mathcal{R}}\ \frac{\Gamma \vdash B[x \leftarrow t]; \Delta}{\Gamma \vdash \exists x^{A}.B; \Delta}$$

Figure 1.2: Inference rules for $\mathrm{LK}_{\mu\tilde{\mu}}$

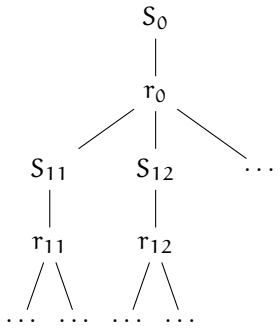
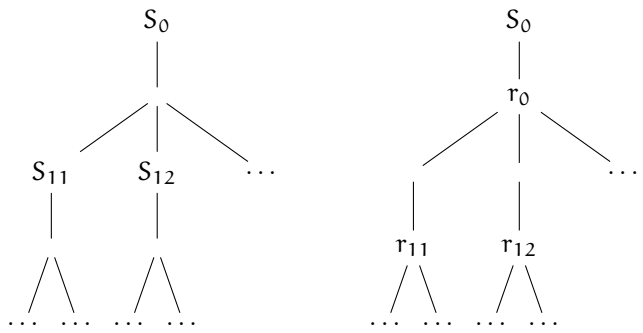


Figure 1.3: A proof trace as a tree of sequents and rules.



(a) Sequents (b) Rules

Figure 1.4: Non redundant proof traces

Conversely, if given an input sequent and an inference rule, then one can deduct the result sequents. By induction, one can reconstitute the complete proof representation from only the initial sequent and the collection of inference rules. The two simplified representations are summarized in figure 1.4.

However, proofs can become very large objects, taking sometimes years to develop (Hales, 2004; Gonthier, 2005). Therefore, there is a need to store and represent proofs in the making, *i.e.* open proofs. In these proofs, one or more sequents are left untouched by inference rules: they are called *open goals*, and the branch of the proof they occupy is called an *open branch*. Dealing with these open proofs is not a problem in the representation of figure 1.4a: the unproved sequents will just be leaves of the tree (sometimes emphasized by using a question mark above them). However, in the representation of figures 1.3 and 1.4b, there is a need for a new inference rule, stating that the associated sequents are unproved. To this end, we introduce the `idtac` rule, which is the identity computation, *i.e.* processes its input sequent into the same output sequent. This rule acts as a placeholder for another inference, allowing for well-formed trace trees.

1.4 A taxonomy of proof commands

Now that the logical framework has been defined, and that the topic of proof representation has been evoked, we can complete the discussion started in the introduction about the classification of proof language constructs.

In modern procedural theorem provers, some tactics correspond to the bare logical inference rules, while others are abstracted and automated versions of these rules. When building a proof, both kinds of tactics are used, but when available the latter is often preferred, because of their ease of use

— the advantages they provide to facilitate the interactive process range from automatic name generation to complex backtracking and branching features.

What is more, proofs are seldom saved in their canonical, tree-like form. Instead, what is retained is the proof script that generated them. While this kind of representation has the advantage of giving a higher-level comprehension of the essence of the proof and facilitate re-reading, it also has the flaw of being easily broken by changes in the proof language semantics. In some cases, the tactics may be over-dimensioned for the goal, and thus provide very few information to the reader. For instance, calling an automatic first-order solver *via* a tactic on a first-order goal masks the effective inference rules and forces the reader looking for a finer-grained proof to re-discover it.

We argue that these two pattern of utilisation for proof languages are equally important in their uses. Hence in establishing a classification of such languages, we will consider the following criterions:

1. Is the language construct used to build the proof?
2. Is the language construct suitable to represent the proof?

Commands as inference rules

The first identifiable set of commands corresponds to the tactics that implement the logical framework's inference rules. The presence of these tactics is by definition a requisite for procedural theorem proving, and they constitute the building bricks of the proof. As such, they are mandatory both to build a proof and its representation. We call them *base tactics*.

Commands as programming constructs

Strategies are used to build other commands, *i.e.* they act as a programming language at the level of the proof language. They are not used directly to construct proofs *per se*, but rather to build the proof builders. Moreover, strategies are not suited for the representation of proofs. Indeed, because they are in essence programming constructs, only the *result* of the programs are to be retained. As an example, take the following proof script:

```
try (apply lemma_1) (apply lemma_2)
```

that attempts to perform a deduction by applying the result of a first lemma, and that applies a second lemma if the first one is unsuccessful. Whichever branch of the `try` strategy gets selected should be recorded in the representation of the proof. However, there is no point in recording the whole script. We call these *programming strategies*.

Yet there are two constructs that are important exceptions to the non representativity of strategies:

- `idtac`, that "does nothing" when applied to a sequent. It is used to represent incomplete proofs, and fill the place where later tactics might be added.
- `;[||]`, which combines commands in a tree structure, isomorphic to the structure of the proof. It is the "glue" that holds the building bricks of the proof together, and thus it is necessary to the representation of non-trivial proofs.

Together with tactics, these two strategies are all that are needed to represent proofs. In reference to their tree-structuring functionality, we call these special programming strategies *bark strategies*.

Commands as super inference rules

We call *extended tactic* a command that is defined as a combination of tactics, by either using strategies or the prover's implementation language. Thus an extended tactic may be seen as a super inference rule, created by the combination of smaller rules. As such, extended tactics are proof builders, on par with base tactics.

At the level of proof representation, extended tactics should be viewed as labelled boxes containing instances of inference rules. Or, equivalently, a combination of base tactics and bark strategies. This enables the proof reader to eventually refine its understanding of the proof by "opening the box" labelled by an extended tactic. For instance, the `apply` tactic mentioned in the previous paragraph should be expandable to display the appropriate cut rule, and the quantifier rules that have been used to achieve unification.

Commands as interaction controls

Building a proof is a largely interactive process, with goals being presented one by one, postponed, changes undone, *etc.*. The last group of commands are the ones that control such interactions. They do not contribute directly to the construction of the proof, neither should they appear in a finished proof script. Examples of such commands include the `postpone` construct evoked in the introduction, PVS's `(hide)` / `(reveal)` or Coq's `focus`. Also, Coq's unassuming '`.`' command, that (literally) punctuates command applications, should be seen as both an evaluation trigger and an interactive control, returning the first subgoal of the active subtree once the evaluation is complete. We call these controls *interactive commands*.

Table 1.1 sums up the behaviour of the different language classes with respect to these two criterions.

Table 1.1: Language classes and purposes

	Proof construction	Proof representation
Base tactics	✓	✓
Extended tactics	✓	✓as a macro
Bark strategies	✓	✓
Programming strategies	×	×
Interaction controls	×	×

2 $\bar{\lambda}\mu\tilde{\mu}$-Calculus and Variations

Here we define the logical framework of our developments, in the form of a proof term calculus; it is declined into four variations: classical, intuitionistic, and their respective minimal weakening.

These frameworks are adaptations of the system $\textsc{lk}_{\mu\tilde{\mu}}$, defined in chapter 1, that present interesting characteristics for interactive theorem proving and proof interoperability. What is more, the simplicity and diversity of the first-order logics they define qualifies them under both postulates 1 and 2.

In this chapter we review the principle of the proof-as-term isomorphism, illustrated with the example of the $\textsc{lk}_{\mu\tilde{\mu}}$ system. Then, by the means of this isomorphism, we detail the logical inference rules of our original systems; we start with the most general logical setting, *i.e.* classical logic, and from there we work our way through more restricted, intuitionistic and minimal, settings. We conclude with the representation of incomplete proofs.

$\star$

The Curry-de Bruijn-Howard correspondence identifies formulas as types, making a sequent into a typing statement, and deduction rules as type inferences. The objects being typed are called *proof terms*, which should not be confused with terms of the predicate language (to this end, we will never omit the 'proof' adjective when referring to proof terms). For instance, a well-known proof term language is the λ-calculus, which labels formulas of minimal natural deduction.

Classical sequent calculus being more expressive than minimal natural deduction, the proof term language also needs to be more expressive than plain λ-calculus. A few languages have been proposed, from Herbelin's $\bar{\lambda}$-calculus (Herbelin, 1995) to Urban's calculus of names and conames (Urban, 2001). In all of these formalisms, the idea is to associate with each inference rule a construction of the proof term language.

Along with the choice of a proof language, one needs to examine to following question: to which element of the sequent is a proof term associated? The problem of the place of the proof term in the sequent arises because, unlike in natural deduction, a deduction in sequent calculus can take place on any formula in a sequent. The use of sequents with active formulas partly solve this question: the polarization marks the target of the computation. The active formula in a sequent is considered as the type of a particular proof term (independently from the choice of a proof term language). Thus sequents are written:

$$\Gamma; \mathtt{t} : A \vdash \Delta \qquad\qquad \Gamma \vdash \mathtt{t} : A; \Delta$$

However in case where there is no active formula, the problem remains. The solution is to have the whole sequent represent the type of the proof term, which we note:

$$\mathtt{t} : (\Gamma \vdash \Delta)$$

Interestingly, this last notation is not much different from the one proposed by Urban, where the proof term is a component of the sequent, at the same level as the antecedent or the succedent.

In the rest of this chapter, we introduce $\bar{\lambda}\mu\tilde{\mu}$, a calculus of proof terms that works with sequents that have active formulas. This presentation uses the flip side of the Curry-de Bruijn-Howard correspondence: we define a calculus of proof terms as a labeling system for a given logical framework, instead of defining the logics as a type inference system for a particular language. Of course, an isomorphism being symmetrical by nature, it is sometimes easier to refer to the typing side of the correspondence. The logical aspects, however, will remain the guiding concern of this development.

2.1 The proof-as-term isomorphism for $\mathbf{lk}_{\mu\tilde{\mu}}$

For the deduction system $\text{LK}_{\mu\tilde{\mu}}$, the proof language used by Herbelin is called the $\bar{\lambda}\mu\tilde{\mu}$-calculus (Curien and Herbelin, 2000; Wadler, 2003; Herbelin, 2005), *i.e.* an extension of λ-calculus with two binders μ and $\tilde{\mu}$ to capture classical logic, and the choice of an active formula in the sequent. What is more, one introduces additional operators to reflect the types $\wedge$ and $\vee$, and a symmetrical to λ to inhabit existential types.

Note that because our purpose is only to give a quick overview of the mechanisms of the calculus, we do not enter its details. All these results can be found and further explained in (Curien and Herbelin, 2000; Herbelin, 2005).

Definition 2.1.1 ($\bar{\lambda}\mu\tilde{\mu}$ proof terms)**.** The syntax of the $\bar{\lambda}\mu\tilde{\mu}$-calculus defines commands c, terms v and environments e:

$$
\begin{aligned}
c &::= \langle v \| e \rangle \\
v, v' &::= x \mid \ltimes \mid \lambda x^A.v \mid e \cdot v \mid \neg(e) \mid (v, v') \mid \text{inj}_r v \mid \text{inj}_l v \mid \mu\alpha^A.c \\
e, e' &::= \alpha \mid \rtimes \mid v \cdot e \mid \lambda\alpha^A.e \mid \neg[v] \mid \text{proj}[x, x', c] \mid [e, e'] \mid \tilde{\mu}x^A.c
\end{aligned}
$$

where A is a formula. $\ltimes$ and $\rtimes$ are constants, respectively called *unit* and *tinu*, linked to the connectors $\top$ and $\bot$.

Remark that this syntax is perfectly symmetrical, notwithstanding the projection operators. The symmetry is extended to the reduction rules of the calculus.

Definition 2.1.2 (Reduction rules for $\bar{\lambda}\mu\tilde{\mu}$)**.** The evaluation rules for $\bar{\lambda}\mu\tilde{\mu}$ proof terms are the same as in (Curien and Herbelin, 2000): one will recognize the evaluation relations for λ, μ and $\tilde{\mu}$ redexes, enriched with the dual

projection and pair reductions.

$$\langle \lambda x^A.v_1 \| v_2 \cdot e \rangle \to \langle v_2 \| \tilde{\mu} x^A.\langle v_1 \| e \rangle \rangle \qquad (\lambda)$$
$$\langle (e_2 \cdot v) \| \tilde{\lambda} \beta^A.e_1 \rangle \to \langle \mu \beta^A.\langle v \| e_1 \rangle \| e_2 \rangle \qquad (\tilde{\lambda})$$
$$\langle \mu \beta^A.c \| e \rangle \to c[\beta \leftarrow e] \qquad (\mu)$$
$$\langle v \| \tilde{\mu} x^A.c \rangle \to c[x \leftarrow v] \qquad (\tilde{\mu})$$
$$\langle \mathrm{inj}_l v \| [e_1, e_2] \rangle \to \langle v \| e_1 \rangle \qquad (\mathrm{inj}_l)$$
$$\langle \mathrm{inj}_r v \| [e_1, e_2] \rangle \to \langle v \| e_2 \rangle \qquad (\mathrm{inj}_r)$$
$$\langle (v_1, v_2) \| \mathrm{proj}[x_1, x_2, c] \rangle \to \langle v_1 \| \tilde{\mu} x_1.\langle v_2 \| \tilde{\mu} x_2.c \rangle \rangle \qquad (\mathrm{proj}_1)$$
$$\langle (v_1, v_2) \| \mathrm{proj}[x_1, x_2, c] \rangle \to \langle v_2 \| \tilde{\mu} x_2.\langle v_1 \| \tilde{\mu} x_1.c \rangle \rangle \qquad (\mathrm{proj}_2)$$

Also a series of η-equivalences can be defined for each of the binders in the proof term syntax:

$$v \leftrightarrow \lambda x.\mu\alpha.\langle v \| x \cdot \alpha \rangle \qquad x \text{ not free in } v \qquad (\eta^R_\lambda)$$
$$e \leftrightarrow \mu\beta.\langle \lambda x.\mu\alpha.\langle x \| \beta \rangle \| e \rangle \cdot \tilde{\mu} y.\langle \lambda x.y \| e \rangle \qquad \beta \text{ not free in } e \qquad (\eta^L_\lambda)$$
$$v \leftrightarrow \tilde{\mu} y.\langle v \| \tilde{\lambda}\alpha.\mu x.\langle y \| \alpha \rangle \rangle \cdot \mu\beta.\langle v \| \tilde{\lambda}\alpha.\beta \rangle \qquad y \text{ not free in } v \qquad (\eta^R_{\tilde{\lambda}})$$
$$e \leftrightarrow \tilde{\lambda}\alpha.\tilde{\mu}x.\langle \alpha \cdot x \| e \rangle \qquad \alpha \text{ not free in } e \qquad (\eta^L_{\tilde{\lambda}})$$
$$v \leftrightarrow \mu\alpha.\langle v \| \alpha \rangle \qquad \alpha \text{ not free in } v \qquad (\eta_\mu)$$
$$e \leftrightarrow \tilde{\mu}x.\langle x \| e \rangle \qquad x \text{ not free in } e \qquad (\eta_{\tilde{\mu}})$$

Note. Let us recall a few customary observations: first of all notice that the rules (μ) and $(\tilde{\mu})$ one the one hand, and (λ) and $(\tilde{\lambda})$ on the other hand, are dual from one another. Second, notice that the rules (μ) and $(\tilde{\mu})$ form a critical pair, and that giving priority to the first reduction imposes a call-by-value reduction strategy, whereas the alternative results in a call-by-name reduction strategy. Note that the rules (proj_1) and (proj_2) also form a critical pair, convergent in the case of a call-by-name strategy but not so for call-by-value.

The definition of the notion of *linearity* on $\bar{\lambda}\mu\tilde{\mu}$ proof terms is a bit stronger than the usual definition of linearity: it implies that binder of linear variables do no overlap.

Definition 2.1.3 (Linearly bound variables)**.** In a proof term t, we say that a variable x is bound linearly by a binder b (either λ, μ or $\tilde{\mu}$) in a subterm u if:

- x is bound by b in u,
- x appears exactly once in u,
- between b and x, there is no occurrence of a binder of the same kind as b.

As highlighted in the introducing paragraph, typing statements for $\bar{\lambda}\mu\tilde{\mu}$ proof terms, *i.e.* sequents in the logical formalism, are of the three following forms:

$$\Gamma; e : A \vdash \Delta \qquad \Gamma \vdash v : A; \Delta \qquad c : (\Gamma \vdash \Delta)$$

where A is a formula of $\mathcal{L}^1_m$, and the contexts Γ and Δ are multisets of labeled formulas of $\mathcal{L}^1_m$. By using multisets instead of plain lists, formula-swapping rules are made implicit.

Definition 2.1.4. Figure 2.1 presents the logical system $\text{LK}_{\mu\tilde{\mu}}$ and a set of type inference rules for $\bar{\lambda}\mu\tilde{\mu}$.

$$\text{ax}_{\mathcal{L}}\ \frac{}{\Gamma; \alpha : A \vdash \alpha : A, \Delta} \qquad \text{ax}_{\mathcal{R}}\ \frac{}{\Gamma, x : A \vdash x : A; \Delta}$$

$$\bot_{\mathcal{L}}\ \frac{}{\Gamma; \rtimes : \bot \vdash \Delta} \qquad \top_{\mathcal{R}}\ \frac{}{\Gamma \vdash \ltimes : \top; \Delta}$$

$$\text{activate}_{\mathcal{L}}\ \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma; \tilde{\mu}x^A.c : A \vdash \Delta} \qquad \text{activate}_{\mathcal{R}}\ \frac{c : (\Gamma \vdash A, \Delta)}{\Gamma \vdash \mu\alpha^A.c : A; \Delta}$$

$$\text{cut}\ \frac{\Gamma \vdash v : A; \Delta \qquad \Gamma; e : A \vdash \Delta}{c : (\Gamma \vdash \Delta)}$$

$$\Rightarrow_{\mathcal{L}}\ \frac{\Gamma \vdash A; \Delta \qquad \Gamma; B \vdash \Delta}{\Gamma; A \Rightarrow B \vdash \Delta} \qquad \Rightarrow_{\mathcal{R}}\ \frac{\Gamma, x : A \vdash v : B; \Delta}{\Gamma \vdash \lambda x^A.v : A \Rightarrow B; \Delta}$$

$$\bigwedge_{\mathcal{L}}\ \frac{c : (\Gamma, x : A, y : B \vdash \Delta)}{\Gamma; \text{proj}[x, y, c] : A \wedge B \vdash \Delta} \qquad \bigwedge_{\mathcal{R}}\ \frac{\Gamma \vdash v : A; \Delta \qquad \Gamma \vdash v' : B; \Delta}{\Gamma \vdash (v, v') : A \wedge B; \Delta}$$

$$\bigvee_{\mathcal{L}}\ \frac{\Gamma; e : A \vdash \Delta \qquad \Gamma; e' : B \vdash \Delta}{\Gamma; [e, e'] : A \vee B \vdash \Delta}$$

$$\bigvee_{1\mathcal{R}}\ \frac{\Gamma \vdash v : A; \Delta}{\Gamma \vdash \text{inj}_l v : A \vee B; \Delta} \qquad \bigvee_{2\mathcal{R}}\ \frac{\Gamma \vdash v : B; \Delta}{\Gamma \vdash \text{inj}_r v : A \vee B; \Delta}$$

$$\neg_{\mathcal{L}}\ \frac{\Gamma \vdash v : A; \Delta}{\Gamma; \neg[v] : \neg A \vdash \Delta} \qquad \neg_{\mathcal{R}}\ \frac{\Gamma; e : A \vdash \Delta}{\Gamma \vdash \neg(e) : \neg A; \Delta}$$

$$\forall_{\mathcal{L}}\ \frac{\Gamma; e : B[x \leftarrow t] \vdash \Delta}{\Gamma; t \cdot e : \forall x^A.B \vdash \Delta} \qquad \forall_{\mathcal{R}}\ \frac{\Gamma \vdash v : B; \Delta}{\Gamma \vdash \lambda x^A.v : \forall x^A.B; \Delta}$$

$$\exists_{\mathcal{L}}\ \frac{\Gamma; e : B \vdash \Delta}{\Gamma; \lambda\alpha^A.e : \exists\alpha^A.B \vdash \Delta} \qquad \exists_{\mathcal{R}}\ \frac{\Gamma \vdash e : B[x \leftarrow t]; \Delta}{\Gamma \vdash t \cdot e : \exists x^A.B; \Delta}$$

Figure 2.1: $\text{LK}_{\mu\tilde{\mu}}$ as a type inference system for $\bar{\lambda}\mu\tilde{\mu}$

Note. In figure 2.1, the usual quantifier side conditions have been omitted for space : in $\forall_{\mathcal{R}}$ (resp. $\exists_{\mathcal{L}}$), the variable x (resp α) does not appear free in Γ or Δ; in $\forall_{\mathcal{L}}$ and $\exists_{\mathcal{R}}$, t and x have the same sort A. This leads us to another remark: an implicit conversion is done between sorts and formulas for the bound variables of the quantifier rules. Because of the trivial injection between ranking system and the formulas of first-order logic, this coercion works as expected.

Figure 1.2 does not include formula management rules, which are presented here. There are four weakening rules:

$$\text{weak}_{1\mathcal{R}}\ \frac{\Gamma \vdash v : C; \Delta}{\Gamma, x : A \vdash v : C; \Delta} \qquad \text{weak}_{2\mathcal{R}}\ \frac{\Gamma \vdash v : C; \Delta}{\Gamma \vdash v : C; x : A, \Delta}$$

$$\text{weak}_{3\mathcal{L}}\ \frac{\Gamma; e : C \vdash \Delta}{\Gamma, x : A; e : C \vdash \Delta} \qquad \text{weak}_{4\mathcal{L}}\ \frac{\Gamma; e : C \vdash \Delta}{\Gamma; e : C \vdash x : A, \Delta}$$

and the two contraction rules are derived using cut and axiom rules.

$$\text{contr}_{\mathcal{R}}\ \frac{\Gamma \vdash v : C; \alpha : C, \Delta}{\Gamma \vdash \mu\alpha^{C}.\langle v \| \alpha \rangle : C; \Delta} \qquad \text{contr}_{\mathcal{L}}\ \frac{\Gamma, x : C; e : C \vdash \Delta}{\Gamma; \tilde{\mu}x^{C}.\langle x \| e \rangle : C \vdash \Delta}$$

Note. This system does not have the cut-elimination property, because it makes use of the cut rule to encode contraction.

Remark that once a formalism for introducing proof terms in sequents is devised, the notion of deductive computation can be replaced by one of type inference of a proof term. Because the whole proof trace can be replaced by a proof term, the proof-as-term morphism offers a compact alternative to the representations of figures 1.3 and 1.4. For instance, the $\bar{\lambda}\mu\tilde{\mu}$ proof term:

$$\lambda x^{A}.\lambda y^{A \Rightarrow B}.\mu\alpha^{B}.\langle y \| x \cdot \alpha \rangle$$

when typechecked in the system $\text{LK}_{\mu\tilde{\mu}}$, builds the proof of the tautology $A \Rightarrow (A \Rightarrow B) \Rightarrow B$.

In the following sections we propose a series of original logical systems derived from $\text{LK}_{\mu\tilde{\mu}}$. We will see how they are adapted to interactive proof construction, and overall proof manipulation.

2.2 The classical system lk

Proof terms for our classical framework LK are expressed in a slight simplification of the $\bar{\lambda}\mu\tilde{\mu}$-calculus, arguably better-suited for interactive proof construction. In the rest of this manuscript, we use the name $\bar{\lambda}\mu\tilde{\mu}$ to designate this simplification. We give first the syntax of the calculus, before developing the reduction rules.

Definition 2.2.1 (Simpler $\bar{\lambda}\mu\tilde{\mu}$ proof terms)**.** The simplified syntax of the $\bar{\lambda}\mu\tilde{\mu}$-calculus defines commands c, terms v and environments e:

$$\begin{aligned} c &::= \langle v \| e \rangle \\ v, v' &::= x \mid \ltimes \mid \lambda x^A.v \mid (v, v') \mid \mathrm{inj}_r v \mid \mathrm{inj}_l v \mid \mu\alpha^A.c \\ e, e' &::= \alpha \mid \rtimes \mid v \cdot e \mid \mathrm{proj}[x, x', c] \mid [e, e'] \mid \tilde{\mu}x^A.c \end{aligned}$$

This syntax is similar to the one of definition 2.1.1, with the environment constructors λ and $\neg$ and the term constructors $\cdot$ and $\neg$ removed.

Definition 2.2.2 (Reduction rules for $\bar{\lambda}\mu\tilde{\mu}$)**.** Following the restriction in the syntax, the reduction rules $(\tilde{\lambda})$, $(\eta^R_{\tilde{\lambda}})$ and $(\eta^L_{\tilde{\lambda}})$ are discarded. All other rules from definition 2.1.2 are applicable to the restricted $\bar{\lambda}\mu\tilde{\mu}$ proof terms.

Typing statements for $\bar{\lambda}\mu\tilde{\mu}$ proof terms, *i.e.* sequents in the logical formalism, are of the two following forms:

$$\Gamma; e : A \vdash \Delta \qquad\qquad \Gamma \vdash v : A; \Delta$$

where the contexts Γ and Δ are multisets of labeled formulas — by using multisets instead of plain lists, formula-swapping rules are made implicit. Remark the absence of any judgements without active formula, *i.e.* of the form $c : (\Gamma \vdash \Delta)$.

Definition 2.2.3 (LK)**.** Figure 2.2 describes the logical inference rules for first-order classical sequent calculus. This system is called LK.

Definition 2.2.4. An classical proof term is an expression of $\bar{\lambda}\mu\tilde{\mu}$ that is well-typed in the system of figure 2.2.

A general remark on the LK logical system is that it retains as much information as possible: in particular, formulas are duplicated in order to forego destructive inferences, and delta expansion of connectors are recorded into the proof term structure:

- the contraction rule is embedded into much of the inferences: in bottom-up application of inferences, when branching is achieved, the contexts in both hypothesis and conclusion are duplicated. Hence $\vee_{\mathcal{L}}$ and $\wedge_{\mathcal{R}}$ are additive rules, while $\wedge_{\mathcal{L}}$ uses the inference for multiplicative conjunction. Incidentally, for the sake of symmetry, a cut rule is embedded in the $\wedge_{\mathcal{L}}$ rule, the alternatives (having two inference rules depending on which formula is focussed upon, or arbitrarily choosing A or B as the principal formula for the rule's hypothesis) being deemed unsatisfactory. Finally, the choice of an additive $\vee_{\mathcal{R}}$ rule is a minor deviation from the aforementioned principle, justified by the coherence with the intuitionistic fragment of this formalism, detailed later. Moreover, the right rule for a multiplicative disjunction is easily simulated using a cut.

$$\mathrm{ax}_{\mathcal{L}}\ \frac{}{\Gamma;\alpha : A \vdash \alpha : A,\Delta} \qquad \mathrm{ax}_{\mathcal{R}}\ \frac{}{\Gamma, x : A \vdash x : A;\Delta}$$

$$\bot_{\mathcal{L}}\ \frac{}{\Gamma;\rtimes : \bot \vdash \Delta} \qquad \top_{\mathcal{R}}\ \frac{}{\Gamma \vdash \ltimes : \top;\Delta}$$

$$\mathrm{cut}_{\mathcal{L}}\ \frac{\Gamma, x : A \vdash v : B;\Delta \qquad \Gamma, x : A; e : B \vdash \Delta}{\Gamma;\tilde{\mu}x^{A}.\langle v \| e\rangle : A \vdash \Delta}$$

$$\mathrm{cut}_{\mathcal{R}}\ \frac{\Gamma \vdash v : B;\alpha : A,\Delta \qquad \Gamma; e : B \vdash \alpha : A,\Delta}{\Gamma \vdash \mu\alpha^{A}.\langle v \| e\rangle : A;\Delta}$$

$$\Rightarrow_{\mathcal{L}}\ \frac{\Gamma \vdash v : A;\Delta \qquad \Gamma; e : B \vdash \Delta}{\Gamma; v \cdot e : A \Rightarrow B \vdash \Delta} \qquad \Rightarrow_{\mathcal{R}}\ \frac{\Gamma, x : A \vdash v : B;\Delta}{\Gamma \vdash \lambda x^{A}.v : A \Rightarrow B;\Delta}$$

$$\bigwedge_{\mathcal{L}}\ \frac{\Gamma, x : A, y : B \vdash v : C;\Delta \qquad \Gamma, x : A, y : B; e : C \vdash \Delta}{\Gamma; \mathrm{proj}[x, y, \langle v \| e\rangle] : A \wedge B \vdash \Delta}$$

$$\bigwedge_{\mathcal{R}}\ \frac{\Gamma \vdash v : A;\Delta \qquad \Gamma \vdash v' : B;\Delta}{\Gamma \vdash (v, v') : A \wedge B;\Delta}$$

$$\bigvee_{\mathcal{L}}\ \frac{\Gamma; e : A \vdash \Delta \qquad \Gamma; e' : B \vdash \Delta}{\Gamma; [e, e'] : A \vee B \vdash \Delta}$$

$$\bigvee_{1\mathcal{R}}\ \frac{\Gamma \vdash v : A;\Delta}{\Gamma \vdash \mathrm{inj}_{l} v : A \vee B;\Delta} \qquad \bigvee_{2\mathcal{R}}\ \frac{\Gamma \vdash v : B;\Delta}{\Gamma \vdash \mathrm{inj}_{r} v : A \vee B;\Delta}$$

$$\neg_{\mathcal{L}}\ \frac{\Gamma \vdash v : A;\Delta}{\Gamma;\tilde{\mu}x^{\neg A}.\langle x \| v \cdot \rtimes\rangle : \neg A \vdash \Delta}$$

$$\neg_{\mathcal{R}}\ \frac{\Gamma; e : A \vdash \Delta}{\Gamma \vdash \mu\alpha^{\neg A}.\langle \lambda y^{A}.\mu\beta^{\bot}.\langle y \| e\rangle \| \alpha\rangle : \neg A;\Delta}$$

$$\forall_{\mathcal{L}}\ \frac{\Gamma; e : B[x \leftarrow t] \vdash \Delta}{\Gamma; t \cdot e : \forall x^{A}.B \vdash \Delta} \qquad \forall_{\mathcal{R}}\ \frac{\Gamma \vdash v : B;\Delta}{\Gamma \vdash \lambda x^{A}.v : \forall x^{A}.B;\Delta}$$

$$\exists_{\mathcal{L}}\ \frac{\Gamma; e : B \vdash \Delta}{\Gamma; \mathrm{proj}[\alpha, \beta, \langle \beta \| e\rangle] : \exists\alpha^{A}.B \vdash \Delta} \qquad \exists_{\mathcal{R}}\ \frac{\Gamma \vdash v : B[x \leftarrow t];\Delta}{\Gamma \vdash (t, v) : \exists x^{A}.B;\Delta}$$

Figure 2.2: Classical inference rules labeled by $\bar{\lambda}\mu\tilde{\mu}$

- these choices have an impact on the quantifiers side. While the dependent product $\forall$ is unsurprisingly labeled by a λ construct, the dependent sum $\exists$ reuses the environment projection operator, instead of introducing a new notation as it is the case in $\text{LK}_{\mu\tilde{\mu}}$. This is made possible by the multiplicativity of the $\wedge_{\mathcal{L}}$ rule, which in turn produced a projection constructor rich enough to label the existential quantifier rule.
- (η_{μ}) and $(\eta_{\tilde{\mu}})$ expansions are used to explicitly store the type of a $\bar{\lambda}\mu\tilde{\mu}$ expression inside a η-redex. For instance, if v is a term of type A, one can record this type information using a variable α by performing the following (η_{μ}) expansion:

  $$v \to \mu\alpha^{A}.\langle v \| \alpha \rangle$$

  This is used in the case of the deduction rules for negation $\neg_{\mathcal{R}}$ and $\neg_{\mathcal{L}}$, to generate non-minimal proof terms that include information about the expansion of negation into an implication. A (η_{μ}) expansion is also performed at the beginning of each proof, in order to store in the proof term the formula that is being proven.

These characteristics make the system LK is well-suited for interactive theorem proving and proof interoperability, two topics that require proof structures as information-rich as possible.

In LK, since there are no typing judgements for standalone commands, they are typed only as subterms of a term or a environment. In short, this consists in a variation of $\text{LK}_{\mu\tilde{\mu}}$, inlining the 'cut' and 'activate' rules into any rule that produces a command. Thus the rules for $\text{cut}_{\mathcal{R}}$, $\text{cut}_{\mathcal{L}}$, $\wedge_{\mathcal{L}}$ and $\exists_{\mathcal{L}}$. A consequence of this design is the following proposition.

Proposition 2.2.5 (Liveliness)**.** *For any sequent in a* LK *proof, there is an active formula.*

Proof. By induction on the structure of the proof. □

The liveliness property is important to us for several reasons, related to the practice of interactive proof systems. First, it is essential to ensure that the user of such systems always knows which formula he is working on. Having sequents where there are distinguished formulas, and others where there are none, arguably adds a level of complexity to the (already involved) comprehension and intuition of a proof formalism. Second, in the case of automated or partly-automated theorem proving, it is conjectured that this property facilitates the design of algorithms, and reduces the search space: this intuition is also mentioned in (Sacerdoti Coen, 2006), and works by Andreoli (Andreoli, 1992) and more recently Saurin (Saurin, 2006) on focussing proofs tend to confirm this hypothesis.

The formula management rules for LK (weakening and contraction) are similar the corresponding rules of $\text{LK}_{\mu\tilde{\mu}}$.

Proposition 2.2.6. *The logical system* LK *is equivalent to the* $\text{LK}_{\mu\tilde{\mu}}$ *formulation of first-order classical sequent calculus, as per definition* 1.2.1*.*

Proof. The equivalence between $\mathcal{L}^1$ and $\mathcal{L}^1_m$, *i.e.* single-sorted and many-sorted first-order languages, is an established result. A many-sorted language can be encoded in a single-sorted language by introducing predicate symbols that represent sort membership. On the logical side, when considering unannotated formulas, the system LK is similar to $\text{LK}_{\mu\tilde{\mu}}$ except for the cut and left conjunction:

- the $\text{cut}_{\mathcal{L}}$ and $\text{cut}_{\mathcal{R}}$ are inlined versions of the usual focus and cut inference rules. The equivalence holds because in $\text{LK}_{\mu\tilde{\mu}}$, the only rule that can follow a focus on the left or on the right is a cut rule.
- the same reasoning holds for the left conjunction rule $\wedge_{\mathcal{L}}$. While embedding a focus rule would have been sufficient, the use of an additional cut rule is not detrimental to the equivalence of both systems, as choosing $C = A$ or $C = B$ and using the weakening rules proves.

The labelling of the proofs by proof terms is slightly different in LK and in $\text{LK}_{\mu\tilde{\mu}}$. It is easy, by using typing information in some cases, to establish a translation between them. In particular, the two symbols $\neg$ and the left λ in $\text{LK}_{\mu\tilde{\mu}}$ are equivalent to constructions using $\cdot$, λ and proj in LK. □

Proposition 2.2.7 (Consistency)**.** *The* LK *logical framework is consistent.*

Proof. This follows from proposition 2.2.6, *i.e.* equivalence with Herbelin's $\text{LK}_{\mu\tilde{\mu}}$ sequent calculus. □

2.3 The intuitionistic system lj

Restricting the framework of classical logic to an intuitionistic formalism is a simple process: it suffices to constrain the conclusion of sequents to contain at most one formula. This constraint has a clearly identifiable counterpart at the proof term level. Indeed, when introducing a μ-abstraction to name the formula one wants to prove next, the previously designated formula and its label are overridden, due to the one-formula-per-consequent constraint. As a consequence, only one environment variable, denoted $*$, is ever used, and the μ-bindings do not overlap: the environment variables for intuitionistic proof terms are linearly bound.

Definition 2.3.1 ($\bar{\lambda}^{*}\tilde{\mu}\tilde{\mu}$ proof terms)**.** The $\bar{\lambda}^{*}\tilde{\mu}\tilde{\mu}$ terms are similar to $\bar{\lambda}\mu\tilde{\mu}$ terms, albeit with a unique environment variable $*$.

$$
\begin{aligned}
c &::= \langle v \| e \rangle \\
v, v' &::= x \mid \ltimes \mid \lambda x^{A}.v \mid (v, v') \mid \text{inj}_r v \mid \text{inj}_l v \mid \mu *^{A}.c \\
e, e' &::= * \mid \rtimes \mid v \cdot e \mid \text{proj}[x, x', c] \mid [e, e'] \mid \tilde{\mu} x^{A}.c
\end{aligned}
$$

The reduction rules given in definition 2.2.2, with the relevant cases pruned, still hold. Also, the form of the typing statements for proof terms isn't changed from section 2.2.

The one formula limitation entails the deprecation of the right contraction rule $\text{contr}_{\mathcal{R}}$ and the right weakening rule $\text{weak}_{2\mathcal{R}}$. Only remain the following weakening and contraction rules:

$$\text{weak}_{1\mathcal{R}}\ \frac{\Gamma \vdash v : C}{\Gamma, x : A \vdash v : C} \qquad \text{contr}_{\mathcal{L}}\ \frac{\Gamma, x : C; e : C \vdash y : D}{\Gamma; \tilde{\mu}x^{C}.\langle x \| e\rangle : C \vdash y : D}$$

$$\text{weak}_{3\mathcal{L}}\ \frac{\Gamma; e : C \vdash y : D}{\Gamma, x : A; e : C \vdash y : D} \qquad \text{weak}_{4\mathcal{L}}\ \frac{\Gamma; e : C \vdash}{\Gamma; e : C \vdash x : A}$$

Definition 2.3.2 (LJ)**.** Figure 2.3 exposes the inference rules for first-order intuitionistic sequent calculus.

Definition 2.3.3. An intuitionistic proof term is an expression of $\bar{\lambda}^{*}\tilde{\mu}\tilde{\mu}$ that is well-typed in the system of figure 2.3.

The intuitionistic formalism being a weakened version of the classical setting, it inherits its liveliness property (proposition 2.2.5).

Proposition 2.3.4 (Liveliness)**.** *For any sequent in a* LJ *proof, there is an active formula.*

Equivalence with the intuitionnistic restriction of $\text{LK}_{\mu\tilde{\mu}}$ can be derived similarly to what is done in LK, entailing consistency.

Proposition 2.3.5 (Consistency)**.** *The* LJ *logical framework is consistent.*

In the rest of this section, we give a few propositions linking the intuitionistic and classical frameworks.

Classical $\bar{\lambda}\mu\tilde{\mu}$ proof terms can be characterized (Sacerdoti Coen, 2006) as expressions that contain *non-linear* environment variables, *i.e.* variables that are not bound by the innermost enclosing μ binder. For instance, the rule (η^{L}_{λ}) is not compatible with the intuitionnistic restriction, because in the right-hand side of the equivalence, the variable α is not linearly bound. Conversely, $\bar{\lambda}\mu\tilde{\mu}$ expressions where environment variables only appear linearly are isomorphic to $\bar{\lambda}^{*}\tilde{\mu}\tilde{\mu}$ expressions, thus intuitionistic. Hence the following proposition:

Proposition 2.3.6. *Any intuitionistic derivation in* LK *is also a valid proof in* LJ*.*

Proof. The translation between $\bar{\lambda}^{*}\tilde{\mu}\tilde{\mu}$ and $\bar{\lambda}\mu\tilde{\mu}$ proof terms is the identity function (modulo renaming of environment variables to $*$). □

$$\mathrm{ax}_{\mathcal{L}}\ \frac{}{\Gamma; * : A \vdash * : A} \qquad \mathrm{ax}_{\mathcal{R}}\ \frac{}{\Gamma, x : A \vdash x : A}$$

$$\bot_{\mathcal{L}}\ \frac{}{\Gamma; \rtimes : \bot \vdash A} \qquad \top_{\mathcal{R}}\ \frac{}{\Gamma \vdash \ltimes : \top}$$

$$\mathrm{cut}_{\mathcal{L}}\ \frac{\Gamma, x : A \vdash v : B \qquad \Gamma, x : A; e : B \vdash C}{\Gamma; \tilde{\mu} x^{A}.\langle v \| e \rangle : A \vdash C}$$

$$\mathrm{cut}_{\mathcal{R}}\ \frac{\Gamma \vdash v : B \qquad \Gamma; e : B \vdash * : A}{\Gamma \vdash \mu *^{A}.\langle v \| e \rangle : A}$$

$$\Rightarrow_{\mathcal{L}}\ \frac{\Gamma \vdash v : A \qquad \Gamma; e : B \vdash C}{\Gamma; v \cdot e : A \Rightarrow B \vdash C} \qquad \Rightarrow_{\mathcal{R}}\ \frac{\Gamma, x : A \vdash v : B}{\Gamma \vdash \lambda x^{A}.v : A \Rightarrow B}$$

$$\bigwedge_{\mathcal{L}}\ \frac{\Gamma, x : A, y : B \vdash v : C \qquad \Gamma, x : A, y : B; e : C \vdash D}{\Gamma; \mathrm{proj}[x, y, \langle v \| e \rangle] : A \wedge B \vdash D}$$

$$\bigwedge_{\mathcal{R}}\ \frac{\Gamma \vdash v : A \qquad \Gamma \vdash v' : B}{\Gamma \vdash (v, v') : A \wedge B}$$

$$\bigvee_{\mathcal{L}}\ \frac{\Gamma; e : A \vdash C \qquad \Gamma; e' : B \vdash C}{\Gamma; [e, e'] : A \vee B \vdash C}$$

$$\bigvee_{1\mathcal{R}}\ \frac{\Gamma \vdash v : A}{\Gamma \vdash \mathrm{inj}_{l} v : A \vee B} \qquad \bigvee_{2\mathcal{R}}\ \frac{\Gamma \vdash v : B}{\Gamma \vdash \mathrm{inj}_{r} v : A \vee B}$$

$$\neg_{\mathcal{L}}\ \frac{\Gamma \vdash v : A}{\Gamma; \tilde{\mu} x^{\neg A}.\langle x \| v \cdot \rtimes \rangle : \neg A \vdash B}$$

$$\neg_{\mathcal{R}}\ \frac{\Gamma; e : A \vdash \rtimes : \bot}{\Gamma \vdash \mu *^{\neg A}.\langle \lambda y^{A}.\mu *^{\bot}.\langle y \| e \rangle \| * \rangle : \neg A}$$

$$\forall_{\mathcal{L}}\ \frac{\Gamma; e : B[x \leftarrow t] \vdash C}{\Gamma; t \cdot e : \forall x^{A}.B \vdash C} \qquad \forall_{\mathcal{R}}\ \frac{\Gamma \vdash v : B}{\Gamma \vdash \lambda x^{A}.v : \forall x^{A}.B}$$

$$\exists_{\mathcal{L}}\ \frac{\Gamma; e : B \vdash C}{\Gamma; \mathrm{proj}[*, *, \langle \beta \| e \rangle] : \exists *^{A}.B \vdash C} \qquad \exists_{\mathcal{R}}\ \frac{\Gamma \vdash v : B[x \leftarrow t]}{\Gamma \vdash (t, v) : \exists x^{A}.B}$$

Figure 2.3: Intuitionistic inference rules labeled by $\bar{\lambda}\tilde{\mu}^{*}\tilde{\mu}$

In the following we extend the previous result, and we show that a classical proof term can be expressed as a proof term of the system LJ+EM, where environment variables are linear.

Definition 2.3.7 (LJ+EM)**.** The logical system LJ+EM is defined as the theory using the inference rules of figure 2.3, extended as follows:

$$\text{excluded middle } \frac{}{\Gamma \vdash \mathbf{em}_{\mathsf{A}} : \mathsf{A} \vee \neg \mathsf{A}}$$

Definition 2.3.8 ($\mathcal{F}$-translation)**.** We define by structural induction the translation function of classical $\bar{\lambda}\mu\tilde{\mu}$-terms to intuitionistic $\bar{\lambda}\mu\tilde{\mu}$-terms enriched with the aforementioned set of constants, noted $\mathcal{F}_{\cdot}(\cdot)$:

$$\mathcal{F}_{\sigma}(\mu\alpha^{\mathsf{A}}.\mathsf{c}) \to \begin{cases} \mu\alpha^{\mathsf{A}}.\mathcal{F}_{\sigma}(\mathsf{c}) & \text{if } \alpha \text{ appears linearly in } \mathsf{c}, \\ \mu\beta^{\mathsf{A}}.\langle \mathbf{em}_{\mathsf{A}} \| [\tilde{\mu}\mathsf{x}^{\mathsf{A}}.\langle \mathsf{x} \| \beta \rangle, \tilde{\mu}\mathsf{a}^{\neg \mathsf{A}}.\mathcal{F}_{(\alpha,\mathsf{a},\mathsf{A})::\sigma}(\mathsf{c})] \rangle & \text{else.} \end{cases} \tag{2.1}$$

$$\mathcal{F}_{\sigma}(\alpha) \to \begin{cases} \tilde{\mu}\mathsf{x}^{\neg \mathsf{A}}.\langle \mathsf{a} \| \mathsf{x} \cdot \rtimes \rangle & \text{if } (\alpha, \mathsf{a}, \mathsf{A}) \in \sigma, \\ \alpha & \text{else.} \end{cases} \tag{2.2}$$

Where σ is a list of triplets containing an environment variable, a term variable and a formula. Note the $\eta_{\tilde{\mu}}$-expansion of β in the second branch of (2.1), used to record type information about the formula A. For the other cases, the translation function is non-destructively applied to subterms of the considered expression.

Proposition 2.3.9. *The result of the $\mathcal{F}$-translation of a well-typed $\bar{\lambda}\mu\tilde{\mu}$ proof term is a well-typed $\bar{\lambda}^{*}\tilde{\mu}\tilde{\mu}$ proof term, modulo renaming of α and β into $*$ and introduction of the* **em** *constant.*

Proof. The intuition is that $\mathcal{F}_{\cdot}(\cdot)$ replaces the non-linear environment variable abstractions by linear ones, and possibly non-linear term variable abstractions. This can be easily proved by induction on the second argument of $\mathcal{F}_{\cdot}(\cdot)$.

The first case is when a variable α appears non-linearly in the body of a μ-abstraction $\mu\alpha^{\mathsf{A}}.\mathsf{c}$ (second branch of (2.1)). In this case, $\mathcal{F}_{\cdot}(\cdot)$ translates the term into $\mu\beta^{\mathsf{A}}.\langle \mathbf{em}_{\mathsf{A}} \| [\tilde{\mu}\mathsf{x}^{\mathsf{A}}.\langle \mathsf{x} \| \beta \rangle, \tilde{\mu}\mathsf{a}^{\neg \mathsf{A}}.\mathcal{F}_{(\alpha,\mathsf{a},\mathsf{A})::\sigma}(\mathsf{c})] \rangle$, which is typable in LJ+EM since, by induction hypothesis, $\mathcal{F}_{(\alpha,\mathsf{a},\mathsf{A})::\sigma}(\mathsf{c})$ is.

The second case is when a non-linear variable is reached: by (2.2) it is translated into a $\tilde{\mu}$-abstraction, typable in LJ+EM. All the other cases are dealt with by trivial induction hypothesis.

Since the resulting terms are linear, then the environment variable namespace can be collapsed to $*$. □

Corollary 2.3.10 (Correctness)**.** *For any classical proof π_{LK} of a formula A, $\pi_{\text{LJ+EM}} = \mathcal{F}_{nil}(\pi_{\text{LK}})$ is a proof of A in* LJ+EM*.*

Proposition 2.3.11 (Completeness)**.** *For any proof term $\pi_{\text{LJ+EM}}$ typable in* LJ+EM*, there exist a classical proof term π_{LK} such that $\pi_{\text{LJ+EM}} = \mathcal{F}_{nil}(\pi_{\text{LK}})$.*

Proof. Since $\pi_{\text{LJ+EM}}$ has got only linear environment variables, then its identity translation into $\bar{\lambda}\mu\tilde{\mu}$ will trivially verify: $\pi_{\text{LJ+EM}} = \mathcal{F}_{nil}(\pi_{\text{LK}})$. □

2.4 The minimal systems lkm and ljm

The framework of minimal logic was investigated by Johansson (Johansson, 1937) in an attempt to minimize the logical content of the implication symbol. In that paper, the result was a logic stricter than Heyting's intuitionistic calculus, where the $\neg\mathfrak{a}$ formula was considered as a macro for $\mathfrak{a} \Rightarrow \curlywedge$, with $\curlywedge$ being an unassuming predicate variable. Note that this is different from what some authors call minimal logic, that is a logic with only the $\Rightarrow$ and $\forall$ connectives (in this manuscript, we call this last definition *minimalistic* logic).

The concept has since then been generalized, for instance to classical frameworks. However the principle remains the same, and can be reformulated as: minimal logic is a framework in which negation is a notation for an implication of the symbol $\bot$, which has no logical content. As a consequence, in minimal frameworks there is no deduction rule for $\bot$, and negation is systematically introduced as an implication.

Because intuitionistic minimal logic is the direct Curry-de Bruijn-Howard counterpart of simply typed lambda-calculus, this formalism has drawn some attention in the recent decades. In particular, this feature has favored the development of the first extraction mechanisms (Hayashi and Nakano, 1988). More recently, the MINLOG theorem prover implements both classical and intuitionistic minimal logic, with a strong emphasis on program extraction (Schwichtenberg, 1993).

The figures 2.4 and 2.5 summarize the inference rules of respectively classical and intuitionistic first-order minimal sequent calculus, which will be referred to as, respectively, LKM and LJM. The weakening and contraction rules are identical to their non-minimal counterparts.

It is easy to prove that the minimal restrictions LKM and LJM of LK and LJ are themselves consistent logical frameworks.

2.5 Incomplete proof representation

Just as the proof representations of chapter 1, the proof-as-terms paradigm can be adapted to describe incomplete proofs. Indeed, a common approach (Magnusson, 1995; Muñoz, 2001a; Jojgov, 2003a; Lengrand, 2006) is to add *metavariables* to the syntax of proof terms, *i.e.* placeholders for the terms whose type will constitute the incomplete branches of the proof. In the case of first-order logic, this extension is fairly simple, simply requiring the addition of metavariables to the syntax (in higher-order systems dealing with dependent types and polymorphism, extensions need to be made to

$$\mathrm{ax}_{\mathcal{L}}\ \frac{}{\Gamma;\alpha : A \vdash \alpha : A,\Delta} \qquad \mathrm{ax}_{\mathcal{R}}\ \frac{}{\Gamma, x : A \vdash x : A;\Delta}$$

$$\top_{\mathcal{R}}\ \frac{}{\Gamma \vdash \aleph : \top;\Delta}$$

$$\mathrm{cut}_{\mathcal{L}}\ \frac{\Gamma, x : A \vdash v : B;\Delta \qquad \Gamma, x : A; e : B \vdash \Delta}{\Gamma;\tilde{\mu}x^{A}.\langle v \| e\rangle : A \vdash \Delta}$$

$$\mathrm{cut}_{\mathcal{R}}\ \frac{\Gamma \vdash v : B;\alpha : A,\Delta \qquad \Gamma; e : B \vdash \alpha : A,\Delta}{\Gamma \vdash \mu\alpha^{A}.\langle v \| e\rangle : A;\Delta}$$

$$\Rightarrow_{\mathcal{L}}\ \frac{\Gamma \vdash v : A;\Delta \qquad \Gamma; e : B \vdash \Delta}{\Gamma; v \cdot e : A \Rightarrow B \vdash \Delta} \qquad \Rightarrow_{\mathcal{R}}\ \frac{\Gamma, x : A \vdash v : B;\Delta}{\Gamma \vdash \lambda x^{A}.v : A \Rightarrow B;\Delta}$$

$$\bigwedge_{\mathcal{L}}\ \frac{\Gamma, x : A, y : B \vdash v : C;\Delta \qquad \Gamma, x : A, y : B; e : C \vdash \Delta}{\Gamma;\mathrm{proj}[x, y, \langle v \| e\rangle] : A \wedge B \vdash \Delta}$$

$$\bigwedge_{\mathcal{R}}\ \frac{\Gamma \vdash v : A;\Delta \qquad \Gamma \vdash v' : B;\Delta}{\Gamma \vdash (v, v') : A \wedge B;\Delta}$$

$$\bigvee_{\mathcal{L}}\ \frac{\Gamma; e : A \vdash \Delta \qquad \Gamma; e' : B \vdash \Delta}{\Gamma; [e, e'] : A \vee B \vdash \Delta}$$

$$\bigvee_{1\mathcal{R}}\ \frac{\Gamma \vdash v : A;\Delta}{\Gamma \vdash \mathrm{inj}_{l} v : A \vee B;\Delta} \qquad \bigvee_{2\mathcal{R}}\ \frac{\Gamma \vdash v : B;\Delta}{\Gamma \vdash \mathrm{inj}_{r} v : A \vee B;\Delta}$$

$$\neg_{\mathcal{L}}\ \frac{\Gamma \vdash v : A;\Delta \qquad \Gamma, \aleph : \bot \vdash \Delta}{\Gamma; v \cdot \aleph : \neg A \vdash \Delta} \qquad \neg_{\mathcal{R}}\ \frac{\Gamma; x : A \vdash \aleph : \bot,\Delta}{\Gamma \vdash \lambda x^{A}.\aleph : \neg A;\Delta}$$

$$\forall_{\mathcal{L}}\ \frac{\Gamma; e : B[x \leftarrow t] \vdash \Delta}{\Gamma; t \cdot e : \forall x^{A}.B \vdash \Delta} \qquad \forall_{\mathcal{R}}\ \frac{\Gamma \vdash v : B;\Delta}{\Gamma \vdash \lambda x^{A}.v : \forall x^{A}.B;\Delta}$$

$$\exists_{\mathcal{L}}\ \frac{\Gamma; e : B \vdash \Delta}{\Gamma;\mathrm{proj}[\alpha, \beta, \langle \beta \| e\rangle] : \exists\alpha^{A}.B \vdash \Delta} \qquad \exists_{\mathcal{R}}\ \frac{\Gamma \vdash v : B[x \leftarrow t];\Delta}{\Gamma \vdash (t, v) : \exists x^{A}.B;\Delta}$$

Figure 2.4: Classical minimal inference rules labeled by $\bar{\lambda}\mu\tilde{\mu}$

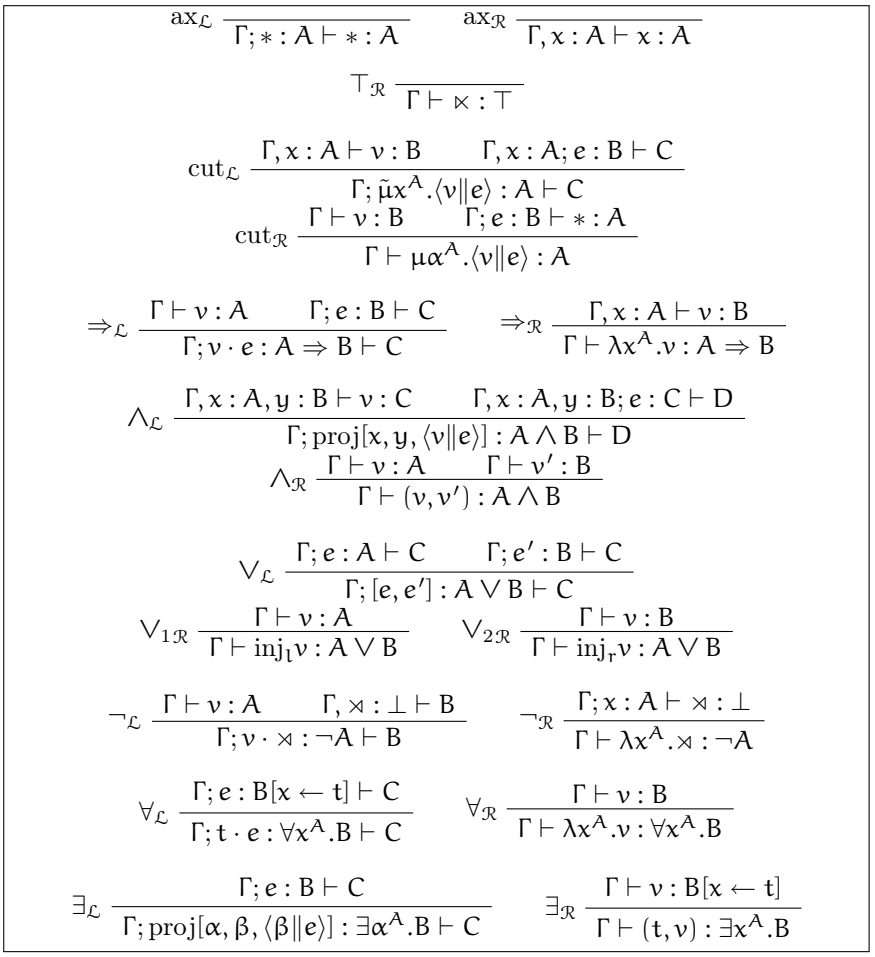


Figure 2.5: Intuitionistic minimal inference rules labeled by $\bar{\lambda}\mu\tilde{\mu}$

the structure of sequents and inference rules to include constraints and their resolution).

Hence term and environment metavariables, which we note with grayed capital letters, are added to the definition of proof terms:

$$v, v' ::= \ldots \mid X$$
$$e, e' ::= \ldots \mid \Xi$$

Proof terms that contain metavariables are called *open terms*, those that do not are called *closed terms*.

Notation. We note $t[X_1, \ldots, X_n]$ to explicitly state that the proof term t contains the metavariables $X_1, \ldots, X_n$ in a pre-defined order: innermost, leftmost to outermost, rightmost.

Definition 2.5.1 (Types for metavariables)**.** In order to be able to type terms containing metavariables, we need to add a new context Υ for metavariables in typing statements, called a *metavariable signature*. The form of the typing judgements is modified to take this signature into account:

$$\Upsilon \therefore \Gamma; e : A \vdash \Delta \qquad\qquad \Upsilon \therefore \Gamma \vdash v : A; \Delta$$

All the type inference rules are extended to this new form, and the following two rules are added:

$$\mathrm{mv}_{\mathcal{R}}\ \frac{}{\Upsilon, X_0 : A \therefore \Gamma \vdash X_0 : A} \qquad \mathrm{mv}_{\mathcal{L}}\ \frac{}{\Upsilon, \Xi_0 : A \therefore \Gamma; \Xi_0 : A \vdash B}$$

In the works that have pioneered their usage, metavariables were used to write proof enumeration algorithms; however here they only serve the purpose of representing incomplete proofs. For this representative usage, there is only a need for a simple explicit substitution mechanism of metavariables, called *grafting*.

Definition 2.5.2 (Grafting on proof terms)**.** Given two proof terms t and u, a grafting of the metavariable X of t by u, denoted $t[X \leftarrow u]$, is the term t where all instances of the metavariable X are *syntactically* replaced by the term u, provided X and u have the same type.

Definition 2.5.3 (Grafting on signatures)**.** Given a metavariable signature $\Upsilon = \Upsilon_1, X : A$, the grafting of the metavariable X in Υ by a proof term t, denoted $\Upsilon[X \leftarrow_{\Upsilon_2} t]$, is defined as the signature Υ_1, Υ_2, provided that:

- the metavariables declared in Υ_2 do not appear in Υ,
- t and X have the same type.

For a more detailed description of the explicit substitution required to deal with metavariable instantiation in dependently typed systems, refer to (Muñoz, 2001b). Note that in the rest of this work, when unambiguous we will leave the metavariable signature implicit, and the grafting operation at the meta-level.

Example 2.5.4. We reuse an example from (Muñoz, 1997, p. 71), *i.e.* the proof in LK of the formula $A \Rightarrow ((A \Rightarrow B) \Rightarrow B)$ in the context $\Gamma = A, B : \mathtt{bool}$. Finding such a proof is merely finding a grafting of a metavariable X_0 such that:

$$\Gamma \vdash X_0 : A \Rightarrow ((A \Rightarrow B) \Rightarrow B)$$

Grafting X_0 with the term $\lambda x^A.\lambda y^{A\Rightarrow B}.X_1$ allows the derivation to progress up to the typing judgement of the new metavariable X_1:

$$2\times\Rightarrow_{\mathcal{R}} \frac{\Gamma, x : A, y : A \Rightarrow B \vdash X_1 : B}{\Gamma \vdash \lambda x^A.\lambda y^{A\Rightarrow B}.X_1 : A \Rightarrow ((A \Rightarrow B) \Rightarrow B)}$$

Now X_1 can be grafted with the μ-abstraction $\mu z^B.\langle y \| \Xi_0 \rangle$, which leads to the derivation:

$$2\times\Rightarrow_{\mathcal{R}} \frac{\mathrm{cut}_{\mathcal{R}} \dfrac{\mathrm{ax}_{\mathcal{R}} \dfrac{}{\Gamma' \vdash y : A \Rightarrow B; z : B} \qquad \Gamma'; \Xi_0 : A \Rightarrow B \vdash z : B}{\Gamma' \vdash \mu z^B.\langle y \| \Xi_0 \rangle : B}}{\Gamma \vdash \lambda x^A.\lambda y^{A\Rightarrow B}.\mu z^B.\langle y \| \Xi_0 \rangle : A \Rightarrow ((A \Rightarrow B) \Rightarrow B)}$$

where $\Gamma' = \Gamma, x : A, y : A \Rightarrow B$. Finally the metavariable Ξ_0 can be grafted with the application $x \cdot z$, which completes the proof:

$$2\times\Rightarrow_{\mathcal{R}} \frac{\mathrm{cut}_{\mathcal{R}} \dfrac{\mathrm{ax}_{\mathcal{R}} \dfrac{}{\Gamma' \vdash y : A \Rightarrow B; z : B} \qquad \Rightarrow_{\mathcal{L}} \dfrac{\mathrm{ax}_{\mathcal{R}} \dfrac{}{\Gamma' \vdash x : A; z : B} \qquad \mathrm{ax}_{\mathcal{L}} \dfrac{}{\Gamma'; z : B \vdash z : B}}{\Gamma'; x \cdot z : A \Rightarrow B \vdash z : B}}{\Gamma' \vdash \mu z^B.\langle y \| x \cdot z \rangle : B}}{\Gamma \vdash \lambda x^A.\lambda y^{A\Rightarrow B}.\mu z^B.\langle y \| x \cdot z \rangle : A \Rightarrow ((A \Rightarrow B) \Rightarrow B)}$$

Remark that the proof of example 2.5.4 is fully described by the following three graftings:

$$[X_0 \leftarrow_{X_1 : B} \lambda x^A.\lambda y^{A\Rightarrow B}.X_1]$$
$$[X_1 \leftarrow_{\Xi_0 : A\Rightarrow B} \mu z^B.\langle y \| \Xi_0 \rangle]$$
$$[\Xi_0 \leftarrow x \cdot z]$$

Hence another representation of proofs can be used: as a series of metavariable graftings.

3 State-based Semantics

With the logical basis of our work set, we turn our attention to the formalization of the concept of proof languages. *We see how the exploration of semantical frameworks for imperative languages can help establish a basis for the semantics of proof languages.*

Procedural proof languages bear a close resemblance to imperative languages, in that they both provide means to modify a persistent object: a proof state, or a memory state. In fact, just as developing a proof consists in exercising a proof language to guide deductive computations, imperative programming similarly consists in performing computations by using side effects on a memory state. Hence the idea, which is the premiss of this chapter, of using the familiar framework of imperative programming as an entry point to the novel and complex problem of the semantics of proof languages.

An imperative language is usually given an operational semantics *à la* Plotkin, focusing on the values of the program and integrating the concept of memory states and side effects through an explicit extension of the syntax of the formalism. Yet in the case of proof languages, we do not care for the values of a script, but only for the side effects it generates on the proof state. Thus in this chapter we show how an alternative, memory state-centered operational semantics can be designed to deal specifically with imperative languages and their extensions. We prove its equivalence with a well-known big-step formalism, and we finish by adapting this framework to proof languages, and discussing its strengths and weaknesses.

$\star$

Small-step operational semantics (Plotkin, 1981) expresses the semantics of a programming language through the use of reduction rules such as:

$$e \longrightarrow e'$$

where e and e' are *expressions* of the language. The length of the step implemented by each reduction is arbitrarily defined and the reductions are repeated until a normal form, the result or *value* of the program, is reached. This way of expressing the semantics allows for a detailed description of the evaluation process; indeed small-step operational semantics have been used in many occasions to prove a number of properties of programming languages (Wright and Felleisen, 1994; Dubois, 2000; Dubois and Boite, 2001; Nipkow, 2003)

Unlike functional languages that carry out computations *via* reduction of expressions, imperative languages heavily rely on side effects, in other words the alteration of a *memory state* by *instructions*, to achieve some computation. Reflecting this particularity usually requires a few changes to the aforementioned semantical formalism. For example, the small-step

operational semantics of the imperative fragment of the programming languages Ocaml (Pottier and Rémy, 2005) or C++ (Wasserrab et al., 2006) uses reduction rules over pairs, also called *configurations*. These pairs are constituted by a program $\mathtt{i}$, made of instructions of the language, and a memory state σ:

$$\langle \mathtt{i}, \sigma \rangle \longrightarrow \langle \mathtt{i}', \sigma' \rangle$$

However a number of problems arise from the introduction of pairs as objects of the reductions. Since imperative programs no longer compute a value, the normal forms of the programs must be adapted. Also, subterms of a pair are not pairs themselves, thus making the expression of a congruence rule quite ticklish.

Besides, any imperative program can be translated to a functional one. This *functional translation* can serve, for example, to prove properties of imperative programs (Filliâtre, 1998).

But as we shall see, the process of considering a program as a function can also spawn the design of a formalism slightly different from the one presented above, for which the central element is the memory state rather than the program values. Although the resulting formalism has some similarities with the one over pairs, it is simpler in several respects: for instance, the definition of normal forms no longer depends on a special value such as `skip`, and a simple congruence rule replaces the more sophisticated context rules. These properties make this formalism well-suited for expressing the semantics of an imperative language, as a straightforward rewrite system. It also proves interesting when studying purely imperative languages, such as proof languages (Delahaye, 2000; Jojgov, 2003b; Martin and Gibbons, 2002), that do not have a notion of returned value. Finally, since the result of an evaluation is a memory state, it is close to the spirit of Winskel's big-step semantics (Winskel, 1993).

Thus in this chapter, after exposing the syntax of the small imperative language IMP, we recall its usual small-step semantics and we review the principle of functionalization of imperative languages. Based on these systems we expose the store-based operational semantics of IMP, a slight deviation of the operational semantics better suited to accommodate the particularities of imperative programming. We then study the evolution of these formalisms when the language evolves to allow exceptions or to consider instructions as a subset of expressions. Finally, after having examined related work, we investigate how this approach can be applied to the semantics of proof languages.

3.1 Syntax and notations

The syntax of the imperative language IMP(Winskel, 1993) can be divided into three parts defining arithmetic expressions $\mathtt{a}$, boolean expressions $\mathtt{b}$ and

commands (or instructions) c:

$$a ::= n \mid X \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2$$
$$b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leqslant a_1 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2$$
$$c ::= \texttt{skip} \mid X := a \mid c_1 ; c_2 \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } b \texttt{ do } c$$

X ranges over a finite set whose elements are called *references*, and n is an integer. The letter e refers to the union of arithmetic and boolean expressions, and v denotes values:

$$e ::= a \mid b$$
$$v ::= n \mid \texttt{true} \mid \texttt{false}$$

The letters σ and σ' denote memory states (also called *stores*). We write $\sigma\{X \mapsto n\}$ for the extension of the memory state σ that associates v to X, *i.e.* for any reference Y,

$$\sigma\{X \mapsto v\}(Y) = \begin{cases} v & \text{if } Y = X \\ \sigma(Y) & \text{else} \end{cases}$$

When needed a store will be noted in extension : $X_1 \mapsto v_1, \ldots, X_n \mapsto v_n$ is the store such that $\forall i \in [1 \ldots n], \sigma(X_i) = v_i$.

3.2 Small-step operational semantics

Formalism

A first definition of the semantics of IMP is derived from the operational semantics of functional languages, where reduction rules read $e \to e'$. When dealing with a imperative languages one must also take into account side effects, which is done by adding stores to terms as objects of the rewrite rules. By definition, in a pair $\langle \text{instruction}, \text{store} \rangle$, addresses defined in the store are bound in the instruction, and addresses are manipulated modulo alpha-conversion.

Thus in a formalism that takes into account side-effects, a reduction rule reads:

$$\langle c, \sigma \rangle \to \langle c', \sigma' \rangle$$

This kind of rules is combined with a notion of context in order to deal with in-depth reduction of instruction. For instance, to design a left-to-right reduction strategy, then contexts are defined as:

$$C ::= [\,] \mid C; c$$

where [] represents the usual notion of *hole*, and C[c] is the context C in which the hole [] is replaced by the instruction c. The congruence rule follows:

$$\frac{\langle c, \sigma\rangle \to \langle c', \sigma'\rangle}{\langle C[c], \sigma\rangle \to \langle C[c'], \sigma'\rangle}$$

In addition, another notion of pairing $(\!|\text{expression}, \text{store}|\!)$ is introduced to allow the computation of arithmetic and boolean expressions in a given memory state. In order to allow the reduction of expressions within instructions, we define the congruence rule:

$$\frac{(\!|e, \sigma|\!) \to e'}{\langle C[e], \sigma\rangle \to \langle C[e'], \sigma\rangle}$$

and we enrich the context C:

$$C ::= [] \mid C; c \mid X := C' \mid \texttt{if } C' \texttt{ then } c_1 \texttt{ else } c_2$$

where C′ is a dedicated context to reduce expressions:

$$\begin{aligned} C' ::= {} & [] \mid C' + a \mid v + C' \mid C' - a \mid v - C' \mid C' \times a \mid v \times C' \\ & \mid C' = a \mid v = C' \mid C' \leqslant a \mid v \leqslant C' \mid C' \wedge a \mid v \wedge C' \mid C' \vee a \mid v \vee C'a \end{aligned}$$

These context rules defines a left-to-right reduction strategy. What is more, by separating context from evaluation rules, the distinction between the specification of elementary reductions and of the evaluation strategy is made explicit.

Evaluation rules

The evaluation rules for expressions are not thoroughly exposed here: they are quite straightforward. For example, the reduction rules for the sum of integers 4 and 3 or for the boolean conjunction of `true` and `false` are:

$$(\!|4 + 3, \sigma|\!) \to 7$$
$$(\!|\texttt{true} \wedge \texttt{false}, \sigma|\!) \to \texttt{false}$$

and access to a store reads:

$$(\!|X, \sigma|\!) \to \sigma(X)$$

The following rules describe the semantics of the instructions of IMP. We start with store affectation, written:

$$\langle X := n, \sigma\rangle \to \langle \texttt{skip}, \sigma\{X \mapsto n\}\rangle \qquad \text{(i)}$$

The instructions for sequence and conditional are evaluated by:

$$\langle \texttt{skip}; c, \sigma\rangle \to \langle c, \sigma\rangle \tag{ii}$$

$$\langle \texttt{if true then } c_1 \texttt{ else } c_2, \sigma\rangle \to \langle c_1, \sigma\rangle \tag{iii}$$

$$\langle \texttt{if false then } c_1 \texttt{ else } c_2, \sigma\rangle \to \langle c_2, \sigma\rangle \tag{iv}$$

Finally the loop unfolds as a fixpoint:

$$\langle \texttt{while b do } c, \sigma\rangle \to \langle \texttt{if b then } (c; \texttt{while b do } c) \texttt{ else skip}, \sigma\rangle \tag{v}$$

Thus in this formalism a series of reductions is written as a succession of pairs $\langle \text{instruction}, \text{store}\rangle$:

$$\langle c, \sigma\rangle \to \langle c_1, \sigma_1\rangle \to \langle c_2, \sigma_2\rangle \to \cdots \to \langle c_n, \sigma_n\rangle$$

In this formalism, the normal form for a well-formed program is a pair $\langle \texttt{skip}, \sigma'\rangle$, with σ' being the memory state at the end of execution.

Example 3.2.1. Given two well-formed programs c_1 and c_2, we illustrate the combined use of context deduction rules and rewriting rules in a small-step evaluation sequence by the reduction, in the memory state $X \mapsto 5$, of the program $\texttt{if } X = 5 \texttt{ then } c_1 \texttt{ else } c_2$:

$$\frac{\overline{(\!|X, X \mapsto 5|\!) \to 5}}{\langle \texttt{if } X = 5 \texttt{ then } c_1 \texttt{ else } c_2, X \mapsto 5\rangle \to \langle \texttt{if } 5 = 5 \texttt{ then } c_1 \texttt{ else } c_2, X \mapsto 5\rangle}$$

then

$$\frac{\overline{(\!|5 = 5, X \mapsto 5|\!) \to \texttt{true}}}{\langle \texttt{if } 5 = 5 \texttt{ then } c_1 \texttt{ else } c_2, X \mapsto 5\rangle \to \langle \texttt{if true then } c_1 \texttt{ else } c_2, X \mapsto 5\rangle}$$

finally,

$$\frac{}{\langle \texttt{if true then } c_1 \texttt{ else } c_2, X \mapsto 5\rangle \to \langle c_1, X \mapsto 5\rangle}$$

3.3 Store-based operational semantics

Functional translation

Another way to deal with imperative language consists in using the concept of program functionalization (Filliâtre, 1998), where an imperative program is treated as a function mapping memory states to memory states. The resulting functional language can be given conventional operational semantics, which we do not fully expose here; yet it is interesting to understand the intuitive signification of the instructions once they have been functionalized.

Instructions are treated as functions over stores. Actually:

- the `skip` instruction is simply the identity function $\lambda\sigma.\sigma$ for stores;
- the sequence instruction ; has the semantics of a functional composition operator for two instructions f and g: $\circ = \lambda f \lambda g \lambda \sigma \cdot (g\ (f\ \sigma))$;
- the loop instruction `while b do c` can be seen as the fixpoint of the function:

$$\lambda f \lambda \sigma \cdot (\texttt{if b then } \langle c \circ f, \sigma \rangle \texttt{ else } \sigma)$$

Moreover the affectation instruction is similar to a function for store manipulation and the instruction `if` refers to the usual conditional functionality. Note that some formalisms use monads to express in a functional way the dynamics of side effects. This is called a monadic translation; it is a generalization of the one developed in this section to accommodate a variety of side-effects: refer to (Moggi, 1991; Wadler, 1995) for further reading on this topic.

Windup: reasoning on types

Define I as the type of instructions, E the type of expressions, V the type of values and S the type of stores.

Section 3.2 presented reduction rules over objects of type $I \times S$. The $\langle\ ,\ \rangle$ and $(\!|\ ,\ |\!)$ symbols were considered as constructors of cartesian products.

On the contrary, in section 3.3, the interpretation of an imperative program as a function over memory states implies that I is identical to $S \to S$ and E to $S \to V$. Here the symbol $\langle\ ,\ \rangle$ of type $I \times S \to S$, as well as the notation $(\!|\ ,\ |\!)$ of type $E \times S \to V$, denote the application function. Note how this allows the evaluation to create terms with nested $\langle\ ,\ \rangle$ and $(\!|\ ,\ |\!)$.

Blending these two concepts yields an alternative approach: to short-circuit the functional interpretation of imperative programs, by using a set of reduction rule interpreting the $\langle\ ,\ \rangle$ symbol as an operator for store construction, that is, as a symbol of type $I \times S \to S$. We will see in the next section how this translates with regards to simplifying the semantical framework, but the attentive reader can already foresee the goal that is being aimed for: amalgamating the simplicity of the definition of operational semantics with the fitness of the functional translation's store-centred approach.

Formalism.

In the store-based formalism, the semantics of IMP is expressed as a set of relations between memory states. We consider the $\langle\ ,\ \rangle$ symbol as a store constructor, hence the normal form of $\langle c, \sigma \rangle$ is a store, namely the final state resulting from the execution of c in the memory state σ.

Here as in the previous section, for all pair $\langle c, \sigma \rangle$, the addresses defined in σ are manipulated modulo alpha-conversion and bound in c, even if σ is not atomic, *i.e.* if it is built with the symbols $\langle\ ,\ \rangle$. Finally, since extensions

can only be carried out on an atomic store, application of extensions on non-atomic stores are deferred — close in this sense to explicit substitution.

Reduction rules are written:

$$\sigma \to \sigma' \ ,$$

and the congruence rule reads:

$$\frac{\sigma \to \sigma'}{\langle c, \sigma\rangle \to \langle c, \sigma'\rangle}$$

where c is an instruction of IMP. Its simplicity triggers the expression of the reduction strategy at the level of the reduction rules. Finally, we reuse the $(\!|\ ,|\!)$ notation for expression evaluation, and allow reduction of subexpression inside expressions and instructions.

Evaluation rules.

The evaluation rules for expressions are the same as in section 3.2, to which are added the context rules. For example, for addition, the context rule reads:

$$(\!|a_1 + a_2, \sigma|\!) \to (\!|(\!|a_1, \sigma|\!) \underline{+} (\!|a_2, \sigma|\!), \sigma|\!)$$

Intermediate notations, such as $\underline{+}$ and $\underline{:=}$ are introduced to mark that evaluation of an expression has already been triggered, and the next step should happen after we get a value. In the case of operators on arithmetical (*resp.* boolean) expressions, the underlined operators can be viewed as their interpretation in $\mathbb{Z}$ (*resp.* $\mathbb{B}$). Remark that this approach is as modular as the usual one, since the context and elementary rules can easily be separated into two disjoint sets of rules.

For instructions, the reduction rules are provided in Fig. 3.1. Observe how now a sequence of reductions is a series of memory states:

$$\sigma_1 \to \sigma_2 \to \cdots \to \sigma_n$$

in which some of the σ_i are syntactically built using $\langle\ ,\rangle$, others are not.

Example 3.3.1. The reduction sequence for the program of example 3.2.1 reads:

$$\begin{aligned}
&\langle \texttt{if } X = 5 \texttt{ then } c_1 \texttt{ else } c_2, X \mapsto 5\rangle \\
&\qquad \to \langle \underline{\texttt{if}}\ (\!|X = 5, X \mapsto 5|\!)\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2, X \mapsto 5\rangle \\
&\qquad \to \langle \underline{\texttt{if}}\ (\!|X, X \mapsto 5|\!) \underline{=} (\!|5, X \mapsto 5|\!)\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2, X \mapsto 5\rangle \\
&\qquad \to \langle \underline{\texttt{if}}\ 5 \underline{=} 5\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2, X \mapsto 5\rangle \\
&\qquad \to \langle \underline{\texttt{if}}\ \texttt{true}\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2, X \mapsto 5\rangle \\
&\qquad \to \langle c_1, X \mapsto 5\rangle
\end{aligned}$$

$$\langle X := a, \sigma\rangle \to \langle X :\equiv (\!|a, \sigma|\!), \sigma\rangle \quad (1)$$
$$\langle X :\equiv n, \sigma\rangle \to \sigma\{X \mapsto n\} \quad (2)$$
$$\langle \texttt{skip}, \sigma\rangle \to \sigma \quad (3)$$
$$\langle c_1; c_2, \sigma\rangle \to \langle c_2, \langle c_1, \sigma\rangle\rangle \quad (4)$$
$$\langle \texttt{if b then } c_1 \texttt{ else } c_2, \sigma\rangle \to \langle \underline{\texttt{if}}\ (\!|b, \sigma|\!)\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2, \sigma\rangle \quad (5)$$
$$\langle \underline{\texttt{if}}\ \texttt{true}\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2, \sigma\rangle \to \langle c_1, \sigma\rangle \quad (6)$$
$$\langle \underline{\texttt{if}}\ \texttt{false}\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2, \sigma\rangle \to \langle c_2, \sigma\rangle \quad (7)$$
$$\langle \texttt{while b do c}, \sigma\rangle \to \langle \texttt{if b then (c; while b do c) else skip}, \sigma\rangle \quad (8)$$

Figure 3.1: Evaluation rules for IMP

Comparison.

In section 3.2 of this note we exposed the usual operational semantics of IMP. Although it is widespread, some of its features are somewhat peculiar: the congruence rule, asymmetric with respect to the elements of the pairs $\langle$instruction, store$\rangle$; or the role of the `skip` instruction in the definition of a normal form. Therefore this semantical framework is often considered as being rather *ad hoc*.

In the present section we have first seen how the functional translation puts the memory state in the midst of the semantics of instructions, a thing that is not well mirrored by the usual operational semantics. This led us to reformulate the small-step semantics of IMP, using a store-centered formalism in which the congruence rule is direct and the `skip` instruction no longer plays any peculiar role. The possibility of writing terms such as $\langle c_2, \langle c_1, \sigma\rangle\rangle$ in this formalism allows for the fairly natural expression of the concept of sequence, as the application of an instruction to the result of another's evaluation. Eventually the evaluation rules of this store-based operational semantics are quite close to the ones of the usual operational semantics. One can recognize in (2) the affectation rule of section 3.2, as well as the rules (6), (7), and (8) for the conditional and the loop. The rules (1) and (5) are the alternatives to the context rule. The intuitiveness of the formalism is best reflected by the rules (3) and (4). Remark that these rules describe exactly the semantics of the functionalized instructions: indeed, (4) and (3) are related to the composition and identity over stores.

For an illustration, consider the evaluation of the program $(X := 4; X := 5)$ in the initial memory state $X \mapsto 8 :: Y \mapsto 12$, using the formalism of the usual operational semantics (Table 3.1) and of our store-based approach

Table 3.1: Example operational semantics reduction

Rules	State
	$\langle(X := 4; X := 5), X \mapsto 8 :: Y \mapsto 12\rangle$
(i)	$\langle(\texttt{skip}; X := 5), X \mapsto 4 :: Y \mapsto 12\rangle$
(ii)	$\langle(X := 5), X \mapsto 4 :: Y \mapsto 12\rangle$
(i)	$\langle(\texttt{skip}), X \mapsto 5 :: Y \mapsto 12\rangle$

Table 3.2: Example state-based semantics reduction

Rules	State
	$\langle(X := 4; X := 5), X \mapsto 8 :: Y \mapsto 12\rangle$
(4)	$\langle(X := 5), \langle(X := 4), X \mapsto 8 :: Y \mapsto 12\rangle\rangle$
(2)	$\langle(X := 5), X \mapsto 4 :: Y \mapsto 12\rangle$
(2)	$X \mapsto 5 :: Y \mapsto 12$

(Table 3.2). Remark how, even though the context inference rules for standard operational semantics have been omitted, the store-based semantics still manages to be simpler with regards to both the normal form and the intermediate steps. Also note that in the case of Table 3.2, another reduction sequence is possible, and yields the same result. Indeed, the rewriting system we propose allows for multiple reduction paths, while remaining confluent. If too itching, the alternative derivations can be suppressed either by specifying that innermost reductions should be given precedence, or by defining a notion of atomic stores $\sigma^\star$ and replacing any σ variable in Fig. 3.1 by a $\sigma^\star$.

Well-behavedness.

We want to make sure that the rules of Fig 3.1 indeed describe the semantics of IMP. This is done by proving that the store-based semantics is equivalent to another, widely accepted as correct, semantics of IMP.

However it is not easy to use the classical operational semantics to this end: as illustrated by Tables 3.1 and 3.2, the reduction sequences generated by both frameworks are quite different. In fact, whereas for a given program there exists only one "conventional" reduction path, there might be many possible "store-based" paths. What we need is a semantical framework that directly associates its value to an expression without mentioning the intermediate steps, i.e., *big-step semantics*. Figure 3.2 describes this setting; for more details on this formalism see for instance (Winskel, 1993).

Let $\rightarrow^*$ be the reflexive transitive closure of the rewriting relation $\rightarrow$. The following proposition is considered common knowledge (Winskel, 1993,

$$\frac{(\!|a, \sigma|\!) \twoheadrightarrow \nu}{\langle X := a, \sigma\rangle \twoheadrightarrow \sigma\{X \mapsto \nu\}} \quad (A)$$

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \twoheadrightarrow \sigma} \quad (B)$$

$$\frac{\langle c_1, \sigma\rangle \twoheadrightarrow \sigma' \quad \langle c_2, \sigma'\rangle \twoheadrightarrow \sigma''}{\langle c_1; c_2, \sigma\rangle \twoheadrightarrow \sigma''} \quad (C)$$

$$\frac{(\!|b, \sigma|\!) \twoheadrightarrow \texttt{true} \quad \langle c_1, \sigma\rangle \twoheadrightarrow \sigma'}{\langle \texttt{if b then } c_1 \texttt{ else } c_2, \sigma\rangle \twoheadrightarrow \sigma'} \quad (D)$$

$$\frac{(\!|b, \sigma|\!) \twoheadrightarrow \texttt{false} \quad \langle c_2, \sigma\rangle \twoheadrightarrow \sigma'}{\langle \texttt{if b then } c_1 \texttt{ else } c_2, \sigma\rangle \twoheadrightarrow \sigma'} \quad (E)$$

$$\frac{(\!|b, \sigma|\!) \twoheadrightarrow \texttt{false}}{\langle \texttt{while b do } c, \sigma\rangle \twoheadrightarrow \sigma} \quad (F)$$

$$\frac{(\!|b, \sigma|\!) \twoheadrightarrow \texttt{true} \quad \langle c, \sigma\rangle \twoheadrightarrow \sigma' \quad \langle \texttt{while b do } c, \sigma'\rangle \twoheadrightarrow \sigma''}{\langle \texttt{while b do } c, \sigma\rangle \twoheadrightarrow \sigma''} \quad (G)$$

Figure 3.2: Big-step semantics for IMP

Exercice 4.10):

Theorem 3.3.2. *The semantics of* IMP *in the classical small-step framework is equivalent to its big-step semantics in the following sense:* $\langle c, \sigma\rangle \twoheadrightarrow \sigma'$ *if and only if* $\langle c, \sigma\rangle \to^* \sigma'$.

We want to prove:

Proposition 3.3.3. *The store-based operational semantics of* IMP *is equivalent to its big-step semantics in the following sense:* $\langle c, \sigma\rangle \twoheadrightarrow \sigma'$ *if and only if* $\langle c, \sigma\rangle \to^* \sigma'$.

Proof. **We prove the left-to-right implication** with a recursion on the structure of the big-step derivation $\langle c, \sigma\rangle \twoheadrightarrow \sigma'$. For a derivation of height 1, only the rule (B) can be used, and the small-step rule (3) is its obvious counterpart. The recursive case is done by examining the topmost command: we will only develop the case of composition as other cases are dealt with similarly.

If $\langle c_1; c_2, \sigma\rangle \twoheadrightarrow \sigma''$ then by (C) the premisses of this rule are $\langle c_1, \sigma\rangle \twoheadrightarrow \sigma'$ and $\langle c_2, \sigma'\rangle \twoheadrightarrow \sigma''$. By induction, we can assume the small-step derivations $\langle c_1, \sigma\rangle \to^* \sigma'$ and $\langle c_2, \sigma'\rangle \to^* \sigma''$. Moreover the rule (4) transforms $\langle c_1; c_2, \sigma\rangle$ into $\langle c_2, \langle c_1, \sigma\rangle\rangle$; which can then be reduced by the two last rules, first to $\langle c_2, \sigma'\rangle$, then to σ''. Hence we have produced the derivation $\langle c_1; c_2, \sigma\rangle \to^* \sigma''$.

The right-to-left implication is proved by induction on the length of the small-step derivation. For a derivation of length 1, as before there is only one possibility, and the rules (3) and (B) are complementary. As in the first part of the proof, we demonstrate the inductive case on the composition, with the proofs of the other cases being comparable.

Assume a reduction path π which starts with the step $\langle c_1; c_2, \sigma\rangle \to \langle c_2, \langle c_1, \sigma\rangle\rangle$. We can assume that $\langle c_1, \sigma\rangle$ is reduced first — if not, all the reductions generated by c_2 will be blocked until c_1 is evaluated. Then π contains the reduction sequence $\langle c_1, \sigma\rangle \to^* \sigma'$, and hence a reduction sequence $\langle c_2, \sigma'\rangle \to^* \sigma''$. By induction, to these two sequences correspond the derivations $\langle c_1, \sigma\rangle \twoheadrightarrow \sigma'$ and $\langle c_2, \sigma'\rangle \twoheadrightarrow \sigma''$, which, combined by the rule (C), yield the derivation $\langle c_1; c_2, \sigma\rangle \twoheadrightarrow \sigma''$. □

By transitivity, theorem 3.3.2 and proposition 3.3.3 allow to conclude to the equivalence of the classical and the store-based operational semantics of IMP.

3.4 Extending IMP

IMP, as a 'toy language', is but an academic exercise as long as non-trivial extensions are left out of the picture. Thus local variable declarations, algebraic types, records, *etc.* can be added to the language, and their semantics can be expressed in either the usual or the store-based semantical frameworks. In this section we deal with two cases in particular: exceptions and expression languages.

Adding exceptions

IMP can be enriched to allow the use of exceptions, by adding two commands `raise` v and `try` c_1 `with` $x \to c_2$ to the syntax of the language. The `raise` operator interrupts the current evaluation and raises an exception. Its counterpart `try` evaluates its first argument c_1 and, if it results in an exception, *restores the original memory state* and proceeds to evaluate c_2 with the variable x replaced by the value carried by the exception.

For instance, the program `try` $(X := 1024;$`raise` $9)$ `with` $z \to (Y := z)$ modifies the memory state $X \mapsto 8 :: Y \mapsto 12$ into $X \mapsto 8 :: Y \mapsto 9$. In particular, note that the affectation of the value 1024 to X before the `raise` command is backtracked during the evaluation of `try`.

As the following paragraphs will show, this language extension illustrates clearly the specificities of both the conventional and the store-based formalisms — in particular relatively to the way they manage information within a reduction sequence. Remark that this extension is also relevant in the case of proof languages, in which backtracking is an essential feature.

Operational semantics.

This formalism being centered around the values of programs, it is natural to use them (i.e. the left element of $\langle\,,\rangle$ pairs) to carry the exception information. Therefore a new value $\texttt{exn}_v$ is introduced, that stands for an exception that carries a value v. The evaluation context is enriched with the `try` C `with` $x \to c$ construction. A first rewrite rule is needed to generate an exception:

$$\langle \texttt{raise}\ v, \sigma\rangle \to \langle \texttt{exn}_v, \sigma\rangle$$

What is more, this exception needs to be propagated until it hits a `try` command. This is done by defining a new context E called *exception context* defined as:

$$E ::= [\,] \mid E; c \mid X := E \mid \texttt{if}\ C'\ \texttt{then}\ c_1\ \texttt{else}\ c_2$$

and adding the rule:

$$E[\texttt{exn}_v] \to \texttt{exn}_v$$

Finally the exception-catching command obeys:

$$\langle \texttt{try}\ v\ \texttt{with}\ x \to c, \sigma\rangle \to \langle v, \sigma\rangle$$
$$\langle \texttt{try}\ \texttt{exn}_v\ \texttt{with}\ x \to c, \sigma\rangle \to \langle c[x \leftarrow v], \sigma\rangle$$

where $c[x \leftarrow v]$ denotes the substitution of the variable x in c by the value v.

Store-based semantics.

The store-based formalism does not rely on programs to relay the exception mechanism. Instead, true to its philosophy that stores are at the center of the computation of imperative programs, it integrates a new store constructor `exn` of type $V \times S \to S$. The congruence rule is untouched, and the following reduction rules are added to the semantical formalism:

$$\langle \texttt{raise}\ v, \sigma\rangle \to \texttt{exn}_v\ \sigma$$
$$\langle c, \texttt{exn}_v\ \sigma\rangle \to \texttt{exn}_v\ \sigma$$
$$\langle \texttt{try}\ c_1\ \texttt{with}\ x \to c_2, \sigma\rangle \to \langle c_1; \underline{\texttt{try}}^x_s\ c_2, \sigma\rangle$$
$$\langle \underline{\texttt{try}}^x_s\ c, \texttt{exn}_v\ \sigma'\rangle \to \langle c_2[x \leftarrow v], \sigma\rangle$$
$$\langle \underline{\texttt{try}}^x_s\ c, \sigma'\rangle \to \sigma'$$

These rules are quite easily understood: `raise` creates an "exception" memory state, which does not allow any command execution except outside of the $\underline{\texttt{try}}$ catching primitive, itself derived from the evaluation of a `try` instruction.

Note. Overall, this section illustrates the polarity of both store-based and conventional approaches. Indeed, the treatment of exceptions, as either special programs or special stores, highlights the way that the continuity of computation is ensured — again, through program values or store values. In short, the information flow is, depending on the formalism, captured either by the program or the memory state, i.e., the left-hand side or the right-hand side of the $\langle\,,\rangle$ construction.

IMP as an expression language

Sections 3.2 and 3.3 showed how, for an imperative language, operational semantics and functional translation articulate. This is an attempt to study the case, occurring in some programming languages, where expressions can trigger side effects.

To this end, we consider a language where all elements of the language IMP are seen as expressions that both return a value and possibly trigger side effects. Hence `skip` or $:=$, in addition to their usual semantics, return an arbitrarily defined value: for instance, `unit` or `end`. As a consequence, the notation $\langle\,,\rangle$ previously used to evaluate instructions is useless here, only the symbol for expression evaluation $(\!|\,,|\!)$ remains.

Operational semantics.

Because expressions now encompass instructions, thus being allowed to trigger side effects, the rule scheme of section 3.2

$$(\!|e, \sigma|\!) \to e'$$

is no longer fit. Pairs need to be used on both sides of the reduction relation to record changes performed to the memory state. Hence the new reduction rules for expressions read:

$$(\!|e, \sigma|\!) \to (\!|e', \sigma'|\!)$$

And a single context rule remains:

$$\frac{(\!|e, \sigma|\!) \to (\!|e', \sigma'|\!)}{\langle C[e], \sigma\rangle \to \langle C[e'], \sigma'\rangle}$$

The reduction rules for the usual expressions simply return the store unchanged along with the result of the computation:

$$\begin{aligned}(\!|4 + 3, \sigma|\!) &\to (\!|7, \sigma|\!)\\ (\!|\texttt{true} \wedge \texttt{false}, \sigma|\!) &\to (\!|\texttt{false}, \sigma|\!)\\ (\!|X, \sigma|\!) &\to (\!|\sigma(X), \sigma|\!)\end{aligned}$$

The reduction rules for instructions are identical to those of the corresponding paragraph in section 3.2, modulo the pair-naming convention.

$$\begin{aligned}
(\!|X := n, \sigma|\!) &\to (\!|\texttt{skip}, \sigma\{X \mapsto n\}|\!) \\
(\!|\texttt{skip}; c, \sigma|\!) &\to (\!|c, \sigma|\!) \\
(\!|\texttt{if true then } c_1 \texttt{ else } c_2, \sigma|\!) &\to (\!|c_1, \sigma|\!) \\
(\!|\texttt{if false then } c_1 \texttt{ else } c_2, \sigma|\!) &\to (\!|c_2, \sigma|\!)
\end{aligned}$$

With the `while` construct being defined as the usual combination of the "`if`" and "`;`" commands.

Store-based semantics.

Here as in section 3.4, the addition of side-effects to expression in the language entails the creation of a new operator to evaluate the expressions: $(\!|\,,\,|\!)$, of type $E \times S \to V \times S$. This operator for Cartesian product construction yields proper expression evaluation, which results in the creation of a pair (value, store) witnessing both the side effects and the value of the computation. The functions $\cdot_1$ and $\cdot_2$ are defined as the usual projections over pairs.

The rules handling arithmetic and boolean expressions, when bringing "imperative" expressions in, become a bit more space-consuming than in section 3.3: the reduction of subexpressions has an impact on stores that needs to be mirrored. For example, addition has for context rule:

$$(\!|a_1 + a_2, \sigma|\!) \to (\!|(\!|a_1, \sigma|\!)_1 \mathbin{\underline{+}} (\!|a_2, (\!|a_1, \sigma|\!)_2|\!)_1, (\!|a_2, (\!|a_1, \sigma|\!)_2|\!)_2|\!)$$

and the reduction rule that returns the final (value, store) pair:

$$(\!|4 \mathbin{\underline{+}} 3, \sigma|\!) \quad \to \quad (7, \sigma)$$

The addition of return values to instructions entail a few modifications in the reduction rules. The store-based rules follow:

$$\begin{aligned}
(\!|X := a, \sigma|\!) &\to (\!|X :\underline{=} (\!|a, \sigma|\!)_1, (\!|a, \sigma|\!)_2|\!) \\
(\!|X :\underline{=} n, \sigma|\!) &\to \sigma\{X \mapsto n\} \\
(\!|\texttt{skip}, \sigma|\!) &\to (\texttt{end}, \sigma) \\
(\!|c_1; c_2, \sigma|\!) &\to (\!|c_2, (\!|c_1, \sigma|\!)_2|\!) \\
(\!|\texttt{if b then } c_1 \texttt{ else } c_2, \sigma|\!) &\to (\!|\underline{\texttt{if}}\ (\!|b, \sigma|\!)_1\ \underline{\texttt{then}}\ c_1\ \underline{\texttt{else}}\ c_2, (\!|b, \sigma|\!)_2|\!) \\
(\!|\underline{\texttt{if}} \texttt{ true } \underline{\texttt{then}} \ c_1 \ \underline{\texttt{else}} \ c_2, \sigma|\!) &\to (\!|c_1, \sigma|\!) \\
(\!|\underline{\texttt{if}} \texttt{ false } \underline{\texttt{then}} \ c_1 \ \underline{\texttt{else}} \ c_2, \sigma|\!) &\to (\!|c_2, \sigma|\!) \\
(\!|\texttt{while b do c}, \sigma|\!) &\to (\!|\texttt{if b then } (c; \texttt{while b do c}) \\
&\qquad\qquad \texttt{else skip}, \sigma|\!)
\end{aligned}$$

An evaluation sequence for an expression e and a memory state σ will then read as a sequence of pairs:

$$(\!|e, \sigma|\!) \to (\!|e_1, \sigma_1|\!) \to \ldots \to (\!|e_n, \sigma_n|\!) \to (v, \sigma')$$

Because it manipulates pairs, this formalism draws noticeably close to the usual operational semantics. The only difference resides in the choice of having a congruence rule, where traditionally an asymmetrical context rule is preferred. Despite the fact that the rules expressing the reduction strategy are consequently quite unwieldy, an advantage is found in having a relatively simple composition rule. Also, this formalism is entirely based on rewrite rules.

3.5 Related work

Following Crank and Felleisen (Crank and Felleisen, 1991), Wright and Felleisen (Wright and Felleisen, 1994) have shown that, for typed programming languages, type safety proofs can be built simply by using typing judgements on the reducts of a program. Because their method required the formulation of a language's semantics as a rewriting system, it generated a strong interest for small-step operating semantics, built on this concept. However, in order to be able to use their methodology, one needs to be able to type the program reducts. While this is not possible in the case of IMP, we indent to develop this in the case of proof languages.

Pitts and others (Ager, 2004; Hannan and Miller, 1992; Pitts, 2002) derive big- and small-step semantics from abstract machines where stores are at the center of program evaluation. Compared to these, the formalism in this chapter is much more abstracted from implementation problems, which facilitates the understanding of the reduction rules and the process of reasoning using them. Moreover, in the case of proof languages, talks about of compilation today make little sense. Note however that these works attest if need be of the relevance of formalism based on stores.

Ariola (Ariola and Sabry, 1998) uses a semantical framework with a compositional sequence operator to prove the correctness of an implementation of the state monad. Messeguer and Rosu (Meseguer and Roşu, 2006) use rewriting logics as a semantical framework for the semantics of programs. This allows them to position themselves at the intersection of the operational and denotational approaches. The also use a particularly rich notion of state, including memory, thread continuations, input-output and global environment — close in this to abstract machines. However both of these semantics are centered on expressions.

3.6 Application: proof-based operational semantics

As mentioned at the beginning of this chapter, a proof language is very similar to an imperative programming language: the state of a proof can be seen as a memory state, which is modified by proof scripts that do not have a return value. Thus the store-based semantics of section 3.3 makes a relevant fit for this kind of language: in particular, there is absolutely no benefit in having tactics return an artificial value.

We show, on a simple example, how the store-based approach can be applied to proof languages. The case in point, intuitionistic minimalistic logic, is a restriction of $\mathcal{L}^{1}_{m}$ and LJM to two logical connectives: implication $\Rightarrow$ and truth `true`. The corresponding inference rules follow:

$$\frac{}{\Gamma; x : A \vdash x : A}\ \mathrm{ax}_{\mathcal{L}} \qquad \frac{}{\Gamma, x : A \vdash x : A}\ \mathrm{ax}_{\mathcal{R}}$$

$$\frac{}{\Gamma \vdash \ltimes : \mathtt{true}}\ \mathtt{true}_{\mathcal{R}}$$

$$\frac{\Gamma \vdash v : A \qquad \Gamma; e : B \vdash C}{\Gamma; v \cdot e : A \Rightarrow B \vdash C} \Rightarrow_{\mathcal{L}} \qquad \frac{\Gamma, x : A \vdash v : B}{\Gamma \vdash \lambda x^{A}.v : A \Rightarrow B} \Rightarrow_{\mathcal{R}}$$

$$\mathrm{cut}_{\mathcal{L}}\ \frac{\Gamma, x : A \vdash v : B \qquad \Gamma, x : A; e : B \vdash C}{\Gamma; \tilde{\mu} x^{A}.\langle v \| e \rangle : A \vdash C}$$

$$\mathrm{cut}_{\mathcal{R}}\ \frac{\Gamma \vdash v : B \qquad \Gamma; e : B \vdash * : A}{\Gamma \vdash \mu \alpha^{A}.\langle v \| e \rangle : A}$$

The proof language built upon this formalism contains the following constructs:

- for each inference rule in the logical system, a similarly named tactic: two nullary tactics $\Rightarrow_{\mathcal{L}}$ and $\mathtt{true}_{\mathcal{R}}$, three unary tactics $\Rightarrow_{\mathcal{R}}$, $\mathrm{ax}_{\mathcal{L}}$ and $\mathrm{ax}_{\mathcal{R}}$ and two binary tactics $\mathrm{cut}_{\mathcal{L}}$ and $\mathrm{cut}_{\mathcal{R}}$;
- the nullary identity strategy `idtac`, that leaves the proof untouched, and the binary sequence strategy $\diamond$, that successively evaluates its two arguments.

Then it is easy to provide local semantical evaluation rules for the tactics:

using a proof representation based on sequents, we have:

$$\langle \mathrm{ax}_{\mathcal{L}}\ x,\ \dfrac{?}{\Gamma; \Xi_0 : A \vdash x : A} \rangle \to \dfrac{}{\Gamma; x : A \vdash x : A} \tag{1}$$

$$\langle \mathrm{ax}_{\mathcal{R}}\ x,\ \dfrac{?}{\Gamma, x : A \vdash X_0 : A} \rangle \to \dfrac{}{\Gamma, x : A \vdash x : A} \tag{2}$$

$$\langle \mathtt{true}_{\mathcal{R}},\ \dfrac{?}{\Gamma \vdash X_0 : \mathtt{true}} \rangle \to \dfrac{}{\Gamma \vdash \ltimes : \mathtt{true}} \tag{3}$$

$$\langle \Rightarrow_{\mathcal{L}},\ \dfrac{?}{\Gamma; \Xi_0 : A \Rightarrow B \vdash C} \rangle \to \dfrac{\dfrac{?}{\Gamma \vdash X_1 : A} \qquad \dfrac{?}{\Gamma; \Xi_1 : B \vdash C}}{\Gamma; X_1 \cdot \Xi_1 : A \Rightarrow B \vdash C} \tag{4}$$

$$\langle \Rightarrow_{\mathcal{R}}\ x,\ \dfrac{?}{\Gamma \vdash X : A \Rightarrow B} \rangle \to \dfrac{\dfrac{?}{\Gamma, x : A \vdash X_1 : B}}{\Gamma \vdash \lambda x^{A}.X_1 : A \Rightarrow B} \tag{5}$$

$$\langle \mathrm{cut}_{\mathcal{L}}\ x\ B,\ \dfrac{?}{\Gamma; \Xi_0 : A \vdash C} \rangle \to \dfrac{\dfrac{?}{\Gamma, x : A \vdash X_1 : B} \qquad \dfrac{?}{\Gamma, x : A; \Xi_1 : B \vdash C}}{\Gamma; \tilde{\mu} x^{A}.\langle X_1 \| \Xi_1 \rangle : A \vdash C} \tag{6}$$

$$\langle \mathrm{cut}_{\mathcal{R}}\ \alpha\ B,\ \dfrac{?}{\Gamma \vdash X_0 : A} \rangle \to \dfrac{\dfrac{?}{\Gamma \vdash X_1 : B} \qquad \dfrac{?}{\Gamma; \Xi_1 : B \vdash \alpha : A}}{\Gamma \vdash \mu \alpha^{A}.\langle X_1 \| \Xi_1 \rangle : A} \tag{7}$$

Notation. The use of pointy brackets $\langle\, , \rangle$ can turn out quite cumbersome at times. As an alternative notation, we place the name of the command on the arrow denoting reduction. For instance, the semantics of the right implication tactic would write:

$$\dfrac{?}{\Gamma \vdash X_0 : A \Rightarrow B} \xrightarrow{\Rightarrow_{\mathcal{R}}} \dfrac{\dfrac{?}{\Gamma, x : A \vdash X_1 : B}}{\Gamma \vdash \lambda x^{A}.X_1 : A \Rightarrow B}$$

Although it could be employed systematically, in this manuscript this notation will only be used when the right element of the pointy brackets is an atomic construct, *i.e.* a proof not built using other brackets.

While these rules are local, *i.e.* they deal with only one open goal at a time, they can be instantly generalized to any proof with an arbitrary number of goals. It suffices, for instance, to chose to apply tactics to the first (in an innermost-leftmost sense) open goal in the proof tree.

Then we can propose the following reduction rules for the two strategies:

$$\langle \mathtt{idtac}, \sigma \rangle \to \sigma \tag{8}$$

$$\langle c_1 \diamond c_2, \sigma \rangle \to \langle c_2, \langle c_1, \sigma \rangle \rangle \tag{9}$$

Finally, as with the store-based formalism, all that is needed to enable in-depth reductions is a simple congruence rule.

Example 3.6.1. Let us consider how the proof of example 2.5.4:

$$\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash y : A \Rightarrow B; z : B}\,\mathrm{ax}_{\mathcal{R}} \qquad \dfrac{\dfrac{}{\Gamma' \vdash x : A; z : B}\,\mathrm{ax}_{\mathcal{R}} \quad \dfrac{}{\Gamma'; z : B \vdash z : B}\,\mathrm{ax}_{\mathcal{L}}}{\Gamma'; x \cdot z : A \Rightarrow B \vdash z : B}\,\Rightarrow_{\mathcal{L}}}{\Gamma' \vdash \mu z^{B}.\langle y \| x \cdot z\rangle : B}\,\mathrm{cut}_{\mathcal{R}}}{\Gamma \vdash \lambda x^{A}.\lambda y^{A\Rightarrow B}.\mu z^{B}.\langle y \| x \cdot z\rangle : A \Rightarrow ((A \Rightarrow B) \Rightarrow B)}\,2 \times \Rightarrow_{\mathcal{R}}$$

with $\Gamma = A, B : \mathtt{bool}$ and $\Gamma' = \Gamma, x : A, y : A \Rightarrow B$, can be obtained by evaluating the proof script:

$$\Rightarrow_{\mathcal{R}} x \diamond \Rightarrow_{\mathcal{R}} y \diamond \mathrm{cut}_{\mathcal{R}}\ (A \Rightarrow B)\ z \diamond \mathrm{ax}_{\mathcal{R}}\ y \diamond \Rightarrow_{\mathcal{L}} \diamond \mathrm{ax}_{\mathcal{R}}\ x \diamond \mathrm{ax}_{\mathcal{L}}\ z$$

on the initial proof X_0. Using proof terms to represent the proof, the evaluation of the script unfolds as follow:

$$\langle \Rightarrow_{\mathcal{R}} x \diamond \Rightarrow_{\mathcal{R}} y \diamond \mathrm{cut}_{\mathcal{R}}\ (A \Rightarrow B)\ z \diamond \mathrm{ax}_{\mathcal{R}}\ y \diamond \Rightarrow_{\mathcal{L}} \diamond \mathrm{ax}_{\mathcal{R}}\ x \diamond \mathrm{ax}_{\mathcal{L}}\ z, X_0\rangle$$

This reduces, by four consecutive applications of the rule (9), into:

$$\langle \mathrm{ax}_{\mathcal{L}}\ z, \langle \mathrm{ax}_{\mathcal{R}}\ x, \langle \Rightarrow_{\mathcal{L}}, \langle \mathrm{ax}_{\mathcal{R}}\ y, \langle \mathrm{cut}_{\mathcal{R}}\ (A \Rightarrow B)\ z, \langle \Rightarrow_{\mathcal{R}} y, \langle \Rightarrow_{\mathcal{R}} x, X_0\rangle\rangle\rangle\rangle\rangle\rangle\rangle$$

Then the evaluation proceeds with the tactics: by dual application of (5),

$$\langle \mathrm{ax}_{\mathcal{L}}\ z, \langle \mathrm{ax}_{\mathcal{R}}\ x, \langle \Rightarrow_{\mathcal{L}}, \langle \mathrm{ax}_{\mathcal{R}}\ y, \langle \mathrm{cut}_{\mathcal{R}}\ (A \Rightarrow B)\ z, \lambda x^{A}.\lambda y^{A\Rightarrow B}.X_2\rangle\rangle\rangle\rangle\rangle\rangle$$

Then, by (7) and (2),

$$\langle \mathrm{ax}_{\mathcal{L}}\ z, \langle \mathrm{ax}_{\mathcal{R}}\ x, \langle \Rightarrow_{\mathcal{L}}, \lambda x^{A}.\lambda y^{A\Rightarrow B}.\mu z^{B}.\langle y \| \Xi_0\rangle\rangle\rangle\rangle$$

And finally, through (4), (2) and (1), the final proof term is obtained:

$$\lambda x^{A}.\lambda y^{A\Rightarrow B}.\mu z^{B}.\langle y \| x \cdot z\rangle$$

Remark that this derivation, although more space-consuming than the one of Table 3.2, shows a strong similarity with it.

The parallel with imperative programming is indisputable: on the one hand, tactics are the equivalent of specific instances of the affectation rule. The comparison between incomplete proofs and stores is even better seen when using proof terms to represent them: for instance, the tactic $\mathrm{true}_{\mathcal{R}}$ transforms a proof term (think: store) t into a proof term (again: store) $t\{X_0 \leftarrow \ltimes\}$ which is exactly the semantics of an affectation. On the other hand, semantically the two strategies $\diamond$ and `idtac` are perfect copycats of the sequence and skip imperative operators, and it is possible to derive proof language equivalents of the conditional, loop and exception operators.

While elegant and simple, this approach has several limitations. First, because of the generalization we did of local semantical rules to complete proofs, tactics are bound to be applied linearly. In order words, there is no

way to apply a tactic to the *second* open goal in the tree — only to the first one. Second, the semantics presented here do not account for tactic failure, success, and progress, thus making it impossible to express the semantics of, say, a strategy such as Coq's `first` that applies different tactics until one of them generates a progress in the proof. Finally, mirroring the behaviour of, for instance, one of PVS's fundamental strategies `then*` (that applies its second argument to *all* of the subgoals generated by the application of its first argument), would be quite complicated.

These problems are solved by introducing a more complex representation of the proof state, that includes a way to designate *current goals*, *i.e.* the goal or set of goals that the evaluation of tactics will affect. The proof representation should also provide a mean to store information about the outcome of a tactic. Finally, it should retain as much as possible of the formalism of this chapter, and in particular the simplicity of a congruence-based system. These problems are addressed in the forthcoming chapter.

4 The Proof Monad

The previous chapter has proposed a semantical framework that is well-suited for tactics, but not rich enough to deal with strategies: in this chapter we turn to this problem. *We present a structure adapted to the representation of proofs and to the semantics of strategies in procedural theorem provers.*

One could compare the procedural development of a proof to a dialog between a proof engine and a scientist, and the representation of this dialog is still a young research topic. Only in the last few years has some research been conducted on the semantics of proof languages and their formalization (Delahaye, 2000; Jojgov, 2003b; Martin et al., 1996), and this area is still the object of ongoing work. However, these analyses — and chapter 3 with them — concentrate on one side of the dialogue, namely the scientist to machine part: given a goal, they identify which commands can be validly applied. Few efforts, if any, have been made to formalize the machine to scientist feedback. In fact, the representations of proofs proposed so far do not contain enough information to accurately mirror *both* sides of the dialogue: as a result, expressing the semantics of sophisticated strategies is made complicated, if not impossible.

Using the branch of mathematics known as category theory, this chapter exposes an innovative way of representing proof states, interpreting constructs of a proof language and modelling the feedback of the proof engine. More specifically, in this chapter we formalize an important aspect of the prover to scientist conversation: the communication by the prover of the current goals. This leads us to formalize the notion of proof trees with current goals; we show that this formalization has the characteristics of a monad, and we investigate its connection with the semantics of chapter 3.

⋆

As seen in chapter 3, a naive encoding of the state of a proof has limitations when it comes to expressing the semantics of tactics, for which more information on the state of the proof in general and its history in particular is required. For instance, a successful tactic application generates some progress at the level of the proof state, which is a difficult thing to identify using this representation. In turn, this makes giving semantics to a strategy that tests for an command's outcome impossible.

A concrete example of the limitations of the usual proof representation is to be found in a recent attempt to formalize the semantics of one of PVS's most feature-rich strategies: `try`. Informally, `(try` $\mathtt{t}_1$ $\mathtt{t}_2$ $\mathtt{t}_3$`)` applies its first argument $\mathtt{t}_1$ to the goal, and if it generates subgoals, it applies $\mathtt{t}_2$ to the subgoals, else it applies $\mathtt{t}_3$. Furthermore, if $\mathtt{t}_2$ fails, for example, because $\mathtt{t}_2$ = `(fail)`, then it initiates a backtracking sequence, which is propagated until it is evaluated as the first member of another `try` construct, in which case it evaluates its third argument. The formal semantics of `try` is given in

(Archer et al., 2003) using five different types of state information: *failure*, *success*, *skip*, *subgoals*, *backtrack*. Using |.| as a semantic evaluator, the semantics of `try` can be expressed as follows:

$$|(\texttt{try t}_1\ \texttt{t}_2\ \texttt{t}_3)| = \left|\begin{array}{lll} |\texttt{t}_3| & \text{if} & |\texttt{t}_1| \in \{\textit{skip}, \textit{backtrack}\} \\ |\texttt{t}_1| & \text{if} & |\texttt{t}_1| \in \{\textit{failure}, \textit{success}\} \\ \textit{backtrack} & \text{if} & |\texttt{t}_1| = \textit{subgoals}, \\ & & |\texttt{t}_2| \in \{\textit{failure}, \textit{backtrack}\} \\ \textit{subgoals} & \text{if} & |\texttt{t}_1| = \textit{subgoals}, \\ & & |\texttt{t}_2| \in \{\textit{skip}, \textit{subgoals}\} \\ \textit{success} & \text{if} & |\texttt{t}_1| = \textit{subgoals}, \\ & & |\texttt{t}_2| = \textit{success}\ , \end{array}\right.$$

where

$$|\texttt{(skip)}| = \textit{skip}$$
$$|\texttt{(fail)}| = \textit{failure}\ .$$

Remark that the information required to deal with the semantics of `try` is quite different from the one found in the usual arborescent representation of proofs, and presents itself as a complement to the latter — in fact, in these semantics of the `try` strategy, the proof tree isn't even mentioned. This hints at the solution: to provide adequate proof search control, some information needs to be added to the encoding of proofs.

Enter monads. Monads are constructs of the theory of categories, introduced in computer science to deal with non-functional constructs in purely functional programming languages. The idea behind monads is to bundle extra information into the objects manipulated by functions. For instance, a function $\mathtt{f}$ on expressions, say mapping $\mathtt{a}$ to $\mathtt{b}$, could be extended to a function $\mathtt{f}'$ on couples, that also increments a counter $\mathtt{x}$: $\mathtt{f}'(\mathtt{a},\mathtt{x}) = (\mathtt{b},\mathtt{x}+1)$: in this simplistic example, the 'bundling' is done by using a pair. While this example illustrates the case of imperative side-effects, monads generalize this approach to any type of side-effects, such as exceptions, input-output, continuations, non-determinism, *etc.*

In this chapter, we demonstrate how adding a simple monadic structure to a generic proof object allows us to give formal semantics to non-trivial procedural proof languages. The use of monads as denotations for proof states permits the description of the scientist-proof engine dialogue without any operational bias. We show that key tactics and strategies can be derived from monadic operators, and we discuss the compatibility of the monadic framework with the semantical framework of chapter 3. Finally we reference complete proof languages that have been designed or documented using this formalism.

4.1 Proof representations

A characteristic of procedural theorem provers is that, because dealing with the whole proof tree would be both cumbersome and confusing, they restrain the user's working field to one or more sequents. As a consequence, in most cases the proof engine's dialogue with the user concerns the working sequents, or *current goals*, rather than the whole proof tree. Hence the semantics of proof language constructs needs to be able to refer to a subset of the whole proof tree.

The representations of a proof detailed in chapter 1 can easily be modified to incorporate the definition of current goals. However, we prefer a more factorized definition of this representation, that takes advantage of the tree-like structure of proofs.

Definition 4.1.1 (Current index)**.** In a given open proof, we call *current index* the nearest common ancestor of the proof's current goals.

Definition 4.1.2 (Indexed proof)**.** An *indexed proof* is a proof parametrized by a current index, that has the following property: there are no open goals that share the ancestor represented by the current index, but that are not current goals.

Hence in indexed proofs, we can refer to the current goals of a proof simply by their nearest common ancestor. The advantage of this definition is that it is arguably abstract enough to be independent from the various representations of proofs.

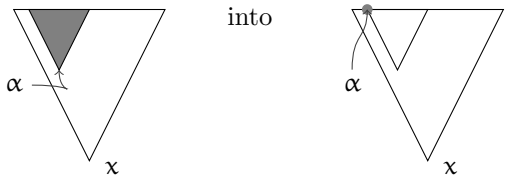
Notation. Let $\mathtt{x}, \mathtt{y}, \mathtt{z}$ be variables that range over proof trees, and α, β, γ variables for proof tree indexes. Let $\mathtt{x}[\alpha]$ the proof tree $\mathtt{x}$ where the current index is α. Let τ be the type of indexed proof trees.

When relevant, we will allow ourselves to omit the mention of the current index, and overload the notation $\mathtt{x}$ to denote indexed proofs.

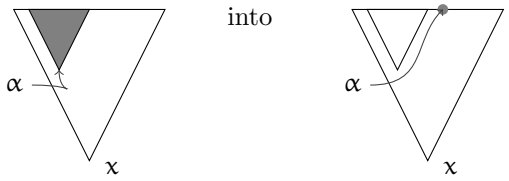
Definition 4.1.3. Let $\downarrow_{\mathtt{i}}$ be the partial function of type $\tau \to \tau$ that changes a proof's current index to the $\mathtt{i}$-th element of the set of current goals, when it exists. Let $\rightsquigarrow$ be the partial function of type $\tau \to \tau$ that changes the current index to the next following open goal, *i.e.* an open goal that is not in the input tree's active goals, if it exists.

Note that these functions are specified, but not detailed: any tree structure, and any tree traversing functions can be used as their implementation.

Example 4.1.4. Let us illustrate using simplified graphics the semantics of the two functions $\downarrow_{\mathtt{i}}$ and $\rightsquigarrow$. In the following graphics, the big triangle represents a proof, of which the small triangle designates a subproof. The grayed area denotes the subtree whose open leafs goals are the current goals. The $\downarrow_{\mathtt{i}}$ provides a way to displace the current index to a leaf of the active subtree. For instance, $\downarrow_1$ would transform the indexed proof tree:



On the other hand, the function $\rightsquigarrow$ would modify the indexed proof:



4.2 Monads

Adding *computational effects* such as side-effects, exceptions or input / output to pure functional languages is an important step towards the usability of these languages. While some languages such as Ocaml or Scheme take the party to augment their semantics with *ad hoc* constructions and rules, others such as Haskell or Gallina look for ways to translate these features into pure functional constructs. One simple idea is to augment the objects that functions manipulate, in order to encompass a memory state, an exception stack or an input / output socket. Moggi's decisive contribution (Moggi, 1989, 1991) was to provide a structure to achieve this which is based on well-known mathematical construct: *monads*. By recognizing the capabilities that monads have of capturing computational effects, Moggi brought a complete toolbox of well-know mathematical operations and properties to this topic.

The following formulation of the theory of categories is inspired by informal lectures by Buronni (Buronni, 2004), other approaches exist but this one has the merit of being quite graphical.

Definition 4.2.1 (Graph)**.** A graph is defined as a quadruplet $(\mathsf{G}_0, \mathsf{G}_1, \mathsf{s}, \mathsf{t})$, where G_0 is the set of *objects* and G_1 is the set of *morphisms* of the graph. For any morphism f in G_1, the objects $\mathsf{x} = \mathsf{s}\,\mathsf{f}$ and $\mathsf{y} = \mathsf{t}\,\mathsf{f}$ are called respectively the *source* and the *target* of f, and we note $\mathsf{f} : \mathsf{x} \to \mathsf{y}$. This definition of a graph can be illustrated by the following diagram:

$$\mathsf{G}_1 \underset{\mathsf{t}}{\overset{\mathsf{s}}{\rightrightarrows}} \mathsf{G}_0$$

Definition 4.2.2 (Category)**.** A category is defined as a triple $(\mathsf{G}, \mathrm{id}, \circ)$, where G is a graph and id and $\circ$ are morphism constructors, called respec-

tively *identity* and *composition*:

$$G_0 \xrightarrow{\text{id}} G_1 \qquad\qquad (G_1, G_1) \xrightarrow{\circ} G_1$$

The elements of a category verify:

- for any morphisms f, g of G such that $t\, f = s\, g$,

$$s\,(f \circ g) = s\, f \qquad\qquad t\,(f \circ g) = t\, g \ .$$

- for any object x of G,

$$s\,(\mathrm{id}\, x) = t\,(\mathrm{id}\, x) = x \ .$$

- for any morphisms f, g, h of G such that $t\, f = s\, g$ and $t\, g = s\, h$,

$$(f \circ g) \circ h = f \circ (g \circ h) \ .$$

- For any morphism $f : x \to y$ of G

$$f \circ (\mathrm{id}\, y) = (\mathrm{id}\, x) \circ f = f \ .$$

Note that there is a one-to-one correspondence between identity morphisms and objects: in other words, one can define an identity morphism for each object of the category.

Definition 4.2.3 (Functor)**.** Functors are morphisms between categories. A functor $F : C \to C'$ between two categories is defined by an homomorphism $h : G \to G'$ between their corresponding graphs such that:

- for any object x of G,

$$h\,(\mathrm{id}\, x) = \mathrm{id}\,(h\, x) \ .$$

- for any morphisms f, g of G such that $t(f) = s(g)$,

$$h\,(f \circ g) = (h\, f) \circ (h\, g) \ .$$

Composition between morphisms is defined by the composition between their corresponding graph homomorphisms.

Definition 4.2.4 (Natural transformation)**.** Natural transformations are morphisms between functors. Given two categories C, C' and two functors $F, F' : C \to C'$, a natural transformation $\phi : F \to F'$ is a family of morphisms indexed by the objects of C such that:

- for any object x of C,

$$s\, \phi_x = F\, x \qquad\qquad t\, \phi_x = F'\, x \ .$$

· for any morphism $f : x \to y$ of C,

$$(F\,f) \circ \phi_y = \phi_x \circ (F'\,f) \ .$$

The diagrammatic representation of a natural transformation between two functors is as follow:

$$C \ \underset{\xrightarrow[F']{}}{\overset{\xrightarrow{F}}{\Downarrow \phi}} \ C'$$

These definitions lay the groundwork for the definition of monads.

Definition 4.2.5 (Monad)**.** A monad over a given category $C = (G, \mathrm{id}, \circ)$ is a triple $(T, \mathit{unit}, \star)$, where $T : C \to C$ is a functor, and the *monadic operators* $\mathit{unit} : \mathrm{id} \to T$ and $\star : T \to (\mathrm{id} \to T) \to T$ are natural transformations such that:

· for any object x of C and morphism $f : C \to T\,C$,

$$\mathit{unit}\,x \star f = f$$

· for any object m of $T\,C$,

$$m \star \mathit{unit} = m$$

· for any object m of $T\,C$ and morphisms $f, g : C \to T\,C$,

$$m \star (f \star g) = (m \star f) \star g$$

In category theory, this definition of a monad is called a *Kleisli triple*.

Now if monads are viewed as means to carry computational effects, one can encode programs as functions mapping from a category of *values* to the monad of *computations* defined over this category. Thus, assuming that the types A and B are objects from the category of values, and $(T, \mathit{unit}, \star)$ is a monad, then a program is represented as a morphism $A \to T\,B$. The monadic operator $\mathit{unit} : A \to T\,A$ initializes the system by turning a value into its trivial computation counterpart, and $\star : T\,A \to (A \to T\,B) \to T\,B$ allows a program of type $A \to T\,B$ to be applied to a computation of type $T\,A$.

Of course, there are as many choices for T as there are computational effects and combinations of these. As an example, let us illustrate this concept with a monad that deals with exceptions.

Example 4.2.6 (The exception monad)**.** For a functor we choose the function of type $A \to (\mathit{Exception} | \mathit{Return}\,B)$. The natural transformations *unit* and $\star$ are defined as:

$$\begin{aligned} \mathit{unit} &= \lambda a \cdot \mathit{return}\,a \\ \star &= \lambda m \cdot \lambda f \cdot \text{match } m \text{ with} \\ &\quad \left| \begin{array}{lcl} \mathit{exception} & \mapsto & \mathit{exception} \\ \mathit{return}\,a & \mapsto & f\,a \end{array} \right. \end{aligned}$$

A wealth of other programming examples as well as an introduction to monads for programmers can be found in (Wadler, 1995).

4.3 The proof monad

This section presents the monadic formalization of the theory of proof languages. It will be used to add additional information into the proof representation, and appropriately mirror the semantics of proof languages.

First we need to define a category for the basic elements of our theory: indexed proofs, and morphisms on them.

Definition 4.3.1 (C_τ, τ)**.** Let C_τ be a category, of which τ is an object. Let $\mathtt{x}[.]$, $\mathtt{y}[.]$ and $\mathtt{z}[.]$ be parametric variables ranging over elements of τ.

Definition 4.3.2 (Proof monad)**.** Let $(\mathcal{M}, \mathit{unit}, \star)$ be a monad over C_τ. Define $\mathcal{M}$ as the datatype constructor:

$$\textbf{data}\ \mathcal{M}\,\tau = \left| \begin{array}{l} \mathtt{x}[\mathit{success}] \\ \mathtt{x}[\mathit{subgoals}\ \mathtt{b}\ \alpha] \\ \mathit{exception}\ \mathtt{s} \end{array} \right.$$

where $\mathtt{x}$ is a proof indexed by α, $\mathtt{b}$ is one of the booleans $\mathtt{true}, \mathtt{false}$ and $\mathtt{s}$ is a symbol. Let "match" be a destructor of this datatype. The monadic operators are defined as follow:

$$\begin{array}{l} \mathit{unit} : \tau \to \mathcal{M}\,\tau \\ \mathit{unit} = \lambda \mathtt{x}[\alpha] \cdot \mathtt{x}[\mathit{subgoals}\ \mathtt{false}\ \alpha] \\ \quad \star : \mathcal{M}\,\tau \to (\tau \to \mathcal{M}\,\tau) \to \mathcal{M}\,\tau \\ \quad \star = \lambda \mathtt{m} \cdot \lambda \mathtt{f} \cdot \text{match}\ \mathtt{m}\ \text{with} \\ \qquad \left| \begin{array}{lcl} \mathtt{x}[\mathit{subgoals}\ \mathtt{b}\ \alpha] & \mapsto & \text{match}\ (\mathtt{f}\,\mathtt{x}[\alpha])\ \text{with} \\ \qquad \left| \begin{array}{lcl} \mathtt{y}[\mathit{subgoals}\ \mathtt{b}'\ \beta] & \mapsto & \mathtt{y}[\mathit{subgoals}\ (\mathtt{b} \oplus \mathtt{b}')\ \beta] \\ * & \mapsto & (\mathtt{f}\,\mathtt{x}[\alpha]) \end{array} \right. & & \\ * & \mapsto & \mathtt{m}\ , \end{array} \right. \end{array}$$

where $\oplus$ is the boolean disjunction. Propositions 4.3.3 and 4.3.4 verify that these operators are correctly defined.

Note. In order for this structure to be a well-defined monad, one needs to complete the definition of C_τ to recursively add as objects of this category all the constructions $\mathcal{M}\,\tau, \mathcal{M}(\mathcal{M}\,\tau), \ldots, \mathcal{M}(\ldots(\mathcal{M}\,\tau)\ldots)$, *etc.*.

With only three constructors, this monadic construction remains quite simple. The intuition is the following: first, a notion of subgoals is attached to the current index. Then we have two special values, for the current index and for the whole proof tree, represented by the constructions *success* and *exception*. Details on the role of these constructors in the semantics of proof commands are given in the upcoming section 4.4.

Proposition 4.3.3. *The monadic operators satisfy the left and right unit properties:*

$$\forall \mathsf{x}[\alpha] : \tau, \forall \mathsf{f} : \tau \to \mathcal{M}\,\tau, (\mathit{unit}\ \mathsf{x}[\alpha]) \star \mathsf{f} = (\mathsf{f}\ \mathsf{x}[\alpha]) \quad (1)$$

$$\forall \mathfrak{m} : \mathcal{M}\,\tau, \mathfrak{m} \star \mathit{unit} = \mathfrak{m} \quad (2)$$

Proof. The proofs of these two properties are easy. For equality (1) we use case reasoning on the outcome of f. Equality (2) is proved by induction on $\mathfrak{m}$. In both proofs we use the fact that the boolean disjunction with `false` is the identity. □

Proposition 4.3.4. *The $\star$ operator is associative:*

$$\forall \mathfrak{m} : \mathcal{M}\,\tau, \forall \mathsf{f}_1, \mathsf{f}_2 : \tau \to \mathcal{M}\,\tau, \mathfrak{m} \star (\lambda \mathsf{x} \cdot (\mathsf{f}_1\ \mathsf{x}) \star \mathsf{f}_2) = (\mathfrak{m} \star \mathsf{f}_1) \star \mathsf{f}_2 \quad (3)$$

Proof. The proof is carried by induction on $\mathfrak{m}$, and then case analysis on the outcome of f_1 and f_2. The associativity of $\oplus$ concludes the proof. □

Additional *map* and *join* operators can also be defined. They are usually seen as a decomposition of the $\star$ operator: $\mathfrak{m} \star \mathsf{k} = \mathit{join}\,(\mathit{map}\,\mathsf{k}\,\mathfrak{m})$.

Definition 4.3.5 (Map)**.** This operator lifts a function on proof trees to a function on computations.

$$\begin{aligned}
&\mathit{map} : (\tau \to \tau) \to (\mathcal{M}\,\tau \to \mathcal{M}\,\tau)\\
&\mathit{map} = \lambda \mathfrak{m} \cdot \lambda \mathsf{f} \cdot \mathfrak{m} \star \lambda \mathsf{x}.\mathit{unit}\,(\mathsf{f}\,\mathsf{x})\\
&\quad = \lambda \mathfrak{m} \cdot \lambda \mathsf{f} \cdot \text{match } \mathfrak{m} \text{ with}\\
&\qquad \left|\begin{array}{lcl} \mathsf{x}[\mathit{subgoals}\ \mathsf{b}\ \alpha] & \mapsto & \mathsf{x}[\mathit{subgoals}\ \mathsf{b}\ \mathsf{f}\ \alpha] \\ * & \mapsto & \mathfrak{m} \end{array}\right.
\end{aligned}$$

Definition 4.3.6 (Join)**.** Join 'flattens' two layers of information into one.

$$\begin{aligned}
&\mathit{join} : \mathcal{M}\,(\mathcal{M}\,\tau) \to \mathcal{M}\,\tau\\
&\mathit{join} = \lambda \mathfrak{m} \cdot \mathfrak{m} \star \lambda \mathsf{x}.\mathsf{x}\\
&\quad = \lambda \mathfrak{m} \cdot \text{match } \mathfrak{m} \text{ with}\\
&\qquad \left|\begin{array}{lcl} \mathsf{x}[\mathit{subgoals}\ \mathsf{b}\ \alpha] & \mapsto & \mathsf{x}[\alpha] \\ * & \mapsto & \mathfrak{m} \end{array}\right.
\end{aligned}$$

4.4 The semantics of proof languages

For a sample proof language, we use a variation on the original LCF language, combined with the tactics of minimalistic logic, as defined in section 3.6. Its strategies include most of the widely used features in modern theorem provers such as Coq, Isabelle or PVS.

Definition 4.4.1 (PRF)**.** The proof language for minimalistic logic PRF consists in the following tactics and strategies:

$$\begin{aligned}\mathsf{c} ::= {} & \mathrm{ax}_{\mathcal{L}}\ \mathtt{x} \mid \mathrm{ax}_{\mathcal{R}}\ \mathtt{x} \mid \mathrm{true}_{\mathcal{R}} \mid \Rightarrow_{\mathcal{L}} \mid \Rightarrow_{\mathcal{R}} \mathtt{x} \mid \mathrm{cut}_{\mathcal{L}}\ \mathtt{x}\ \mathsf{B} \mid \mathrm{cut}_{\mathcal{R}}\ \mathtt{x}\ \mathsf{B} \\ & \mid \texttt{postpone} \mid \mathsf{c}.\mathsf{c} \\ & \mid \texttt{idtac} \mid \mathsf{c};\mathsf{c} \mid [\mathsf{c}_{\mathrm{list}}] \\ \mathsf{c}_{\mathrm{list}} ::= {} & \mathit{nil} \mid \mathsf{c}::\mathsf{c}_{\mathrm{list}}\end{aligned}$$

where $\mathtt{x}$ and B are respectively a well-formed variable and a well-formed formula of $\mathcal{L}^1_{\mathfrak{m}}$. The tactics correspond to the inference rules of minimalistic logic, *as per* section 3.6.

We describe the informal semantics of PRF by case:

- tactics apply once their corresponding inference rule to *each* of the current goals: in particular, if the current index points a subtree in the proof, the inferences are applied to all of the open goals in this subtree;
- "`postpone`" is a strategy to delay the treatment of the current goals and '.' is a toplevel command evaluator: in the taxonomy of section 1.4, these two commands are *interactive commands*;
- "`idtac`" and ';' are the identity and composition strategies. Composition is different from the simplistic $\diamond$ operator of section 3.6: it applies its second argument to *each* of the subgoals generated by the application of its first argument. Finally, the "$[\mathsf{c}_{\mathrm{list}}]$" construct applies a list of length $\mathfrak{n}$ of commands to the same number $\mathfrak{n}$ of open goals, on a ordered one-to-one basis. These strategies are *programming strategies* in the taxonomy of section 1.4.

The following define the set of mathematical objects that are manipulated by the proof commands. In particular it defines τ as the type of indexed proofs.

Definition 4.4.2. Define the set of indexed proofs τ, with $\mathtt{x}, \mathtt{y}, \mathtt{z}$ indexed proof variables. Let C_τ be the category of proofs built upon τ, and $\downarrow_{\mathfrak{i}}$ a morphism of C_τ. Let $(\mathcal{M}, \mathit{unit}, \star)$ be a monad over the category C_τ defined *as per* definition 4.3.2.

The key element of these semantics is the application function, that pits PRF commands against indexed proofs, returning monadic constructions. While in the previous paragraphs this operators was denoted by the simple juxtaposition of its two arguments, here we use the symbols $\langle\,,\rangle$.

Definition 4.4.3 (Application)**.** Let $\langle\,,\rangle$ be the application operator of proof commands to indexed proofs. For any command c and object $\mathtt{x}[\alpha]$ in τ, then $\langle \mathsf{c}, \mathtt{x}[\alpha]\rangle$ builds an object in $\mathcal{M}\,\tau$.

Note. A few intuitions on the choices made with this formalization:

- commands are functions from proofs to computations, i.e., of type $\tau \to \mathcal{M}\,\tau$. In this sense, our treatment of proof languages is very similar to that of functional programming languages;
- $\langle \mathtt{s}, \mathtt{x}[\alpha] \rangle$ denotes the application of the command $\mathtt{c}$ to the whole subtree of $\mathtt{x}$ designated by α. In particular, if $\mathtt{c}$ is a tactic, this denotes its systematic application to every open goal of α;
- *subgoals* `b` indicates whether or not subgoals have been generated by the command. By convention *subgoals* `false` means that the current goals were not modified;
- *success* indicates that the command has discharged (proved) the current goals;
- *exception* `s` indicates that the command has raised the exception `s`. The exceptions are raised by the user (through the use of the appropriate tactic) or the proof engine (if a tactic fails to apply correctly).

The semantics of tactics can simply be lifted from the semantics given in section 3.6. It is easy to assign a monadic constructor to each possible case of tactic applications.

Definition 4.4.4 (Semantics of tactics)**.** The result of the evaluation of a tactic `t` on an indexed proof $\mathtt{x}[\alpha]$ is derived from the semantics of its local application to a given sequent, *as per* section 3.6, by the following algorithm. If there is only one current goal:

- if the topmost symbol of the active formula does not match the symbol `s` required for the logical inference rule, then the result is the monadic construction *exception* `Not a s`;
- else, if there are no premisses in the inference rule, and `y` is the proof `x` with the previously current goal closed, then the result is the monadic construction $\mathtt{y}[\mathit{success}]$;
- else, assume there are `n` premisses in the inference rule, and `y` is the proof `x` extended with the `n` subgoals, then the result is the monadic construction *subgoals* `n` $\mathtt{y}[\alpha]$. Remark that this constructor is the counterpart to IMP's `exn` store constructor (section 3.4).

This is extended to the case where there are `m` current goals. The tactic `t` is recursively evaluated on each of the goals using an innermost leftmost path:

- if any of the evaluation generates an exception, then return this exception;
- else if the evaluation closes the current goal, and `z` is the proof `x` with all its previously current goals closed, then the result is the monadic construction $\mathtt{z}[\mathit{success}]$;

· else if for each goal n subgoals are generated, and z is the proof x extended with the nm subgoals, then the result is the monadic construction *subgoals* $nm\ z[\alpha]$

This algorithm is deterministic, hence from each tactic local evaluation rule (as defined in section 3.6) one can automatically derive the semantics of its evaluation on a global indexed proof.

Example 4.4.5. Take for instance, the semantics of $\Rightarrow_{\mathcal{L}}$, locally given as:

$$\langle \Rightarrow_{\mathcal{L}}, \overset{?}{\overline{\Gamma; \Xi_0 : A \Rightarrow B \vdash C}} \rangle \rightarrow \frac{\overset{?}{\overline{\Gamma \vdash X_1 : A}} \qquad \overset{?}{\overline{\Gamma; \Xi_1 : B \vdash C}}}{\Gamma; X_1 \cdot \Xi_1 : A \Rightarrow B \vdash C}$$

adding the monad constructors, this can be extended to:

$$\langle \Rightarrow_{\mathcal{L}}, \overset{?}{\overline{\Gamma; \Xi_0 : A \Rightarrow B \vdash C}} \rangle \rightarrow \textit{subgoals}\ 2\ \frac{\overset{?}{\overline{\Gamma \vdash X_1 : A}} \qquad \overset{?}{\overline{\Gamma; \Xi_1 : B \vdash C}}}{\Gamma; X_1 \cdot \Xi_1 : A \Rightarrow B \vdash C}$$

$$\langle \Rightarrow_{\mathcal{L}}, \overset{?}{\overline{\Gamma \vdash C}} \rangle \rightarrow \textit{exception}\ \texttt{Not an implication}$$

Then this definition is again extended to deal with any number of current goals: if there are k current goals, then either $\Rightarrow_{\mathcal{L}}$ applies to all of them, and *subgoals* $2k$ is generated. Else the *exception* constructor is used.

The following aside allows us to rest the difference between programming and interactive commands on a mathematical characterization of their semantics.

Definition 4.4.6 (Programming commands characterization)**.** For any programming command c such that $\langle c, x[\alpha] \rangle = \textit{subgoals}\ b\ y[\beta]$, then $\alpha = \beta$. In other terms, the programming commands do not modify the position of the current index.

Definition 4.4.7 (Interactive commands characterization)**.** Interactive commands are the only constructs of a proof language that can modify the position of the current index in the proof.

Definition 4.4.8 (Semantics of strategies)**.** Figures 4.1 and 4.2 complete the definition of the evaluation function for strategies. Remark that the monadic operators *unit* and $\star$ are exact denotations for the "`idtac`" and ';' strategies. An intermediary function *fold* is used to inductively define the semantics of the list evaluation. The monadic operator *map* is also implicitly used to implement "`postpone`". In fact any trivial operation on proofs (renaming of formulas, changing their order, etc.) of type $\tau \rightarrow \tau$ can be lifted to $\tau \rightarrow \mathcal{M}\,\tau$ using the *map* operator.

$$
\begin{array}{l}
\langle \texttt{idtac}, \mathtt{x}[\alpha]\rangle = \mathit{unit}\,\mathtt{x}[\alpha] \\
\langle \mathtt{c}_1;\mathtt{c}_2, \mathtt{x}[\alpha]\rangle = \langle \mathtt{c}_1, \mathtt{x}[\alpha]\rangle \star \mathtt{c}_2 \\
\langle [\mathtt{c} :: \mathtt{l}], \mathtt{x}[\alpha]\rangle \to \mathit{fold}_{\alpha,2}\ \mathtt{l}\ \langle \mathtt{c}, \mathtt{x}[\downarrow_1 \alpha]\rangle
\end{array}
$$

with

$$
\begin{array}{l}
\mathit{fold}_{\alpha,\mathtt{i}}\ \mathit{nil}\ \mathtt{m} \to \text{match } \mathtt{m} \text{ with} \\
\qquad \left|\begin{array}{lcl} \mathtt{x}[\mathit{subgoals}\ \mathtt{b}\ \beta] & \mapsto & \mathtt{x}[\mathit{subgoals}\ \mathtt{b}\ \alpha] \\ * & \mapsto & \mathtt{m} \end{array}\right. \\
\mathit{fold}_{\alpha,\mathtt{i}}\ \mathtt{c} :: \mathtt{l}\ \mathtt{m} \to \text{match } \mathtt{m} \text{ with} \\
\quad \left|\begin{array}{lcl}
\mathtt{x}[\mathit{subgoals}\ \mathtt{b}\ \beta] & \mapsto & \text{match } \langle \mathtt{c}, \mathtt{x}[\downarrow_{\mathtt{i}} \alpha]\rangle \text{ with} \\
\quad \left|\begin{array}{lcl} \mathtt{y}[\mathit{subgoals}\ \mathtt{b}'\ \gamma] & \mapsto & \mathit{fold}_{\alpha,\mathtt{i}+1}\ \mathtt{l}\ \mathtt{y}[\mathit{subgoals}\ (\mathtt{b} \oplus \mathtt{b}')\ \gamma] \\ \mathtt{y}[\mathit{success}] & \mapsto & \mathit{fold}_{\alpha,\mathtt{i}+1}\ \mathtt{l}\ \mathtt{y}[\mathit{subgoals}\ \mathtt{b}\ \beta] \\ \mathit{exception}\ \mathtt{s} & \mapsto & \mathit{exception}\ \mathtt{s} \end{array}\right. \\
\mathtt{x}[\mathit{success}] & \mapsto & \mathit{fold}_{\alpha,\mathtt{i}+1}\ \mathtt{l}\ \langle \mathtt{c}, \mathtt{x}[\downarrow_{\mathtt{i}} \alpha]\rangle \\
\mathit{exception}\ \mathtt{s} & \mapsto & \mathit{exception}\ \mathtt{s}
\end{array}\right.
\end{array}
$$

Figure 4.1: Semantics of the programming commands

$$
\begin{array}{l}
\langle \texttt{postpone}, \mathtt{x}[\alpha]\rangle = \mathit{subgoals}\ \texttt{false}\ \mathtt{x}[\rightsquigarrow \alpha] \\
\qquad \langle \mathtt{c}_1.\mathtt{c}_2, \mathtt{x}[\alpha]\rangle = \text{match } \langle \mathtt{c}, \mathtt{x}[\alpha]\rangle \text{ with} \\
\qquad\qquad \left|\begin{array}{lcl}
\mathtt{y}[\mathit{subgoals}\ \mathtt{b}\ \beta] & \mapsto & \langle \mathtt{c}_2, \mathtt{y}[\downarrow_1 \beta]\rangle \\
\mathtt{y}[\mathit{success}] & \mapsto & \langle \mathtt{c}_2, \mathtt{y}[\rightsquigarrow \alpha]\rangle \\
\mathit{exception}\ \mathtt{s} & \mapsto & \mathit{exception}\ \mathtt{s}
\end{array}\right.
\end{array}
$$

Figure 4.2: Semantics of the interactive commands

$$\langle \texttt{spread}\ \mathtt{l}, \mathtt{x}[\alpha] \rangle = \text{if length } \mathtt{l} > \text{card } \alpha$$
$$\text{then } \langle [\text{truncate (card } \alpha) \text{ l}], \mathtt{x}[\alpha] \rangle$$
$$\text{else if length } \mathtt{l} < \text{card } \alpha$$
$$\text{then } \langle [\text{extend (card } \alpha) \text{ l}], \mathtt{x}[\alpha] \rangle$$
$$\text{else } \langle [\mathtt{l}], \mathtt{x}[\alpha] \rangle \ ,$$

where

- length $\mathtt{l}$ and card α return respectively the length of the list $\mathtt{l}$ and the number of open goals in α.
- truncate $\mathtt{n}$ $\mathtt{l}$ and extend $\mathtt{n}$ $\mathtt{l}$ respectively truncate the list $\mathtt{l}$ at length $\mathtt{n}$ and extend the list $\mathtt{l}$ to length $\mathtt{n}$, using its last element as a filler.

Figure 4.3: A hardened list strategy

Note. The list operator [.] is a programming command that seems to modify the current index. However, it is easy to demonstrate that this modification is only internal to the evaluation of the construct, and that globally it does not alter the position of the current index.

Note. The list operator [.] does not handle well cases where the number of open goals and the number of elements in the list are not equal. In implementations of this strategy where such a mismatch can occur, tests are added to detect and correct these cases. Figure 4.3 shows the semantics of such a strategy, called "`spread`": if there are less goals than elements in the list, then truncate the list; if there are more then apply the last element of the list to the all the additional goals. This treatment of special cases can also be easily applied to generate hardened versions of the commands "`postpone`" and '.'.

This prototype proof language and its formal semantics can easily be extended with exception constructors and destructors (*e.g.*, `throw` and `catch`), progress testers (*e.g.*, `orelse`) and more. Most of these extensions have been included in the implementations we made of this concept. But in fact, we argue that the result of this formalization of the proof monad is stronger than this: it allows for the formal characterization of a language of strategies. However in order to get to this characterization, we need to be able to consider the $\mathtt{n}+1$-ary strategy $\mathtt{c}_0; [\mathtt{c}_1 :: \ldots :: \mathtt{c}_\mathtt{n} :: \mathit{nil}]$ as an implementation of a monadic composition operator.

Conjecture 4.4.9. There exists an extension of the monadic operator $\star$,

noted $\star_n$ that respects the monad laws and is n-ary on the right. The monad laws for this construction are the left and right unit:

$$\begin{aligned} \texttt{idtac} \star_n (c_1, \ldots, c_n) &= [c_1 :: \ldots :: c_n :: nil] \\ c \star_n (\texttt{idtac}, \ldots, \texttt{idtac}) &= c \end{aligned}$$

and associativity:

$$c_0 \star_n (c_0^1 \star_n (c_1^1, \ldots, c_{n_1}^1), \ldots, c_0^m \star_n (c_1^m, \ldots, c_{n_m}^m)) = c_0 \star_n (c_0^1, \ldots, c_0^m) \star_n (c_1^1, \ldots, c_{n_m}^m)$$

Corollary 4.4.10. *Given such an extension, the triple* $(\mathcal{M}, \texttt{idtac}, \star_n)$ *is a monad.*

Conjecture 4.4.11. A strategy language consists in a programming language combined with a proof monad $(\mathcal{M}, \texttt{idtac}, \star_n)$, and a monadic datatype destructor such as a `match` or a `case`.

Finally, we make the link with the proof-based formalism of chapter 3. First, remark that the system presented in this chapter can be considered as a rewriting system, relying on a congruence rule to achieve in-depth reductions. The commands "`idtac`" and ';' are still linked to the identity and sequence of the formalism, and it is easy to overload the definition of the evaluation symbol $\langle , \rangle$ in order to have $\langle c, m \rangle = m \star c$, thus recovering the rule $\langle c_1; c_2, x[\alpha] \rangle = \langle c_2, \langle c_1, x[\alpha] \rangle \rangle$.

What is more, since we have shown that the semantics for tactics can be derived from their simple expression as rewriting rules on local sequents, it is possible to refer directly to these simplified rules when dealing with the semantics of tactics. This is an illustration of the modularity of the combination of the two frameworks: for tactics, simple rules are enough, and for strategies the addition of the proof monad provides a rich and capable formalism.

4.5 The proof monad in PVS

PVS has an extensive proof language (Shankar et al., 1999). Initial attempts to formalize its semantics (Archer et al., 2003), although incomplete, have highlighted its complexity. In particular, it uses two kinds of exceptions, called *failure* and *backtrack*. In (Kirchner and Muñoz, 2006), we have proposed formal semantics for a large part of this language, based on a monadic mathematical structure. The theory of the proof monad is a direct extension of this work, and the results obtained in (Kirchner and Muñoz, 2006) can be easily translated into this formalism.

The results of the formalization of the semantics of PVS's strategies are twofold: first, the formalization of the proof structure allowed for a better

documentation of PVS's proof language. A number of commands, testing the state of the prover, have been written to aid the user in controlling the prover's power. For instance we have implemented the monadic datatype destructor as a strategy named `test`, that simulates the application of an command on the proof state. Depending on the outcome of this masquerade, it applies one of it five remaining arguments.

syntax: `(test` *`step step1 step2 step3 step4 step5`*`)`

usage:
```
(test (grind)
    (skip-msg "grind will generate subgoals")
    (skip-msg "grind won't do anything")
    (skip-msg "grind will prove the goal")
    (skip-msg "grind will fail")
    (skip-msg "grind will backtrack"))
```

errors: No error messages other than those generated by the given commands.

Second, a streamlined proof language was designed based on the proof structure, with the aim of providing commands with very elements semantics to the user. This language, called PVS#, has been implemented as a PVS package, and is available to download from the Internet.

4.6 Related work

Delahaye (Delahaye, 2000) made the first attempt to formalize the concept of a proof language. He proposed a prototyped proof language called $\mathcal{L}_{pdt}$, and gave its formal big-step semantics. However, these semantics were coupled with the semantics of lower-level tactics, making it difficult to abstract the principles of his design from his implementation. Using the inherited LCF language as a basis, Delahaye also enriched Coq's proof language with modest yet powerful programming constructs, and documented this work using informal big-step semantics.

Jojgov (Jojgov, 2004) followed in the steps of Delahaye, using a notion of parametrized metavariables to describe unproved branches in incomplete proofs. He proposed small-step operational semantics for proof languages, but he did not recognize the modularity of the formalisms for tactics and strategies: as a result, the rules in his framework have to deal with whole proofs. What is more, the case of the sequence and identity strategies are only mentioned as an aside, and the semantics are focussed on the case of tactics.

Martin and Gibbons (Martin and Gibbons, 2002) in an unpublished note remarked that Angel's proof language had a monadic structure, and generalized their observations to a generic proof language. While their note is

based on the same intuition as our work, their development had two drawbacks: first, they intentionally avoided any description of a proof datatype, only mapping proof language constructs to monadic operators. Second, their proof language was quite minimal, and their formalism was not easily extendible to nowadays developments. On the other hand, we wanted to give an insight into which structures were *specifically* necessary and sufficient to implement or describe a modern proof language.

Finally, rewriting strategies (Cirstea et al., 2003) use some state information very similar to our monadic structure to provide means to program strategies. Moreover, the structure of proof trees is quite similar to the structure of the majority of the rewriting system targets, *i.e.* terms. On the down side, rewriting strategy languages are often quite simple compared to proof languages: for instance, there are no interactive strategies in rewriting languages.

5 A Typing system for Proof Languages

Another aspect of the dialog between the proof engine and the scientist lies in the discrimination of well-formed inputs, or proof scripts. *In this chapter we aim at providing an additional formal foundation for proof languages, in the form of a typing system.*

A first application of the semantics exposed in the two previous chapters is to derive a type-safety result for proof languages. This is obtained by providing a typing mechanism for proof languages, and the definition of the type of tactics and strategies entails brings us to study the type of the proofs they manipulate. In a sense, the raw types for proofs and tactics provided in the proof monad formalism implicitly announced the system of types presented in this chapter: the latter can be seen as a refinement of the former.

The type system presented here guarantees that proof commands apply to, and generate sequents where the active formula has the indented form. Thus it does more than just ensure that proof commands operate on proofs, but it does not seek to ensure that a proof script will correctly close a give set of goals — this role is left to the semantical formalism.

In this chapter, we incrementally build a typing system for proofs and proof commands: after a simple example and the definition of the form of types, we define well-formed typing judgements for indexed proofs, and we work our way through tactics, strategies, and application and abstraction over these constructs. We conclude by using the semantics of the previous chapters to demonstrate a type-safety result.

$\star$

An important question with proof languages is whether a certain class of proof scripts behaves correctly. This class is formed by well-typed scripts in a given typing system, and their correct behaviour is called *type safety*. In the case of programming languages, this topic has been addressed since the last century with the first typing systems and typed languages (Whitehead and Russell, 1910; ANSI, 1966, 1996; Leroy et al., 2003; McBride, 2004). For instance, the type of the list concatenation operation:

$$\texttt{append} : \texttt{list}(\mathtt{n}) \to \texttt{list}(\mathtt{m}) \to \texttt{list}(\mathtt{n} + \mathtt{m})$$

ensures both that `append` manipulates finite lists, and not say integers, and that the resulting list has a correct length. As a result, if the development of a program includes a type-checking pass, then during that phase parts of that program that break the previous properties can be quickly identified and corrected, thus not requiring a tedious (and potentially incomplete) debugging process. Hence proof systems provide an additional level of both safety and comfort for the programmer.

But in for proof languages, no such system exists yet, perhaps due to the relative simplicity of the proof scripts written up to recently. However, the

development of a community of strategy programmers (Archer et al., 2006) has created a demand for the additional comfort provided by a typing system in programming with proof languages. For instance, we want to detect that the tactic that corresponds to the $\text{cut}_{\mathcal{R}}$ rule generates two subgoals, and that as a consequence it cannot be chained within a strategy that assumes that it will return exactly three subgoals. We also want to detect more fine-grained errors, such as the one triggered by the following proof script:

$$\text{cut}_{\mathcal{R}}\ x\ (A \Rightarrow B)\ ;\ [\ \texttt{true}_{\mathcal{R}}\ |\ \texttt{idtac}\ ]$$

where the right introduction rule of the `true` connective cannot take place, since the sequents generated by the $\text{cut}_{\mathcal{R}}$ rule have $A \Rightarrow B$ as an active formula.

5.1 Types for tactics: a simple example.

Let us give a taste of what's ahead with an example based on two tactics exposed in section 3.6: $\Rightarrow_{\mathcal{R}}$ and $\Rightarrow_{\mathcal{L}}$. Recall the local semantics of these commands:

$$\frac{?}{\Gamma \vdash X_0 : A \Rightarrow B} \xrightarrow{\Rightarrow_{\mathcal{R}}\ x} \frac{\dfrac{?}{\Gamma, x : A \vdash X_1 : B}}{\Gamma \vdash \lambda x^A.X_1 : A \Rightarrow B}$$

$$\frac{?}{\Gamma, \Xi_0 : A \Rightarrow B \vdash C} \xrightarrow{\Rightarrow_{\mathcal{L}}} \frac{\dfrac{?}{\Gamma \vdash X_1 : A} \qquad \dfrac{?}{\Gamma; \Xi_1 : B \vdash C}}{\Gamma; X_1 \cdot \Xi_1 : A \Rightarrow B \vdash C}$$

In the first rule, the evaluation of $\Rightarrow_{\mathcal{R}}$ takes place on a sequent containing a metavariable X_0 of the expected type, and return a structure where X_0 has been instantiated, and a new metavariable X_1 has been introduced as part of the instantiated proof term. The type of the metavariable X_0 determines whether the tactic applies or not; the type of X_1 corresponds to the new active formula, and can be seen as the result of the tactic. The same reasoning applies in the second rule, except that in this case two metavariables are generated: the result of the tactic in this case is the couple (X_1, Ξ_1).

Hence, using the symbol $\rightarrow$ as a constructor for functional tactic types, the idea is to write:

$$\Rightarrow_{\mathcal{R}} x : \overbrace{(A \Rightarrow B)}^{X_0} \rightarrow \underbrace{B}_{X_1}$$

$$\Rightarrow_{\mathcal{L}} : \overbrace{(A \Rightarrow B)}^{\Xi_0} \rightarrow (\underbrace{A}_{X_1}, \underbrace{B}_{\Xi_1})$$

This example however is too simplistic, as the typing system needs to be enriched to handle a bit more information:

- the type of the tactic's parameter is a fundamental element of the tactic's behaviour. For instance, the fact that $\mathtt{x} : \mathsf{A}$ in `intro x`. As such, it is very desirable to incorporate it in the typing judgement;
- in general, a metavariable in an open proof term is grafted with another open proof term, which in turn may contain a number of metavariables. Hence, more than one metavariable can be created in a semantic step;
- this example takes place on a local modification of the proof. In order to consider the typing of global commands, such as strategies, the types need to take into account that there are possibly several current goals;
- the types have to take into account the type of the monadic computation. Since in chapter 4 we have defined the type of the proof monad depending on the type of indexed proofs, all we need is the latter.

5.2 Proof elements and values

We illustrate our purpose using the proof language PRF, as presented in definition 4.4.1. Recall the grammar of the language:

$$\begin{aligned}\mathsf{c} ::= {} & \mathrm{ax}_{\mathcal{L}}\ \mathsf{x}^{\mathsf{A}} \mid \mathrm{ax}_{\mathcal{R}}\ \mathsf{x}^{\mathsf{A}} \mid \mathrm{true}_{\mathcal{R}} \mid \Rightarrow_{\mathcal{L}} \mid \Rightarrow_{\mathcal{R}}\ \mathsf{x}^{\mathsf{A}} \mid \mathrm{cut}_{\mathcal{L}}\ \mathsf{x}^{\mathsf{A}}\ \mathsf{B} \mid \mathrm{cut}_{\mathcal{R}}\ \mathsf{x}^{\mathsf{A}}\ \mathsf{B} \\ & \mid \mathtt{postpone} \mid \mathsf{c}.\mathsf{c} \\ & \mid \mathtt{idtac} \mid \mathsf{c}; \mathsf{c} \mid [\mathsf{c}_{\mathrm{list}}] \\ \mathsf{c}_{\mathrm{list}} ::= {} & \mathit{nil} \mid \mathsf{c}::\mathsf{c}_{\mathrm{list}}\end{aligned}$$

where x and B are respectively a variable and a well-formed formula of $\mathcal{L}^1_{\mathrm{m}}$. Remark that we use Church-like notations to embed the types of the tactic variable parameter into their syntax. The tactics correspond to the inference rules of minimalistic logic, *as per* section 3.6. The objects manipulated by PRF are the indexed proofs and the proof monad as defined in chapter 4. Also as in chapter 4, we note application with the symbols $\langle\,,\rangle$. To complement it, we explicit the notion of abstraction, for which we use the traditional λ binder.

Definition 5.2.1 (Proof elements)**.** We define *proof elements*, and we note ϵ as the compilation of indexed proof trees, proof commands, and abstractions and applications operator.

$$\epsilon ::= \mathsf{x} \mid \mathsf{x}[\alpha] \mid \mathsf{c} \mid \lambda \mathsf{x}^{\mathsf{T}}.\epsilon \mid \langle \epsilon_1, \epsilon_2 \rangle$$

where T is the type of a proof element, defined a bit later. Note that one shouldn't confuse constants for proof trees $\mathsf{x}[\alpha]$ with proof element variables x.

Note. Proof states as used in chapters 3 and 4 are a subset of proof elements, and the application symbol $\langle\,,\rangle$ is a generalization of the application for proof

states. In fact, the definition of proof elements regroups into one category all the objects that we want to type in our system.

Definition 5.2.2 (Substitution of proof elements)**.** We define a notion of implicit substitution on proof elements, *i.e.* the replacement of a proof elements variable modulo α-conversion. We note $\epsilon_1\{x \leftarrow \epsilon_2\}$ the element ϵ_1 where the variable x is replaced by the element ϵ_2, without capture.

Definition 5.2.3 (Values)**.** Standalone indexed proof trees are called *values*, together with unapplied abstractions and proof commands.

$$\epsilon ::= x[\alpha] \mid c \mid \lambda x^{T}.\epsilon$$

5.3 Types and typing judgements

Based on these remarks, we introduce a framework to assign types to proof commands. We proceed gradually, by first assigning types to open goals, and then working our way up to the types of higher-order proof commands:

- the type of an open goal, *i.e.* a sequent whose active formula A is labelled by a metavariable, is the formula A. The syntax of well-formed formulas is given by figure 1.1;
- the type of a list of current goals, *i.e.* a list of open goals, is built with the list constructors $\{\}$. If $A_1, \ldots, A_j$ are the types of the j current goals, then globally their type is $\{A_1, \ldots, A_j\}$;
- the type of indexed proofs, denoted τ in chapter 4, is build by combining the type of its current goals and the type of its other open goals, using the constructor $[]$. Given a proof with n open goals, if $A_{i+1}, \ldots, A_{i+j}$ are the types of the j current goals, $A_1, \ldots, A_i$ are the types of the first i open goals and $A_{i+j+1}, \ldots, A_n$ the types of its last open goals, then the type of the whole indexed proof is denoted:

$$[A_1, \ldots, A_i, \{A_{i+1}, \ldots, A_{i+j}\}, A_{i+j+1}, \ldots, A_n]$$

- the type of monadic constructions respects the definition 4.4.2: given an indexed proof tree of type τ, a monadic computation has type $\mathcal{M}\,\tau$;
- the type of tactics reflects their condition as functions on indexed proofs. Using the type operator for functional abstraction $\rightarrow$, we write $\tau \rightarrow \mathcal{M}\,\tau$ the type of a tactic that maps an indexed proof of type τ into a computation of type $\mathcal{M}\,\tau$;
- the type of strategies, *i.e.* functions on tactics, is built either as the type of a tactic, or as an arrow type $\rightarrow$.

To sum things up, the following condenses the aforementioned comments into one definition:

Definition 5.3.1 (Proof language type)**.** The type T of a proof state construct is the sum of the types of indexed proofs, tactics and strategies:

$$\begin{aligned}
\tau &::= [A_1, \ldots, A_i, \{A_{i+1}, \ldots, A_j\}, A_{j+1}, \ldots, A_n] \\
\mathsf{T_t} &::= \tau \to \mathcal{M}\,\tau' \\
\mathsf{T_s} &::= \mathsf{T_t} \mid \mathsf{T_s} \to \mathsf{T_s'} \\
\mathsf{T} &::= \tau \mid \mathsf{T_t} \mid \mathsf{T_s}
\end{aligned}$$

where $A_1, \ldots, A_n$ are well-formed formulas of $\mathcal{L}^1_m$.

We can then proceed to give the definition of proof language typing judgements.

Definition 5.3.2 (Typing judgements)**.** We write $\Gamma \Vdash \epsilon : \mathsf{T}$ to state that the proof element ϵ has type T in the context Γ.

Example 5.3.3. The typing judgements for the two tactics $\Rightarrow_{\mathcal{R}}$ and $\Rightarrow_{\mathcal{L}}$, used as examples at the beginning of this section, write:

$$\begin{aligned}
\Vdash \Rightarrow_{\mathcal{R}} x^A &: [\ldots\{(A \Rightarrow B), \ldots, (A \Rightarrow B)\}\ldots] \to \mathcal{M}\,[\ldots, \{B, \ldots, B\}\ldots] \\
\Vdash \Rightarrow_{\mathcal{L}} &: [\ldots\{(A \Rightarrow B), \ldots, (A \Rightarrow B)\}\ldots] \to \mathcal{M}, [\ldots, \{A, B, \ldots, A, B\}\ldots]
\end{aligned}$$

Note. We introduce a different kind of inference rules which are double-lined, and a new typing judgement $\Vdash$, in order to clearly separate the logical and proof element typing realms. An exception, the symbol ':' is quite overloaded, being used as it is in the typing of elements of $\mathcal{L}^1_m$, proof terms and proof commands. However, the form of inference rules, typing judgements, and types being clearly different, no need was felt to introduce a new notation here.

Note. Already we see that, as was the case with the semantics of tactics, there is some redundancy in specifying tactics over the whole proof. Indeed types of tactics can be defined locally, and extended automatically to the complete proof when needed.

Notation. For clarity and when unambiguous, the types of proof elements are noted as arrows mapping the type of a single open goal to the type of a list of current goals. Hence the previous example can be written, without loss of generality, in a more readable format:

$$\begin{aligned}
\Vdash \Rightarrow_{\mathcal{R}} x^A &: \{A \Rightarrow B\} \to \mathcal{M}\{B\} \\
\Vdash \Rightarrow_{\mathcal{L}} &: \{A \Rightarrow B\} \to \mathcal{M}\{A, B\}
\end{aligned}$$

As we will see later, this simplification cannot be applied to typing judgements for interactive proof commands, because they change the current set of goals.

Note. The types of the proof elements are implicitly polymorphic. For instance, the aforementioned type of the tactic $\Rightarrow_{\mathcal{R}}$ would, explicitly, write:

$$\Vdash \Rightarrow_{\mathcal{R}} x^{A} : \forall A, B : \text{bool}.\{A \Rightarrow B\} \to \mathcal{M}\{B\}$$

Definition 5.3.4 (Types of pre-defined proof commands)**.** To each proof command is associated a type T. We call † the mapping from proof commands to types. For instance, we have: $(\Rightarrow_{\mathcal{R}} x)^{\dagger} = \{A \Rightarrow B\} \to \mathcal{M}\{B\}$.

Note. In practice, this means that a proof languages such as PRF is presented in the form of a typed signature.

We now state the type inference rules for proof elements, *i.e.* the rules that allow for well-formed typing judgements. We proceed gradually, starting with the types of indexed proof, then tactics, strategies, and finally proof elements in general.

5.4 Type inference rules: indexed proofs

Definition 5.4.1. Well-formed typing judgements for indexed proofs write:

$$\frac{}{\Gamma \Vdash x[\alpha] : [A_1, \ldots, A_i, \{A_{i+1}, \ldots, A_j\}, A_{j+1}, \ldots, A_n]}\ C$$

with C a side condition that states that $A_1, \ldots, A_n$ are the types of the metavariables in the well-formed proof x, and more specifically $A_{i+1}, \ldots, A_j$ are the types of the metavariables in the current goals.

Example 5.4.2. For instance if we chose a proof term representation for proofs where we underline the subterm corresponding to the active subtree, then the following typing judgement is well-formed:

$$\frac{}{\Vdash \lambda x^{A}.\lambda y^{A \Rightarrow B}.\mu z^{B}.\langle X_0 \| \underline{x \cdot \Xi_0} \rangle : [(A \Rightarrow B), \{B\}]}\ C$$

with C being: $\lambda x^{A}.\lambda y^{A \Rightarrow B}.\mu z^{B}.\langle X_0 \| x \cdot \Xi_0 \rangle$ is a well-formed proof term, and X_0 and Ξ_0 are of type respectively $A \Rightarrow B$ and B.

Note. The side-condition of the inference rule can be verified by using the logical proof derivation rules of minimalistic logic with particular rules for metavariables. If the derivation is complete, then the side-condition is verified. To achieve this we need to add two inference rules to the logical system of minimalistic logic:

$$\text{amv}_{\mathcal{R}}\ \frac{}{\Upsilon, \underline{X_0} : A \mid \Gamma \vdash \underline{X_0} : A} \qquad \text{amv}_{\mathcal{L}}\ \frac{}{\Upsilon, \underline{\Xi_0} : A \mid \Gamma; \underline{\Xi_0} : A \vdash B}$$

and for any non-metavariable term, the underlining is discarded by the inference rules. The other logical frameworks (LK, LJ and their minimal counterparts) can be extended similarly. For instance, in example 5.4.2, the side condition amounts to the deduction:

$$2\times\Rightarrow_{\mathcal{R}}\ \dfrac{\mathrm{cut}_{\mathcal{R}}\ \dfrac{\mathrm{mv}\ \dfrac{}{\Upsilon\mid\Gamma'\vdash X_0:A\Rightarrow B;z:B}\qquad\pi}{\Upsilon\mid\Gamma'\vdash\mu z^{B}.\langle X_0\|\underline{x\cdot\Xi_0}\rangle:B}}{\Upsilon\mid\Gamma\vdash\lambda x^{A}.\lambda y^{A\Rightarrow B}.\mu z^{B}.\langle X_0\|\underline{x\cdot\Xi_0}\rangle:A\Rightarrow((A\Rightarrow B)\Rightarrow B)}$$

with $\pi =$

$$\Rightarrow_{\mathcal{L}}\ \dfrac{\mathrm{ax}_{\mathcal{R}}\ \dfrac{}{\Upsilon\mid\Gamma'\vdash x:A;z:B}\qquad\mathrm{amv}\ \dfrac{}{\Upsilon\mid\Gamma';\underline{\Xi_0}:B\vdash z:B}}{\Upsilon\mid\Gamma';\underline{x\cdot\Xi_0}:A\Rightarrow B\vdash z:B}$$

where $\Gamma = A, B : \mathtt{bool}$, $\Gamma' = \Gamma, x : A, y : A\Rightarrow B$ and $\Upsilon = X : A\Rightarrow B, \underline{\Xi_0} : B$. Remark that a possible way of presenting this information would be to stack the logical derivation on top of the indexed proof typing derivation.

5.5 Type inference rules: tactics

Definition 5.5.1. Assume given a parametrized tactic $\mathbf{t}$, that has some parameters $x_1,\dots,x_p$ that are fresh elements the proof. Assume that $\mathbf{t}$ instantiates a metavariable X_0 of type A_0 with a proof term containing the metavariables $X_1,\dots,X_n$ with types *resp.* $A_1,\dots,A_n$. In other words, $\mathbf{t}^\dagger = A_0\to\{A_1,\dots,A_n\}$. A well-formed typing judgement for this tactic is written:

$$\dfrac{}{\Gamma\Vdash\mathbf{t}:\mathbf{t}^\dagger}\ \rhd x_1:P_1\dots\rhd x_p:P_p$$

As with the side condition for indexed proof type inferences, the parameter well-formation judgements can be stacked on top of the tactic type inference rule:

$$\dfrac{\rhd x_1:P_1\dots\rhd x_p:P_p}{\Gamma\Vdash\mathbf{t}:\mathbf{t}^\dagger}$$

These judgements are then derived using the rules of figure 1.1.

Example 5.5.2. We develop the typing rules for the base tactics of PRF, which entails extending the † mapping to deal with all of them. Let us begin with the implicative tactics:

$$\dfrac{}{\Gamma\Vdash\Rightarrow_{\mathcal{L}}:\{A\Rightarrow B\}\to\mathcal{M}\{A,B\}}\qquad\dfrac{}{\Gamma\Vdash\Rightarrow_{\mathcal{R}}\ x^{A}:\{A\Rightarrow B\}\to\mathcal{M}\{B\}}$$

The $\Rightarrow_{\mathcal{L}}$ tactic takes no parameter, and instantiates a metavariable of type $A\Rightarrow B$ with a proof term containing two metavariables of type A and B. The $\Rightarrow_{\mathcal{R}}$ tactic instantiates a metavariable of type $A\Rightarrow B$ by a proof term containing a metavariable of type B. The axiom and truth tactics type inference rules write:

$$\dfrac{\dfrac{}{\Gamma\Vdash\mathrm{ax}_{\mathcal{L}}\ x^{A}:\{A\}\to\mathcal{M}\{\}}\qquad\dfrac{}{\Gamma\Vdash\mathrm{ax}_{\mathcal{R}}\ x^{A}:\{A\}\to\mathcal{M}\{\}}}{\Gamma\Vdash\ltimes_{\mathcal{R}}:\{\mathtt{true}\}\to\mathcal{M}\{\}}$$

where the empty result list {} signals that the corresponding rules do not generate any new metavariables. The two axiom rules specify that the metavariable in the sequent is instantiated and no other metavariable is introduced. Since their parameter x designates a formula in the proof, it is not checked. The truth rule simply instantiates any metavariable of type `true` with the constant $\ltimes$. Finally well-formed types of the cut rules are written:

$$\frac{\rhd x : A \qquad \rhd B : \mathrm{bool}}{\Gamma \Vdash \mathrm{cut}_{\mathcal{L}}\ x^A\ B : \{A\} \to \mathcal{M}\{B, B\}} \qquad \frac{\rhd \alpha : A \qquad \rhd B : \mathrm{bool}}{\Gamma \Vdash \mathrm{cut}_{\mathcal{R}}\ \alpha^A\ B : \{A\} \to \mathcal{M}\{B, B\}}$$

Both rules verify that their parameter variable and formula are well-formed, and replace the metavariable in the current goal with two new metavariables whose type is the new formula.

This example can be extended to tactics for quantifiers:

$$\frac{\rhd t : A}{\Gamma \Vdash \forall_{\mathcal{L}}\ t : \{\forall x^A.B\} \to \mathcal{M}\{B[x \leftarrow t]\}} \qquad \frac{}{\Gamma \Vdash \forall_{\mathcal{R}}\ x^A : \{\forall x^A.B\} \to \mathcal{M}\{B\}}$$

where the $\forall_{\mathcal{L}}$ tactic operates a substitution in the type of its metavariable, after having checked that the term used for the substitution is well-formed. $\forall_{\mathcal{R}}$ performs the introduction of the quantified variable, using the variable x which is already present in the proof. The type system can easily be extended to tactics for the other logical connectives of $\mathcal{L}^1_m$.

Example 5.5.3. Well-formed types for interactive commands require the use of the complete types of indexed proofs: they need to be able to express modifications in the position of the current index (and thus designate a new set of current goals in the proof). The typing rules for "`postpone`" follow:

$$\frac{}{\begin{array}{l}\Gamma \Vdash \mathtt{postpone} : \ [A_1, \ldots, A_i, \{A_{i+1}, \ldots, A_j\}, A_{j+1}, \ldots, A_n] \\ \qquad\qquad \to \mathcal{M}\,[A_1, \ldots, A_i, A_{i+1}, \ldots, A_j, \{A_{j+1}\}, A_{j+2}, \ldots, A_n]\end{array}}$$

We can already note that much of the simplicity of this typing system comes from the fact that there is always an active formula in the sequent. This liveliness property (proposition 2.2.5) allows the framework to focus on a single formula, instead of having to encode an entire sequent. In particular, since the number of succedent formulas in a sequent is not relevant in this system, it means that the typing rules for classical and intuitionistic tactics are identical. Presumably, this system could be extended to logical formalisms where there is no ambiguity on the active formula, *e.g.* intuitionistic natural deduction.

5.6 Type inference rules: strategies

Definition 5.6.1. Let $\mathbf{s}$ be an n-ary strategy, that combines proof commands of respective types $\forall i \in [1, \ldots, n], (\tau \to \mathcal{M}\tau)$. In other words, $\mathbf{s}^\dagger = (\tau \to \mathcal{M}\tau) \to \ldots \to (\tau \to \mathcal{M}\tau)$ (with n arrows). The well-formed type of the strategy $\mathbf{s}$ is given by the following inference rule:

$$\frac{}{\Gamma \Vdash \mathbf{s} : \mathbf{s}^{\dagger}}$$

Example 5.6.2. In this example we give the typing rules, using the simplified notation for indexed proof types, of the two bark strategies of PRF.

$$\frac{}{\Gamma \Vdash \texttt{idtac} : \{A\} \to \mathcal{M}\{A\}}$$

The `idtac` proof command does not modify any part of the proof: current goals are left untouched.

$$\frac{}{\Gamma \Vdash \; ; \; : \mathsf{T}_1 \to \mathsf{T}_2 \to \{A\} \to \mathcal{M}\{C_1^1, \ldots, C_n^m\}}$$

where $\mathsf{T}_1 = \{A\} \to \mathcal{M}\{B_1, \ldots, B_n\}$,
and $\mathsf{T}_2 = \{B_i\} \to \mathcal{M}\{C_i^1, \ldots, C_i^m\}$

The ';' sequencing tactic combines a tactic that for each current goal generates n subgoals, with a tactic that generates m subgoals for each of the $i \in [1, \ldots, n]$ subgoals. One can easily verify that this corresponds to the type of the *unit* and $\star$ monadic operators as defined in chapter 4. Finally, the list construction $[.:: \ldots ::.::\mathit{nil}]$ of length m has verifies the following type inference:

$$\frac{}{\Gamma \Vdash [.:: \ldots ::.::\mathit{nil}] : \mathsf{T}_1 \to \ldots \to \mathsf{T}_n \to \mathsf{T}}$$

where $\forall i \in [1, \ldots, m], \mathsf{T}_i = \{A^i\} \to \mathcal{M}\{B_1^i, \ldots, B_{n_i}^i\}$
and $\mathsf{T} = \{A_1, \ldots, A_m\} \to \mathcal{M}\{B_1^1, \ldots, B_{n_m}^m\}$.

The list construction maps a list of tactics to a same-sized collection of current goals, on a one-to-one basis.

Example 5.6.3. In this example we give the typing rules of the interactive strategy '.'.

$$\frac{}{\Gamma \Vdash \; . \; : \mathsf{T}_1 \to \mathsf{T}_2 \to \mathsf{T}}$$

where $\mathsf{T}_1 = [\ldots\{A_{i+1}, \ldots, A_j\}\ldots] \to \mathcal{M}\,[\ldots\{B_1, \ldots, B_m\}\ldots]$
and $\mathsf{T}_2 = [\ldots\{A_{j+1}\}\ldots] \to \mathcal{M}\,[\ldots\{C_1, \ldots, C_k\}\ldots]$
and $\mathsf{T} = [A_1, \ldots, A_i, \{A_{i+1}, \ldots, A_j\}, A_{j+1}, \ldots A_n] \to$
$\mathcal{M}\,[A_1, \ldots, A_i, B_1, \ldots, B_m, \{C_1, \ldots, C_k\}, A_{j+2}, \ldots, A_n]$

That is, '.' applies its first argument to the current goals, and then postpones them before applying its second argument.

Example 5.6.4. We conclude our review of strategic types with the well-formed type of the monadic datatype destructor "`match`".

$$\frac{}{\Gamma \Vdash \texttt{match . with} \;\Big|\; \begin{array}{l} \mathit{success} \mapsto . \\ \mathit{subgoals}\ b\ \alpha \mapsto . \quad : \mathsf{T}_1 \to \mathsf{T}_2 \to \mathsf{T}_3 \to \mathsf{T}_4 \to \mathsf{T}_5 \\ \mathit{exception}\ s \mapsto . \end{array}}$$

where $\forall i \in [1, \ldots, 5], \mathsf{T}_i = \{A_i\} \to \mathcal{M}\{B_i^1, \ldots, B_i^n\}$.

5.7 Type inference rules: application and abstraction

Definition 5.7.1. Well-formed types for abstraction and application are as given by th following inference rules:

$$\frac{\Gamma, x : T_1 \Vdash \epsilon : T_2}{\Gamma \Vdash \lambda x^{T_1}.\epsilon : T_1 \to T_2} \qquad \frac{\Gamma \Vdash \epsilon_1 : T_1 \to T_2 \qquad \Gamma \Vdash \epsilon_2 : T_1}{\Gamma \Vdash \langle \epsilon_1, \epsilon_2 \rangle : T_2}$$

Note. In particular, these rules hold for first-order application, *i.e.* the application of a tactic to a proof tree: take $T_1 = [\ldots\{A_1, \ldots, A_n\}\ldots]$, and $T_2 = \mathcal{M}[\ldots\{A_1^1, \ldots, A_1^m, \ldots, A_n^1, \ldots, A_n^m\}\ldots]$. They also cover the case of second-order application, *i.e.* the application of a strategy to a proof command, with $T_1 = \tau_1 \to \mathcal{M}\,\tau_1$ and $T_2 = \tau_2 \to \mathcal{M}\,\tau_2$.

Note. In the case of n-ary strategies, we take the liberty to condense the n application inferences into one.

Example 5.7.2. As a first example, take the first step of the typechecking derivation of the application of the sequence strategy to two tactics of PRF:

$$\frac{\Vdash ;\ : (\{A \Rightarrow B\} \to \mathcal{M}\{B\}) \to (\{B\} \to \mathcal{M}\{\}) \to \{A \Rightarrow B\} \to \mathcal{M}\{\} \qquad \Vdash \Rightarrow_{\mathcal{R}} x^A : T_1 = \{A \Rightarrow B\} \to \mathcal{M}\{B\} \qquad \Vdash \mathrm{ax}_{\mathcal{R}}\ y^B : T_2 = \{B\} \to \mathcal{M}\{\}}{\Vdash \Rightarrow_{\mathcal{R}}\ x^A\ ;\ \mathrm{ax}_{\mathcal{R}}\ y^B : \{A \Rightarrow B\} \to \mathcal{M}\{\}}$$

Note that the type of the strategy ';' can be inferred from the types of its arguments and of the result, making its typing judgement redundant. Hence, without loss of generality, we can simplify the derivation step to:

$$\frac{\Vdash \Rightarrow_{\mathcal{R}} x^A : \{A \Rightarrow B\} \to \mathcal{M}\{B\} \qquad \Vdash \mathrm{ax}_{\mathcal{R}}\ y^B : \{B\} \to \mathcal{M}\{\}}{\Vdash \Rightarrow_{\mathcal{R}}\ x^A\ ;\ \mathrm{ax}_{\mathcal{R}}\ y^B : \{A \Rightarrow B\} \to \mathcal{M}\{\}}$$

Example 5.7.3. A more elaborate example consists in typechecking one case of the definition of the `focus` extended tactic, defined as:

$$\texttt{focus}\ x^B\ y^A ::= \mathrm{cut}_{\mathcal{R}}\ y^A\ B\ ;\ [\mathrm{ax}_{\mathcal{R}}\ x^B\ |\ \texttt{idtac}]$$

if $x : B$ is a hypothesis, and when the active formula is in the succedent. Intuitively, this tactic consists in naming the active formula of the sequent y, and making hypothesis formula x active. The typing derivation, using the simplified notation introduced in the previous example, writes:

$$\frac{\dfrac{\dfrac{\rhd y : A \qquad \overline{\rhd B : \mathrm{bool}}}{\Vdash \mathrm{cut}_{\mathcal{R}}\ y^A\ B : \{A\} \to \mathcal{M}\{B, B\}} \qquad \dfrac{}{\Vdash \mathrm{ax}_{\mathcal{R}}\ x^B : \{B\} \to \mathcal{M}\{\}} \qquad \dfrac{}{\Vdash \texttt{idtac} : \{B\} \to \mathcal{M}\{B\}}}{\Vdash \mathrm{cut}_{\mathcal{R}}\ y^A\ B\ ;\ [\mathrm{ax}_{\mathcal{R}}\ x^B\ |\ \texttt{idtac}] : \{A\} \to \mathcal{M}\{B\}}}$$

The statement $\rhd B : \mathrm{bool}$ is verified because $x : B$ is a hypothesis of the current goal. Remark that the typing derivation would have failed if the type of the parameter x of the axiom rule did not match the parameter formula of the cut rule. Finally one can introduce the following "shortcut" for this typing derivation:

$$\frac{\rhd y : A}{\Vdash \texttt{focus}\ x^{B}\ y^{A} : \{A\} \to \mathcal{M}\{B\}}$$

The other cases of the definition of `focus` are typechecked similarly.

Definition 5.7.4. As in all type systems, the following rule is used to verify trivial typing judgements:

$$\frac{}{\Gamma, \epsilon : T \Vdash \epsilon : T}$$

Recall that types are implicitly polymorphic. As a consequence, we consider that this last rule also encompasses the case when the type on the right of the judgement is an instance of the type on the left.

Example 5.7.5. This last example illustrates the use of strategic abstraction. The definition of LCF's strategy `orelse`, that applies its first argument and then its second only if the first has failed to apply, can be formalized as:

$$\begin{aligned}\texttt{orelse} ::= \lambda \mathbf{s_1}^{T_1 \to T_1'}.\lambda \mathbf{s_2}^{T_2 \to T_2'}.&\texttt{match}\ \mathbf{s_1}\ \texttt{with}\\ &\mid \textit{success} \mapsto \mathbf{s_1}\\ &\mid \textit{subgoals}\ \texttt{true}\ \alpha \mapsto \mathbf{s_1}\\ &\mid \textit{subgoals}\ \texttt{false}\ \alpha \mapsto \mathbf{s_2}\\ &\mid \textit{exception}\ s \mapsto \mathbf{s_2}\end{aligned}$$

where $T_1 = \{A\}$, $T_1' = \mathcal{M}\{A_1, \ldots, A_n\}$, $T_2 = \{B\}$ and $T_2' = \mathcal{M}\{B_1, \ldots, B_m\}$. Remark that there is an implicit application of the `match` strategy to $\mathbf{s_1}$ and $\mathbf{s_2}$. In the following typing derivation, we note $\bigoplus(\mathbf{s_1}, \mathbf{s_1}, \mathbf{s_2}, \mathbf{s_2})$ the applied `match` construct, and we define $T_3 = \{C\}$, $T_3' = \mathcal{M}\{C_1, \ldots, C_k\}$ and $T_f = (T_1 \to T_1') \to (T_2 \to T_2') \to (T_3 \to T_3')$:

$$\dfrac{\dfrac{\dfrac{}{s_1 : T_1 \to T_1', s_2 : T_2 \to T_2' \Vdash \mathbf{s_1} : T_1 \to T_1'} \quad \dfrac{}{s_1 : T_1 \to T_1', s_2 : T_2 \to T_2' \Vdash \mathbf{s_2} : T_2 \to T_2'}}{s_1 : T_1 \to T_1', s_2 : T_2 \to T_2' \Vdash \bigoplus(\mathbf{s_1}, \mathbf{s_1}, \mathbf{s_2}, \mathbf{s_2}) : T_3 \to T_3'}}{\Vdash \lambda \mathbf{s_1}^{T_1 \to T_1'}.\lambda \mathbf{s_2}^{T_2 \to T_2'}.\bigoplus(\mathbf{s_1}, \mathbf{s_1}, \mathbf{s_2}, \mathbf{s_2}) : T_f}$$

5.8 The type safety property for proof languages

This typing system for proof commands can be combined with the structural operational semantics exposed in chapters 3 and 4, to derive type safety results. Indeed, the reduction relation $\to$ can be extended to deal with not only proof states, but all the proof elements as specified by definition 5.2.1. To the reduction rules of the proof-based semantics we add the β-reduction entailed by the use of the abstraction and application operations:

$$\langle \lambda x^{T}.\epsilon_1, \epsilon_2 \rangle \to \epsilon_2\{x \leftarrow \epsilon_1\}$$

Now we can state and demonstrate some of the usual results of type theory.

Proposition 5.8.1 (Preservation of type by reduction)**.** *Assume given a proof element* ϵ *and its reduct* ϵ' *such that* $\epsilon \to \epsilon'$*. If* ϵ *is well-typed and of type* T*, then* ϵ' *is also well-typed, and of type* T*.*

Proof. We need to prove: if $\Gamma \Vdash \epsilon : \mathsf{T}$ and $\epsilon \to \epsilon'$ then $\Gamma \Vdash \epsilon' : \mathsf{T}$. This is done by induction on the structure of ϵ, and by case. □

Proposition 5.8.2 (Progression)**.** *If a proof element* ϵ *is typable, then it is either a value, or there exists* ϵ' *such that* $\epsilon \to \epsilon'$*.*

Proof. By induction on the structure of σ, and by case. □

Proposition 5.8.3 (Type safety)**.** *If a proof element* ϵ *is typable and reduces in finitely many steps to an irreducible script* ϵ'*, then* ϵ' *is a value.*

Proof. Assume $\Gamma \Vdash \epsilon : \mathsf{T}$ By proposition 5.8.1, then $\Gamma \Vdash \epsilon' : \mathsf{T}$. And by proposition 5.8.2, ϵ' is a value. □

Intuitively, the type safety property for the application of a proof script to a proof guarantees that the execution of the script does indeed return the appropriate result.

Even at the level of simple scripts, it is easy to see the added programming comfort that this system supplies. The ongoing trend of using more complex strategies to program tactics using exceptions, side-effects and pattern matching will only make this feature more helpful.

The difficulty may reside in the presentation of this information to the proof programmer. Indeed, this formalism constitutes yet another type system in a domain that already counts a few of them, and could be a risk of confusion. We hope that the simplicity of the system presented in this chapter will contribute to alleviate this risk.

6 Interoperability

Here we demonstrate various ways to export proofs in first-order predicate logic into other proof frameworks.

Building upon the logical and semantical frameworks developed earlier in this manuscript, in this chapter we examine the problem of designing interoperability mechanisms towards a variety of proof systems. In particular, in accordance with the heterogeneity of the formal proof tool ecosystem, we propose several translation algorithms that take advantage of various characteristics of these systems. Thus we argue that our mechanism can fit most systems, regardless of their features and specificities.

Some of the translations in this chapter generate proof script results, and are illustrated on a case-per-case basis: we present translations towards Coq's and PVS's proof languages, although we conjecture that any other proof system can be adapted. An important fact is that the proof of correction of the translations constitute a major application of the semantical formalism for proof commands. Indeed, we give in this chapter the semantics of a subset of Coq's and PVS's proof languages.

In order to achieve the translation, the working (or *source*) framework retrained is first-order logic, as exposed in chapter 2. We provide a total of four translations, each of them more or less general, and more or less subject to the aforementioned critics, onto varying *target* frameworks. In particular, we expose and prove the correction of a translation towards a Frege-Hilbert style deduction system, that only uses the *modus ponens* and axiom deduction rules, and does not verify the deduction lemma. We also review translations towards the proof term calculus of natural deduction, and the proof languages of Coq and PVS; finally we provide a translation towards natural language. We discuss the advantages and drawbacks of these translations in the last section of this chapter.

⋆

Converting proofs is, in general, a difficult problem. Many logical frameworks have non-overlapping feature sets, different levels of expressiveness, when not downright disparate proof strengths. A reasonable approach to this first problem is to restrict the conversion to developments that lie within a common logical framework. Then, the naive way would be to establish a translation between the different inference rules and axiomatizations considered. However, this alternative is rapidly toppled by the facts: the implementations that various proof assistants make of inference rules are by and large uneven, and often very fragile:

- The semantics of the inference steps are often based on an idealization of the implemented theory, and not their actual encoding as tactics. Thus given two concrete systems implementing of the same presumptive framework, the same rule can behave quite differently depending

on which system it runs upon. Simplification and re-ordering of formulas, automatic proof of trivial premisses, *etc.*, all these features — that are necessary for an enlightened interaction the user — are as many obstacles to a tactic-for-tactic translation.

· These semantics are often modified, for instance because of newer advances in proof theory or to accommodate the requests of its user base. Even minor modifications such as formula numbering or the handling of implicit arguments would completely break a translation system. Thus, supposing the semantic issues are resolved, basing proof interoperability on such volatile grounds as the implementation of inference rules, would be a sure recipe for a maintainability nightmare[1].

Hence having a translation function acting as a morphism on proof commands cannot be heralded as a one-stop solution to proof interoperability. We show in this chapter that an approach based on more basic objects than inferences, such as sequents is doable. We also look at transformations based on closed proof terms, *i.e.* terms that contain no metavariables.

6.1 A Frege-Hilbert style deduction system

We call *proof steps* the set of sequents that are bound together by the application of an inference rule. The first translation views proofs in the source formalism as a series of proof steps. It is tied exclusively to the language $\mathcal{L}^1_m$ of first-order logic, and not to proof terms or inference rules. Proof steps are translated into the target system as a series of inference rules.

We use a deduction system *à la* Frege-Hilbert as a target of this first translation. This kind of systems was pioneered by Hilbert and Frege, and constituted the first modern attempt to formalize the notion of demonstration. The principle is to have a number of axioms to encode the behaviour of logical connectives, and only a few deduction rules used to apply these axioms. For instance, early formalizations of intuitionistic logic have been given in this type of framework (Heyting, 1930).

Definition 6.1.1 ($\mathcal{L}^{\Rightarrow}$)**.** We call $\mathcal{L}^{\Rightarrow}$ a language that is able to express the syntax of many-sorted first-order predicate logic, *as per* definition 1.1.2. In order to differentiate the constructions of the two languages, in $\mathcal{L}^{\Rightarrow}$ we use connectives with a round superscripts, *e.g.* $\Rightarrow$ for implication.

The language $\mathcal{L}^1_m$ is isomorphic to $\mathcal{L}^{\Rightarrow}$: a translation function $\backslash.\backslash$ between the two syntaxes is the identity function (up to renaming of the logical connectives). We extend this function to translate sequents of formulas in $\mathcal{L}^1_m$ into formulas of $\mathcal{L}^{\Rightarrow}$:

$$\backslash A_1, \ldots, A_n \vdash B\backslash = \mathcal{C}(\backslash A_1\backslash \Rightarrow \ldots \Rightarrow \backslash A_n\backslash \Rightarrow \backslash B\backslash)$$

[1] A prototype implementation of such a translator, named League, was carried out in 2005, and promptly validated the aforementioned statement.

where, given a formula F of $\mathcal{L}^{\geqslant}$, $\mathcal{C}(F)$ denotes its universal closure. Note that in this translation, we don't distinguish active formulas. List of sequents can also be translated, as a conjunction of their elements:

$$\backslash \Gamma_1 \vdash \Delta_1 \dots \Gamma_n \vdash \Delta_n \backslash = \backslash \Gamma_1 \vdash \Delta_1 \backslash \wedge \dots \wedge \backslash \Gamma_n \vdash \Delta_n \backslash$$

Definition 6.1.2 (H_{mp})**.** Let H_{mp} be a logical framework capable of expressing the *modus ponens* and axiom inference rules:

$$\mathrm{mp}\ \frac{A \qquad A \Rightarrow B}{B} \qquad \mathrm{axiom}\ \frac{}{A}\ A \in \mathcal{A}$$

We define the set of axioms $\mathcal{A}$ as containing all formulas of the form

$$\backslash S_l \backslash \Rightarrow \backslash s \backslash$$

where s (*resp.* S_l) is the conclusion (*resp.* premisses) of a deduction rule of LJ, *as per* definition 2.3.2. Rules with no premiss generate axioms of the form:

$$\backslash s \backslash$$

Finally the two following formulas are considered as elements of $\mathcal{A}$:

$$A \Rightarrow B \Rightarrow (A \wedge B) \tag{$*_1$}$$
$$(A \Rightarrow B) \Rightarrow ((A \wedge C) \Rightarrow (B \wedge C)) \tag{$*_2$}$$

Note. Let us bring up two important remarks about the H_{mp} framework:

- the sequents in H_{mp} only have a consequent formula, and no antecedents (hence we omit the turnstile symbol). Indeed, the framework does not have any inference rule that grow the context: it can thus be treated as a global constant, which corresponds to the theory's set of axioms $\mathcal{A}$.
- the set of axioms $\mathcal{A}$ contains the following formula, that corresponds to the $\forall_{\mathcal{R}}$ rule of LJ:

  $$\forall x^s.((A_1 \wedge \dots \wedge A_n) \Rightarrow A) \Rightarrow ((A_1 \wedge \dots \wedge A_n) \Rightarrow \forall x^s.A)$$

  This axiom is correct, because of the freshness condition that the rule $\forall_{\mathcal{R}}$ enforces on the variable x *wrt.* its context formulas $A_1, \dots, A_n$. A similar remark can be made about the axiom of $\mathcal{A}$ that corresponds to the $\exists_{\mathcal{L}}$ rule of LJ.

Notation. In the rest of this section we use a macro inference rule, named "auto", which given an input formula:

1. if the formula is of the form $B \Rightarrow (A \wedge B)$, applies the $(*_1)$ axiom using *modus ponens*,

2. if the formula is of the form $(A \wedge C) \Rightarrow (B \wedge C)$, applies the $(*_2)$ axiom using *modus ponens*,
3. immediately proves the resulting formula, by applying the "axiom" rule.

Having defined the target logical framework, we can propose a translation between proofs in LJ and in H_{mp}. Each inference step in a proof tree is interpreted using the inference rules of H_{mp}:

$$\left\backslash \frac{\dfrac{\pi_1}{s_1} \quad \cdots \quad \dfrac{\pi_n}{s_n}}{s} \right\backslash^{\frac{\pi_l}{S_l}} = \mathrm{mp}\ \frac{\left\backslash \frac{\pi_1}{s_1} \right\backslash^{\frac{\pi_2}{s_2} :: \cdots :: \frac{\pi_n}{s_n} :: \frac{\pi_l}{S_l}} \qquad \mathrm{auto}\ \dfrac{}{\backslash s_1 \ldots s_n S_l \backslash \Rightarrow \backslash s S_l \backslash}}{\backslash s S_l \backslash}$$

$$\backslash \overline{s} \backslash^{\frac{\pi_1}{s_1} :: \frac{\pi_l}{S_l}} = \mathrm{mp}\ \frac{\left\backslash \frac{\pi_1}{s_1} \right\backslash^{\frac{\pi_l}{S_l}} \qquad \mathrm{auto}\ \dfrac{}{\backslash s_1 S_l \backslash \Rightarrow \backslash s\, s_1 S_l \backslash}}{\backslash s\, s_1 S_l \backslash}$$

$$\backslash \overline{s} \backslash^{nil} = \mathrm{auto}\ \frac{}{\backslash s \backslash}$$

Note that this translation uses a continuation of subproofs (in superscript on the translation function) to encode an innermost-leftmost proof tree traversal. However any bottom-up tree traversing strategy would be adequate here: the objective is simply to deal with one premise (and its subtree) at a time. Alternatively, one could consider a modus ponens rule with an arbitrary number of premisses, allowing to deal in parallel with all the premisses of the source inference.

In order to translate the proofs in LK and LKM into this intuitionistic framework, we extend the translation function to deal with multiple consequents, and Gödel's double negation translation is applied on each formula in the target inference tree. We note $\backslash . \backslash_{\mathsf{NNPP}}$ the combination of the $\backslash . \backslash$ translation with this systematic double negation translation.

Example 6.1.3. Writing down the inference trees is rapidly very space-costly, thus the case in point is quite small. Take the proof of the classical tautology $((\neg B \Rightarrow \neg A) \Rightarrow A \Rightarrow B)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{ax}_{\mathcal{L}}\ \dfrac{}{A; B \vdash B}}{\neg_{\mathcal{R}}\ \dfrac{}{A \vdash \neg B; B}} \qquad \dfrac{\mathrm{ax}_{\mathcal{R}}\ \dfrac{}{A \vdash A; B}}{\neg_{\mathcal{L}}\ \dfrac{}{A; \neg A \vdash B}}}{\Rightarrow_{\mathcal{L}}\ \dfrac{}{A; \neg B \Rightarrow \neg A \vdash B}}}{\mathrm{focus}\ \dfrac{}{\neg B \Rightarrow \neg A, A \vdash B}}}{\Rightarrow_{\mathcal{R}}\ \dfrac{}{\neg B \Rightarrow \neg A \vdash A \Rightarrow B}}}{\Rightarrow_{\mathcal{R}}\ \dfrac{}{\vdash (\neg B \Rightarrow \neg A) \Rightarrow A \Rightarrow B}}$$

with the "focus" inference being a macro for the composition of a trivial cut (here, on $\neg B \Rightarrow \neg A$) and an axiom rule. Since the translation does not mind which formula is active, this rule is transparent for this purpose, and will be ignored for the sake of conciseness. The $\backslash.\backslash$-translation generates the following proof (in the form of russian headstocks, for typesetting purposes):

$$\dfrac{\dfrac{\pi_1 \qquad \text{auto}\ \dfrac{}{\mathbf{P_1} \Rightarrow \mathbf{P_1}}}{\mathbf{P_1} \equiv (\neg B \Rightarrow \neg A) \Rightarrow A \Rightarrow B}\ \text{mp} \qquad \text{auto}\ \dfrac{}{\mathbf{P_1} \Rightarrow \mathbf{P_1}}}{\mathbf{P_1} \equiv (\neg B \Rightarrow \neg A) \Rightarrow A \Rightarrow B}\ \text{mp}$$

with π_1:

$$\dfrac{\dfrac{\pi_2 \qquad \text{auto}\ \dfrac{}{\mathbf{P_3} \Rightarrow \mathbf{P_2}}}{\mathbf{P_2} \equiv (A \Rightarrow (\neg B \vee B)) \wedge (A \Rightarrow \neg A \Rightarrow B)}\ \text{mp} \qquad \text{auto}\ \dfrac{}{\mathbf{P_2} \Rightarrow \mathbf{P_1}}}{\mathbf{P_1} \equiv (\neg B \Rightarrow \neg A) \Rightarrow A \Rightarrow B}\ \text{mp}$$

and π_2:

$$\dfrac{\dfrac{\text{auto}\ \dfrac{}{\mathbf{P_5} \equiv A \Rightarrow (A \vee B)} \qquad \text{auto}\ \dfrac{}{\mathbf{P_5} \Rightarrow \mathbf{P_4}}}{\mathbf{P_4} \equiv A \Rightarrow \neg A \Rightarrow B}\ \text{mp} \qquad \text{auto}\ \dfrac{}{\mathbf{P_4} \Rightarrow \mathbf{P_3}}}{\mathbf{P_3} \equiv (A \Rightarrow B \Rightarrow B) \wedge (A \Rightarrow \neg A \Rightarrow B)}\ \text{mp}$$

where the $\equiv$ symbol is used as a labelling operator, and the bold labels represent their respective formulas.

Proposition 6.1.4. *For each formula A of $\mathcal{L}^1_m$ that has a* LJ*-proof π_A, then $\backslash\pi_A\backslash$ is a valid proof in* H_{mp} *of the formula $\backslash A\backslash$ of $\mathcal{L}^{\geqslant}$.*

Proof. By double induction on the structure of the proof π_A and of the translation continuation. If π_A is a leaf, then we proceed by case on the structure of the stack of subproofs:

- if the continuation is empty, then the proof is translated into a single "auto" rule (last equation of the proof translation rules). Since π_A is a leaf, then its conclusion s is an axiom of LJ, whose translation $\backslash s\backslash$ is an element of $\mathcal{A}$. Thus "auto" proves the goal by applying the "axiom" rule of H_{mp}.
- if the continuation is non-empty, then the translated proof uses *modus ponens* to discard the goal and pop the continuation (second equation of the proof translation rules). Since π_A is a leaf, then its conclusion s is an axiom of LJ, whose translation $\backslash s\backslash$ is an element of $\mathcal{A}$. Thus "auto" proves the premiss $\backslash s_1 S_l\backslash \Rightarrow \backslash s s_1 S_l\backslash$ by applying the formula $(*_1)$ *via modus ponens*, which yields the formula $\backslash s\backslash$. This formula is then discarded by the "axiom" rule. The other premiss is a valid proof of $\backslash s s_1 S_l\backslash$, by induction hypothesis.

If π_A is a tree, then the translated proof uses *modus ponens* to discard the goal, then prove the translation of the first premiss while pushing into the continuation the translation of the other premisses (first equation of the proof translation rule). Then the premiss $\backslash s_1 \dots s_n S_l \backslash \Rightarrow \backslash s S_l \backslash$ is proven by "auto" by applying the axiom $(*_2)$ if needed (*i.e.* if the continuation is non-empty), and then discarding the formula $\backslash s_1 \dots s_n \backslash \Rightarrow \backslash s \backslash$ using the "axiom" rule. The other premiss if a valid proof of $\backslash s_1 \dots s_n S_l \backslash$, by induction hypothesis. □

Corollary 6.1.5. *For each formula* A *of* $\mathcal{L}^1_m$ *that has a* LK*-proof* π_A*, then* $\backslash \pi_A \backslash_{\mathrm{NNPP}}$ *is a valid proof in* H_{mp} *of the formula* $\backslash A \backslash$ *of* $\mathcal{L}^{\Rightarrow}$.

Note. As illustrated by example 6.1.3 with the translation of the first two $\Rightarrow_{\mathcal{R}}$ inferences, some of the native inference steps generate superfluous *modus ponens* steps of the form:

$$\mathrm{mp}\ \frac{A \qquad A \Rightarrow A}{A}$$

These steps do not contribute to the translated proof, hence a simple optimization of the interpretation function can be done to discard such irrelevant proof steps.

To conclude this section, we emphasize the contribution of the system H_{mp} to the design of Frege-Hilbert formalisms. Proposition 6.1.4 asserts the equivalence of H_{mp} with LJ, yet our formalism importantly differs from the usual formulation of Frege-Hilbert systems on the following points:

- our framework does not have quantifier deduction rules: introduction of the universal quantifier and elimination of the existential quantifier.
- it does not verify the deduction lemma, which states: if B is derivable from A and F then $(A \Rightarrow B)$ is derivable from F.

These two specificities are related to the use of context-free sequents. They represent a significant simplification of Frege-Hilbert style formalisms.

6.2 Natural deduction: the λ1-calculus

A second translation can be made at the level of closed proof terms: it supposes that both the source and the target system are based on the Curry-de Bruijn-Howard isomorphism. In particular, the target system in this case will use λ-calculus with dependent types, which at the logical level represent proofs in first-order natural deduction.

Because of the need for terms to encode non-minimal logical connectors, a subset of the consensual λ1-calculus is used as the target for the translation.

Definition 6.2.1 ($\lambda\mathbb{1}$-calculus)**.** Terms of the $\lambda\mathbb{1}$-calculus are given the following syntax:

$$\begin{aligned} t, u, A, B ::= {} & x \mid c \mid (t\ t') \mid \lambda x^A.t \\ & \mid I \mid (R^A\ t) \\ & \mid \pi_1^A\ t \mid \pi_2^A\ t \mid (p^A\ t\ u) \\ & \mid (\delta^{A,B,C}\ f\ g\ t) \mid i^{A,B}\ t \mid j^{A,B}\ t \\ & \mid \top \mid \bot \mid A + B \\ & \mid \Pi x^A.B \mid \Sigma x^A.B \\ & \mid \mathit{Kind} \mid \mathit{Type} \end{aligned}$$

and equipped with an equivalence relation $\equiv$. The reduction rules as follow:

$$\begin{aligned} (\lambda x^A.t\ u) &\to t[x \leftarrow u] \\ (\pi_1\ (p\ t\ u)) &\to t \\ (\pi_2\ (p\ t\ u)) &\to u \\ (\delta\ f\ g\ (i\ t)) &\to (f\ t) \\ (\delta\ f\ g\ (j\ a)) &\to (g\ t) \end{aligned}$$

The type inference rules for $\lambda\mathbb{1}$ are given by figure 6.1.

We begin with the translation of the language of $\mathcal{L}^{\mathbb{1}}_m$ into constructs of $\lambda\mathbb{1}$, which is relatively straightforward. We note $\|.\|$ the translation function. For types:

$$\begin{aligned} \|\mathrm{bool}\| &= \mathit{Type} \\ \|k\| &= c_k \\ \|s \to r\| &= \Pi x^{\|s\|}.\, \|r\| \qquad\qquad x \text{ fresh} \end{aligned}$$

for terms:

$$\begin{aligned} \|x\| &= x \\ \|f(t_1, \ldots, t_n)\| &= (c_f\ \ \|t_1\| \ldots \|t_n\|) \end{aligned}$$

$$\frac{}{\Gamma \vdash \mathit{Type} : \mathit{Kind}} \qquad \frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{}{\Gamma \vdash \top : \mathit{Type}} \qquad \frac{}{\Gamma \vdash I : \top}$$

$$\frac{}{\Gamma \vdash \bot : \mathit{Type}} \qquad \frac{\Gamma \vdash t : \bot}{\Gamma \vdash (R^A\ t) : A}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Kind}}{\Gamma \vdash \Pi x^A.B : \mathit{Kind}}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Type}}{\Gamma \vdash \Pi x^A.B : \mathit{Type}}$$

$$\frac{\Gamma \vdash t : \Pi x^A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B[x \leftarrow u]}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Kind} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : \Pi x^A.B}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Type} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : \Pi x^A.B}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Type}}{\Gamma \vdash \Sigma x^A.B : \mathit{Type}}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, x : A \vdash B : \mathit{Type} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B[x \leftarrow A]}{\Gamma \vdash (p^{\Sigma x^A.B}\ t\ u) : \Sigma x^A.B : \mathit{Type}}$$

$$\frac{\Gamma \vdash t : \Sigma x^A.B}{\Gamma \vdash (\pi_1^{\Sigma x^A.B}\ t) : A}$$

$$\frac{\Gamma \vdash t : \Sigma x^A.B}{\Gamma \vdash (\pi_2^{\Sigma x^A.B}\ t) : B[x \leftarrow (\pi_1^{\Sigma x^A.B}\ t)]}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma \vdash B : \mathit{Type}}{\Gamma \vdash A + B : \mathit{Type}}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma \vdash B : \mathit{Type} \qquad \Gamma \vdash t : A}{\Gamma \vdash (i^{A,B}\ t) : A + B}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma \vdash B : \mathit{Type} \qquad \Gamma \vdash t : B}{\Gamma \vdash (j^{A,B}\ t) : A + B}$$

$$\frac{\Gamma \vdash t : A + B \qquad \Gamma \vdash C : \mathit{Type} \qquad \Gamma \vdash f : A \to C \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash (\delta^{A,B,C}\ f\ g\ t) : C}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : \mathit{Type} \qquad \Gamma \vdash A : \mathit{Type} \qquad A \equiv B}{\Gamma \vdash t : B}$$

Figure 6.1: Typing rules for $\lambda 1$

and for formulas:

$$
\begin{aligned}
\|\texttt{true}\| &= \top \\
\|\texttt{false}\| &= \bot \\
\|p(t_1,\ldots,t_n)\| &= (c_p\ \ \|t_1\|\ldots\|t_n\|) \\
\|\neg A\| &= \Pi x^{\|A\|}.\bot && x \text{ fresh} \\
\|A \Rightarrow B\| &= \Pi x^{\|A\|}.\,\|B\| && x \text{ fresh} \\
\|A \wedge B\| &= \Sigma x^{\|A\|}.\,\|B\| && x \text{ fresh} \\
\|A \vee B\| &= \|A\| + \|B\| \\
\|\forall x^s.A\| &= \Pi x^{\|s\|}.\,\|A\| \\
\|\exists x^s.A\| &= \Sigma x^{\|s\|}.\,\|A\|
\end{aligned}
$$

Now the proof terms themselves can be translated into $\lambda 1$. The process is akin to Prawitz' translation (Prawitz, 1965). The idea is that derivations in sequent calculus are recipes for constructing deductions in natural deduction. Recursively, for intuitionistic proof terms:

$$
\begin{aligned}
\|\ltimes\| &= I \\
\|x\| &= x \\
\left\|\lambda x^A.v\right\| &= \lambda x^{\|A\|}.\,\|v\| \\
\|(v,v')\| &= (p\ \ \|C\|\ \ \|v\|\ \ \|v'\|) && \text{with } (v,v') : C \\
\|\mathrm{inj}_r v\| &= (j^{\|A\|,\|B\|}\ \ \|v\|) && \text{with } \mathrm{inj}_r v : A \vee B \\
\|\mathrm{inj}_l v\| &= (i^{\|A\|,\|B\|}\ \ \|v\|) && \text{with } \mathrm{inj}_r v : A \vee B \\
\left\|\mu *^A.c\right\| &= \|c\|
\end{aligned}
$$

for environments:

$$
\begin{aligned}
\|\rtimes\|^t &= (R^A\ t) && \text{where } A \text{ is inferred} \\
\|*\|^t &= t \\
\|v \cdot e\|^t &= \|e\|_{(t\ \|v\|)} \\
\|\mathrm{proj}[x,y,c]\|^t &= \left\|c[x \leftarrow (\pi_1^{\|C\|}\ t)][y \leftarrow (\pi_2^{\|C\|}\ t)]\right\| && \text{with } \mathrm{proj}[x,y,c] : C \\
\|[e,e']\|^t &= (\delta^{A,B,C}\ (\lambda z^A.\,\|e\|^z)\ (\lambda z^B.\,\|e'\|^z)\ t) && \text{where } C \text{ is inferred} \\
\left\|\tilde{\mu} x^A.c\right\|^t &= \text{let } (x^A := t) \text{ in } \|c\|
\end{aligned}
$$

finally, for commands:

$$
\|\langle v \| e \rangle\| = \|e\|^{\|v\|}
$$

Additionally, the first η_μ redex is discarded by the translator, which uses it solely to retrieve the final type of the term. This in turn allows the type inference constraints, mentioned in some translation cases, to be satisfied[2].

For classical proof terms, the $\mathcal{F}$-translation can be used to generate LJ+EM terms. Then the translation can proceed using the intuitionistic rules, extended with the correspondence between $\mathbf{em}_A$ and the term of classical $\lambda 1$ that inhabits $A + \Pi x^A.\bot$. We note $\|.\|_{\mathbf{em}}$ this extended translation.

Proposition 6.2.2. *If π_A is a proof term of type A in* LJ*, then $\|\pi_A\|$ is a proof term of type $\|A\|$ in $\lambda 1$.*

Proof. By induction on the structure on terms, and by case analysis. □

Corollary 6.2.3. *If π_A is a proof term of type A in* LK*, then $\|\mathcal{F}_{nil}(\pi_A)\|_{\mathbf{em}}$ is a proof term of type $\|A\|_{\mathbf{em}}$ in $\lambda 1$.*

6.3 Coq and PVS proof scripts

Generating Coq proof scripts

The original syntax of Coq terms is based in the Calculus of Inductive Constructions, a powerful, yet bare formalism. The `Init` library provides global definitions more elaborate constructs, such as the common logical constructions. Finally, tactics are used to incrementally build proof terms by successive instantiation of open terms. In what follows, we describe a simplified subset of this layering, using when necessary the results of the previous chapters. We then provide a translation from $\lambda\mu\tilde{\mu}$ proof terms to Coq proof terms, using the bias of the tactic mechanism.

Definition 6.3.1 (CIC: the calculus of inductive constructions)**.** We follow the definition of (Paulin-Mohring, 1993) using the usual vectorial notation $\vec{.}$ for sequences:

$$
\begin{aligned}
s &::= \texttt{Set} \mid \texttt{Type} \\
A_r &::= s \mid \Pi x^A.A_r \\
C &::= x_I \mid (C\ t) \mid \Pi x^A.C \\
t, u, A, B &::= x \mid c \mid s \mid (t\ t') \mid \lambda x^A.t \mid \Pi x^A.B \\
&\quad \mid \texttt{Ind}(x : A_r)\{\vec{C}\} \mid \texttt{Constr}(i, t) \mid \texttt{Elim}(A_r, t)\{\vec{u}\}
\end{aligned}
$$

where C is a form constructor *wrt.* x_I of type A_r, and in the dependent product for constructors, A satisfies a positivity condition[3]. The letter i denotes a natural number of the meta-theory. Figure 6.2 recaps the type inference rules for this calculus. Note that in the constructor and elimination

[2] This type inference process can be avoided by using a translation function that keeps track of the type of its argument.

rule, I denotes $\texttt{Ind}(x : A_r)\{C_1 \dots C_n\}$ with $A_r \equiv \Pi\vec{x}^{\vec{A}}.s$. The notation $C\{x_I, A, t\}$ is used to denote the appropriate instantiation[3] of the constructor C by the terms x_I, A and t. Finally the $\to$ notation is used to denote non-dependent products.

$$\frac{}{\Gamma \vdash \texttt{Set} : \texttt{Type}} \qquad \frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x^A.B : s_2}\ (s_1, s_2) \in \{\texttt{Set}, \texttt{Type}\}^2$$

$$\frac{\Gamma \vdash t : \Pi x^A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B[x \leftarrow u]} \qquad \frac{\Gamma \vdash \Pi x^A.B : s \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : \Pi x^A.B}$$

$$\frac{s \in A_r \qquad \forall i \in [1 \dots n], (\Gamma, x_I : A_r \vdash C_i : s) \qquad \forall i \in [1 \dots n], x_I \in C_i}{\Gamma \vdash \texttt{Ind}(x_I : A_r)\{\vec{C}\} : A_r}$$

$$\frac{\Gamma \vdash I : A_r \qquad 1 \leqslant i \leqslant n}{\Gamma \vdash \texttt{Constr}(i, I) : C_i[x_I \leftarrow I]}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : (I\ \vec{a}) \qquad \Gamma \vdash A : \Pi\vec{x}^{\vec{B}}.(I\ \vec{x}) \to s' \\ \forall i \in [1 \dots n], (\Gamma \vdash u_i : C_i\{I, A, \texttt{Constr(i,I)}\})\end{array}}{\Gamma \vdash \texttt{Elim}(A, t)\{\vec{u}\} : (A\ \vec{a}\ t)}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \equiv B}{\Gamma \vdash t : B}$$

Figure 6.2: Typing rules for CIC

Notation. In accordance with the `Init` library, we introduce the following concrete syntax:

$$A\ \texttt{->}\ B = A \to B$$
$$\texttt{forall}\ x : A,\ B = \Pi x^A.B$$

and definitions:

$$\texttt{True} = \texttt{Ind}(x_I : \texttt{Set})\{x_I\}$$
$$\texttt{False} = \texttt{Ind}(x_I : \texttt{Set})\{\}$$
$$\texttt{~}A = A \to \texttt{False}$$
$$A\ \texttt{/\textbackslash}\ B = \texttt{Ind}(x_I : \texttt{Set})\{A \to B \to x_I\}$$
$$A\ \texttt{\textbackslash/}\ B = \texttt{Ind}(x_I : \texttt{Set})\{A \to x_I, B \to x_I\}$$
$$\texttt{exists}\ x : A,\ B = \texttt{Ind}(x_I : \texttt{Set})\{\Pi y^A.(P[x \leftarrow y]) \to (x_I\ A\ \lambda x^A.P)\}$$

[3] More on positivity and all that in (Paulin-Mohring, 1993).

Definition 6.3.2 (Syntax of Coq proof commands)**.** The following subset of the tactic language of Coq is used for the translation:

$$\begin{aligned}\mathbf{t} ::= &\ \texttt{exact t} \mid \texttt{intro x} \mid \texttt{apply x} \mid \texttt{cut A} \\ \mid &\ \texttt{destruct x as [}x^1_1 \ldots x^1_{m_1} \mid \ldots \mid x^n_1 \ldots x^n_{m_n}\texttt{]} \mid \texttt{rename x y} \\ \mid &\ \texttt{constructor} \mid \texttt{left} \mid \texttt{right} \mid \texttt{split} \mid \texttt{exists t}\end{aligned}$$

A simplistic strategy language is also considered:

$$\mathbf{s} ::= \mathbf{t} \mid \mathbf{t}\ \texttt{;}\ \mathbf{s} \mid \mathbf{t}\ \texttt{;}\ \texttt{[}\mathbf{s_1}\ \texttt{|}\ \ldots \texttt{|}\ \mathbf{s_n}\texttt{]}$$

Definition 6.3.3 (Semantics of Coq proof commands)**.** Figure 6.3 presents the semantics of tactics within the formalism of chapter 3, *i.e.* as a set of rewrite rules over a simplified proof state. The semantics of strategies is akin to the one developed in chapter 4.

The translation of $\mathcal{L}^1_m$ in Coq is achieved *via* the use of the translation function /./. For types:

$$\begin{aligned}/\text{bool}/ &= \texttt{Set} \\ /\mathtt{k}/ &= \mathtt{c_k} \\ /\mathtt{s} \to \mathtt{r}/ &= \texttt{(/s/ -> /r/)}\end{aligned}$$

terms:

$$\begin{aligned}/\mathtt{x}/ &= \mathtt{x} \\ /\mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n})/ &= (\mathtt{c_f}\ /\mathtt{t_1}/\ldots/\mathtt{t_n}/)\end{aligned}$$

and formulas:

$$\begin{aligned}/\texttt{true}/ &= \texttt{True} \\ /\texttt{false}/ &= \texttt{False} \\ /\mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})/ &= (\mathtt{c_p}\ /\mathtt{t_1}/\ldots/\mathtt{t_n}/) \\ /\neg \mathtt{A}/ &= \texttt{~}/\mathtt{A}/ \\ /\mathtt{A} \Rightarrow \mathtt{B}/ &= \texttt{(/A/ -> /B/)} \\ /\mathtt{A} \wedge \mathtt{B}/ &= /\mathtt{A}/\ \texttt{/\textbackslash}\ /\mathtt{B}/ \\ /\mathtt{A} \vee \mathtt{B}/ &= /\mathtt{A}/\ \texttt{\textbackslash/}\ /\mathtt{B}/ \\ /\forall \mathtt{x}^{\mathtt{s}}.\mathtt{A}/ &= \texttt{(exists x : /s/, /A/)} \\ /\exists \mathtt{x}^{\mathtt{s}}.\mathtt{A}/ &= \texttt{(forall x : /s/, /A/)}\end{aligned}$$

The translation of proof terms stems from the idea of emulating the type inference rules of $\bar{\lambda}\mu\tilde{\mu}$ in Coq. However, while in sequent calculus both hypothesis and succedents can be active, it is not the case in natural

$$\frac{?}{\Gamma \vdash X : A} \xrightarrow{\texttt{exact t}} \frac{}{\Gamma \vdash t : A}$$

$$\frac{?}{\Gamma \vdash X : \Pi x^A.B} \xrightarrow{\texttt{intro x}} \frac{\dfrac{}{\Gamma \vdash \Pi x^A.B : s} \quad \dfrac{?}{\Gamma, x : A \vdash Y : B}}{\Gamma \vdash \lambda x^A.Y : \Pi x^A.B}$$

$$\frac{?}{\Gamma, x : \Pi \vec{y}^{\vec{A}}.B \vdash X : B} \xrightarrow{\texttt{apply x}} \frac{\pi_1 \qquad \pi_2}{\Gamma, x : \Pi \vec{y}^{\vec{A}}.B \vdash (x\ Y) : B}$$

where $\pi_1 = \dfrac{}{\Gamma, x : \Pi \vec{y}^{A}.B \vdash x : \Pi \vec{y}^{\vec{A}}.B}$

and $\pi_2 = \dfrac{?}{\Gamma, x : \Pi \vec{y}^{\vec{A}}.B \vdash Y : A}$

$$\frac{?}{\Gamma \vdash X : B} \xrightarrow{\texttt{cut A}} \frac{\dfrac{?}{\Gamma \vdash Y : \Pi x^A.B} \quad \dfrac{?}{\Gamma \vdash Z : A}}{\Gamma, \vdash (Y\ Z) : B}$$

$$\frac{?}{\Gamma, x : (I\ \vec{a}) \vdash X : (A\ \vec{a}\ t)} \xrightarrow{\texttt{destruct x as l}} \frac{\pi_1 \qquad \pi_2 \qquad \pi_3^1 \dots \pi_3^n}{\Gamma, x : (I\ \vec{a}) \vdash \texttt{Elim}(A, x)\{\vec{u}\} : (A\ \vec{a}\ t)}$$

where $l = [x_1^1 \dots x_{m_1}^1 | \dots | x_1^n \dots x_{m_n}^n]$

and $\pi_1 = \dfrac{}{\Gamma, x : (I\ \vec{a}) \vdash x : (I\ \vec{a})}$

and $\pi_2 = \dfrac{}{\Gamma, x : (I\ \vec{a}) \vdash A : \Pi \vec{x}^{\vec{B}}.(I\ \vec{x}) \to s'}$

and $\pi_3^i = \dfrac{\dfrac{?}{\Gamma, x : (I\ \vec{a}), x_{j_i}^{\vec{i}} : \vec{B_i} \vdash X_i : D_i}}{\Gamma, x : (I\ \vec{a}) \vdash \Gamma \vdash \lambda x_{j_i}^{\vec{i}}{}^{\vec{B_i}}.X_i : \Pi x_{j_i}^{\vec{i}}{}^{\vec{B_i}}.D_i}$

and $\Pi x_{j_i}^{\vec{i}}{}^{\vec{B_i}}.D_i = C_i\{I, A, \texttt{Constr(i,I)}\}$

$$\frac{?}{\Gamma, x : A \vdash X : B} \xrightarrow{\texttt{rename x y}} \frac{?}{\Gamma[x \leftarrow y], y : A[x \leftarrow y] \vdash X : B[x \leftarrow y]}$$

$$\frac{?}{\Gamma \vdash X : \Pi \vec{x}^{B}.C} \xrightarrow{\texttt{constructor i}} \frac{\dfrac{\pi_1 \qquad \pi_2}{\Gamma, \vec{x} : \vec{B} \vdash (\texttt{Constr}(i, I)\ \vec{X}) : D}}{\Gamma \vdash \lambda \vec{x}^{\vec{B}}.(\texttt{Constr}(i, I)\ \vec{X}) : \Pi \vec{x}^{\vec{B}}.D}$$

where $\Pi \vec{x}^{\vec{B}}.D = C_i[x_I \leftarrow I]$

and $\pi_1 = \dfrac{}{\Gamma, \vec{x} : \vec{B} \vdash \texttt{Constr}(i, I) : \Pi \vec{x}^{\vec{B}}.D}$

$\pi_2 = \dfrac{?}{\Gamma, \vec{x} : \vec{B} \vdash \vec{X} : \vec{B}}$

Moreover,

$$\begin{aligned} \texttt{left} &= \texttt{split} = \texttt{constructor 1} \\ \texttt{right} &= \texttt{constructor 2} \\ \texttt{exists t} &= \texttt{constructor 1} \\ &\quad \text{with t being used as a clue for type unification} \end{aligned}$$

Figure 6.3: The semantics of Coq base tactics

deduction. Hence the translation function /./ gets appended a superscript: we note $/t/^{H}$ the translation of the proof term t in which the hypothesis H is active. The following rules hold in the case of intuitionistic proofs; for terms:

$$
\begin{aligned}
/x/ &= \texttt{exact x} \\
/\ltimes/ &= \texttt{constructor 1} \\
/\lambda x^{A}.\nu/ &= \texttt{intro x ; } /\nu/ \\
/(\nu,\nu')/ &= \texttt{split ; [}/\nu/ \texttt{ | } /\nu'/\texttt{]} \\
/(t,\nu)/ &= \texttt{exists } /t/ \texttt{ ; } /\nu/ \\
/\mathrm{inj}_{l}\nu/ &= \texttt{left ; } /\nu/ \\
/\mathrm{inj}_{r}\nu/ &= \texttt{right ; } /\nu/ \\
/\mu *^{A}.\langle \nu \| e\rangle/ &= \texttt{cut } /B/ \texttt{ ; [intro H ; } /e/^{H:B} \texttt{ | } /\nu/\texttt{]}
\end{aligned}
$$

and for environments:

$$
\begin{aligned}
/*/^{H:A} &= \texttt{exact H} \\
/\rtimes/^{H:A} &= \texttt{contradiction} \\
/\nu \cdot e/^{H:A\Rightarrow B} &= \texttt{cut } /B/ \texttt{ ;} \\
&\quad \texttt{[intro H' ; } /e/^{H':B} \\
&\quad \texttt{| apply H ; } /\nu/\texttt{]} \\
/t \cdot e/^{H=\forall x:A.B} &= \texttt{cut } /B[x \leftarrow t]/\texttt{;} \\
&\quad \texttt{[intro H' | apply H];} \ /e/^{H':B[x\leftarrow t]} \\
/\mathrm{proj}[x,y,\langle \nu \| e\rangle]/^{H:A\wedge B} &= \texttt{destruct H as [x y] ; } /\mu *.\langle \nu \| e\rangle/ \\
/\mathrm{proj}[x,y,\langle y \| e\rangle]/^{H:\exists x:A.B} &= \texttt{destruct H as [x H'] ; } /e/^{H':B} \\
/[e,e']/^{H:A\vee B} &= \texttt{destruct H as [H'|H''] ;} \\
&\quad \texttt{[}/e/^{H':A} \texttt{ | } /e/^{H'':B}\texttt{]} \\
/\tilde{\mu} x^{A}.\langle \nu \| e\rangle/^{H:A} &= \texttt{rename H x ; cut } /B/ \texttt{ ;} \\
&\quad \texttt{[intro H' ; } /e/^{H':B} \texttt{ | } /\nu/\texttt{]}
\end{aligned}
$$

Classical proofs are dealt with using the $\mathcal{F}$-translation, and the appropriate Coq rule for excluded middle:

$$/\mathbf{em}_{A}/ = \texttt{exact (classic } /A/\texttt{)}$$

Example 6.3.4. Assume given a type $\mathbb{N}$ for natural numbers, as well as the two constructor 0 and S. Add the two predicates *Even* and *Odd* and the

following formula definitions:

$$\mathsf{E}_0 = (\mathit{Even}\ 0)$$
$$\mathsf{O}_\mathsf{S} = \forall \mathfrak{n}^{\mathbb{N}}.(\mathit{Even}\ \mathfrak{n}) \Rightarrow (\mathit{Odd}\ (\mathsf{S}\ \mathfrak{n}))$$
$$\mathsf{E}_\mathsf{S} = \forall \mathfrak{n}^{\mathbb{N}}.(\mathit{Odd}\ \mathfrak{n}) \Rightarrow (\mathit{Even}\ (\mathsf{S}\ \mathfrak{n}))$$

The proof that $(\mathsf{S}\ (\mathsf{S}\ 0))$ is even, in the context that contains the formulas E_0, O_S and E_S, is the $\bar{\lambda}\mu\tilde{\mu}$-term:

$$\mu*^{(\mathit{Even}\ 2)}.\langle \mu*^{(\mathit{Even}\ 2)}.\langle \mathsf{E}_\mathsf{S} \| 1 \cdot \mu*^{\mathit{Odd}\ 1}.\langle \mathsf{O}_\mathsf{S} \| 0 \cdot \mu*^{(\mathit{Even}\ 0)}.\langle \mathsf{E}_0 \| * \rangle \cdot * \rangle \cdot * \rangle \| * \rangle$$

where the decimal notations 1 and 2 are used for the terms $(\mathsf{S}\ 0)$ and $(\mathsf{S}\ (\mathsf{S}\ 0))$. The /./-translation yields a Coq proof script of $(\mathsf{S}\ (\mathsf{S}\ 0))$'s evenness:

```
1  cut (forall n:nat, ((Odd n) -> (Even (S n)))) ;
2    [ intro H ;
3      cut ((Odd (S O)) -> (Even (S (S O)))) ;
4      [ intro H'
5      | apply H ] ;
6        cut (Even (S (S O))) ;
7          [ intro H'' ; exact H''
8          | apply H' ;
9            cut (forall n:nat, ((Even n) -> (Odd (S n)))) ;
10           [ intro H''' ;
11             cut ((Even O) -> (Odd (S O))) ;
12               [ intro H''''
13               | apply H'''] ;
14                 cut (Odd (S O)) ;
15                   [ intro H''''' ; exact H''''' 
16                   | apply H'''' ;
17                     cut (Even O) ;
18                       [ intro H'''''' ; exact H''''''
19                       | exact EO]]
20           | exact OS]]
21   | exact ES ]
```

Note that we discarded the first η_μ-redex, for clarity.

Proposition 6.3.5. *If π_A is a proof of the formula A of $\mathcal{L}^1_\mathrm{m}$ in* LJ*, then $/\pi_\mathsf{A}/$ is a proof in Coq of the formula $/\mathsf{A}/$.*

Proof. It is easy (but tedious) to check that each Coq tactic generates the appropriate proof term of CIC. The second part of the proof consists in linking $\bar{\lambda}\mu\tilde{\mu}$ proof terms to CIC term, which is a trivial consequence of proposition 6.2.2. □

Corollary 6.3.6. *If π_A is a proof of the formula A of $\mathcal{L}^1_\mathrm{m}$ in* LK*, then $/\mathcal{F}_{nil}(\pi_\mathsf{A})/_{\mathbf{em}}$ is a proof in Coq of the formula $/\mathsf{A}/$.*

Generating PVS proof scripts

PVS is based on higher-order logics, and uses tactics to build a tree of indexed formulas rather than proof terms. The target framework for this translation is a simplified subset of the semantics of PVS, as presented in (Owre and Shankar, 1999). As such, it is to some level an idealized version of the actual implementation of this framework.

Definition 6.3.7 (HOL for PVS)**.** The syntax of PVS defines types and terms with built-in abstraction, application, tuples and projections:

$$\begin{aligned} s, r &::= \texttt{type+} \mid \texttt{bool} \mid k \mid \texttt{[}s\ \texttt{->}\ r\texttt{]} \mid \texttt{[}s\texttt{,}r\texttt{]} \\ t, u, A, B, C &::= x \mid f \mid \lambda(x : s) : t \mid t(u) \mid (t, u) \mid p_i(t) \end{aligned}$$

Three particular function symbols are defined:

$$\begin{aligned} \texttt{true} &: \texttt{bool} \\ \texttt{false} &: \texttt{bool} \\ \texttt{if} &: \texttt{[bool,s,s] -> s} \\ = &: \texttt{[s,s] -> bool} \end{aligned}$$

Figure 6.4 provides proof inference rules for these objects. While sequents in PVS are pairs of lists of indexed formulas, this presentation abstracts from the ordering and numbering process. Also remark that, unlike in (Owre and Shankar, 1999), the typing context and the formula antecedents are mixed, which saves some redundancy.

Notation. The logical connectives are introduced as notations for terms:

$$\begin{aligned} \texttt{not}\ A &= \lambda(x : \texttt{bool}) : x = \texttt{false} \\ A\ \texttt{and}\ B &= \texttt{if (}A\texttt{,}B\texttt{,false)} \\ A\ \texttt{or}\ B &= \texttt{if (}A\texttt{,true,}B\texttt{)} \\ A\ \texttt{=>}\ B &= \texttt{if (}A\texttt{,}B\texttt{,true)} \\ \texttt{forall}\ (x : s) : A &= (\lambda(x : s) : A) = (\lambda(x : s) : \texttt{true}) \\ \texttt{exists}\ (x : s) : A &= \texttt{not}\ (\lambda(x : s) : \texttt{not}\ A) \end{aligned}$$

The proof rules for these connectives can be derived using the rules of figure 6.4.

Definition 6.3.8 (Syntax of PVS proof commands)**.** The tactics used are very close to the inference rules for the logical connectives, and sport limited automation:

$$\begin{aligned} \mathbf{t} ::= &\texttt{(propax)} \mid \texttt{(lemma "x")} \mid \texttt{(flatten-disjunct)} \mid \texttt{(split i 1)} \\ &\mid \texttt{(skolem! i)} \mid \texttt{(instantiate i ("t") F)} \mid \texttt{(case "A")} \end{aligned}$$

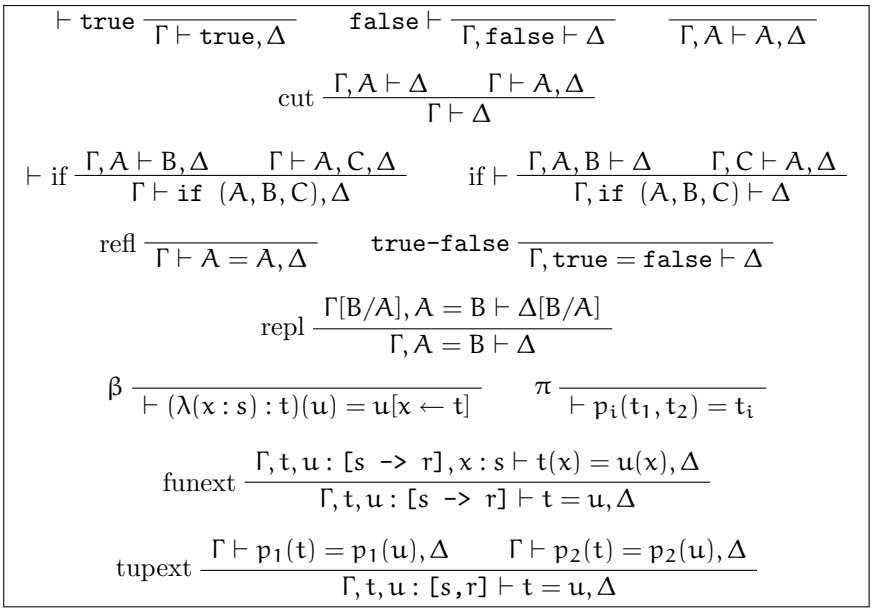


Figure 6.4: Proof rules for PVS

where i, j are formula indexes. We also make use of some of the PVS# strategies defined in chapter 4:

$$s ::= t \mid \texttt{(then t s)} \mid \texttt{(spread t (}s_1 \ \ldots s_n\texttt{))}$$

Definition 6.3.9 (Semantics of PVS proof commands)**.** Figure 6.5 exposes the semantics of these constructs, within the formalism of chapter 3. Note that the indexes of formulas in a sequent are at times made explicit, overloading the colon symbol: $i : A$ for a formula A at index i.

The translation, denoted $/./$, of $\mathcal{L}^1_m$ into PVS syntax is straightforward. For types:

$$\begin{aligned} /\text{bool}/ &= \texttt{bool} \\ /k/ &= k \\ /s \to r/ &= \texttt{[}/s/ \texttt{ -> } /r/\texttt{]} \end{aligned}$$

terms:

$$\begin{aligned} /x/ &= x \\ /f(t_1, \ldots, t_n)/ &= f_f(/t_1/, \ldots, /t_n/) \end{aligned}$$

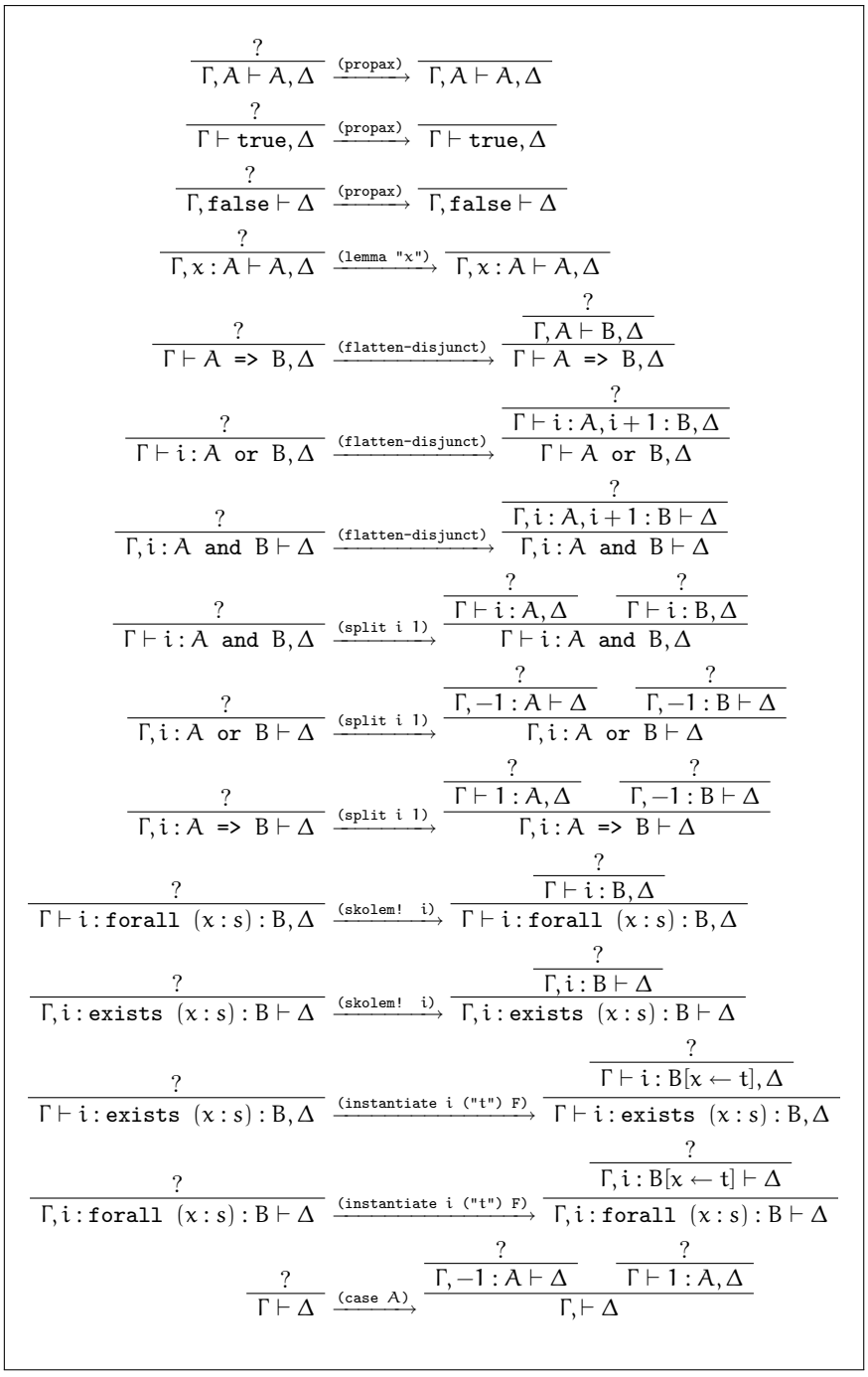


Figure 6.5: The semantics of PVS base tactics

and formulas:

$$
\begin{aligned}
/\texttt{true}/ &= \texttt{true} \\
/\texttt{false}/ &= \texttt{false} \\
/p(t_1,\ldots,t_n)/ &= f_p(/t_1/,\ldots,/t_n/) \\
/\neg A/ &= \texttt{not } /A/ \\
/A \Rightarrow B/ &= \texttt{(}/A/ \texttt{ => } /B/\texttt{)} \\
/A \wedge B/ &= /A/ \texttt{ and } /B/ \\
/A \vee B/ &= /A/ \texttt{ or } /B/ \\
/\forall x^s.A/ &= \texttt{(exists (x : }/s/\texttt{): } /A/\texttt{)} \\
/\exists x^s.A/ &= \texttt{(forall (x : }/s/\texttt{): } /A/\texttt{)}
\end{aligned}
$$

As PVS uses classical sequent calculus, the core of this translation is much more straightforward than the previous one, and holds whether the proof is constructive or not. Note that PVS does not use string labels, but rather formula numbers: this makes the corresponding rules much less robust in the face of potential numbering implementation changes. Let /./ be the proof terms evaluator, carrying as an exponent notation the active formula and its index. Then, recursively, for terms:

$$
\begin{aligned}
/\ltimes/ &= \texttt{(propax)} \\
/x/ &= \texttt{(else\# (propax) (lemma "x"))} \\
/\lambda x^A.\nu/^{i:A\Rightarrow B} &= \texttt{(then\# (flatten-disjunct i 1) } /\nu/^{i:B}\texttt{)} \\
/\lambda x^A.\nu/^{i:\forall x^A.B} &= \texttt{(then\# (skolem! i) } /\nu/^{i:B}\texttt{)} \\
/(\nu,\nu')/^{i:A\wedge B} &= \texttt{(spread\# (split i 1) (}/\nu/^{i:A} \texttt{ } /\nu'/^{i:B}\texttt{))} \\
/(t,\nu)/^{i:\exists x^A.B} &= \texttt{(then\# (instantiate i ("t") F)} \\
&\qquad\qquad /\nu/^{i:B[x\leftarrow t]}\texttt{)} \\
/\mathrm{inj}_l\nu/^{i:A\vee B} &= /\mathrm{inj}_r\nu/^{i:A\vee B} \\
/\mathrm{inj}_r\nu/^{i:A\vee B} &= \texttt{(then\# (flatten-disjunct i 1)} \\
&\qquad\qquad /\nu/^{i:A,i+1:B}\texttt{)} \\
/\mu\alpha^A.\langle\nu\|e\rangle/^{i:A} &= \texttt{(spread\# (case "B") (}/e/^{-1:B} \texttt{ } /\nu/^{1:B}\texttt{))}
\end{aligned}
$$

and for environments:

$$
\begin{aligned}
/\rtimes/ &= \texttt{(propax)} \\
/\alpha/ &= \texttt{(else\# (propax) (lemma "}\alpha\texttt{"))} \\
/v \cdot e/^{i:A\Rightarrow B} &= \texttt{(spread\# (split i 1) (}/e/^{-1:B}\ /v/^{1:A}\texttt{))} \\
/t \cdot e/^{i:\forall x^A.B} &= \texttt{(then\# (instantiate i ("t") F)} \\
&\qquad /e/^{i-1:B[x\leftarrow t]}\texttt{)} \\
/\mathrm{proj}[x,y,\langle v\|e\rangle]/^{i:A\wedge B} &= \texttt{(then\# (flatten-disjunct i 1)} \\
&\qquad /\mu\alpha.\langle v\|e\rangle/^{i:A,i+1:B}\texttt{)} \\
/\mathrm{proj}[x,y,\langle y\|e\rangle]/^{i:\exists x^A.B} &= \texttt{(then\# (skolem! i)} /e/^{i:B}\texttt{)} \\
/[e,e']/^{i:A\vee B} &= \texttt{(spread\# (split i 1)} \\
&\qquad \texttt{(}/e/^{-1:A}\ /e/^{-1:B}\texttt{))} \\
/\tilde{\mu}x^A.\langle v\|e\rangle/^{i:A} &= \texttt{(spread\# (case "B") (}/e/^{-1:B}\ /v/^{1:B}\texttt{))}
\end{aligned}
$$

The intuitionistic case is a simple restriction of the transformation to terms that have only one environment variable.

Example 6.3.10. We illustrate the /./-translation to PVS scripts with the proof from example 6.3.4:

$$\mu *^{(Even\ 2)}.\langle \mu *^{(Even\ 2)}.\langle E_S \| 1 \cdot \mu *^{Odd\ 1}.\langle O_S \| 0 \cdot \mu *^{(Even\ 0)}.\langle E_0 \| *\rangle \cdot *\rangle \cdot *\rangle \| *\rangle$$

This proof term is mapped to the following script:

```
(spread# (case "forall (n:int): (Odd(n) => Even(S(n)))")
  ( (then# (instantiate -1 ("1") F)
      (spread# (split -2 1)
        ( (propax)
          (spread# (case "forall (n:int): (Even(n) => Odd(S(n)))")
            ( (then# (instantiate -1 ("0") F)
                (spread# (split -2 1)
                  ( (propax)
                    (spread# (case "Even(0)")
                      ( (propax)
                        (else# (propax) (lemma "E0"}))))))
              (else# (propax) (lemma "OS")))))))
  (else# (propax) (lemma "ES")))))
```

As previously, for brevity the first η_μ-redex is not considered.

Proposition 6.3.11. *If π_A is a proof of the formula A of $\mathcal{L}^1_m$ in* LK*, then $/\pi_A/$ is a proof in PVS of the formula $/A/$.*

Proof. By induction on the length of the typing derivation of π_A, and by case by logical connective. □

Corollary 6.3.12. *If π_A is a proof of the formula A of $\mathcal{L}^1_m$ in* LJ*, then $/\pi_A/$ is a proof in PVS of the formula $/A/$.*

6.4 Natural language

As a testament to the richness of the $\bar{\lambda}\mu\tilde{\mu}$ datatype, we expose natural language rendering function. While this section is obviously not as formal as the previous, it answers nonetheless to an essential necessity: make the proofs available for understanding by a human mind.

The challenge in such a translation comes from the opulence of natural language: a much detailed translation into a scarce subset of the human language is all but readable. Here the advantage of having additional binders — μ and $\tilde{\mu}$ — for proof terms, as well as native constructors for logical connectives, is a great asset. The translation process takes full advantage of these constructs to allow high level notions to pervade the final rendering.

This translation, denoted $|.|$, builds upon a similar work (Sacerdoti Coen, 2006). Here a superscript annotation is used to disambiguate the type of the translated proof term. Recursively, starting with terms:

$$
\begin{aligned}
|x| &= \textit{by } x \\
|\ltimes| &= \textit{by definition of True} \\
\left|\lambda x^A.v\right|^{A\Rightarrow B} &= \textit{assume } A \textit{ (} x \textit{), } |v| \\
\left|\lambda x^A.v\right|^{\forall x^A.B} &= \textit{consider an arbitrary but fixed } x \textit{ of type } A\textit{, } |v| \\
|(v,v')|^{A\wedge B} &= |v| \textit{ and } |v'| \\
|(t,v)|^{\exists x^A.B} &= |v| \\
|\mathrm{inj}_l v| &= |v|\textit{, trivial} \\
|\mathrm{inj}_r v| &= |v|\textit{, trivial} \\
\left|\mu\alpha^A.\langle v\|e\rangle\right| &= \textit{we need to prove } A\textit{: (} |c| \textit{)}
\end{aligned}
$$

for environments:

$$
\begin{aligned}
|\alpha| &= \textit{done}\\
|\rightarrowtail| &= \textit{absurd}\\
|v \cdot e|^{A \Rightarrow B} &= \textit{and}\, |v|, |e|\\
|t \cdot e|^{\forall x^A . B} &= \|e\|\\
|\mathrm{proj}[x, y, \langle v \| e \rangle]|^{A \wedge B} &= \textit{we proved}\, A\ (x)\ \textit{and}\, B\ (y),\, |e|\\
|\mathrm{proj}[x, y, \langle y \| e \rangle]|^{\exists x^A . B} &= \textit{let}\, x\ \textit{be the element of type}\, A\ \textit{which satisfies the property},\, |e|\\
|[e, e']| &= \textit{by cases: first case: (by case hypothesis}\, |e|\textit{)}\\
&\qquad\qquad \textit{second case: (by case hypothesis}\, |e'|\textit{)}\\
\left|\tilde{\mu} x^A . \langle v \| e \rangle\right| &= \textit{we proved}\, A\ (x),\, |c|
\end{aligned}
$$

and for commands:

$$|\langle v \| e \rangle| = |v|, |e|$$

Note that the witnesses in the case of the quantifiers are hidden. This choice was motivated by the wish to produce a relatively high-level description of the proof. Technical details such as the substitution of variables tend to clutter the proof, and are easily inferred by the reader.

Example 6.4.1. Assume given a type N, a binary predicate symbol $p : N \to N \to \mathrm{bool}$, and two formulas:

$$
\begin{aligned}
A_1 &= \exists y^N . \forall x^N . (p\ x\ y)\\
A_2 &= \forall x^N . \exists y^N . (p\ x\ y)
\end{aligned}
$$

We translate in natural language the intuitionistic proof of $A_1 \Rightarrow A_2$, *i.e.* that existential quantifiers can be pushed under universal quantifiers [4].

To improve readability, we indent the resulting script to reflect the ex-

[4] In particular, if N is the poset equipped with the partial ordering $p = \prec$, then this is the property of correctness of its supremum.

pression's parenthesizing.

$$\left|\begin{array}{l}\mu*^{A_1\Rightarrow A_2}.\langle\lambda H_1^{A_1}.\mu*^{A_2}.\langle H_1\|\ \mathrm{proj}[y,z,\\ \quad\langle z\|\ \tilde{\mu}H_2^{\forall x^N.(p\ x\ y)}.\langle\lambda x^N.(y,\mu*^{(p\ x\ y)}.\langle H_2\|x\cdot *\rangle)\|*\rangle\rangle]\rangle\|*\rangle\end{array}\right| =$$

we need to prove $A_1 \Rightarrow A_2$*:*
(assume A_1 *(*H_1 *),*
we need to prove A_2*:*
(by H_1*,*
let y *be the element of type* N *that satisfies the property,*
we proved $\forall x^N.(p\ x\ y)$ *(*H_2 *),*
consider an arbitrary but fixed x *of type* N*,*
we need to prove $(p\ x\ y)$*:*
(by H_2*,*
done),
done),
done)

Remark that this translation can be easily extended to randomize natural language constructions, in order to further the ease of reading rendered proofs.

6.5 Discussion

Let us spare a few words on the advantages and drawbacks of the various translations presented in this chapter. We consider several aspects of the problem, ranging from extensibility to robustness, from performance to practical availability.

This dissertation in general, and this chapter in particular, puts a large emphasis on two proof assistants: Coq and PVS. All the translation algorithms are designed with these provers in mind, thus one could ask if these ideas can apply to other formal tools. The short answer to this question is: yes, provided these tools provide an input method for their proofs. The form of the input is not relevant, because the translation can accommodate both proof terms *à la* Curry-de Bruijn-Howard[5], and tactics. A more detailed answer lies in the examination of the details of the considered prover: if one wants to use the "proof scripts" approach, reliable semantics for the tactic language are mandatory. For the "Frege-Hilbert system" method, the obstacle is the availability of the "auto" tactic.

As exposed in the first words of this chapter, of great concern is the resilience of the translation algorithms: how vulnerable they are to changes in the target formalisms. Needless to say, the most stable translation is the

[5] Although there exist proof term mechanisms that do not rely on λ-calculus (and $\bar{\lambda}\mu\tilde{\mu}$-calculus is an easy illustration of this fact) most of them are convertible to it.

natural language rendering: it is difficult to imagine an API change in this domain. The morphism of proof terms is almost as good as that: proof terms are at the core of the proof engine, and very rarely altered, at least in their root constructs (*i.e.* the subset of these languages that correspond to λ1). The translations to Frege-Hilbert systems and to proof scripts are both hampered by their reliance on the proof language. For the first one, the critical elements are reduced to the two tactics "modus ponens" and "auto". For the second one however, the tactics for first-order inferences, and a few strategies constitute a non-negligible number of sources for breakdowns. This vulnerability is worsened by the relative immaturity of proof languages in general, and the instability that results. Yet in some cases (*e.g.* PVS), such translations are the only way to communicate a proof.

But maybe most important is the efficiency of the translated proofs in the target frameworks. Indeed, if large contributions are to be exchanged between various formal libraries, then it is fundamental to have each proof checked in a reasonably short period of time. On this aspect, since the 20th century, the human mind is no longer the fastest tool available. However our translations to machine tools are not equal in efficiency. Generating and checking a tree of "modus ponens" and "auto" is very costly, especially when large numbers of lemmas are used in the context. Using proof scripts as targets is somewhat more efficient, but some performance is lost in the unification algorithm and the input / output mechanism. When possible, the use of proof terms as target for the translation yields the fastest proof-checking results.

Overall, all these approaches have advantages and drawbacks. In some cases, some optimizations can be made to take advantage of the specificities of the target prover. But in general, all these approaches are necessary to be able to deal as robustly as possible with the large number of frameworks available nowadays.

6.6 Looking forward

In this chapter, we have constructively detailed several translations methods for proofs between frameworks sporting various features. We compared their strengths and weaknesses, allowing when possible to choose between the different methods to better meet expectations.

This work on proofs is trivially extended to deal with toplevel declarations, and it is well known that this translation does not impact the proofs themselves (Severi and Poll, 1994). As a result, it is possible to process complete specification files, and translate them into corresponding specifications in the target frameworks. Thus, interoperability is achieved.

All these algorithms for proof translation have been implemented, and to a certain extent tested, in Fellowship (more on this tool in chapter 7). The next step would be to integrate such translation tools directly into the

mainstream provers. This would allow, for instance PVS to generate Coq specifications and proofs, and vice-versa. Another direction for further work could be to try and enhance the source framework with richer constructs. The deciding factor here lies in the target system, and less in the strength of its formalism than in the availability, and validity, of its semantics.

7 Fellowship is a Super Prover

In this chapter we describe the specifications of a piece of software, named Fellowship, serving as an interoperable front end to other procedural provers.

Fellowship serves as a real-world validation platform for the concepts introduced in this manuscript. Its design embraces the $\bar{\lambda}\mu\tilde{\mu}$ logical formalism, and includes a proof language that implements the local semantics of tactics and the proof monad-based semantics for strategies; it leverages the features of these concepts to facilitate scientist to proof engine dialogue. Finally it leverages the translation algorithms to achieve interoperability with other theorem provers, thus being called a "super prover".

Although the Fellowship tool can be used to build and check proofs, it is mainly a way to organize and export formal developments. Thus we argue that Fellowship can be called a *proof manager*, in the sense that it handles proofs in a more general sense than any other proof assistant. As we will see in this chapter, this orientation has several consequences in the overall design choices and structure of the implementation of Fellowship.

⋆

The Fellowship tool was originally though out as a repository of proofs in first-order predicate sequent calculus, with exporting mechanisms towards other formal tools such as Coq and PVS. While this focus on interoperability remains unchanged, an interactive layer was adjoined to permit the development of proofs within Fellowship itself, thus making it into a proof assistant. The combination of these two aspects now allows the simultaneous development of specifications and proofs for both provers.

Technically, Fellowship is implemented in Ocaml by Claudio Sacerdoti Coen and the author. The code is about 4000 lines in total, licensed under the CeCILL free software license, version 2.0. It features a toplevel mode for interactive proof development, and a compiler mode for batch proof-checking. Fellowship can be found online at (Kirchner and Sacerdoti Coen, 2007); it is distributed since 2005 and is still under active development.

Fellowship is intended to be used for developing formal specification and proof libraries that are common to several provers, and do not make much use of higher-order logics. Such libraries are somewhat widespread amongst the various formal methods communities: often efforts had to be replicated to re-implement local developments into different tools (Mayero, 2001; Muñoz and Mayero, 2001; Daumas et al., 2001; Boldo and Muñoz, 2006).

7.1 Design principles

Fellowship, as a proof assistant, is required to meet postulates 1 and 2. But it is aimed at being more than a proof assistant: to be a proof manager. As

highlighted in chapter 6, this change of finality has several consequences at the level of the logical framework. Thankfully these constraints come close to the ones entailed by the de Bruijn postulates: Fellowship is based on a simple framework with understandable semantics, and it provides a choice of logics to work with.

The logical framework of the prover is based on the work presented in chapter 2. The specification language is $\mathcal{L}^{1}_{\mathrm{m}}$, allowing for multisorted first-order constructions. Classical, intuitionistic, and minimal logics are made available, and comply with the definitions of LJ, LK, LJM and LKM. Finally, proof terms are written in the $\bar{\lambda}\mu\tilde{\mu}$ and $\bar{\lambda}^{*}\mu\tilde{\mu}$-calculi. These proof terms are generated on the fly, as in Alf, rather than being produced during the proof post-processing phase (Coq takes such an approach). This allows the display of additional information, derived from the proof terms, during the proof editing process. For instance, one can wish to overview in such a manner the construction of the exported proof.

The conception of the proof language is based on the work of chapter 5. In particular, it articulates around a datatype similar to the proof monad, enriched to deal with instructions. In addition to this mechanism, tactics take advantage of the liveliness property of the sequent calculus: there is always an active formula in a sequent, and as a consequence there is always an active topmost formula constructor. Hence only four base tactics are introduced:

cut which corresponds either to the $\mathrm{cut}_{\mathcal{L}}$ or the $\mathrm{cut}_{\mathcal{R}}$ inference rule,

axiom which refers to either the $\mathrm{ax}_{\mathcal{L}}$ or $\mathrm{ax}_{\mathcal{R}}$ inference,

elim which is the union of all left and right the connector rules, and introduces (or eliminates, from a bottom-up perspective) the active topmost formula constructor,

weaken which corresponds to either the left or the right weakening rule.

More elaborate tactics are derived by combining the base tactics, either using Ocaml code or the strategy language. The latter is based on the proof monad operators, and provides basic combinatory facilities.

The core feature of this prover is, as mentioned earlier, interoperability. The design of this particular functionality has to meet two criteria: first, maintainability, due as highlighted in chapter 6 to the changing nature of proof assistants. Extensibility is second, because in the long run the aim is to address the majority of tomorrow's seasoned formal systems; thus compatibility with these should be easy to implement. As a consequence, the interoperability layer is implemented by associating fields containing particular prover-specific syntax to fields representing abstracted common manipulations (*e.g.* "declare a variable", "initiate a proof" or "apply an inference rule"). This kind of record datatype is easily serviceable and augmented,

even with little knowledge in Ocaml, and it is easily convertible if need be into a full-fledged, language agnostic database.

Finally, an important design principle is one of modularity. It is not a gratuitous constraint, but more of the developers taking note of the pace of innovation in the domain of formal methods. The added benefit is one of maintainability, each of the software components being implemented independently. In particular, the three aforementioned features — the logical framework, the proof language, and the interoperability functions — are as many interchangeable modules.

7.2 Structure

This implementation of a theorem prover is in general similar to other developments: it consists in a proof engine, with which the user interacts by using a proof language to build proofs. However, while in most other provers the finality of this process is to discard proof challenges and perhaps constitute a library by collecting them, in Fellowship the goal is to export the finished proofs and libraries into other theorem provers.

In Fellowship, all these features are implemented as several more or less independent modules, each of them in charge of a specific functionality. To each module corresponds one physical `ml` file.

- A module contains the definition of the abstract syntax of the prover: terms, sorts and formulas, as well as tactics and strategies. It also contains the functions closely related to these objects: for instance the type-checking algorithms.
- A module defines the proof structure: sequents, proof state, proof monad; and all the related functions: initialization, accessing, pretty-printing, conversion, *etc.*.
- A module regroups the collection of the functions that are used to interact with the prover outside of the proof-editing mode. This includes the front-end for the exporting mechanisms.
- A module contains the set of functions that are used to interact with the prover in proof-editing mode: the implementation of tactics and strategies.
- Two modules provide the back-end implementation of the exporting algorithms. They also contain the datatype describing the mapping into the languages of the target systems.

These modules are complemented with a lexer and parser implementation, a main program entry point and a help message database.

Table 7.1: Concrete notation for $\mathcal{L}^1_m$ in Fellowship

Abstract syntax	Concrete syntax
bool	`bool`
$s \to r$	`s -> r`
$\top$	`true`
$\bot$	`false`
$f(t_1, \ldots, t_n)$	`f(t1)...(tn)`
$p(t_1, \ldots, t_n)$	`p[t1]...[tn]`
$\neg A$	`~ A`
$A \Rightarrow B$	`A -> B`
$A \wedge B$	`A /\ B`
$A \vee B$	`A \/ B`
$\forall x^s.A$	`forall x:s, A`
$\exists x^s.A$	`exists x:s, A`

7.3 The logical frameworks

The logics implemented faithfully follow the systems LJ, LK, LJM and LKM described in chapter 2. Proof terms are generated by successive instantiation of open terms. Depending on the logics, either the $\bar{\lambda}\mu\tilde{\mu}$ or the $\bar{\lambda}^*\mu\tilde{\mu}$-calculus are used.

The correspondence between abstract and concrete syntax of first-order predicate formulas is given by Table 7.1. Remark that function symbols use parenthesis to structure their arguments, while predicate symbols use square brackets. The proposition $(p\ t) \wedge (q\ u)$, where `p` and `q` are unary predicate symbols and `t` and `u` are terms, read `(p[t] \/ q[u])` in Fellowship syntax.

7.4 Syntax and semantics of the interaction language

The proof language follows closely the theory in chapters 4 and 5. A number of instructions are added to deal with the proof management feature.

Definition 7.4.1 (Instruction language)**.** The following instructions are part of the non-proof-editing mode of Fellowship:

i ::= `lj` | `lk` | `minimal` | `full` | `help t`
| `declare [x:s|x:t|x:A]` | `theorem x:A` | `next` | `prev`
| `qed` | `undo` | `open f` | `checkout p`
| `discard [theorem|all]` | `quit`

Their semantics are given informally:

- `lj`, `lk`, `minimal` and `full` are flags enabling the corresponding logical frameworks. These can only be used at the beginning of a development, *i.e.* while no proof has been carried.
- `declare` adds sort, term or formula constants, as well as axioms, to a global environment.
- `theorem` starts the proof of a formula. It enters proof-editing mode, sets the formula as the consequence of an empty sequent, and performs an η-expansion of the proof term in order to retain the theorem formulation.
- `next` and `prev` are interactive proof management instructions. They trigger the switch to the next (*resp.* previous) open goal, as formalized in chapter 4.
- `qed` concludes a proof where all branches are closed. It stores the theorem in the global environment, and saves with its proof term.
- `checkout` exports the global environment into a file, and runs the corresponding prover on this file.
- `discard` either clears the theorem being locally proved, or the whole proof state.

Definition 7.4.2 (The base tactic language)**.** The following tactics implement the inference rules of the logical framework of Fellowship:

$$\mathbf{t} ::= \texttt{axiom}\ \chi \mid \texttt{cut}\ \chi\ (A) \mid \texttt{weaken}\ \chi \mid \texttt{idtac} \mid \texttt{elim}\ \xi$$

where the argument of `elim`, ξ, can either be empty, or be en identifier, or a sequence of two identifiers and a formula, or the `left` or `right` keyword, or a term.

Figures 7.1, 7.2 and 7.3 provide the small-step operating semantics of the four basic tactics, in a classical logical framework, using the formalism developed in chapter 3. The intuitionistic and minimal cases are trivial variations on these semantics.

A few remark follow on these semantics:

- The semantics of these tactics in the case of intuitionistic logics is similar to the classical case, except that the number of formulas in the right-hand side of the sequent is restricted to one.
- For minimal logics, the introduction rule for $\bot$ is obviously removed. The negation symbol is treated as an implication, and discharged as such.
- A few facilities have been implemented in these tactics in order to simplify the management of identifiers. Hence `axiom` can be used with no argument, and will search the sequent for a proposition that matches

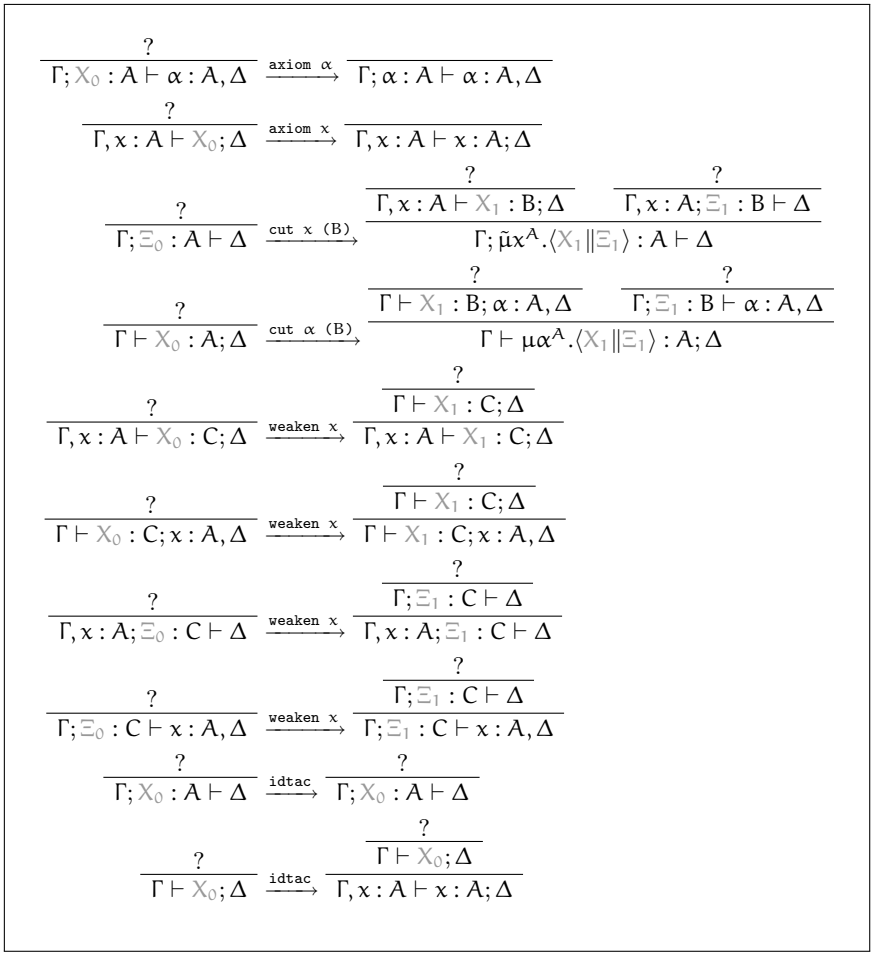


Figure 7.1: The semantics of Fellowship's `axiom`, `cut`, `weaken` and `idtac` tactics

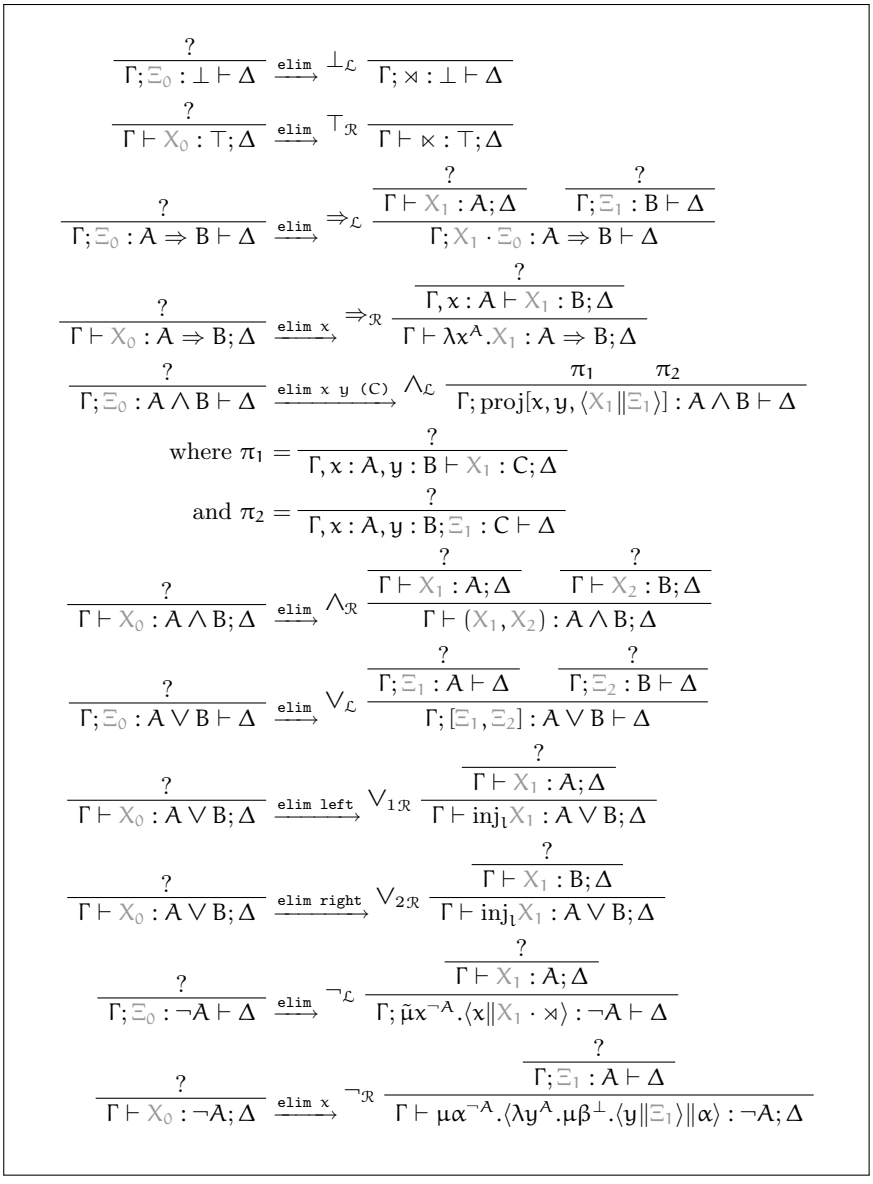


Figure 7.2: The semantics of Fellowship's `elim` tactic — propositional connectives

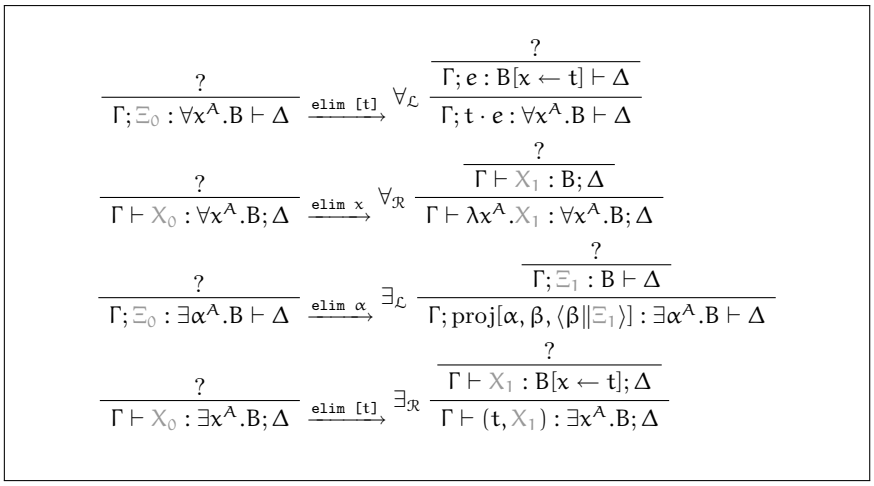


Figure 7.3: The semantics of Fellowship's `elim` tactic — quantifiers

the active formula. Quantifier introduction can be achieved in batch mode, *i.e.* treating a list of similarly-quantified variables in one step. Finally, although it goes against good coding practice, automatic identifier generation is available.

Definition 7.4.3 (Extended tactics)**.**

$$\mathbf{t}' ::= \texttt{focus x y} \mid \texttt{contraction x} \mid \texttt{apply x}$$

These are constructs based in the base tactics, whose definitions are:

`focus` x y = `cut` y `(`A`)` `;` `[axiom | idtac]`
where x : A is a hypothesis
`focus` x y = `cut` y `(`A`)` `;` `[idtac | axiom]`
where x : A is a conclusion
`contraction` x = `cut` x `(`A`)` `;` `[axiom | idtac]`
if the active formula A is in the hypothesis
`contraction` x = `cut` x `(`A`)` `;` `[idtac | axiom]`
if the active formula A is in the conclusion
`apply` x = `elim [t] ; elim ; elim ; [axiom ; idtac]`
if $x : \forall y^s.\exists z^r.A \Rightarrow B$ is a hypothesis
and the active fornula $B[y \leftarrow t]$ is in the hypothesis
`apply` x = `elim [t] ; elim ; elim ; [idtac ; axiom]`
if $x : \forall y^s.\exists z^r.A \Rightarrow B$ is a hypothesis
and the active fornula $B[y \leftarrow t]$ is in the conclusion

where `;` and `;[||]` denote strategies (see definition 7.4.4). Also remark that the semantics of `apply` extend to the case where any number of quantified variables are mixed in the active formula, or when the implication $A \Rightarrow B$ collapses into B.

The tactic language of Fellowship is the union of the base tactic and the extended tactic languages: $\mathbf{t} \cup \mathbf{t}'$.

Definition 7.4.4 (Strategy language)**.**

$$\mathbf{s} ::= \mathbf{t}\ \texttt{in x y} \mid \mathbf{t}\ \texttt{;}\ \mathbf{s} \mid \mathbf{t}\ \texttt{;}\ \texttt{[}\mathbf{s_1}\ \texttt{|}\ \ldots \texttt{|}\ \mathbf{s_n}\texttt{]}$$

The semantics of most of these strategies are similar to the one provided in chapter 4. As for the remaining **t** `in x y` construct, it is a macro for the script `focus x y ;` **t**.

7.5 Interoperability

This tool implements all the translation functions defined in chapter 6, adapted for their use in Coq and PVS. In detail, this means that in the first translation function \.\, the generic tactics "modus ponens" and "auto" are instantiated respectively:

- in Coq by the proof scripts `cut` and `firstorder`;

· in PVS by the proof scripts `(case)` and `(then (flatten) (assert) (grind))`.

What is more, the morphism of proof terms ‖.‖ is trivially adapted to Coq's syntax. The remaining translations are implemented as they are exposed in chapter 6.

7.6 Example

The proof that 2 is even illustrates the unfolding of a proof in Fellowship. The ascii symbols "\" and ";:" are used to respectively represent the λ and μ symbols. In the proof editing part, the open lambda-term and the partial natural language rendering of that term are printed at each step.

First we launch Fellowship: the "`-ascii`" option forces the formatting of the output.

```
% ./launcher.sh -ascii

    > Welcome! This is FSP version 0.2.0.
      To get help type "help" followed by a dot.
      Current logic: intuitionistic sequent calculus.
```

Before anything, we specify the logical setting of our development: here, LJM. Then the various sort, function and predicate symbols of the theory are declared: the sort of natural numbers, their constructors, and two predicate symbols "`Even`" and "`Odd`". We also provide an axiomatic definition of these predicate symbols.

```
fsp < minimal. lj.
      declare N: type.
      declare O: N.
      declare S: N -> N.
      declare Even: N -> bool.
      declare Odd: N -> bool.
      declare EO: (Even [O]).
      declare OS: (forall n:N, Even [n] -> Odd [S n]).
      declare ES: (forall n:N, Odd [n] -> Even [S n]).

    > Current logic: minimal intuitionistic sequent calculus.
      N defined.
      O defined.
      S defined.
      Even defined.
      Odd defined.
      EO defined.
      OS defined.
      ES defined.
```

Now the proof can begin: we state the theorem that is proven, and Fellowship enters the proof-editing mode. Remark that the metavariable in the proof term are printed using a numbered question mark, here "`?1`". In natural rendering, it is represented by an ellipsis "`...`", and in addition the current goal is marked with an arrow "`<======`".

```
fsp < theorem even_2: (Even [S (S O)]).

    > Proof term:
      ;:thesis:Even (S (S O)).<?1||thesis>
      Natural language:
      we need to prove Even (S (S O))
        .... (1) <======
      done

      1 goal yet to prove!

       |-----  1
      *:Even (S (S O))
```

The first tactic applied consists in focussing on the axiom that states the inductive case of the definition of the "`Even`" predicate symbol. The axiom, which was left implicit in the antecedent, is now printed; the active formula, denoted by a "`*`" prefix, is switched to this axiom.

```
fsp < focus ES th.

    > Proof term:
      ;:thesis:Even (S (S O)).<;:th:Even (S (S O)).<ES||?1.2>
      ||thesis>
      Natural language:
      we need to prove Even (S (S O))
        we need to prove Even (S (S O))
          by ES
        ...(1.2) <======
      done

      1 goal yet to prove!

      *:forall n:N,Odd n->Even (S n)
       |-----  1.2
      th:Even (S (S O))
```

Then we apply the $\forall_{\mathcal{L}}$ rule, and we provide a term "`S O`" for the instantiation of the quantified variable.

```
fsp < elim [S O].
```

```
> Proof term:
  ;:thesis:Even (S (S O)).<;:th:Even (S (S O)).<ES||S O*
  ?1.2.1>||thesis>
  Natural language:
  we need to prove Even (S (S O))
    we need to prove Even (S (S O))
      by ES
    ...(1.2.1) <======
  done

  1 goal yet to prove!

  *:Odd (S O)->Even (S (S O))
   |-----  1.2.1
  th:Even (S (S O))
```

The next step is to introduce the implicative connective, which generates two subgoals: the first one is that 1 is odd, and the second one is that 2 is even assuming that 2 is even. The latter being trivial, it is discarded using the $\text{ax}_{\mathcal{L}}$ rule.

```
fsp < elim ; [idtac | axiom].
```

```
> Proof term:
  ;:thesis:Even (S (S O)).<;:th:Even (S (S O)).<ES||S O*
  ?1.2.1.1*th>||thesis>
  Natural language:
  we need to prove Even (S (S O))
    we need to prove Even (S (S O))
      by ES
      and .... (1.2.1.1) <======
    done
  done

  1 goal yet to prove!

   |-----  1.2.1.1
  *:Odd (S O)
```

The remaining goal is treated as the previous one, using the inductive case of the definition of the "Odd" symbol, and then the base case of the definition of the "Even" symbol. Remark that, while tactics here have been kept simple, the "apply" tactic of Fellowship would have sped the proof, by automatically finding and implementing the appropriate introductions to complete the proof.

```
fsp < focus OS th; elim [O] ; elim; [focus EO th; axiom
      | axiom].

    > Closed the last branch:
      Proof completed!

      Proof term:
      ;:thesis:Even (S (S O)).<;:th:Even (S (S O)).<ES||S O*
      ;:th:Odd (S O).<OS||O*;:th:Even O.<EO||th>*th>*th>
      ||thesis>
      Natural language:
      we need to prove Even (S (S O))
        we need to prove Even (S (S O))
          by ES
          and we need to prove Odd (S O)
            by OS
            and we need to prove Even O
              by EO
            done
          done
        done
      done
```

Finally, we save the proof and exit the proof-editing mode, before exporting it to Coq and PVS.

```
fsp < qed.

    > even_2 defined.


fsp < checkout coq.

    > Proof checked successfully by Coq.


fsp < checkout proof term coq.

    > Proof checked successfully by Coq.


fsp < checkout pvs.

    > Proof checked successfully by PVS.
```

```
fsp < quit.

    > Out.
```

7.7 Future work

As a prototype proof assistant, Fellowship succeeds in proving the feasibility and elegance of basing a proof assistant on the proof monad, and a structured proof language. And as a prototype proof manager, it serves as a testbed and validator for interoperability explorations. While still a beta piece of software, relatively large developments have been carried out using it.

As usual in this type of project, a number of possible future implementations are possible. The ones that follow, all are efforts in attempt to provide the working scientist more of the choice advocated by postulate 2.

Investigation has started to understand how to include the subtraction (Rauszer, 1974; Crolard, 2001) in our logical frameworks. This would provide the syntax with a dual to the implication operator and its inference rules. However this is not without problems, one of them being the definition of the intuitionistic restriction in the presence of this operator. A necessary and sufficient condition (Dyckhoff, 1992) is to apply this restriction only to the upper sequent of the $\Rightarrow_{\mathcal{R}}$ inference rule. By duality, this restriction would extend to the $-_{\mathcal{L}}$ rule of subtractive logic, which should be made to contain at most one hypothesis (Crolard, 2001).

At the level of the proof language, some time can still be spared to implement the most advanced features in chapter 5. In particular, the prospect of having a typing system for this language would dramatically improve ease of use. Another request is to design a declarative proof language with close ties to the natural language used in chapter 6. It should be interesting to study the prospects offered in this domain by the $\bar{\lambda}\mu\tilde{\mu}$-calculus.

Finally, on the interoperability topic, work should go on in adding other proof assistants to the translation function. Support for the Isabelle framework is already underway, but the task requires a good understanding of the target formalisms, which is not always the authors' case. What is more, the prospect of "popping the hood" and tinkering with Ocaml code is not always appealing to external proof assistant maintainers. One way to alleviate the burden of such a work would be to base the translation on simple xml files, allowing for simple if not painless edition.

8 Classes

Throughout this manuscript the basis of our work has been a weak logical framework: first-order logic. Now that we have demonstrated the relevance of this framework for interoperability purposes, we need to prove that its domain of application is not limited to trivial examples. In this chapter, we present a theory that validates this requirement. *We expose a formalism that allows the expression of any theory with one or more axiom schemes into first-order predicate logics, using a finite number of axioms.*

This allows us to give finite first-order axiomatizations of second-order theories such as arithmetic and real closed fields theory. We also derive a presentation of arithmetic in deduction modulo that has a finite number of rewrite rules. Overall, this formalization relies on a weak calculus of explicit substitutions to provide a simple and finite framework.

This work is part of a long term project investigating the possibility to base proof checkers on weaker frameworks, such as first-order logic. The maturity of these frameworks make them very secure centerpieces of formal tools in general, and proof checkers in particular; some designs (e.g. (Ridge and Margetson, 2005)) have already taken advantage of them. It is essential in this project to implement strong theories, such as arithmetic, real closed fields or set theory, with a finite number of axioms. The main result of this chapter is a systematic way of conducting these implementations, using a theory of classes.

$\star$

In mathematics, some theories — such as arithmetic or set theory — are often expressed using an infinite number of axioms. This is achieved through the use of one or more *axiom schemes*, i.e. sets of axioms, often described within the meta-theory. For instance the induction scheme in arithmetic can be expressed as: for any proposition P,

$$\mathsf{P}(0) \Rightarrow \forall y, (\mathsf{P}(y) \Rightarrow \mathsf{P}(S(y))) \Rightarrow \forall z, \mathsf{P}(z)$$

This scheme, parametrized by the *schematic variable* P that takes values in the set of formulas of arithmetic, generates an infinite number of axioms.

The use of these axiom schemes can be avoided though, by introducing a new sort of objects, *classes* (which we will distinguish from other objects by using uppercase letters), and a membership symbol $\in$. Using classes as representatives for propositions, the induction scheme is re-written as a single axiom:

$$\forall E, (0 \in E \Rightarrow \forall y, (y \in E \Rightarrow S(y) \in E) \Rightarrow \forall z, z \in E)$$

However, in order for classes to soundly emulate propositions, one needs to guarantee that any proposition has an associated class. This is assured by

the comprehension axiom scheme, which states: for any proposition P that is well formed in the original language,

$$\exists E, \forall x, (x \in E \Leftrightarrow P)$$

This extension of arithmetic is *conservative*, meaning that any formula that is provable in the theory of arithmetic plus classes, involving symbols of arithmetic only, was already provable in arithmetic. Also note that some axiom schemes might have propositions with *two* free variables or more, and that to deal with this in the most general way, one needs not only classes of objects but classes of n-tuples of objects. Therefore, one will for instance introduce a sort of lists, list constructors and adapt the symbol $\in$ and the comprehension axiom scheme to this new structure.

In the end, however, the theory would still have an axiom scheme that generates an infinite number of axioms.

This is not a zero-sum game, though. Previous works (von Neumann, 1925; Bernays, 1958; Gödel, 1940; Mendelson, 1997; Vaillant, 2002) have shown that, in the case of set theory, it is possible to reduce the comprehension axiom scheme to a finite number of axioms. However we believe that the notion of class is independent of set theory and can be extended to express any theory containing axiom schemes in a finite first-order axiomatization. Moreover, unlike in the previous systems, we will see that the axioms in the theory of classes can easily be oriented as rewrite rules.

8.1 A theory with the comprehension scheme

We consider a language $\mathcal{L}^{1=}$ in first-order predicate logic with equality, and we call Σ its *finite* signature. Let $\mathcal{T}$ be an intuitionistic theory of this language that has one or more axiom schemes, i.e. axioms of the form:

$$s(P(t_1^1, \ldots, t_n^1), \ldots, P(t_1^p, \ldots, t_n^p))$$

where n and p are natural numbers depending on each scheme, s is a p-ary first-order formula, the t_j^i are terms of $\mathcal{L}^{1=}$ and P is a schematic formula variable. Remark that, although the axiom schemes in this chapter only have one schematic variable P, the generalization of our results to a system with arbitrary axiom schemes is straightforward.

Definition 8.1.1 ($\mathcal{L}^{cs}$)**.** Let $\mathcal{L}^{cs}$ be a many-sorted language with three sort symbols: the sort of objects X, lists L, and classes C. In $\mathcal{L}^{cs}$, Σ is finitely extended into a signature Σ^{cs} with a predicate symbol $\in$ of rank (L, C) and two function symbols:

$$\begin{aligned} &\mathit{nil} : L \\ &:: : (X, L)L \end{aligned}$$

Notation. We use $\langle x_1, \ldots, x_n \rangle$ as syntactic sugar for the term $x_1 :: \ldots :: x_n :: nil$.

Definition 8.1.2 ($\mathcal{T}^{cs}$)**.** Define $\mathcal{T}^{cs}$ as the theory of the language $\mathcal{L}^{cs}$ derived from $\mathcal{T}$ by adding the comprehension scheme:

$$\exists E : C, \forall x_1, \ldots, \forall x_n : X, \qquad (\langle x_1, \ldots, x_n \rangle \in E \Leftrightarrow P) \qquad (\mathcal{T}^{cs}_1)$$

where P is built with the symbols of $\mathcal{L}^{1=}$, and may contain some of the x_i as free variables. Axiom schemes are replaced by axioms:

$$\forall E : C, \qquad s(\langle t^1_1, \ldots, t^1_n \rangle \in E, \ldots, \langle t^p_1, \ldots, t^p_n \rangle \in E) \qquad (\mathcal{T}^{cs}_2)$$

Example 8.1.3. Following the translation of definition 8.1.2, the induction axiom scheme:

$$P(0) \Rightarrow \forall y, (P(y) \Rightarrow P(S(y))) \Rightarrow \forall z, P(z)$$

is replaced by the following axiom scheme and axiom:

$$\exists E : C, \forall x_1, \ldots, \forall x_n : X, (\langle x_1, \ldots, x_n \rangle \in E \Leftrightarrow P)$$
$$\forall E : C, \langle 0 \rangle \in E \Rightarrow \forall y, (\langle y \rangle \in E \Rightarrow \langle S(y) \rangle \in E) \Rightarrow \forall z, \langle z \rangle \in E$$

Proposition 8.1.4. *$\mathcal{T}^{cs}$ is an extension of $\mathcal{T}$.*

Proof. We need to show that in $\mathcal{T}^{cs}$, for any P,

$$s(P(t^1_1, \ldots, t^1_n), \ldots, P(t^p_1, \ldots, t^p_n))$$

is provable. This is immediate, by the comprehension scheme ($\mathcal{T}^{cs}_1$) and the axioms ($\mathcal{T}^{cs}_2$). □

We have now all ingredients to move towards our goal, finding a finite axiomatization of the comprehension scheme.

8.2 Finite class theory

Setting and notations.

Definition 8.2.1 ($\mathcal{L}^{ws}$)**.** We extend $\mathcal{L}^{cs}$ with the function symbols:

$1 : X$	$\varnothing : C$
$S : (X)X$	$\cap, \cup, \supset : (C, C)C$
$\cdot[\cdot] : (X, L)X$	$\mathcal{P}, \complement : (C)C$

To each predicate symbol p is associated a function symbol $\dot{p}$ of similar arity, which constructs elements of sort C. We call $\mathcal{L}^{ws}$ this language, and Σ^{ws} its finite signature.

Definition 8.2.2 ($\mathcal{T}^{ws}$)**.** Let $\mathcal{T}^{ws}$ be the theory of the language $\mathcal{L}^{ws}$ formed with the following axioms for explicit substitutions:

$$\forall x : X, \quad x[nil] = x \qquad (\mathcal{T}^{ws}_1)$$
$$\forall \ell : L, \forall x : X, \quad 1[x::\ell] = x \qquad (\mathcal{T}^{ws}_2)$$
$$\forall \ell : L, \forall x, y : X, \quad S(y)[x::\ell] = y[l] \qquad (\mathcal{T}^{ws}_3)$$
$$\forall \ell : L, \forall x_1, \ldots, x_n : X, \quad f(x_1, \ldots, x_n)[\ell] = f(x_1[\ell], \ldots, x_n[\ell]) \qquad (\mathcal{T}^{ws}_4)$$

and the axioms for proposition encoding:

$$\forall \ell : L, \forall x_1, \ldots, x_n : X, \quad \ell \in \dot{p}(x_1, \ldots, x_m) \Leftrightarrow p(x_1[\ell], \ldots, x_m[\ell]) \qquad (\mathcal{T}^{ws}_5)$$
$$\forall A, B : C, \forall \ell : L, \quad \ell \in A \cap B \Leftrightarrow \ell \in A \wedge \ell \in B \qquad (\mathcal{T}^{ws}_6)$$
$$\forall A, B : C, \forall \ell : L, \quad \ell \in A \cup B \Leftrightarrow \ell \in A \vee \ell \in B \qquad (\mathcal{T}^{ws}_7)$$
$$\forall A, B : C, \forall \ell : L, \quad \ell \in A \supset B \Leftrightarrow \ell \in A \Rightarrow \ell \in B \qquad (\mathcal{T}^{ws}_8)$$
$$\forall \ell : L, \quad \ell \in \varnothing \Leftrightarrow \bot \qquad (\mathcal{T}^{ws}_9)$$
$$\forall A : C, \forall \ell : L, \quad \ell \in \mathcal{P}(A) \Leftrightarrow \exists x, x::\ell \in A \qquad (\mathcal{T}^{ws}_{10})$$
$$\forall A : C, \forall \ell : L, \quad \ell \in \mathcal{C}(A) \Leftrightarrow \forall x, x::\ell \in A \qquad (\mathcal{T}^{ws}_{11})$$

A couple of remarks on this formalization:

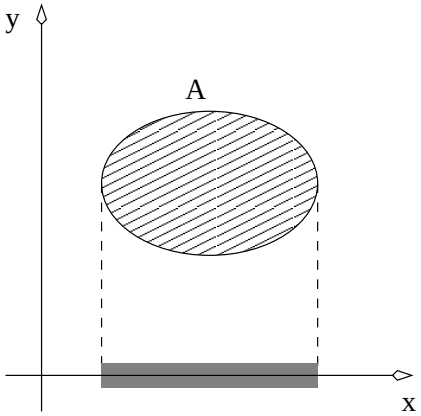
- The symbols 1 and S in $\mathcal{L}^{ws}$ are constructors of de Bruijn indices. It is important for them to be given the sort X, because the symbol S will need to be applied to non-substitutable variables of X to lift them out of the substitution's reach[1].
- This axiom system features a weak calculus of explicit substitutions (Hardin et al., 1996): the substitutions are propagated over the elements of the language via the symbols $\in$ and $\cdot[\cdot]$, and no lift is introduced by the $\mathcal{P}$ or $\mathcal{C}$ binders (axioms ($\mathcal{T}^{ws}_{10}$) and ($\mathcal{T}^{ws}_{11}$)).
- Axiom scheme ($\mathcal{T}^{ws}_4$) (resp. ($\mathcal{T}^{ws}_5$)) represents a finite number of first-order axioms, as there is one such axiom for each function (resp. predicate) symbol of arity n (resp. m) in the language $\mathcal{L}^{ws}$ (which as a finite extension of $\mathcal{L}^{cs}$ is finite).
- The $\mathcal{P}$ and $\mathcal{C}$ operators are respectively called *projection* and *cylindrification* in Algebra. Figure 8.1 illustrates the semantics of these combinators in a two-dimensional space.

From now on in this paper, we will spare the type of variables in quantifiers when no ambiguities hold.

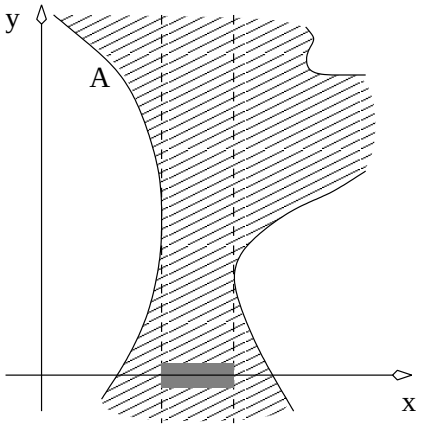
Expressiveness

We want to associate to each proposition a characteristic class, constructed with the symbols exposed in definition 8.2.1.

[1] This operation is called *pre-cooking* in (Dowek et al., 2001).



(a) Projection



(b) Cylindrification

Figure 8.1: Two-dimensional operators

Example 8.2.3. Assume given a predicate symbol $<$ and the associate characteristic set constructor $\dot{<}$. Using infix notations and the decimal representations 2 and 3 for the Peano numbers $S(1)$ and $S(S(1))$, the class of objects x such that $\exists y, \exists z, (x < z \wedge z < y)$ is written $\mathcal{PP}(3\dot{<}1 \cap 1\dot{<}2)$. Indeed,

$$\begin{aligned}
&\langle x\rangle \in \mathcal{PP}(3\dot{<}1 \cap 1\dot{<}2)\\
&\quad \Leftrightarrow \exists y, \exists z, (\langle z, y, x\rangle \in 3\dot{<}1 \cap 1\dot{<}2)\\
&\quad \Leftrightarrow \exists y, \exists z, (\langle z, y, x\rangle \in 3\dot{<}1 \wedge \langle z, y, x\rangle \in 1\dot{<}2)\\
&\quad \Leftrightarrow \exists y, \exists z, (x < z \wedge z < y)
\end{aligned}$$

Example 8.2.4. Assume the function symbols f, 0, $-$ and $|.|$ and the predicate symbols $<$ and $>$ given. For readability's sake, we will use infix notations. Let us try to define the class of real points r around which f is continuous, i.e. the class E such that:

$$\langle r\rangle \in E \Leftrightarrow \forall \varepsilon, \exists \eta, \forall x, \varepsilon > 0 \Rightarrow \eta > 0 \Rightarrow |x - r| < \eta \Rightarrow |f(x) - f(r)| < \varepsilon.$$

Given the sets $\dot{<}$, and $\dot{>}$ associated to the predicates $<$ and $>$, and using the axiom $(\mathcal{T}^{ws}_5)$ of definition 8.2.2, the right hand side of this formula is equivalent to:

$$\begin{aligned}
\forall \varepsilon, \exists \eta, \forall x, \langle \varepsilon, \eta, x, r\rangle \in 1 \dot{>} 0 \Rightarrow \langle \varepsilon, \eta, x, r\rangle \in 2 \dot{>} 0 \Rightarrow\\
\langle \varepsilon, \eta, x, r\rangle \in |3 - 4| \dot{<} 2 \Rightarrow \langle \varepsilon, \eta, x, r\rangle \in |f(3) - f(4)| \dot{<} 1
\end{aligned}$$

Now the axiom $(\mathcal{T}^{ws}_8)$ for implication allows for the following factorization:

$$\forall \varepsilon, \exists \eta, \forall x, \langle \varepsilon, \eta, x, r\rangle \in 1 \dot{>} 0 \supset 2 \dot{>} 0 \supset |3 - 4| \dot{<} 2 \supset |f(3) - f(4)| \dot{<} 1$$

Axioms $(\mathcal{T}^{ws}_{10})$ and $(\mathcal{T}^{ws}_{11})$ allow us to land the final equivalent formula:

$$\langle r\rangle \in \mathcal{CPC}(1 \dot{>} 0 \supset 2 \dot{>} 0 \supset |3 - 4| \dot{<} 2 \supset |f(3) - f(4)| \dot{<} 1)$$

Remark that we only used the axioms of definition 8.2.2 to prove this instance of the axiom scheme of comprehension.

We first need a little lemma to prove that the term substitution axioms are complete:

Lemma 8.2.5. *For all term u and variables $x_1, \ldots, x_n$ of $\mathcal{L}^{1=}$, there exists a term t of $\mathcal{L}^{ws}$ in which none of the x_i appear, such that $u = t[x_1 :: \ldots :: x_n :: nil]$ is provable in $\mathcal{T}^{ws}$.*

Proof. We proceed inductively on the structure of u.

- If u is one of the x_i, then we take $S^{i-1}(1)$ for t and by using axioms $(\mathcal{T}^{ws}_2)$ and $(\mathcal{T}^{ws}_3)$, $x_i = S^{i-1}(1)[x_1 :: \ldots :: x_n :: nil]$ is provable.
- If u is a variable y different from the x_i, we take $t = S^n(y)$, and by axioms $(\mathcal{T}^{ws}_1)$, $(\mathcal{T}^{ws}_2)$ and $(\mathcal{T}^{ws}_3)$ we can prove $y = S^n(y)[x_1 :: \ldots :: x_n :: nil]$.

· Finally, if u is a term $f(u_1, \ldots, u_m)$, then by induction hypothesis there exist $t_1, \ldots, t_m$ such that for any $0 < i \leqslant m$, $u_i = t_i[x_1 :: \ldots :: x_n :: nil]$. Axiom $(\mathcal{T}^{ws}_4)$ allows to conclude:

$$f(u_1, \ldots, u_m) = f(t_1, \ldots, t_m)[x_1 :: \ldots :: x_n :: nil] \qquad \square$$

We now show that the comprehension axiom scheme holds in our formalism. Translated in our framework, the scheme states:

Proposition 8.2.6. *For any formula P built with the symbols of $\mathcal{L}^{1=}$, the formula:*

$$\exists E, \forall x_1, \ldots, \forall x_n, (\langle x_1, \ldots, x_n \rangle \in E \Leftrightarrow P)$$

is provable in $\mathcal{T}^{ws}$.

Proof. We show by induction on the structure of P that for any formula P there exists a term E of sort C such that, for all $x_1, \ldots, x_n$ of sort X, $(\langle x_1, \ldots, x_n \rangle \in E) \Leftrightarrow P$.

If P is an atomic proposition $p(u_1, \ldots, u_m)$, where p is a predicate symbol and $u_1, \ldots, u_n$ are terms, lemma 8.2.5 allows us to equate this proposition to $p(t_1[l], \ldots, t_m[l])$ where $l = \langle x_1, \ldots, x_n \rangle$. Then by Axiom $(\mathcal{T}^{ws}_5)$, E is $\dot{p}(t_1, \ldots, t_m)$.

For the propositional connectors, using Axioms $(\mathcal{T}^{ws}_6)$ to $(\mathcal{T}^{ws}_9)$, we prove that:

- if $P = P' \wedge P''$ then we can take $E = E_{P'} \cap E_{P''}$,
- if $P = P' \vee P''$ then we can take $E = E_{P'} \cup E_{P''}$,
- if $P = \bot$ then we can take $E = \varnothing$,
- if $P = P' \Rightarrow P''$ then we can take $E = E_{P'} \supset E_{P''}$,

where, by induction hypothesis, $E_{P'}$ and $E_{P''}$ are the classes characterized by, respectively, the formulas P' and P''.

For the case $P = \exists y, A$, the bi-dimensional illustration of figure 8.1a provides the intuition: the class such that there exists a value y for which A holds — and E_A is characterized — is its projection $\mathcal{P}(E_A)$. Indeed, we can derive:

$$\begin{aligned} \langle x_1, \ldots, x_n \rangle \in E &\Leftrightarrow \langle x_1, \ldots, x_n \rangle \in \mathcal{P}(E_A) \\ &\Leftrightarrow \exists y, \langle y, x_1, \ldots, x_n \rangle \in E_A \qquad \text{by axiom } (\mathcal{T}^{ws}_{10}) \\ &\Leftrightarrow \exists y, A \end{aligned}$$

Similarly for the universal quantifier: if $P = \forall y, A$ then $E = \mathcal{C}(E_A)$. $\square$

Remark that while our theory $\mathcal{T}^{ws}$ is intuitionistic, this proof unfolds just as well in a classical setting. In this case, we can additionally discard the $\cup$, $\cap$ and $\mathcal{C}$ symbols and related axioms, and to use the De Morgan equivalences to carry out the proof.

Because this entails that $\mathcal{T}^{\mathsf{ws}}$ is an extension of $\mathcal{T}^{\mathsf{cs}}$, and that $\mathcal{T}^{\mathsf{cs}}$ is an extension of $\mathcal{T}$, we have:

Proposition 8.2.7 (Extension)**.** *$\mathcal{T}^{\mathsf{ws}}$ is an extension of $\mathcal{T}$.*

This property ensures that the formalism we proposed is sufficiently expressive; however it should not be too strong: it should not be able to prove propositions that were not provable by using axiom schemes.

Proposition 8.2.8 (Conservativity)**.** *$\mathcal{T}^{\mathsf{ws}}$ is a conservative extension of $\mathcal{T}$.*

Proof. We prove this proposition by showing that for each model $\mathcal{M}$ of $\mathcal{T}$ there is a model of $\mathcal{T}^{\mathsf{ws}}$ validating the same $\mathcal{T}$-built formulas.

Let $\mathcal{M}$ be a model of $\mathcal{T}$, and consider the structure $\mathcal{M}'$ defined as follows:

- $[\![X]\!]^{\mathcal{M}'}$ is the set of functions from $\mathcal{M}^{\omega}$ to $\mathcal{M}$, where $\mathcal{M}^{\omega}$ is the set of infinite lists of elements of $\mathcal{M}$.
- For any n-ary function symbol f of $\mathcal{L}^{1=}$, $[\![f]\!]^{\mathcal{M}'}$, is the function:

$$a_1 \mapsto \ldots \mapsto a_n \mapsto u \mapsto [\![f]\!]^{\mathcal{M}}(a_1(u), \ldots, a_n(u))$$

  where $\forall i, a_i \in [\![X]\!]^{\mathcal{M}'}$ and $u \in \mathcal{M}^{\omega}$.
- The denotation of a predicate symbol p of $\mathcal{L}^{1=}$ in $\mathcal{M}'$ is the function mapping elements $a_1, \ldots, a_n$ of $[\![X]\!]^{\mathcal{M}'}$ to 1 if and only if for all infinite lists u, $[\![p]\!]^{\mathcal{M}}(a_1(u), \ldots, a_n(u)) = 1$.

Note that the denotation of a formula P of $\mathcal{L}^{1=}$ in $\mathcal{M}'$ is entirely known when given $[\![X]\!]^{\mathcal{M}'}$ and the denotation of the symbols of $\mathcal{L}^{1=}$.

Upon such a basis, we can define denotations for the symbols of $\mathcal{L}^{\mathsf{ws}}$:

- We say that a set e of sequences of elements of $[\![X]\!]^{\mathcal{M}'}$ is definable if and only if there exists a formula P in $\mathcal{L}^{1=}$ such that the sequence $a_1, \ldots, a_n$ is a member of e if and only if $[\![P]\!]^{\mathcal{M}'}_{a_1/x_1,\ldots,a_n/x_n} = 1$.
- $[\![L]\!]^{\mathcal{M}'}$ is the set of finite sequences of elements of $[\![X]\!]^{\mathcal{M}'}$, and $[\![C]\!]^{\mathcal{M}'}$ is the set of definable subsets of $[\![L]\!]^{\mathcal{M}'}$.
- $[\![1]\!]^{\mathcal{M}'}$ is the function that, to each infinite list, associates its first element. And $[\![S]\!]^{\mathcal{M}'}$ maps a function f into a function g, such that $g(u_1, u_2, \ldots) = f(u_2, u_3, \ldots)$.
- The denotation of $t[\ell]$ is the element of $[\![X]\!]^{\mathcal{M}'}$ that associates to any list $u = u_1, u_2, \ldots$ the term $[\![t]\!]^{\mathcal{M}'}(b_1(u), \ldots, b_n(u), u_1, u_2, \ldots)$ where the finite sequence $b_1, \ldots, b_n$ is the denotation of ℓ.
- $[\![\dot{p}]\!]^{\mathcal{M}'}$ is the set of sequences defined by the corresponding predicate symbol p.
- The denotation of the rest of the symbols of $\mathcal{L}^{\mathsf{ws}}$ is self-evident: $\cap$ is set intersection, $\in$ is set membership, etc.

Remark that this definition of definability and the denotation of the symbols of $\mathcal{L}^{ws}$ make the following sentence a tautology: if e is the set of sequences defined by a proposition P then for all $a_1, \ldots, a_n$,

$$[\![\langle x_1, \ldots, x_n\rangle \in E]\!]^{\mathcal{M}'}_{a_1/x_1,\ldots,a_n/x_n,e/E} = [\![P]\!]^{\mathcal{M}'}_{a_1/x_1,\ldots,a_n/x_n} \qquad (*)$$

Indeed, both are interpreted as 1 when $a_1, \ldots, a_n$ is a member of e.

To prove that $\mathcal{M}'$ is a model of $\mathcal{T}^{ws}$, we check that it validates axioms $(\mathcal{T}^{ws}_1)$ to $(\mathcal{T}^{ws}_{11})$. We also check that the translation we did of the original axiom schemes (e.g. the induction scheme of example 8.1.3) is also valid in $\mathcal{M}'$. Let

$$s(P(t^1_1, \ldots, t^1_{n_1}), \ldots, P(t^p_1, \ldots, t^p_{n_p}))$$

be an axiom scheme of $\mathcal{T}$, and

$$\forall E, s(\langle t^1_1, \ldots, t^1_{n_1}\rangle \in E, \ldots, \langle t^p_1, \ldots, t^p_{n_p}\rangle \in E)$$

its translation in $\mathcal{T}^{ws}$. We prove that for any definable set e and sequence $a_1, \ldots, a_n$,

$$[\![s(\langle t^1_1, \ldots, t^1_{n_1}\rangle \in E, \ldots, \langle t^p_1, \ldots, t^p_{n_p}\rangle \in E)]\!]^{\mathcal{M}'}_{a_1/x_1,\ldots,a_n/x_n,e/E} = 1$$

Assume e is defined by the proposition Q of $\mathcal{L}^{1=}$. Since the particular scheme instance $s(Q(t^1_1, \ldots, t^1_{n_1}), \ldots, Q(t^p_1, \ldots, t^p_{n_p}))$ is valid in $\mathcal{M}$,

$$\begin{aligned}
&[\![s(Q(t^1_1, \ldots, t^1_{n_1}), \ldots, Q(t^p_1, \ldots, t^p_{n_p}))]\!]^{\mathcal{M}'}_{a_1/x_1,\ldots,a_n/x_n} \\
&= [\![s(Q(t^1_1, \ldots, t^1_{n_1}), \ldots, Q(t^p_1, \ldots, t^p_{n_p}))]\!]^{\mathcal{M}}_{a_1/x_1,\ldots,a_n/x_n} \\
&= 1 \ ,
\end{aligned}$$

and all we need to do is to prove:

$$\begin{aligned}
&[\![s(\langle t^1_1, \ldots, t^1_{n_1}\rangle \in E, \ldots, \langle t^p_1, \ldots, t^p_{n_p}\rangle \in E)]\!]^{\mathcal{M}'}_{a_1/x_1,\ldots,a_n/x_n,e/E} \\
&= [\![s(Q(t^1_1, \ldots, t^1_{n_1}), \ldots, Q(t^p_1, \ldots, t^p_{n_p}))]\!]^{\mathcal{M}'}_{a_1/x_1,\ldots,a_n/x_n}
\end{aligned}$$

But this is simply a consequence of $(*)$. Hence $\mathcal{M}'$ is a model of $\mathcal{T}^{ws}$.

Finally, a formula of the language of $\mathcal{L}^{1=}$ has, obviously, the same denotation in $\mathcal{M}$ and in $\mathcal{M}'$. Thus we can conclude the conservativity of $\mathcal{T}^{ws}$ over $\mathcal{T}$. □

This last proof holds in classical logic. However it can be extended to an intuitionistic proof by using Heyting algebra based models instead of classical models.

8.3 Applications

The result of propositions 8.2.7 and 8.2.8 can be applied to any theory that uses axiom schemes. For instance, Zermelo's set theory accepts a conservative extension, built by applying these propositions to the traditional formulation of the theory. The same holds for the binary replacement axiom scheme of Zermelo-Fraenkel's theory, or the three schemes that result from Dowek and Miquel's encoding of set theory in a theory of pointed graphs (Dowek and Miquel, 2006). We detail two examples: arithmetic and real analysis.

A finite theory of arithmetic

In the following, we will explore Heyting's arithmetic. While our formalism applies to the original formulation of the theory, HA, we consider here a slightly more elaborate presentation of the theory where the universe of discourse is not restricted to natural numbers. This theory, called HA_N, was presented in (Dowek and Werner, 2005) by Dowek and Werner.

Definition 8.3.1 (HA_N)**.** The theory HA_N of arithmetic is defined in first-order logic using the symbols 0, *Succ*, $+$, $\times$, *Pred*, $=$, *Null* and N. It consists of the axioms:

$$\begin{array}{rr}
\mathsf{N}(0) & \forall x,(\mathsf{N}(x) \Rightarrow \mathsf{N}(\mathit{Succ}(x))) \\
\mathit{Pred}(0) = 0 & \forall x,(\mathit{Pred}(\mathit{Succ}(x)) = x) \\
\mathit{Null}(0) & \forall x,(\neg \mathit{Null}(\mathsf{S}(x))) \\
\forall y,(0 + y = y) & \forall x, \forall y,(\mathit{Succ}(x) + y = \mathit{Succ}(x + y)) \\
\forall y,(0 \times y = 0) & \forall y,(\mathit{Succ}(x) \times y = x \times y + y)
\end{array}$$

the axiom and axiom scheme for equality:

$$\forall x, (x = x)$$
$$\forall x, \forall y, (x = y \Rightarrow \mathsf{P}(x) \Rightarrow \mathsf{P}(y))$$

and the induction scheme:

$$\mathsf{P}(0/x) \Rightarrow \forall y, (\mathsf{P}(y/x) \Rightarrow \mathsf{P}(\mathit{Succ}(y)/x)) \Rightarrow \forall n, (\mathsf{N}(n) \Rightarrow \mathsf{P}(n/x))$$

In (Dowek and Werner, 2005), the authors define a translation $|\cdot|$ between the languages of HA and HA_N, and prove:

Proposition 8.3.2. *HA_N is a conservative extension of HA in the sense that if A is a closed proposition formed in the language of HA then A is provable in HA if and only if $|A|$ is provable in HA_N.*

Finitizing the presentation of these axiom schemes is achieved by introducing lists and classes and a set of axioms that allows one to express comprehension, as per Sect. 8.2.

Definition 8.3.3 ($\mathsf{HA}^{\mathsf{ws}}_{\mathsf{N}}$)**.** Define $\mathsf{HA}^{\mathsf{ws}}_{\mathsf{N}}$ as an extension of HA_{N}, composed of the ranked signature:

$$\begin{array}{rr} 0, 1 : \mathsf{X} & \mathsf{S}, \mathit{Succ}, \mathit{Pred} : (\mathsf{X})\mathsf{X} \\ +, \times : (\mathsf{X}, \mathsf{X})\mathsf{X} & = : (\mathsf{X}, \mathsf{X}) \\ \mathsf{N}, \mathit{Null} : (\mathsf{X}) & \cdot[\cdot] : (\mathsf{X}, \mathsf{L})\mathsf{X} \\ \varnothing : \mathsf{C} & \mathcal{P}, \mathcal{C} : (\mathsf{C})\mathsf{C} \\ \cap, \cup, \supset : (\mathsf{C}, \mathsf{C})\mathsf{C} & \in : (\mathsf{L}, \mathsf{C}) \end{array}$$

axioms ($\mathcal{T}^{\mathsf{ws}}_{1}$) to ($\mathcal{T}^{\mathsf{ws}}_{11}$), the axioms of arithmetic:

$$\begin{array}{rr} \mathsf{N}(0) & \forall x, (\mathsf{N}(x) \Rightarrow \mathsf{N}(\mathit{Succ}(x))) \\ \mathit{Pred}(0) = 0 & \forall x, (\mathit{Pred}(\mathit{Succ}(x)) = x) \\ \mathit{Null}(0) & \forall x, (\neg \mathit{Null}(\mathit{Succ}(x))) \\ \forall y, (0 + y = y) & \forall x, \forall y, (\mathit{Succ}(x) + y = \mathit{Succ}(x + y)) \\ \forall y, (0 \times y = 0) & \forall y, (\mathit{Succ}(x) \times y = x \times y + y) \end{array}$$

the equality axioms:

$$\forall x, (x = x)$$
$$\forall x, \forall y, (x = y \Rightarrow \forall A, (\langle x \rangle \in A \Rightarrow \langle y \rangle \in A))$$

and the induction axiom:

$$\forall n, \forall A, (\langle 0 \rangle \in A \Rightarrow \forall y, (\langle y \rangle \in A \Rightarrow \langle \mathit{Succ}(y) \rangle \in A) \Rightarrow \forall n, (\mathsf{N}(n) \Rightarrow \langle n \rangle \in A))$$

In particular, axiom ($\mathcal{T}^{\mathsf{ws}}_{4}$) has four instances (*Succ*, *Pred*, + and ×) and axiom ($\mathcal{T}^{\mathsf{ws}}_{5}$) three (=, N and *Null*), for a total of 29 axioms.

Remark that there are two sets of integer constructors in $\mathsf{HA}^{\mathsf{ws}}_{\mathsf{N}}$: the native arithmetic integers, build with 0 and *Succ*; and the de Bruijn indices formed by the symbols 1 and S.

Propositions 8.2.7 and 8.2.8 applied to HA_{N}, composed with proposition 8.3.2, allow us to state:

Proposition 8.3.4. $\mathsf{HA}^{\mathsf{ws}}_{\mathsf{N}}$ *is as a conservative extension of* HA.

We can define a slight variant of $\mathsf{HA}^{\mathsf{ws}}_{\mathsf{N}}$ by replacing the class induction axiom and Leibniz's equality axiom by the equivalences:

$$\forall x, \forall y, (x = y \Leftrightarrow \forall A, (\langle x \rangle \in A \Rightarrow \langle y \rangle \in A))$$
$$\forall n, (\mathsf{N}(n) \Leftrightarrow \forall A, (\langle 0 \rangle \in A \Rightarrow \forall y, (\langle y \rangle \in A \Rightarrow \langle \mathit{Succ}(y) \rangle \in A) \Rightarrow \forall n, \langle n \rangle \in A))$$

And we can drop the three axioms $\forall x, (x = x)$, $\mathsf{N}(0)$ and $\forall x, (\mathsf{N}(x) \Rightarrow \mathsf{N}(\mathit{Succ}(x)))$ that have become superfluous.

Definition 8.3.5 (HA^+)**.** Let HA^+ be this shortened theory.

Lemma 8.3.6. *HA^+ is equivalent to $\mathsf{HA}^{\mathsf{ws}}_{\mathsf{N}}$, and counts 26 axioms instead of 29.*

Proof. Trivial. □

Arithmetic as a theory modulo

A theory modulo (Dowek et al., 2003) is a theory in which formulas are identified modulo a congruence, defined as a rewriting system. In particular, the theory of arithmetic has been expressed in such a framework (Dowek and Werner, 2005), but this formalization had an infinite number of rewrite rules. The goal of this section is to show how the result of section 8.2 allows a finite formulation of arithmetic modulo.

Definition 8.3.7 ($\mathsf{HA}^{\mathsf{mod}}$)**.** The language of the theory $\mathsf{HA}^{\mathsf{mod}}$ is the same as the theory HA^+. The congruence $\equiv_{\mathcal{R}}$ associated with this theory is given by the rewrite system $\mathcal{R}$ of figure 8.3.

The system is split between rules dealing with substitutions, rules for arithmetic operations and rules defining relations (equality, etc.). This formalism counts a total of 26 rules, which is reduced to 22 or 23 in classical logic using the fact that all the connectors and quantifiers can be defined from 2 or 3 primitive ones.

Application to real closed fields

Real numbers or their approximation are used in exact arithmetic, programming languages, computer algebra and formal systems. The following formalization is quite common (Lelong-Ferrand and Arnaudies, 1972), and is used *e.g.* in the proof assistant Coq to implement the theory of real numbers.

Definition 8.3.8 ($\mathbb{R}^{\mathsf{cs}}$)**.** The language of the theory of real numbers $\mathbb{R}^{\mathsf{cs}}$ is formed by the symbols 0, 1, $+$, $\times$, the opposite $-$, inverse $1/\cdot$, the symbol $\lceil\cdot\rceil$ that maps real numbers to natural numbers, and the predicates $<$ and $=$. We note $\leqslant$ the disjunction of the two aforementioned predicates. The

Substitutions rules

$$t[nil] \to t \qquad (HA_1^{mod})$$
$$1[t::\ell] \to t \qquad (HA_2^{mod})$$
$$S(n)[t::\ell] \to n[\ell] \qquad (HA_3^{mod})$$
$$Succ(t)[\ell] \to Succ(t[\ell]) \qquad (HA_4^{mod})$$
$$Pred(t)[\ell] \to Pred(t[\ell]) \qquad (HA_5^{mod})$$
$$(t_1 + t_2)[\ell] \to t_1[\ell] + t_2[\ell] \qquad (HA_6^{mod})$$
$$(t_1 \times t_2)[\ell] \to t_1[\ell] \times t_2[\ell] \qquad (HA_7^{mod})$$

Arithmetic rules

$$Pred(0) \to 0 \qquad (HA_8^{mod})$$
$$Pred(Succ(x)) \to x \qquad (HA_9^{mod})$$
$$0 + y \to y \qquad (HA_{10}^{mod})$$
$$0 \times y \to 0 \qquad (HA_{11}^{mod})$$
$$Succ(x) + y \to Succ(x + y) \qquad (HA_{12}^{mod})$$
$$Succ(x) \times y \to x \times y + y \qquad (HA_{13}^{mod})$$

Proposition rules

$$\ell \in \dot{Null}(t) \to Null(t[\ell]) \qquad (HA_{14}^{mod})$$
$$\ell \in \doteq(t_1, t_2) \to t_1[\ell] = t_2[\ell] \qquad (HA_{15}^{mod})$$
$$\ell \in \dot{N}(t) \to N(t[\ell]) \qquad (HA_{16}^{mod})$$
$$x = y \to \forall A, (\langle x \rangle \in A \Rightarrow \langle y \rangle \in A)) \qquad (HA_{17}^{mod})$$
$$N(n) \to \forall A, (\langle 0 \rangle \in A \Rightarrow \forall y, (\langle y \rangle \in A \Rightarrow \langle Succ(y) \rangle \in A) \Rightarrow \langle n \rangle \in A) \qquad (HA_{18}^{mod})$$
$$Null(0) \to \top \qquad (HA_{19}^{mod})$$
$$Null(Succ(x)) \to \bot \qquad (HA_{20}^{mod})$$
$$\ell \in A \cap B \to \ell \in A \wedge \ell \in B \qquad (HA_{21}^{mod})$$
$$\ell \in A \cup B \to \ell \in A \vee \ell \in B \qquad (HA_{22}^{mod})$$
$$\ell \in A \supset B \to \ell \in A \Rightarrow \ell \in B \qquad (HA_{23}^{mod})$$
$$\ell \in \varnothing \to \bot \qquad (HA_{24}^{mod})$$
$$\ell \in \mathcal{P}(A) \to \exists n, n::\ell \in A \qquad (HA_{25}^{mod})$$
$$\ell \in \mathcal{C}(A) \to \forall n, n::\ell \in A \qquad (HA_{26}^{mod})$$

Figure 8.2: Rewrite system $\mathcal{R}$ for arithmetic

axioms follow:

$$(1 = 0) \Rightarrow \bot \tag{$\mathbb{R}_1$}$$
$$\forall x, \forall y, \quad x + y = y + x \tag{$\mathbb{R}_2$}$$
$$\forall x, \forall y, \forall z, \quad (x + y) + z = x + (y + z) \tag{$\mathbb{R}_3$}$$
$$\forall x, \quad x + (-x) = 0 \tag{$\mathbb{R}_4$}$$
$$\forall x, \quad x + 0 = x \tag{$\mathbb{R}_5$}$$
$$\forall x, \forall y, \quad x \times y = y \times x \tag{$\mathbb{R}_6$}$$
$$\forall x, \forall y, \forall z, \quad (x \times y) \times z = x \times (y \times z) \tag{$\mathbb{R}_7$}$$
$$\forall x, \quad ((x = 0) \Rightarrow \bot) \Rightarrow (1/x) \times x = 1 \tag{$\mathbb{R}_8$}$$
$$\forall x, \quad 1 \times x = x \tag{$\mathbb{R}_9$}$$
$$\forall x, \forall y, \forall z, \quad x \times (y + z) = x \times y + x \times z \tag{$\mathbb{R}_{10}$}$$
$$\forall x, \forall y, \quad (x < y) \vee (x = y) \vee (y < x) \tag{$\mathbb{R}_{11}$}$$
$$\forall x, \forall y, \quad x < y \Rightarrow y < x \Rightarrow \bot \tag{$\mathbb{R}_{12}$}$$
$$\forall x, \forall y, \forall z, \quad x < y \Rightarrow y < z \Rightarrow x < z \tag{$\mathbb{R}_{13}$}$$
$$\forall x, \forall y, \forall z, \quad y < z \Rightarrow x + y < x + z \tag{$\mathbb{R}_{14}$}$$
$$\forall x, \forall y, \forall z, \quad 0 < x \Rightarrow y < z \Rightarrow x \times y < x \times z \tag{$\mathbb{R}_{15}$}$$
$$\forall x, \quad x < \lceil x \rceil \wedge (\lceil x \rceil + (-x) \leqslant 1) \tag{$\mathbb{R}_{16}$}$$

One way of formulating the completeness theorem of real closed fields is to use classes and bounds. Thus we consider classes of reals, manipulated using the *nil*, :: and $\in$ symbols, and the comprehension axiom scheme:

$$\forall E, \forall x_1, \ldots, x_n, (\langle x_1, \ldots, x_n \rangle \in E \Leftrightarrow P) \tag{$\mathbb{R}_{17}$}$$

The last four axioms of this theory follow: the first three define the semantics of the predicate symbols $isUB(\cdot, \cdot)$, $bounded(\cdot)$ and $isLUB(\cdot, \cdot)$; the fourth is the completeness axiom.

$$\forall E, \forall m, (\forall x, \langle x \rangle \in E \Rightarrow x \leqslant m) \Leftrightarrow isUB(E, m) \tag{$\mathbb{R}_{18}$}$$
$$\forall E, (\exists m, isUB(E, m)) \Leftrightarrow bounded(E)) \tag{$\mathbb{R}_{19}$}$$
$$\forall E, \forall m, (isUB(E, m) \wedge (\forall b, isUB(E, b) \Rightarrow m \leqslant b)) \Leftrightarrow isLUB(E, m) \tag{$\mathbb{R}_{20}$}$$
$$\forall E, (bounded(E) \Rightarrow (\exists x, \langle x \rangle \in E) \Leftrightarrow (\exists m, isLUB(E, m))). \tag{$\mathbb{R}_{21}$}$$

Following the result of Sect. 8.2, we give a conservative finite first-order presentation of this theory.

Definition 8.3.9 ($\mathbb{R}^{ws}$)**.** Define $\mathbb{R}^{ws}$ as an extension of $\mathbb{R}^{cs}$, formed with the

ranked signature:

$$\begin{array}{rr} 0, 1 : \mathsf{X} & \mathsf{S}, -, 1/\cdot : (\mathsf{X})\mathsf{X} \\ +, \times : (\mathsf{X}, \mathsf{X})\mathsf{X} & <, = : (\mathsf{X}, \mathsf{X}) \\ \lceil\cdot\rceil : (\mathsf{X})\mathsf{X} & \cdot[\cdot] : (\mathsf{X}, \mathsf{L})\mathsf{X} \\ \varnothing : \mathsf{C} & \mathcal{P}, \mathcal{C} : (\mathsf{C})\mathsf{C} \\ \cap, \cup, \supset : (\mathsf{C}, \mathsf{C})\mathsf{C} & \in : (\mathsf{L}, \mathsf{C}) \end{array}$$

axioms ($\mathcal{T}^{\mathsf{ws}}_1$) to ($\mathcal{T}^{\mathsf{ws}}_{11}$), the axioms ($\mathbb{R}_1$) to ($\mathbb{R}_{16}$) and ($\mathbb{R}_{18}$) to ($\mathbb{R}_{21}$).

In particular, axiom ($\mathcal{T}^{\mathsf{ws}}_4$) has six instances ($\mathsf{S}$, $-$, $1/\cdot$, $+$, $\times$ and $\lceil\cdot\rceil$) and axiom ($\mathcal{T}^{\mathsf{ws}}_5$) two ($=$ and $<$), for a total of 37 axioms. Propositions 8.2.7 and 8.2.8 applied to $\mathbb{R}$ allow us to state:

Proposition 8.3.10. $\mathbb{R}^{\mathsf{ws}}$ *is as a conservative extension of* $\mathbb{R}^{\mathsf{cs}}$.

Remark that in the ranked signature of $\mathbb{R}^{\mathsf{ws}}$, $\lceil\cdot\rceil$ has the rank $(\mathsf{X})\mathsf{X}$, which is too general. This is because there is no notion of natural numbers in the formalism of definition 8.3.9. This can be rectified by introducing the appropriate sort X', and the language and theory of natural arithmetic as in the previous section; then writing $\lceil\cdot\rceil : (\mathsf{X})\mathsf{X}'$. However a more lightweight way of solving this issue is to emulate natural numbers within the sort of real numbers X. Indeed, we can define N as the smallest class of real numbers that satisfy the conjunction of the formulas:

$$0 \in \mathsf{N} \qquad\qquad \forall x : \mathbb{R}, x \in \mathsf{N} \Rightarrow x + 1 \in \mathsf{N}$$

Now the signature of $\lceil\cdot\rceil$ would still read $(\mathsf{X})\mathsf{X}$, however the semantics of the operator would restrict its values to elements of the class N.

Implementation: a theory of reals in Fellowship

An implementation of the theory of real closed fields as a library for Fellowship, using the theory of classes, is in progress. This theory is a good example of second-order development whose implementation is being largely reproduced between theorem provers. For instance, a library of reals has been developed in Coq by Mayero during her Ph.D (Mayero, 2001), who then ported it to PVS during her post-doctoral studies (Muñoz and Mayero, 2001). By using the theory of classes and the interoperability features of Fellowship, we make it possible to avoid this duplication of efforts.

To this day, the library of real closed fields constitutes the biggest example carried out within Fellowship: the current development totals more than 1500 lines of code and over 150 proofs. The table 8.1 contains a few examples of the included theorems. Although automation is not much developed in Fellowship, this development is roughly only 1.5 times larger than in similar implementations in other theorem provers. This library is available online

Table 8.1: Theorems from the library of reals in Fellowship

$$\forall r_1, r_2, r_1 < r_2 \vee r_1 > r_2 \Leftrightarrow r_1 \neq r_2$$
$$\forall r_1, r_2, r_3, r_4, r_1 \leqslant r_2 \wedge r_2 < r_3 \Rightarrow (r_1 \leqslant r_2 \wedge r_2 < r_4) \vee (r_4 \leqslant r_2 \wedge r_2 < r_3))$$
$$\forall r_1, r_2, r_1 \neq 0 \wedge r_2 \neq 0 \Rightarrow r_1 \times r_2 \neq 0)$$
$$\forall r_1, r_2, r_3, r_1 \neq 0 \Rightarrow (r_1/r_2) \times (r_3/r_1) = r_3 \times (1/r_2)$$
$$\forall r_1, r_2, 1 \leqslant r_1 \Rightarrow r1 < r2 \Rightarrow 1/r_2 < 1/r_1$$
$$\forall r_1, r_2, r_1^2 + r_2^2 = 0 \Rightarrow r_1 = 0 \wedge r_2 = 0$$
$$\forall r_1, r_2, (\forall \varepsilon, 0 < \varepsilon \Rightarrow r_1 \leqslant r_2 + \varepsilon) \Rightarrow r_1 \leqslant r_2$$

from Fellowship's website. In the close future, tools such as (Ayache and Filliâtre, 2006) could be used to speed-up the development of the library by automatically importing specifications from other provers' existing libraries.

8.4 Related work

The work we present in the previous sections is related to von Neumann, Bernays and Gödel's formalism for set theory (NBG) (Mendelson, 1997) that rehabilitated the notion of class used by 19th century mathematicians (Bourbaki, 1968). However it improves on a couple of points:

- Classes and the NBG approach have largely been associated to set theory (Bourbaki, 1968). We generalize it to any theory that has axiom schemes.
- By clarifying the classes/set distinction, not only is the system simplified, but we also allow a more structured hierarchy of objects. In $\mathcal{T}^{\mathsf{ws}}$ the sorts X, L and C are clearly separate entities, while in NBG the sorts of objects and classes are indistinctly embedded into one another.
- Using lists and explicit substitutions to instantiate predicate free variables also greatly clarifies the argument-passing process, and allows us to bypass a couple of permutation axioms. Also, because we use native lists we are spared the tedious process of re-encoding them using sets, as is done in NBG.
- There is no easy way to orient NBG's permutation axioms to generate a well-behaved rewrite system. On the contrary, and as illustrated in section 8.3, the rules ($\mathcal{T}^{\mathsf{ws}}_1$) to ($\mathcal{T}^{\mathsf{ws}}_{11}$) are easily orientable.

What is more, our formalism applied to Zermelo's set theory would use 15 axioms (in the classical case) vs. 14 for NBG, thus we feel it allows for a more understandable presentation of set theory without being overly bloated.

Vaillant (Vaillant, 2002) gives a presentation of set theory using explicit substitutions (in the form of the λσ-calculus) to manipulate classes. The axiomatization we propose differs from it in the following ways:

- While Vaillant's paper, following NBG, was focused on set theory, our method applies to any theory with one or more axiom schemes.
- We have shown that a weak substitution calculus is strong enough to allow the complete manipulation of substitutions in this type of framework. This allows us to greatly reduce both the language's signature (neither lift, shift nor compose operators) and the number of axioms in our presentation.

If a comparison of the size of the two formalizations could reinforce the reader's opinion that our system is lighter, consider that Vaillant's intuitionistic theory uses a total of 42 axioms, while ours would only require 18 and still express full-blown Zermelo set theory. These axiom numbers comparisons might seem pointless without any experimental data, thus irrelevant in the scope of automated reasoning. However one should note that this work is destined to be implemented in proof assistants, where the low number of axioms, in particular for variable substitution, will allow for faster and less tedious computational steps.

Finally, while Megill's work on a finite formal predicate calculus (Megill, 1995) also uses a form reification, its approach is more invasive than the ones presented above, as the whole logical system (including inference rules) is finitized. This would make this solution hard to implement in a existing, general purpose prover.

8.5 Conclusion

We have exposed a generic formalization of theories with axiom schemes, which has the property of being finite. This was achieved through the use of classes and the recourse to weak explicit substitutions to cope straightforwardly with variable instantiation. This operating protocol was applied to give a finite axiomatization of the theories of arithmetic and of real closed fields, and a finite formalization of the former in deduction modulo.

Comparing to other methods such as von Neumann, Bernays and Gödel's or Vaillant's, it appears this way of formalizing theories using axiom schemes has links with both works. The use of a weak calculus allows us to keep a reduced number of axioms, and provides an intuitive, direct mechanism for substitutions — all of which are highly desirable properties in a proof checking environment.

Such a result easily fits into the trend set by the previous works done to formalize arithmetic and set theory into computer proof assistants (Belinfante, 1999; Boyer et al., 1986; Paulson, 1993; Quaife, 1992; Dowek and Miquel, 2006). Moreover, the fact that our axioms are easily orientable is a

major asset when dealing with theories in deduction modulo. An implementation of the theory of real numbers into Fellowship using this technique is currently underway, and demonstrates the feasibility of using weak frameworks such as first-order logic as a basis for proof-checkers.

Burel has derived from this work the following result (Burel and Kirchner, 2007): given a proof in $n+1$-th order arithmetic one can find a proof of linear size in n-th order arithmetic, within the framework of deduction modulo. In other terms, proof speed-up in deduction modulo is linear: this demonstrates that the speed-up conjectured by Gödel (Gödel, 1936) can be expressed as a computation and does not come from the deduction part of proofs.

Perspectives

Higher-order sequent calculi

The duality between $\bar{\lambda}\mu\tilde{\mu}$-calculus and its variants on the one hand, and first-order sequent calculus on the other hand, is quite a recent topic, but is now relatively well understood. The next frontier is higher-order settings, and in particular pure type systems. In this field, scientists are still chipping away.

Dyckhoff and Pinto have proposed an extension of sequent calculus with dependent types (Pinto and Dyckhoff, 1998), using several restrictions. In their setting, only proof terms in normal form are considered, and the types are only allowed to contain pure λ-terms, falling short of a pure type system.

Lately, Lengrand, Dyckhoff and McKinna (Lengrand et al., 2006) have coined the term *pure type sequent calculus* to refer to a sequent calculus for pure type systems, and have developed a theory for such systems. Unfortunately, they can only deal with intuitionistic constructs. Lengrand and Miquel (Lengrand and Miquel, 2006) have proposed a classical extension of $\bar{\lambda}\mu\tilde{\mu}$ to F_ω, *i.e.* with polymorphism and type constructors. It has the drawback of using mono-sided sequents, which does away with the μ / $\tilde{\mu}$ differences, and thus blurs the duality of the calculus (there is no way to express call-by-value / call-by-name reduction strategies, for instance).

Generalizing this approach while preserving duality is a difficult challenge, as highlighted by Herbelin in (Herbelin, 2005), because dependent types break the symmetry of sequents. Yet pushing towards an hypothetical '$\bar{\lambda}\mu\tilde{\mu}\Pi$' could well prove worthwhile: Cousineau and Dowek (Cousineau and Dowek, 2007) have shown that any PTS can be conservatively embedded in the $\lambda\Pi$-calculus modulo. Thus the question can be displaced to the field of deduction modulo: can this framework be adapted to sequent calculus? Recent work by Brauner and Houtmann suggest so, but their answer avoids $\bar{\lambda}\mu\tilde{\mu}$ in favor of Urban's calculus (Urban and Bierman, 1999).

Formal proof languages

The formal frameworks for proof languages introduced in this work are tested on relatively simple systems. A natural course of events would for this testing to continue and expand towards more complex examples — yielding, in turn, improvements and refinements in these frameworks.

The use of a monadic construction to reflect the semantics of strategies sparks an interesting question: is there a deeper categorical foundation behind these languages? The peculiar form of the proof monad seems to suggest so, and the tree-like representation and modification of indexed proofs hints at a possible direction of investigation. Also, the study of strategies from a proof search point of view, *i.e.* as commands used to shape the search

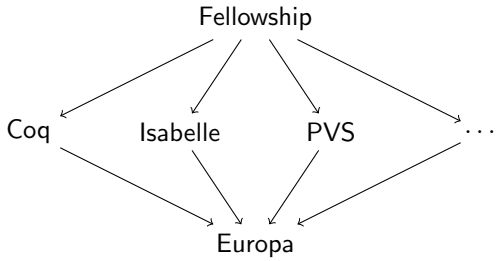
space, should yield additional insight on the definition of these concepts. Incidentally, the study of the relationship between strategies in rewriting frameworks (Clavel and Meseguer, 1996; Visser, 2001; Cirstea et al., 2003) and in proof language could provide another bridging point between proof theory and rewriting systems, which would cross-fertilize both fields.

Finally, the idea of a typing system for proof languages, exposed in chapter 5, can surely be extended with more intricate constructions. The system exposed in this work could be compared to a framework of simple types, with the arrow symbol $\rightarrow$ as a single type constructor: future work should study the feasibility and use of adding extra constructions, such as dependent types and sub-typing, to this system. It should also be interesting to examine the link between the logical framework of the proof checker, and the typing framework of the proof language.

Interoperability anywhere?

Work on the interoperability of theorem proving systems has already made strides, and mostly takes place around the HOL tool. Connections with NuPrl (Howe, 1996; Naumov et al., 2001), Isabelle (Obua and Skalberg, 2006) and Coq (Denney, 2000) have been made, yet most of them only translate specifications, and the ones that deal with proofs do so in a very specialized fashion, that can hardly be reused.

Our work presents a series of approaches to the proof translation problem, based on a simple premiss: proofs can be written in a separate framework, and then translated to any formally-defined tool. The nascent Europa system, initiated by Dowek, intends to use the $\lambda\Pi$-modulo framework to express proofs coming from PTS-based tools. Thus a lattice of importing provers could be sketched (arrows indicate the direction of proof translation functions between the systems):



Now the stake of the search for a $\bar{\lambda}\mu\tilde{\mu}\Pi$-calculus modulo becomes even higher: this structure would enable the merging of the exporting functions of Fellowship with the importing functions of Europa, thus creating truly transparent interoperability between proof systems.

Finally, although the subject of this manuscript lies in proof assistant compatibility, it should be interesting to examine the case of other tools for

formal verification, such as model-checker or SAT-solvers. The concept of a 'blackboard', as found in (Rushby, 2006), is indeed quite close to the notion of 'proof manager' in our work. Yet extending our method to different kinds of formal tools certainly requires a good deal of work, for instance to adapt the base logical formalism to enable the expression of *ad hoc* assertions required to communicate with these tools.

Bibliography

Mark Aagaard, Miriam Leeser, and Phil Windley. Toward a super duper hardware tactic. In Jeffrey Joyce and Carl-Johan Seger, editors, *Proc. 6th Int. Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 400–414. University of British Columbia, Springer-Verlag, August 1993. 1

Mads Sig Ager. From natural semantics to abstract machines. In Sandro Etalle, editor, *Proc. 3rd Int. Symp. on Logic-based Program Synthesis and Transformation*, volume 3573 of *Lecture Notes in Computer Science*, pages 245–261. Springer-Verlag, 2004. 49

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992. 24

ANSI. *Programming language, Fortran: X3.9-1966.* FIPS Publication. American National Standards Institute, 1966. 71

ANSI. *Programming language, Common LISP: X3.226-1994.* FIPS Publication. American National Standards Institute, 1996. 71

Mila Archer, Ben Di Vito, and César Muñoz. Developing user strategies in PVS: A tutorial. In *Proc. Int. Workshop on Design and Application of Strategies/Tactics in Higher Order Logics*, number CP-2003-212448 in NASA Conference Publication, pages 16–42, September 2003. 55, 68

Myla Archer, Thierry Boy de la Tour, and César Muñoz, editors. *Proc. 6th Int. Workshop on Strategies in Automated Deduction*, August 2006. 72

Zena Ariola and Amr Sabry. Correctness of monadic state: An imperative call-by-need calculus. In *Proc. 25th ACM Symp. on Principles of Programming Languages*, pages 62–74, 1998. 49

Nicolas Ayache and Jean-Christophe Filliâtre. Combining the coq proof assistant with first-order decision procedures. Draft Communication, March 2006. 138

Johan Belinfante. Computer proofs in Gödel's class theory with equational definitions for composite and cross. *Journal of Automated Reasoning*, 22 (2):311–339, 1999. 139

Paul Bernays. *Axiomatic Set Theory.* Dover Publications, 1958. 124

Sylvie Boldo and César Muñoz. Provably faithful evaluation of polynomials. In Hisham Haddad, editor, *Proc. 21st ACM Symp. on Applied Computing*, pages 1328–1332. ACM Press, 2006. 109

Richard Boulton. Boyer-moore automation for the HOL system. In Luc Claesen and Michael Gordon, editors, *Proc. 5th Int. Workshop on Higher Order Logic Theorem Proving and its Applications*, volume A-20 of *IFIP Transactions*, pages 133–142. Elsevier Science, 1992. 1

Nicolas Bourbaki. *Éléments de Mathématique: Théorie des ensembles*, volume 1 à 4. Masson, 1968. ISBN 2-225-81909-2. 138

Robert Boyer and J Moore. *A Computational Logic*. Academic Press, 1979. 2

Robert Boyer and J Moore. A computational logic handbook. *PERSPEC: Perspectives in Computing*, 23, 1988. 2

Robert Boyer, Ewing Lusk, William McCune, Ross Overbeek, Mark Stickel, and Larry Wos. Set theory in first-order logic: Clauses for Gödel's axioms. *Journal of Automated Reasoning*, 2(3):287–327, 1986. 139

Guillaume Burel and Claude Kirchner. Unbounded proof-length speed-up in deduction modulo. 2007. Draft communication. 140

Albert Buronni. Théorie des catégories, 2004. Informal lectures. 58

Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Rewrite strategies in the rewriting calculus. In Bernhard Gramlich and Salvador Lucas, editors, *Proc. 3rd Int. Workshop on Reduction Strategies in Rewriting and Programming*, volume 86 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2003. 70, 142

Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In José Meseguer, editor, *Proc. 1st Int. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 125–147. Elsevier Science, 1996. 142

Robert Constable, Stuart Allen, Mark Bromley, Rance Cleaveland, James Cremer, Robert Harper, Doug Howe, Todd Knoblock, Nax Mendler, Prakash Panangaden, James Sasaki, and Scott Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986. 2

Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In Bruno Buchberger, editor, *Proc. European Conf. on Computer Algebra*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184. Springer-Verlag, 1985. 2

Denis Cousineau and Gilles Dowek. Embedding pure type systems in lambda Pi calculus modulo. Submitted to TLCA'07, 2007. 141

Erik Crank and Matthias Felleisen. Parameter-passing and the lambda calculus. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 233–244. ACM Press, January 1991. 49

Tristan Crolard. Subtractive logic. *Theoretical Computer Science*, 254, 2001. 122

Pierre-Louis Curien and Hugo Herbelin. The duality of computation. *ACM sigplan notices*, 35(9), 2000. 18

Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theoretical Computer Science*, 325(1):69–110, 2004. 1

Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library for floating-point numbers and its application to exact computing. In Richard Boulton and Paul Jackson, editors, *Proc. 14th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 169–184. Springer-Verlag, 2001. 109

Nicolaas de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In Michel Laudet, Daniel Lacombe, Louis Nolin, and Marcel Schützenberger, editors, *Proc. of Symp. on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970. 2

Michel de Certeau. *The Practice of Everyday Life*. Univ. of California Press, 1948. translation by Steven Rendall. 3

David Delahaye. *Conception de Langages pour Décrire les Preuves et les Automatisations dans les Outils d'Aide à la Preuve*. Thèse de doctorat, Université Paris 6, 2001. 2

David Delahaye. A tactic language for the system Coq. In Michel Parigot and Andrei Voronkov, editors, *Proc. 7th Int. Conf. on Logic for Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer-Verlag, November 2000. 36, 55, 69

Ewen Denney. A prototype proof translator from HOL to coq. In Mark Aagaard and John Harrison, editors, *Proc. 13th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 108–125. Springer-Verlag, 2000. 142

Gilles Dowek and Alexandre Miquel. Cut elimination for Zermelo's set theory. Submitted to RTA 2006, 2006. 132, 139

Gilles Dowek and Benjamin Werner. Arithmetic as a theory modulo. In Jürgen Giesl, editor, *Proc. 16th Int. Conf. on Rewriting Techniques and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pages 423–437. Springer-Verlag, 2005. 132, 134

Gilles Dowek, Thérèse Hardin, and Claude Kirchner. HOL-λσ: an intentional first-order expression of higher-order logic. *Math. Structures in Computer Science*, 11(1):21–45, 2001. 126

Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, 2003. 134

Catherine Dubois. Proving ML type soundness within Coq. In Mark Aagaard and John Harrison, editors, *Proc. 13th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 126–144. Springer-Verlag, 2000. ISBN 3-540-67863-8. 35

Catherine Dubois and Olivier Boite. Proving type soundness of a simply typed ML-like language with references. In Richard Boulton and Paul Jackson, editors, *Proc. 14th Int. Conf. on Theorem Proving in Higher Order Logics*, pages 69–84, 2001. 35

Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57:795–807, 1992. 122

Steven Eker, Merrill Knapp, Keith Laderoute, Patrick Lincoln, and Carolyn Talcott. Pathway logic: Executable models of biological networks. *Electronic Notes in Theoretical Computer Science*, 71, 2002. 1

Jean-Christophe Filliâtre. Proof of imperative programs in type theory. In *Proc. 1998 Int. Workshop on Proofs and Programs*, volume 1657 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1998. 36, 39

Gerhard Gentzen. Untersuchungen über das logisches schließen. *Mathematische Zeitschrift*, 1:176–210, 1935. 9

Peter Gluck and Gerard Holzmann. Using spin model checking for flight software verification. In *Proc. of the 2002 Aerospace Conference*. IEEE Comp. Soc. Press, March 2002. 1

Kurt Gödel. Uber die lange von beweisen. *Ergebnisse Eines Math. Koll.*, 7: 23–24, 1936. 140

Kurt Gödel. *The Consistency of the Axiom of Choice and of the Generalized Continuum-Hypothesis with the Axioms of Set Theory*, volume 3 of *Annals of Mathematics Studies*. Princeton Univ. Press, 1940. 124

Georges Gonthier. A computer-checked proof of the Four Colour Theorem. Technical report, Microsoft Research Cambridge, 2005. 12

Michael Gordon and Thomas Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge Univ. Press, 1993. 2

Michael Gordon, Robin Milner, Lockwood Morris, Malcolm Newey, and Christopher Wadsworth. A metalanguage for interactive proof in LCF. In *Proc. 5th ACM Symp. on Principles of Programming Languages*, pages 119–130. ACM, ACM Press, January 1978. 2

Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979. 2

Thomas Hales. Formalizing the proof of the kepler conjecture. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Proc. 17th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 3223 of *Lecture Notes in Computer Science*, page 117. Springer-Verlag, 2004. 12

John Hannan and Dale Miller. From operational semantics to abstract machines. *Math. Structures in Computer Science*, 2(4):415–459, December 1992. 49

Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional back-ends within the lambda-sigma calculus. In *Proc. 1st ACM Sigplan Int. Conf. on Functional Programming*, pages 25–33. ACM Press, 1996. 126

Susumu Hayashi and Hiroshi Nakano. *PX, a Computational Logic*. Foundations of Computing. MIT Press, 1988. 29

Hugo Herbelin. *Séquents qu'on Calcule*. PhD thesis, Université Paris VII, January 1995. 17

Hugo Herbelin. C'est maintenant qu'on calcule, 2005. Habilitation à Diriger les Recherches. 10, 18, 141

Arendt Heyting. Die formalen reglen der intuitionistischen logik. *Sitzungsberichete der Preussischen Akademie der Wissenschaften, Physikalische-Mathematische Klasse*, pages 42–56, 1930. 84

Douglas Howe. Importing mathematics from HOL into Nuprl. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Proc. 9th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, 1996. 142

Common Criteria for Information Technology Security Evaluation. ISO/IEC Standard 15408, version 2.0 edition, November 1998. 1

Ingebrigt Johansson. Der minimalkalkül, ein reduzierter intuitionistischer formalismus. *Compositio Mathematica*, 4:119–136, 1937. 29

Gueorgui Jojgov. Holes with binding power. In *Proc. 2002 Int. Workshop on Proofs and Programs*. Springer-Verlag, 2003a. 29

Gueorgui Jojgov. Tactics and parameters. In Herman Geuvers and Fairouz Kamareddine, editors, *Mathematics, Logic and Computation*, volume 85 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2003b. 36, 55

Gueorgui Jojgov. *Incomplete proofs and terms and their use in interactive theorem proving*. PhD thesis, Technische Universiteit Eindhoven, 2004. 69

Florent Kirchner. A finite first-order theory of classes. In *Proc. 2006 Int. Workshop on Proofs and Programs*, Lecture Notes in Computer Science. Springer-Verlag, 2006. 5

Florent Kirchner. PVS#: A streamlined tactical language. Technical report, National Institute of Aerospace, 2005a. Subsumed by (Kirchner and Muñoz, 2006). 5

Florent Kirchner. Store-based operational semantics. In *Seizièmes Journées Francophones des Langages Applicatifs*. Institut National de Recherche en Informatique et en Automatique, Unité Rocquencourt, 2005b. 5

Florent Kirchner and César Muñoz. PVS#: Streamlined tacticals for PVS. In *Proc. 6th Int. Workshop on Strategies in Automated Deduction*, August 2006. 5, 68, 150

Florent Kirchner and Claudio Sacerdoti Coen. The Fellowship proof manager. `www.lix.polytechnique.fr/Labo/Florent.Kirchner/fellowship/`, 2007. 109

Florent Kirchner and François-Régis Sinot. Rule-based operational semantics for and imperative language. In *Proc. 7th Int. Workshop on Rule Based Programming*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 35–47. Elsevier Science, August 2006. 5

Jean-Louis Krivine. *Lambda-calculus, Types and Models*. Ellis Horwood, 1993. Translated from the 1990 French original by René Cori. 7

Jacqueline Lelong-Ferrand and Jean-Marie Arnaudies. *Cours de Mathématiques. Tome 2 : Analyse*. Dunod, 1972. 134

Stéphane Lengrand. *Normalisation & Equivalence in Proof Theory & Type Theory*. PhD thesis, Université Paris 7 & University of St Andrews, 2006. 29

Stéphane Lengrand and Alexandre Miquel. A classical version of F_ω. In Stephen van Bakel and Stefano Berardi, editors, *Proc. 1st Workshop on Classical logic and Computation*, July 2006. 141

Stéphane Lengrand, Roy Dyckhoff, and James McKinna. A sequent calculus for Pure Type Systems. In Zoltan Esik, editor, *Proc. 15th Conf. for Computer Science Logic*, volume 4207 of *Lecture Notes in Computer Science*, pages 441–455. Springer-Verlag, September 2006. 141

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon). *The Objective Caml system, Documentation and user's manual*, release 3.07 edition, 2003. 71

Helen Lowe and David Duncan. XBarnacle: Making theorem provers more accessible. In William McCune, editor, *Proc. 14th Int. Conf. on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 404–407. Springer-Verlag, July 1997. 1

Lena Magnusson. *The Implementation of ALF: A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Chalmers University of Technology and Göteborg University, January 1995. 29

Andrew Martin and Jeremy Gibbons. A monadic interpretation of tactics, 2002. 36, 69

Andrew Martin, Paul Gardiner, and Jim Woodcock. A tactic calculus. In Joanne Allison, editor, *Formal Aspects of Computing*, 1996. 55

Micaela Mayero. *Formalisation et Automatisation de Preuves en Analyses Réelle et numérique*. PhD thesis, Université Paris VI, December 2001. 109, 137

Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer-Verlag, 2004. 71

Norman Megill. A finitely axiomatized formalization of predicate calculus with equality. *Notre Dame Journal of Formal Logic*, 36(3):435–453, Summer 1995. 139

Elliott Mendelson. *Introduction to Mathematical Logic (4th ed.)*. Chapman & Hall, 1997. ISBN 0-412-80830-7. 124, 138

José Meseguer and Grigore Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, to appear, 2006. 49

Eugenio Moggi. Computational lambda-calculus and monads. In *Proc. 4th IEEE Symp. Logic in Computer Science*. IEEE Comp. Soc. Press, 1989. Superseded by (Moggi, 1991). 58

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991. 40, 58, 151

César Muñoz. *Un calcul de substitutions pour la représentation de preuves partielles en théorie de types*. Thèse de doctorat, Université Paris 7, 1997. English version available as INRIA research report RR-3309. 33

César Muñoz. Proof-term synthesis on dependent-type systems via explicit substitutions. *Theoretical Computer Science*, 266:407–440, 2001a. 29

César Muñoz. Dependent types and explicit substitutions. *Math. Structures in Computer Science*, 11(1):91–129, February 2001b. It also appears as report NASA/CR-1999-209722 ICASE No. 99-43. 32

César Muñoz and Micaela Mayero. Real automation in the field. Technical Report NASA/CR-2001-211271 Report 39, ICASE Nasa-Langley Research Center, December 2001. 109, 137

Pavel Naumov, Mark-Oliver Stehr, and José Meseguer. The HOL/NuPRL proof translator: A practical approach to formal interoperability. In Richard Boulton and Paul Jackson, editors, *Proc. 14th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 329–345. Springer-Verlag, 2001. 142

Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. In Helmut Schwichtenberg and Klaus Spies, editors, *Proc. Marktobberdorf Summer School 2003*. IOS Press, 2003. 35

Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

Steven Obua and Sebastian Skalberg. Importing HOL into isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *Proc. 3rd Int. Joint Conf. on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 2006. 142

Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report CR-1999-209321, Computer Science Laboratory, SRI International, May 1999. 98

Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992. 2

Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Proc. 1st Int. Conf. on Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 328–345, 1993. 92, 93

Lawrence Paulson. Experience with isabelle : A generic theorem prover. Technical Report UCAM-CL-TR-143, University of Cambridge, Computer Laboratory, August 1988. 2

Lawrence Paulson. Set theory for verification: I: from foundations to functions. *Journal of Automated Reasoning*, 11(3):353–389, 1993. 139

Luís Pinto and Roy Dyckhoff. Sequent calculi for the normal terms of the λΠ- and λΠΣ-calculi. In Didier Galmiche, editor, *Proc. Workshop on Proof Search in Type-Theoretic Languages*, volume 17 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, July 1998. 141

Andrew Pitts. Operational semantics and program equivalence. In Gilles Barthe, Peter Dybjer, and João Saraiva, editors, *Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer-Verlag, 2002. 49

Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981. 35

Robert Pollack. *The Theory of Lego: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994. 2

François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005. 36

Dag Prawitz. *Natural Deduction: a Proof-Theoretical Study*, volume 3 of *Stockholm Studies in Philosophy*. Almqvist & Wiksell, 1965. 91

Art Quaife. Automated deduction in von Neumann-Bernays-Gödel set theory. *Journal of Automated Reasoning*, 8(1):91–147, 1992. 139

Cecylia Rauszer. Semi-boolean algebras and their applications to intuitionistic logic with dual operations. *Fundamenta Mathematicae*, 83(3):219–249, 1974. 122

Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In Joe Hurd and Thomas Melham, editors, *Proc. 18th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 294–309. Springer-Verlag, August 2005. ISBN 3-540-28372-2. 123

John M. Rushby. Harnessing disruptive innovation in formal verification. In *Proc. 4th IEEE Int. Conf. on Software Engineering and Formal Methods*, volume 0, pages 21–30. IEEE Comp. Soc. Press, 2006. 143

Claudio Sacerdoti Coen. Explanation in natural language of $\bar{\lambda}\mu\tilde{\mu}$-terms. In *Proc. 4th Int. Conf. on Mathematical Knowledge Management*, volume 3863 of *Lecture Notes in Artificial Intelligence*, pages 234–249. Springer-Verlag, 2006. 24, 26, 103

Alexis Saurin. Focussing as proof normalization, 2006. Informal discussions. 24

Helmut Schwichtenberg. Minimal logic for computable functions. In Friedrich Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Series F: Computer and Systems Sciences*, pages 289–320. Springer-Verlag, 1993. 29

Paula Severi and Erik Poll. Pure type systems with definitions. In Anil Nerode and Yuri Matiyasevich, editors, *Proc. 3rd Int. Symp. on Logical Foundations of Computer Science*, volume 813 of *Lecture Notes in Computer Science*, pages 316–328. Springer-Verlag, 1994. 106

Natarajan Shankar, Sam Owre, John Rushby, and David Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, September 1999. 68

Andrzej Trybulec. The Mizar-QC/6000 logic information language. In *Association for Literary and Linguistic Computing Bulletin*, volume 6, pages 136–140, 1978. 2

Dictionary of Military and Associated Terms. United States Departement of Defense, April 2001. Joint Publication 1-02. 3

Christian Urban. Strong normalisation for a gentzen-like cut-elimination procedure. In Samson Abramsky, editor, *Proc. 5th Int. Conf. on Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 415–429. Springer-Verlag, 2001. 17

Christian Urban and Gavin Bierman. Strong normalisation of cut-elimination in classical logic. In *Proc. 4th Int. Conf. on Typed Lambda Calculi and Applications*, pages 365–380. Springer-Verlag, 1999. ISBN 3-540-65763-0. 141

Stéphane Vaillant. A finite first-order presentation of set theory. In *Proc. 2002 Int. Workshop on Proofs and Programs*, pages 316–330, 2002. 124, 139

Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Aart Middeldorp, editor, *Proc. 12th Int. Conf. on Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer-Verlag, 2001. 142

John von Neumann. Eine axiomatisierung der mengenlehre. *Journal für die reine und angewandte Mathematik*, 154:219–240, 1925. 124

Philip Wadler. Call-by-value is dual to call-by-name. *ACM sigplan notices*, 38(9):189–201, September 2003. 18

Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995. 40, 61

Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *OOPSLA '06: Object oriented programming, systems, languages, and applications*. ACM Press, 2006. 36

Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge Univ. Press, 1910. 71

Freek Wiedijk. *The Seventeen Provers of the World*. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2006. 2

Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing. MIT Press, 1993. WIN g2 93:1 P-Ex. 36, 43

Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994. 35, 49

This dissertation was typeset using the LaTeX typesetting system, the memoir class, and the BibTeX bibliography database system. The body text is set 10/12pt on a 27.4pc measure with Computer Modern Roman. Mathematics are set with Euler Roman and Script. Other fonts include Sans, Smallcaps, Italic, Slanted and Typewriter, all from the Computer Modern family.