Applications of Foundational Proof Certificates in theorem proving

PAR

M. Roberto Blanco Martínez

Thèse de doctorat

DE

l'**université**
PARIS-SACLAY

PRÉPARÉE À

ÉCOLE
POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY
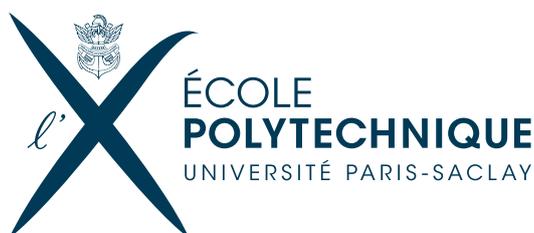
Spécialité de doctorat: Informatique

École doctorale n° 580
Sciences et technologies de l'information et de la communication

NNT: XXX

Thèse présentée et soutenue à Palaiseau, le XX décembre 2017

Composition du Jury:

Mme Laura Kovács	Professeure des universités	Rapporteuse
	Technische Universität Wien &	
	Chalmers tekniska högskola	
M. Dale Miller	Directeur de recherche	Directeur de thèse
	Inria & LIX/École polytechnique	
M. Claudio Sacerdoti Coen	Professeur des universités	Rapporteur
	Alma Mater Studiorum - Università di Bologna	

D
D
E
E
F
F

Applications of Foundational Proof Certificates in theorem proving

For Jason.

Contents

List of figures

Preface

Like any scientific undertaking, a thesis does not develop in a vacuum, and personal as it is, it is informed by the indirect as well as the direct influence of many people. In the roll call of important actors that must of need accompany such a document, regrettable omissions may be nigh inevitable. Indeed, as with test cases and program bugs, a mention in a preface may show the presence of an acknowledgment, not its absence—and in the flurry of writing and its companion events, in keeping with the software analogy, slips are not altogether unlikely to occur. In the world of computer science, at least, there is hope for getting things just right; this work marks the point at which I start walking in that direction, resolutely and in earnest.

First and foremost in importance and intensity, I am indebted to Dale for the pleasure and the privilege of working with him. Much more than a *directeur*, he is a teacher and a mentor, knowledgeable and wise, and the nicest person you will ever meet; words do not do justice. May our paths continue to meet.

From the moment I entered the scene in April 2014, arriving on the plateau de Saclay after the summer, Inria and École polytechnique have been wonderful places to work and study. Dale's team Parsifal takes most of the credit for being friendly and collegial to the utmost. Much has changed on the plateau in the intervening three years, the primordial constant being a pleasant and productive life thanks to my fellow Parsifalians: an eclectic group of talented and wonderful people, among whom I am honored to count myself. With the permanent members I have partaken of all manners of conversations in and out of science, meals in and out of the lab, teaching and offices, mostly with Kaustuv Chaudhuri, Beniamino Accattoli, Stéphane Graham-Lengrand, and more recently Gabriel Scherer. In addition to these activities, I have shared the experience of these formative years with Zakaria Chihani and the rest of my student cohort: Quentin Heath, Sonia Marin, Ulysse Gérard, and Matteo Manighetti. And, of course, the

perspective and company of the young researchers we must become, especially Taus Brock-Nannestad, Danko Ilik, Tomer Libal, Giselle Reis, and Marco Volpe.

In addition, Alberto Momigliano has been a genial colleague since we met during his visit to Parsifal in 2016. We have a lot of interesting research to do, and hopefully we shall find the time and the opportunity to continue to work together, as has been my pleasure until now.

In fulfilling my teaching duties at Polytechnique, Eric Goubault, Benjamin Werner, Frank Nielsen and Stéphane allowed me to participate actively in the development of the classes they coordinated and welcomed my suggestions, improvements and proposals. Benjamin, along with Kaustuv and Julien Signoles, with whom I had the pleasure to co-teach, and Dale, were all energetically supportive despite the nebulous bureaucracy involved; without them a smooth run would have been hardly conceivable, and for this they have my sincere gratitude. None of this would have meaning without my students at the *cycle d'ingénieur polytechnicien*, always hard-working, insightfully clever and invariably polite, in sum the finest soil a professor could ask to assist in cultivating.

The administrative team at Inria, including Valérie Berthou, Catherine Bensoussan and very especially Christine Biard, helped me navigate the labyrinth of bureaucracy—an everchanging maze whose strange corners I visited more than once. On occasion the whole edifice looked Escherian indeed, but they never failed to steer me in the right direction and dexterously remove most of the pain from paperwork and procedure.

Scientific evaluation is the more enriching part of administrative procedure. Sylvain Conchon and Hugo Herbelin accepted to serve in the committee of my *suivi approfondi à mi-parcours* and provided detailed commentary on my developing research project.

As the formal culmination of my doctoral studies and the act of scientific evaluation par excellence, the defense is given meaning by the senior colleagues who ratify the content of the present work. In particular, at this point in time, I am indebted to Laura Kovács and Claudio Sacerdoti Coen, who have kindly agreed to serve as reporters. As I write these lines they are the two confirmed members of my jury; other examiners will join in, and be duly recognized, in the near future. Dale's comments and conversations, always insightful, have percolated through and enriched the thesis, as have the observations of those who have gracefully offered to read and proofread the manuscript.

Outside the lab I have benefited from the vibrant scientific community around Paris, much more so as I became proficient enough to grok the depth and the significance of at least a decent part of what goes on around me. The hospitality and the seminars of such groups as Deducteam, Gallium, LRI, LSV, Prosecco... continue to enrich me and broaden my perspective.

Last, never least, my family and friends, unsung heroes of science, have always been supportive and understanding in the many days when I have "*pulled a Roberto*" on them and become engrossed in this stuff of beauty that is logic. Misty left us during this time; she is dearly missed.

With the—unlikely—exception of false premises coming from erroneous (and therefore uncertified) proofs, any remaining errors are, of course, my own.

Palaiseau, October 2017

1 Introduction

The unifying theme of this thesis is how to establish trust that a certain property, i.e., a theorem about some formalizable artifact, holds. If the abstract answer to this question is "with a proof," the issue becomes then to establish how proofs are represented and how to trust them. A document or more generally an act of communication purporting to represent a proof of a theorem cannot be blindly trusted: it must be checkable independently from its origin. In a world of mechanized proofs of formidable size and complexity, in addition to a convenience automation becomes a necessity.

Thus, it would be desirable to represent pieces of proof evidence from very different provenances in a unified format: proofs would then be represented by a document, or certificate, to which would be attached the description of the syntax of the proof and its semantics. With this information, independent checking becomes possible. As it is proofs we are dealing with, the branch of logic that studies proofs as mathematical objects, proof theory, is an obvious candidate for such an undertaking. From these theoretical foundations, the ProofCert project has developed broad-spectrum proof formats called Foundational Proof Certificates (FPC). The resulting framework establishes how to define certificate formats and build simple, trustworthy checkers that consume them. A checker results from the combination of a proof checking kernel and the definition of a proof certificate format.

Until now, research on FPCs has validated it and seen it applied to a range of proof systems in proofs of concept, but had yet to graduate to certificates "in the large." In addition, with the desirable theoretical properties of the FPC framework established, it remains to consider how an implementation of the framework can be trusted: who watches the watchmen? A further point of inquiry is whether proof certificates can represent more than inert evidence of a proof: whether in addition to checking them, proof certificates can grow and

shrink, whether they can be queried and interacted with. We shall consider all these questions in turn. These investigations will be placed in relation to both great groups of tools: automated and interactive theorem provers. The document is structured in three parts.

Part I collects the general background common to the two main parts that follow it; it is complemented by more specialized background at the beginning of each of Part II and Part III. It is composed by two chapters.

Chapter 2 commences with an overview of the concept of proof and its abstract treatment by the discipline of proof theory. It concentrates on sequent calculi as the proof systems of choice and their treatment of canonical forms of proofs to the point to which normalization can be gained by the development of focused proofs.

Chapter 3 continues where the preceding chapter leaves off by further motivating the use of proof evidence as the foundation of trust and developing the sequent calculus into a logically sound, programmable framework where proof construction can be flexibly steered by the Foundational Proof Certificates that are at the center of this research. It complements this discussion with a number of examples which play various roles in subsequent chapters; these are the kind of small, textbook systems with which the FPC framework has been used to date with small, handcrafted examples.

Part II studies developments taking place in a classical logic without fixed points, where the open-world assumption holds. Logics such as these underlie standard logic programming languages and automated theorem provers.

Chapter 4 completes the background of Part I with the technical setting specific to this part of the document. It briefly discusses the uses of logic in computation to concentrate on the use of logic itself as a paradigm for computation, that is, logic programming. After reviewing the common foundations of Prolog, it presents the salient features of λProlog, the language used throughout this part, and demonstrates the implementation of the kernel of a proof checker for the FPC framework in that language, which can be used to execute the examples in Chapter 3 as well as the original developments of future chapters.

Chapter 5 explores the idea of combining sources of proof evidence by defining a combinator that attempts to construct a proof of a formula using two separate pieces of proof evidence in parallel. This opens rich possibilities of combination of proof search strategies and of elaboration of proof evidence. Using standard features of logic programming, this pairing combinator can be used to

enrich a proof certificate with additional information (making it more efficient), compacting it by removing inessential information (making it potentially slower), or querying and extracting information about proofs. If a certificate is enriched to be essentially a trace of the logical computation, the reproduction of this trace is performed as a determinate computation, which does not resort to the distinguishing features of logic programming

Chapter 6 exploits traces of computation of a proof used as proof evidence, which constitute not a general logic computation, but a particular case of functional computation. This suggest that we can implement specialized proof checkers for the determinate fragment of the FPC framework as functional programs, which rely on simpler languages and runtimes and avoid the hard problems that a logic programming language must integrate. Going further, the operation of these simplified proof checkers can be modeled in a proof assistant like Coq and proved correct. The result is a verified implementation of a simplified proof. Since the FPC definitions from Chapter 5 always allow the extraction of a full trace, any proof certificate defined in the framework can be run in the simplified, verified setting by that intermediate translation step.

Chapter 7 begins to move the FPC framework from the realm of small examples and prototypes into the domain of theorem provers and proof formats as they are used in practice. Here we look at a family of automated theorem provers that solve instances of the classical boolean satisfiability problem. These tools are widely used and have already recognized the necessity for independent validation of results by defining their own proof output formats. We develop a proof theoretic view of the proofs underlying those formats representing proofs of unsatisfiability of a propositional formula, and show how to interpret their information as a proof certificate capable of guiding the reconstruction of a proof in our proof checkers.

Chapter 8 proceeds in the same vein and seeks to apply the FPC framework to certify proofs produced by a concrete automated theorem prover. The general recipe involves analyzing the proof evidence produced by the concrete tool and modeling it in terms of a proof calculus, which in turn can be described by an FPC definition. We perform this study for Prover9, a venerable prover based on a resolution proof calculus like many leading automated theorem provers. Starting from a corpus of publicly available proofs, we apply the methods of Chapter 5 to obtain several certificates corresponding to the same proof with increasing amounts of information and study the effects on certificate size and checking time.

In addition to the general-purpose checkers built in λProlog, the determinate checkers of Chapter 6 are employed as well.

Part III studies developments in the setting of intuitionistic logics to which fixed points are added; here, the closed-world assumption replaces the open world of Part II. These logics are typically used in the design of proof assistants.

Chapter 9 begins the background specific to this part by complementing Chapter 2 with a logic where inductive and coinductive definitions, as well as term equality, are featured as first-class logical connectives. A constructive logic such as those that commonly serve as the foundations of proof assistants is presented with those new logical features; the logic is then extended to support focused proofs, and then the FPC framework on top of those.

Chapter 10, analogue to Chapter 4, completes the background specific to Part III by studying how logic programming is affected by extending the logic with the features in the previous chapter. The proof assistant Abella serves as the programming language of choice for this part. After a short description of the system, its use is exemplified by an implementation of the proof checking kernel used in the chapters that follow.

Chapter 11 begins by inspecting the common structure of many proof developments both in informal mathematics and in the formalized setting of proof assistants, and endeavors to support these schematic styles of proof in the FPC framework. To this end, we define families of proof certificates that express succinctly the high-level structure of a proof by resorting to high-level concepts such as induction and use of previously proved lemmas. Several variations of this general scheme experiment with the amount of detail added to these descriptions and their effects on proof reconstruction by the checker at runtime. The resulting proof outlines are closely related to the less strict languages of tactics and tacticals that proof assistants expose to their users for the composition of proof scripts.

Chapter 12 adapts the FPC framework to the complement of proving formulas: disproving them. Much like programs may contain bugs, logical developments may contain errors either by attempting to prove properties about the wrong things or by attempting to prove the wrong properties about things. When a proof assistant is confronted with a proof obligation, it may try to quickly find a counterexample that falsifies the property, which indicates to the user that either the specification or the property are wrong, and allows for correction before much effort has been invested in a proof attempt that is doomed to fail. Compact representations of the strategies that are commonly used to find counterexamples

are given as proof certificates: exhaustive and random generation of test data is followed by testing of the properties with the generated data. The expressive logics considered in this part make it possible to apply these techniques to the domain of the metatheory of programming languages.

Chapter 13 integrates these developments in Abella. It considers the technical and algorithmic extensions that go into thoroughly integrating the FPC framework in a proof assistant. Proof certificates become a complement and even a substitute for the built-in tactics language, which is replaced with another language, that of certificates, which is not only programmable but offers the formal guarantees of correctness of the framework.

Often a subject is treated in the context of one of the two main parts while being in whole or in part applicable to the other part. Each chapter is completed by a Notes section that complements the preceding sections with extended discussions and background, as well as connections to other chapters.

Part I

Logical foundations

2 Structural proof theory

2.1 Concept of proof

Logic—the scientific study of reasoning and deduction—has the concept of *proof* at its core: how is new knowledge created from what is already known? One presents the fruit of deduction as a proof, but what exactly is a proof, and how can we recognize one, that is, ensure that a claimed proof is correct? Our modern understanding of the concept harkens back to David Hilbert's vigorous efforts to infuse all of mathematics with complete and absolute (metamathematical) rigor—formalized by means of the kind of axiomatic proof systems which were pioneered by such bodies of work as Gottlob Frege's *Begriffsschrift* and Alfred North Whitehead and Bertrand Russell's *Principia Mathematica*. These same efforts sparked foundational controversies with mathematicians like Frege and L. E. J. Brouwer, and spurred Kurt Gödel to develop his incompleteness theorems—which demonstrated that Hilbert's lofty ambitions, at least in their original form, were unattainable. The study of proofs as first-class mathematical objects developed in close connection with these advances.

Vis-à-vis the purely mathematical conception of proof are the philosophical and sociological faces of argumentation as a mental activity and an act of communication. It is recognized that proof has a dual nature by which it can be seen alternatively as *proof-as-message* or as *proof-as-certificate*. Taken as a message, the purpose of a proof is the transmission of insight and understanding between mathematicians: to convey the lines of argumentation followed to arrive at a conclusion and to *convince* of the truth of that conclusion—the focus is on meaning, on *semantics*. Seen instead as a certificate, the purpose of proofs is the transmission and mechanical verification of knowledge: the use of the symbols and rules of a formal language to unambiguously derive new, correct phrases—the focus here is on *syntax*. In practice, both functions are closely related, and although

formal study tends to emphasize the side of proof-as-certificate, proof-as-message considerations will be pervasive in much of the work that follows (especially throughout Part III, where *proof assistants* and user interaction are key subjects).

The rest of the chapter is organized as follows. Section 2.2 traces the development of proof theory and describes its major formal systems. Section 2.3, in parallel, outlines the most important division in the taxonomy of standard logics, that separating classical and intuitionistic logics, both of which will be called upon over the course of our study. Section 2.4 presents the sequent calculus, one of the major deductive systems, on which our formal studies will be based. Section 2.5 introduces the discipline of focusing, used to structure the proofs of the sequent calculus in more organized, abstract forms. Section 2.6 summarizes some important considerations on the relation between proof systems and the logics they model. Section 2.7 concludes the chapter.

2.2   Evolution of proof theory

Proof theory is the branch of mathematical logic that studies proofs as objects of mathematics. It is widely acknowledged that the modern study of proof has its roots in the axiomatizing undertakings of Hilbert's Program. Among the disciplines of proof theory, *structural proof theory* studies the structure and properties of proofs (in the sense of proof-as-certificate of the previous section). The concrete objects of its study are *proof calculi*: formalized systems of deduction where formulas and proofs are inductively defined objects, and the steps of deduction are carried out syntactically by the application of inference rules which transform formulas and in the process construct proof objects. There are three principal families of proof calculi, each of which will be presented and placed in its proper context in this section.

First, *Hilbert systems* take their name from the refined calculus developed by Hilbert for the development of his Program. They reflect the longstanding tradition of organizing mathematical developments as a sequence of steps which starts from instances of a collection of logical axioms—for example, that every property P implies itself: $P \supset P$—and follows by applications of inference rules which derive new facts from previously known ones. Logical reasoning has historically relied on this discursive style, from Aristotle to Gottlob Frege—after whom these calculi are sometimes named *Frege systems*, as we shall reference in Chapter 11. Often, a single inference rule, that of *modus ponens*, is used. This

rule states that if we know that a fact Q is implied by another fact P, and we also know that P holds, then we can infer that Q holds. For this reason (and closer to the uniform terminology that will be used shortly) modus ponens is also called *implication elimination* and depicted schematically as follows:

$$\frac{P \supset Q \quad P}{Q}$$

Second, *natural deduction* systems arose as a response to the linear and unstructured proofs of Hilbert-style systems, to better reflect the "natural" way in which proofs are built by mathematicians—in fact, although Hilbert systems have later been inspected under the lens of structural proof theory, it is in natural deduction that the discipline has its proper genesis. Natural deduction was developed by Gerhard Gentzen (1935) in his landmark dissertation with the goal of accurately reflecting the mental process of reasoning and its dependencies. Centrally, it employs the concept of *assumptions* made over the course of a proof attempt and which may be closed at some later point. An important question in this more structured system is whether there exist *normal forms* of natural deduction proofs, so that many shallowly different derivations of the same property may be given a common representation. In his dissertation, Gentzen attempted to prove such a normalization property, succeeding in the intuitionistic case but failing in the classical case—eventually, Dag Prawitz (1965) succeeded in doing so within the edifice of natural deduction, and we now know Gentzen himself persevered in his efforts until the knot was untangled (von Plato, 2008).

The third and last great family of deductive systems is the *sequent calculus*, which Gentzen developed to work around the difficulties of the proof of normalization for natural deduction. Its technical motivation was to serve as a sort of meta-calculus in which the derivability relation of natural deduction could be expressed and reasoned about: the sequent calculus is more pedantic, but also more practical. In this formalism, an analog of normalization for sequent calculus proofs called *cut elimination* could be proved, and this result in turn rendered the original pursuit—i.e., a proof of normalization for intuitionistic natural deduction—unnecessary. Sequent calculi proved to be fertile theoretical ground, more mechanistic than their forebears and far more relevant in the looming age of computer science. They form the immediate substrate of the present work and will be discussed extensively in the pages that follow; after a brief orthogonal discussion, they will be properly introduced in Section 2.4.

2.3 Classical and intuitionistic logics

Perhaps the most fundamental division in modern logic concerns the distinction between the standard logics: *classical logic* and *intuitionistic logic*. Classical logic is a "logic of truth," concerned with the assignment of truth values (say, true or false) to formal statements. The traditional study of logic starting with Aristotle can be framed in this tradition. By contrast, intuitionistic logic is a "logic of construction." It was developed from L. E. J. Brouwer's philosophy, notably by Arend Heyting. In proving properties about mathematical objects, an intuitionistic proof provides a way to construct objects exhibiting the properties being proved. For example, to have a proof of "*A* and *B*," we need separately a proof of *A* and a proof of *B*.

Under the classical interpretation, the negation of a statement is an assertion of its falsity. Conversely, the intuitionistic negation of a statement points to the existence of a counterexample. As an illustration, consider *proof by contradiction*, a standard proof technique commonly used in mathematics. A proof by contradiction of a property *p* starts by assuming that *p* does not hold and proceeds to arrive at a contradiction. Many commonplace results—such as the classic proof of the irrationality of $\sqrt{2}$—make use of this method. However, such non-constructive arguments are invalid in intuitionistic logic.

In practice, classical logic and intuitionistic logic can be related by disallowing in the latter the non-constructive parts of the former—that is, the principle of the *excluded middle*, which asserts that for every statement either itself or its negation is true. By virtue of this constraint, intuitionism rejects the aberrant "proofs" of classical logic which rely on non-constructive arguments. A priori, because intuitionistic logic can be defined as a restriction of classical logic, not only can intuitionistic logic not be stronger than classical logic, but it would moreover seem to be *weaker*—every intuitionistically provable theorem is classically provable, but there exist classical theorems which are not intuitionistically provable.

Nonetheless, the relation between the expressive powers of both logics is subtler than their hierarchical relation might suggest. In the settings of propositional and first-order logic, there exist translation functions such that, if a formula is a theorem of classical logic, its translation is a theorem of intuitionistic logic: such functions are called *double-negation translations* (Ferreira and Oliva, 2012)—in fact, what such a function does is making explicit each use of the excluded middle by means of an encoding based on double negations. (However, a formula is not

intuitionistically equivalent to its translation, and no general mappings exist in the opposite direction.) This leads to the observation that—in these settings—intuitionistic logic is more *expressive*, or *finer* than classical logic: if a formula is classically but not intuitionistically provable, we can find another formula which is intuitionistically provable and classically indistinguishable from the original.

If classical logic can be seen as the logic of traditional mathematics, intuitionistic logic takes on the mantle of the logic of computer science. In the context of theorem proving, classical logic serves as the foundation of choice for reasoning in automated theorem provers, where a computer program attempts to find proofs of theorem candidates. In interactive theorem provers (or proof assistants), where the user drives the search for proofs, the foundational role is assumed by intuitionistic logic. Both classical and intuitionistic logics will be employed extensively as the bases for Part II and Part III, respectively.

2.4 Sequent calculus

This section introduces Gentzen's original sequent calculus, the proof theoretical foundation of our investigations. We concentrate on the calculus for classical logic and reference intuitionistic logic when appropriate. Classical logic will be the focus of Part II, while Part III will adopt intuitionistic logic as its vehicle; additional background is given in Chapter 9.

Classical logic contains the familiar set of *logical constants* or *connectives*. In the propositional fragment, we have the nullary constants *true* (t) and *false* (f); the unary constant *negation* (or *not*, $\neg$); and the binary constants *conjunction* (or *and*, $\wedge$), *disjunction* (or *or*, $\vee$), and *implication* ($\supset$). First-order logic extends propositional logic with the *universal quantifier* ($\forall$) and the *existential quantifier* ($\exists$). The standard classical equivalences establish the interrelations between these constants. We make no attempt to reduce the set of connectives to a small, or minimal, functionally complete set of operators.

Added to these logical constants will be a *type signature* of *non-logical constants* that will function as *atomic propositions* (or simply *atoms*). A *literal* is either an atom or a negated atom. In first-order logic, as quantifiers are introduced, atoms can be parameterized by *terms*, also part of the signature. Quantifiers bind names within their scope and can be instantiated by the operation of *substitution*—all the usual subtleties and caveats about free and bound variables, capture-avoiding substitution, etc., apply here. Given a formula A, $[t/x]A$ designates the substitu-

tion of a term t for a (free) variable x in A. A more computational view of all these aspects is deferred until Chapter 4.

Sequent calculi are formal logic systems organized around the concept of *sequents*. In its basic form, a sequent combines two arbitrary lists of formulas separated by a *turnstile*, say:

$$A_1, \ldots, A_m \vdash B_1, \ldots, B_n$$

The list of formulas to the left of the turnstile is called the *left-hand side* (LHS) or the *antecedent* of the sequent and abbreviated Γ. The list of formulas to the right of the turnstile is called the *right-hand side* (RHS) or *succedent* of the sequent and abbreviated Δ. The classical semantics of a sequent states that, if all the formulas in the left-hand side are true, at least one formula in the right-hand side is true, as well. Equivalently, the antecedent list represents a conjunction of formulas, and the succedent list represents a disjunction of formulas.

Every sequent calculus is presented as a collection of *inference rules* on sequents. Each inference rule has one *conclusion* below the line and any number of *premises* above the line—possibly complemented by provisos and side conditions. The inference rules are actually *rule schemata*: they are essentially "universally quantified" over all their variables (representing arbitrary formulas, list of formulas, etc.), so that any combination of values for those syntactic variables constitutes an *instance* of that inference rule. The rules of the standard sequent calculus for classical logic are presented in Figure 2.1. (These rules do not cover the nullary logical constants true and false, which can be trivially defined in terms of the other connectives.)

The rules of the calculus are organized in three distinct groups. First, logical rules, also called *introduction rules*, analyze each logical connective on both sides of the turnstile in the conclusion of the rule and relate the conclusion to the necessary premises for the introduction of the connective. Second, *identity rules* treat the (symmetric) cases where the same formula appears on both sides of the turnstile: the *axiom* rule in the conclusion; and the *cut* rule in the premises. And third, *structural rules* manipulate the structure of the sequent without inspecting its formulas: they are *exchange*, which reorders the components of the sequent; *weakening*, which introduces new formulas in the conclusion; and *contraction*, which makes copies of formulas in the premise. This presentation showcases the deep, remarkable symmetry of classical logic.

INTRODUCTION RULES

$$\dfrac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge_L^1 \qquad\qquad \dfrac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee_R^1$$

$$\dfrac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge_L^2 \qquad\qquad \dfrac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee_R^2$$

$$\dfrac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \vee_L \qquad\qquad \dfrac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \wedge_R$$

$$\dfrac{\Gamma_1 \vdash A, \Delta_1 \qquad \Gamma_2, B \vdash \Delta_2}{\Gamma_1, \Gamma_2, A \supset B \vdash \Delta_1, \Delta_2} \supset_L \qquad\qquad \dfrac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \supset B, \Delta} \supset_R$$

$$\dfrac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \neg_L \qquad\qquad \dfrac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \neg_R$$

$$\dfrac{\Gamma, [t/x]A \vdash \Delta}{\Gamma, \forall x.A \vdash \Delta} \forall_L \qquad\qquad \dfrac{\Gamma \vdash [y/x]A, \Delta}{\Gamma \vdash \forall x.A, \Delta} \forall_R \;\dagger$$

$$\dfrac{\Gamma, [y/x]A \vdash \Delta}{\Gamma, \exists x.A \vdash \Delta} \exists_L \;\dagger \qquad\qquad \dfrac{\Gamma \vdash [t/x]A, \Delta}{\Gamma \vdash \exists x.A, \Delta} \exists_R$$

IDENTITY RULES

$$\dfrac{}{A \vdash A} \; axiom \qquad\qquad \dfrac{\Gamma_1 \vdash A, \Delta_1 \qquad \Gamma_2, A \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2} \; cut$$

STRUCTURAL RULES

$$\dfrac{\Gamma_1, B, A, \Gamma_2 \vdash \Delta}{\Gamma_1, A, B, \Gamma_2 \vdash \Delta} E_L \qquad\qquad \dfrac{\Gamma \vdash \Delta_1, B, A, \Delta_2}{\Gamma \vdash \Delta_1, A, B, \Delta_2} E_R$$

$$\dfrac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} W_L \qquad\qquad \dfrac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta} W_R$$

$$\dfrac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta} C_L \qquad\qquad \dfrac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta} C_R$$

2.1 Figure   The *LK* proof system for classical logic with two-sided sequents (Gentzen, 1935). Here, A and B are arbitrary formulas; Γ and Δ are lists of formulas. The proviso marked as † is the usual eigenvariable restriction that y must not be free in the components of the premise (Γ, A, and Δ).

INTRODUCTION RULES

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge_L \qquad\qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee_R$$

$$\frac{\Gamma_1, A \vdash \Delta_1 \qquad \Gamma_2, B \vdash \Delta_2}{\Gamma_1, \Gamma_2, A \vee B \vdash \Delta_1, \Delta_2} \vee_L \qquad\qquad \frac{\Gamma_1 \vdash A, \Delta_1 \qquad \Gamma_2 \vdash B, \Delta_2}{\Gamma_1, \Gamma_2 \vdash A \wedge B, \Delta_1, \Delta_2} \wedge_R$$

2.2   **Figure**   The multiplicative fragment of the *LK* proof system for classical logic with two-sided sequents. These rules replace $\wedge_L^i$, $\vee_R^i$ (for $i \in \{1, 2\}$), $\vee_L$, and $\wedge_R$ in Figure 2.1; the rest of the system remains unchanged. Presentation conventions are shared with Figure 2.1.

An introduction rule corresponds to the analysis of a formula on a certain side of the conclusion sequent with a certain top-level connective. The remaining contents of the conclusion (the Γ and Δ lists) are called the *context*. In Figure 2.1, we have given a modern presentation of the original *LK* calculus as developed by Gentzen. Such a presentation is said to be *additive* because of the relation between the parts of the conclusion sequent and the parts of the premise sequents—namely, the context is the same across all premises (and coincides with the context in the conclusion). In contrast, a *multiplicative* reading requires that the parts of the conclusion exactly match the sum of the parts of the premises—contexts are disjoint across premises and merged in the conclusion. Two kinds of changes are made to enforce this regime. First, the split one-premise and-right (resp. or-left) rules are merged by requiring both conjuncts (resp. disjuncts) to be available in the premise. Second, the two-premise rules split the lists of formulas Γ and Δ so that the provenance of each is recorded. The alternative rules are shown in Figure 2.2. In classical logic, these two views are interadmissible in the presence of weakening and contraction—i.e., both resulting calculi prove exactly the set of theorems of classical logic.

The initial rules determine how a single formula is introduced or eliminated from both sides of a sequent simultaneously. The axiom rule is the sole proper initial rule of the classical calculus, in that it has no premises and can be used at all times. Gentzen's main result is the admissibility of the rule of cut under the form of the crucial *cut-elimination theorem*, which states that any theorem whose proof makes use of the cut rule possesses another proof which does not employ the cut rule. This result is the cornerstone of the theoretical study of sequent calculi and among the first important properties to establish for a new calculus,

from which other related results generally follow easily. It corresponds to the notion of normal forms of proofs and strong proof normalization in systems like natural deduction, and underlies the original motivation for the conception of the sequent calculus.

Finally, structural rules are the administrative elements of the calculus. First, the exchange rules allow arbitrary reorderings of the components of a sequent. As the introduction rules are defined to analyze formulas at specific positions—here, the rightmost left-hand side formula or the leftmost right-hand side formula, i.e., on the immediate neighbors of the turnstile symbol—it is necessary to rearrange the formulas on which work is to be carried out. Second, the contraction rules state that if a formula is present in a side of the sequent, its cardinality does not matter, as we can always make more copies. Together with the ultimate irrelevance of ordering by way of exchange, these rules substantiate the view of the LHS and RHS as *sets* of formulas. More commonly, they are modeled as *multisets*, which removes the exchange rules—this makes the instantiation of rule schemata somewhat less concrete. Third and last, the weakening rules allow the insertion of arbitrary formulas in the conclusion sequent.

In addition to the standard *two-sided* presentation of sequent calculus—where formulas appear on both sides of the turnstile—classical logic also admits a *one-sided* presentation, where all formulas are located on the right-hand side. In this version of the classical sequent calculus, two significant changes take place relative to the two-sided calculus. First, implication is defined as its classical reading, so that: $A \supset B \equiv \neg A \vee B$, for all A and B. Second, negation is only allowed to have atomic scope, i.e., as part of a literal. In consequence, all formulas are assumed to be in *negation normal form*, or NNF. To push negations down to the atomic level, the De Morgan dual equivalences are used in combination with the double-negation translation. The resulting rewrite system implementing the NNF translation is strongly normalizing and confluent—hence, straightforward application of the translation rules terminates and arrives at the unique NNF. For arbitrary formulas A and B:

$$
\begin{array}{rclcrcl}
\neg(A \wedge B) & \rightarrow & \neg A \vee \neg B & \qquad & \neg \forall x.A & \rightarrow & \exists x.\neg A \\
\neg(A \vee B) & \rightarrow & \neg A \wedge \neg B & \qquad & \neg \exists x.A & \rightarrow & \forall x.\neg A \\
\neg\neg A & \rightarrow & A
\end{array}
$$

INTRODUCTION RULES

$$\frac{\vdash A, \Gamma}{\vdash A \vee B, \Gamma} \vee^1 \qquad\qquad \frac{\vdash B, \Gamma}{\vdash A \vee B, \Gamma} \vee^2 \qquad\qquad \frac{\vdash A, \Gamma \qquad \vdash B, \Gamma}{\vdash A \wedge B, \Gamma} \wedge$$

$$\frac{\vdash [y/x]A, \Gamma}{\vdash \forall x.A, \Gamma} \, \forall \, \dagger \qquad\qquad \frac{\vdash [t/x]A, \Gamma}{\vdash \exists x.A, \Gamma} \, \exists$$

IDENTITY RULES

$$\frac{}{\vdash A, \neg A} \; axiom \qquad\qquad \frac{\vdash A, \Gamma_1 \qquad \vdash \neg A, \Gamma_2}{\vdash \Gamma_1, \Gamma_2} \; cut$$

STRUCTURAL RULES

$$\frac{\vdash \Gamma_1, B, A, \Gamma_2}{\vdash \Gamma_1, A, B, \Gamma_2} \, E \qquad\qquad \frac{\vdash \Gamma}{\vdash A, \Gamma} \, W \qquad\qquad \frac{\vdash A, A, \Gamma}{\vdash A, \Gamma} \, C$$

2.3 Figure The *LK* proof system for classical logic with one-sided sequents (Schütte, 1950; Tait, 1968). The system consists of versions of the right rules of the two-sided *LK* where formulas are by definition in negation normal form. Rules for negation and implication are therefore no longer part of the system, and the negation of a formula represents the NNF of its negation. Presentation conventions are shared with Figure 2.1.

The one-sided calculus has the advantage of being more concise and therefore simpler to implement. Therefore, it will become the basis for the *kernel* of the proof checker presented in Section 4.4. Figure 2.3 shows this sequent calculus.

Gentzen observed that a sequent calculus for intuitionistic logic, called *LJ*, results as a particular case of the (two-sided) classical calculus *LK* by imposing the simple restriction that the right-hand side of each sequent contain at most one conclusion: this is called the *intuitionistic restriction*. Due to this fundamental asymmetry, a one-sided calculus for intuitionistic logic—where left-hand side and right-hand side are confused—cannot be formulated. Provided that this proviso is threaded throughout all inference rules, Figure 2.1 can be used to describe *LJ* (a rewrite with a small amount of syntactic simplification is also feasible).

Finally, note that based on these rules of Figure 2.1, we can connect the stated semantics of sequents to their single-formula equivalent, also in sequent form:

$$\vdash (A_1 \wedge \cdots \wedge A_m) \supset (B_1 \vee \cdots \vee B_n)$$

In the one-sided calculus of Figure 2.3, the context is simpler and always disjunctive, and this is equivalent to:

$$\vdash \neg A_1, \ldots, \neg A_m, B_1, \ldots, B_n$$

A *derivation* in sequent calculus is a *proof tree* whose edges are (correct) applications of the inference rules of the calculus, and whose nodes are sequents. A *proof* corresponds to a complete derivation, whose root sequent at the bottom is the formula that is proved, and whose leaves are all vacuous and derived by the *axiom* rule. Two operational readings of a proof are possible. The *top-down* interpretation starts from the axiom and composes them into more complex sequents. The *bottom-up* interpretation starts from the sequent to be proved and decomposes it in smaller proof obligations until it arrives at instances of the axioms. Under this latter view, the cut rule corresponds to the application of a *lemma* inside the proof of a theorem: one premise provides a proof of the lemma (as the goal of the sequent); the other premise uses the lemma to further the proof of the theorem (as a new hypothesis). Cut elimination, in turn, corresponds to the *inlining* of lemmas: instead of proving them once, building them from scratch at each point where they are needed.

2.4.1 Example Consider the two right introduction rules for $\vee$ and $\exists$ from Figure 2.1 where the two separate rules for disjunction (one for each disjunct) are compacted into a single rule:

$$\frac{\Gamma \vdash A_i, \Delta}{\Gamma \vdash A_1 \vee A_2, \Delta} \ \vee_R$$

If one attempts to prove sequents by reading these rules from conclusion to premises, the rules need either additional information from some external source (e.g., an oracle providing the disjunct $i \in \{1, 2\}$ or the term t) or some implementation support for non-determinism (e.g., unification and backtracking search). Indeed, it is difficult to meaningfully use Gentzen's sequent calculus to directly support proof automation. Consider attempting to prove the following sequent:

$$\Gamma \vdash \exists x. \exists y. (p \ x \ y) \vee ((q \ x \ y) \vee (r \ x \ y))$$

Here, assume Γ contains, say, a hundred formulas. The search for a (cut-free) proof of this sequent confronts the need to choose from among 101 potentially applicable introduction rules. If we choose the right-side introduction rule, we will then be left with, again, 101 introduction rules to apply to the premise. Thus,

reducing this sequent to, say, $\Gamma \vdash (q\ t\ s)$ requires picking one path of choices in a space of 101^4 possible choice combinations.

2.5 Focusing

It is clear from the most cursory inspection that the unadorned sequent calculus exhibits prodigious levels of bureaucracy and nondeterminism: on the one hand, copious applications of the structural rules are required to constantly transform the sequents into forms suitable for the other two groups of rules; on the other, work can take place anywhere in the sequent at any time, resulting in an exponential number of equivalent interleavings. The resulting proofs are profoundly non-canonical and too unstructured to offer any kind of realistic support for automation. The practical question becomes, then, how to remedy this chaotic situation and thereby bring more order into the operation of the calculus.

A series of advances in proof theory—stemming from the study of proofs to describe the semantics of logic programming, notably the *uniform proofs* of Miller et al. (1991)—have shown that imposing certain reasonable restrictions on the proofs of the sequent calculus allow these proofs to be structured in alternating phases of two distinct types. These results crystallized, in the linear logic setting, in the form of the discipline of *focusing* and its associated *focused proof systems*, introduced by Andreoli (1992); in the practical plane, these strategies have been applied to describe computational aspects of theorem proving by Chaudhuri et al. (2008b). Further developments by Liang and Miller (2009) extended this discipline by obtaining focused sequent calculi for classical and intuitionistic logic. This will be our starting point. Again, for the time being we continue to focus on classical logic. Starting from the one-sided *LK*, its corresponding focused version is called *LKF*, and shown in Figure 2.4.

First, at the level of formulas, the notion of *polarity* is central to focused systems. Recall how, in the previous section, the inference rules for conjunction and disjunction admit two interchangeable presentations: additive (in Figure 2.1) and multiplicative (in Figure 2.2). Instead of choosing one conjunction and one disjunction, making both variations of each connective available in the proof system can lead to greater control over the inferences. This control will be achieved by defining two conjunctions and two disjunctions, and assigning to each one of admissible the rules for the connective, additive or multiplicative.

ASYNCHRONOUS INTRODUCTION RULES

$$\frac{}{\vdash \Gamma \Uparrow t^-, \Theta} \qquad \frac{\vdash \Gamma \Uparrow A, \Theta \quad \vdash \Gamma \Uparrow B, \Theta}{\vdash \Gamma \Uparrow A \wedge^- B, \Theta}$$

$$\frac{\vdash \Gamma \Uparrow A, B, \Theta}{\vdash \Gamma \Uparrow A \vee^- B, \Theta} \qquad \frac{\vdash \Gamma \Uparrow [y/x]B, \Theta}{\vdash \Gamma \Uparrow \forall x.B, \Theta} \ {\dagger}$$

$$\frac{\vdash \Gamma \Uparrow \Theta}{\vdash \Gamma \Uparrow f^-, \Theta}$$

SYNCHRONOUS INTRODUCTION RULES

$$\frac{}{\vdash \Gamma \Downarrow t^+} \qquad \frac{\vdash \Gamma \Downarrow B_1 \quad \vdash \Gamma \Downarrow B_2}{\vdash \Gamma \Downarrow B_1 \wedge^+ B_2}$$

$$\frac{\vdash \Gamma \Downarrow B_i}{\vdash \Gamma \Downarrow B_1 \vee^+ B_2} \ i \in \{1, 2\} \qquad \frac{\vdash \Gamma \Downarrow [t/x]B}{\vdash \Gamma \Downarrow \exists x.B}$$

IDENTITY RULES

$$\frac{}{\vdash \neg P_a, \Gamma \Downarrow P_a} \ init \qquad \frac{\vdash \Gamma \Uparrow B \quad \vdash \Gamma \Uparrow \neg B}{\vdash \Gamma \Uparrow \cdot} \ cut$$

STRUCTURAL RULES

$$\frac{\vdash \Gamma, C \Uparrow \Theta}{\vdash \Gamma \Uparrow C, \Theta} \ store \qquad \frac{\vdash P, \Gamma \Downarrow P}{\vdash P, \Gamma \Uparrow \cdot} \ decide \qquad \frac{\vdash \Gamma \Uparrow N}{\vdash \Gamma \Downarrow N} \ release$$

2.4 Figure   The *LKF* focused proof system for classical logic (Liang and Miller, 2009). Here, P is a positive formula; N is a negative formula; P_a is a positive literal; C is a positive formula or a negative literal; A and B are arbitrary formulas; and $\neg B$ is the negation of B, itself in negation normal form. The proviso marked as $\dagger$ is the usual eigenvariable restriction that y must not be free in the components of the premise (Γ, B, and Θ).

More generally, a focused proof system operates on *polarized formulas*, obtained from regular, unpolarized formulas by replacing each logical connective with a polarized version of it, positive or negative, and by assigning a polarity, again positive or negative, to each non-logical constant. In the one-sided focused sequent calculus for classical logic, the four connectives $\wedge$, $\vee$, t and f exist in both positive and negative versions, signified respectively by a $^+$ or $^-$ superscript. The universal quantifier $\forall$ is always negative, whereas the existential quantifier $\exists$ is always positive. The polarity of a non-atomic formula is that of its top-level connective, and the polarity of an atomic formula is assigned arbitrarily. Because formulas of the one-sided calculus are always in negation normal form, the scope of negation is always atomic, and negation is subsumed by polarity and corresponds to a simple polarity flip: if an atom is defined as positively polarized, its negation is defined as negative, and vice versa—the polarity of each atom is arbitrary, but all instances of an atom must share the chosen polarity. From now on, when we discuss formulas we will usually mean polarized formulas; the distinction between unpolarized and polarized formulas will be made clear when it is not clear from the context.

Second, at the level of inference rules, the same general classification in groups of rules remains, but is built upon a more basic distinction that divides sequents into two separate sequent types. Both these types divide their collection of formulas in two: a *storage zone* (abbreviated Γ, like the original, unfocused RHS) and a *workbench* (sometimes abbreviated Θ)—the two zones are separated by an *arrow sign*. Proofs will now be structured in groups of inference rules of the same kind constituting distinct *phases*:

1. **Up-arrow sequents** $\vdash \Gamma \Uparrow \Theta$ are related to the *asynchronous phase* (variously called negative, invertible, and up-arrow phase). Here, Γ is a multiset of formulas and Θ is a list of formulas. Each inference rule in this phase is invertible—i.e., its premises are true iff its conclusion is true, so that they can be moved at the end of a proof without loss of completeness. Moreover, these rules involve exclusively up-arrow sequents in both conclusion and premises. Owing to these properties, invertible rules can be applied to saturation indistinctly in any order. In fact, Θ is modeled as a list to enforce an order of evaluation in which asynchronous rules are always applied to the head of the list. This phase corresponds to *don't-care nondeterminism*.

2. **Down-arrow sequents** $\vdash \Gamma \Downarrow B$ are related to the *synchronous phase* (variously called positive, non-invertible, and down-arrow phase). Here, Γ is

a multiset of formulas and B is a single formula. Inference rules in this phase are not necessarily invertible, and some are indeed non-invertible. Throughout a synchronous phase, the system is *focused* on the single formula in the workbench and sequentially applies these non-invertible *choices* until no more such rules are applicable (at which point the phase ends). Because these choices are not reversible, we may need to *backtrack* and try other possibilities if they choices are not good. This phase corresponds to *don't-know nondeterminism*.

The three groups of inference rules remain with the following changes to their structure and organization:

1. Introduction rules are now subdivided into two groups depending on the phase in which they operate: asynchronous introduction rules operate on negative connectives, and synchronous introduction rules operate on positive connectives. For each phase, there is one introduction rule for each propositional connective of matching polarity—except for f^+—and one rule for the corresponding quantifier. Because each polarized connective is introduced by at most one rule, inference rule names are simply those of the connectives they introduce.

2. Initial rules do not experiment substantial modifications. The interesting change occurs in the atomic *init* rule, which replaces the *axiom* rule. The scope of this rule is now limited to a positive literal as a focused formula whose negated complement is contained in the storage zone. Indeed, Gentzen proved that all instances of *init* can be eliminated except for those that operate on atomic formulas, which is the criterion the focused calculus adopts. Whereas the *init* rule is part of the positive phase, the cut rule operates in the negative phase.

3. Structural rules undergo the most significant changes with respect to the unfocused calculus. First, the standard data structures eliminate the need for an exchange rule. More importantly, weakening and contraction are now integrated in other rules: contraction is used in the new *decide* rule, and weakening in the *init* rule, above. The structural rules streamline the flux of formulas between the zones of the sequent and arbitrate the phase transitions. Because they are conceived to enable proof search, they are best interpreted by a bottom-up reading, unlike the top-down we have

followed until now. First, in the asynchronous phase, when the head of the workbench list is positive or a literal—and therefore no asynchronous rules apply to it—the *store* rule moves it into the storage zone. Second, when all formulas of the asynchronous workbench have been processed, the *decide* rule selects a positive formula from the storage zone as focus of the synchronous phase that begins. And third, when a negative formulas is encountered in the positive phase—and therefore no synchronous rules apply to it—the *release* rule removes the focus from it and starts the next asynchronous phase. (Note that the cut rule presents an alternative to the decide rule that prolongs the asynchronous phase instead of switching to the asynchronous phase.)

The resulting *focused proofs* are therefore structured as an alternation of negative and positive phases. The combination of a positive phase followed by a negative phase is called a *bipole*. This aggregation of rules greatly decreases the amount of nondeterminism in the calculus and organizes proofs into larger coherent units. Under the bottom-up reading—representing the aspect of proof search which focusing is designed to automate—once the decide rule focuses on a formula, this formula (and the sequence of synchronous choices made during the positive phase) guides the evolution of the proof up to the boundaries between the branching negative phases that follow the positive phase and the next set of positive phases. For this reason, bipoles are described as *synthetic inference rules*.

It remains to prove the connection between the original classical calculus *LK* and its focused version *LKF*. An important result that will be referenced in subsequent chapters is the following:

2.5.1 Theorem   Let B be a formula of classical logic. B is a theorem of classical logic iff the *entry point sequent* $\vdash \cdot \Uparrow \hat{B}$ is provable in *LKF*, where $\cdot$ is the empty storage and $\hat{B}$ is an arbitrary polarization of B.

Proof. Proved in Liang and Miller (2009).   □

In other words, *LKF* is sound and complete w.r.t. classical logic. Moreover, an *LKF* proof reveals an underlying *LK* proof by removing all polarities and up- and down-arrow sequent annotations—and possibly adapting structural rules to the presentation of choice. In addition, *LKF* enjoys the cut-elimination property. Finally, it must be noted that although polarization does not affect provability, it influences the shapes and sizes of the proofs that can be found for a given

theorem. Examples of the choice of polarities and their consequences on proof size will be given in Chapter 3. In particular, compare Sections 3.3 and 3.4 and the quantitative analyses in Section 8.5.

2.5.2   Example   Returning now to Example 2.4.1, consider the synchronous introduction rules for $\vee^+$ and $\exists$ as refined by Figure 2.4. Focused proof systems address the kinds of combinatorial explosions witnessed in the previous example by organizing introduction rules into two distinct phases. Two rules of interest are synchronous, i.e., they operate on a single formula marked as under focus.

As a result, it is easy to see that—after introducing the focusing decorations in the previously considered sequents—reducing proving the original sequent $\Gamma \vdash \exists x \exists y[(p\ x\ y)\vee((q\ x\ y)\vee(r\ x\ y))] \Downarrow$ to its reduced form $\Gamma \vdash (q\ t\ s) \Downarrow$ involves only those choices related to the formula marked for focus: no interleaving of other choices needs to be considered.

2.6   Soundness and completeness

Theorem 2.5.1 in the previous section states the properties of *soundness* and *completeness* of the system *LKF* for classical logic. Because these concepts recur in presentations of subsequent systems and frameworks, we collect here some essential remarks. A proof system is sound with respect to a logic if every statement it proves is a theorem of the logic. Conversely, a proof system is complete with respect to a logic if for every theorem of the logic there exist proofs of its statement in the proof system.

In sequent calculus, part of the interest of cut elimination lies on its connection to the related property of *consistency*. Consider again the system *LK* in Figure 2.1. Except for the cut rule, all inference rules satisfy the *subformula property*: that is, every formula in the conclusion sequent is a subformula of one of the premise sequents. If we can eliminate cut, we are left with a system that globally satisfies the subformula property. From this, a proof of consistency is simple: if it were possible to obtain proof of a contradiction (i.e., prove false), this contradiction should be found as part of a premise for some inference rule in the calculus; because none of the rules allow for this propagation of false from conclusion to premise, the calculus is consistent. (Furthermore, because it globally satisfies the subformula property, proof search can be easily automated.)

By similar arguments, *LJ* is found to be sound and complete w.r.t. intuitionistic logic (Liang and Miller, 2009). Starting in Chapter 3, we will study methods to

restrict completeness in such a way that soundness is preserved by construction. The motivation will be to start from a system that is, say, "complete enough" and sculpt out of it another system, possibly less complete, but also "complete enough" for a certain domain of interest, in the hope that this leads to increased expressiveness and efficiency.

2.7 Notes

The present work develops in the rich soils of computational logic in general and structural proof theory in particular. As general overviews of the field, a number of monographs serve as good references, expounding the bases of structural proof theory and covering the needed logical preliminaries in various styles of presentation: Takeuti (1987); Buss (1998); Troelstra and Schwichtenberg (2000); Negri and von Plato (2001); von Plato (2013).

While we shall cover both classical and intuitionistic logic, a wealth of other logics exist. *Linear logic*, the "logic of resources" developed by Girard (1987), is especially relevant. Concepts like the distinction between additive and multiplicative inference rules (and connectives)—already present in relevant logics (Anderson and Belnap, 1975; Read, 1988)—, are ubiquitous in linear logic. In fact, the notions of polarity and focused proof developed by Andreoli (1992) extend cleanly to classical and intuitionistic settings, as well as to fixed points (Baelde and Miller, 2007; Liang and Miller, 2009) and serve to give a proof theoretical reading to the complementary discipline of model checking (Heath and Miller, 2017). Linear logic is one of the primary representatives of the family of *substructural logics*, so called because they limit the application of structural rules—and at least along this axis constitute a weakening of, say, classical logic. This logic reoccurs during the introduction of fixed points in Chapter 9.

The original sequent calculi by Gentzen use the homonymous structures as their fundamental building block. However, various extensions and refinements to the paradigm have been proposed, of which focusing is one of the most important. One way to obtain more general systems is to extend the data structures: calculi based on *hypersequents* (Avron, 1991, 1996) do this by replacing simple sequents with a list (or a multiset) of sequents, usually interpreted disjunctively (like the formulas in a one-sided sequent); the motivation is to study general frameworks in which to easily model many families of interesting logics. Another extension to the basic data structures of the calculus employs *nested sequents* (Brünnler,

2010), whose components can be not only formulas, but also other sequents, thus turning the latter into a recursive data type. This last idea is related to *deep inference* (Guglielmi, 2007, 2015), a design methodology for proof systems which, applied to the sequent calculus, allows inference rules to be applied anywhere in a sequent instead of being limited to the top-level connective of a certain formula. Structural generalizations also extend to the arena of focused systems, where *multi-focusing* (Chaudhuri et al., 2008a) allows the decide rule to operate on a set of formulas, in this case to further increase the canonicity of sequent calculus proofs that is the hallmark of focused systems.

The classic treatise on the nature of proof-as-message is the book by Lakatos (1976). The proof-as-certificate counterpart is covered by MacKenzie (2001), which traces the development of mechanized proofs with which we are directly concerned. More recent overviews include Asperti (2012) and, under the prism of Foundational Proof Certificates, Miller (2014). Although we are primarily concerned with the certifying nature of proofs, it is arguable that work in the latter chapters—Part III, and even sections of Part II—addresses the proof-as-message half by a flexible linkage between formally defined certificates and the messages conveyed by them, namely concise, readable descriptions of proofs made syntactically—and semantically—precise while being usable by mathematicians.

The rule of cut serves very different purposes in proofs created for (or by) machines or mathematicians. In addition to the important theoretical results derived from it—bearing witness to certain desirable characteristics of a logic, like consistency—the existence of cut-free proofs is instrumental to the automation of proof search. In turn, for the mathematician, the ability to organize proofs using lemmas is an absolute conceptual necessity. Beyond cognitive and stylistic considerations, cuts are indispensable—also for machines—to manage the complexity and size of proofs, as the combinatorial explosion that results from inlining every auxiliary lemma at every point of use quickly becomes intractable (D'Agostino and Mondadori, 1994).

3 Foundational Proof Certificates

3.1 Proof as trusted communication

Chapter 2 opens by examining the mathematical concept of proof and briefly charting its formal study. As we concentrate on the more solidified and structured states of the malleable substance—and, through rigorous efforts, further seek to reduce entropy and obtain a flawless crystallization of mathematical truth—, the dual functions of proof-as-message and proof-as-certificate seemingly begin to coalesce. In fact, inasmuch as computer programs can not only build proofs, but also write and read them (and check them), those external representations of machine proofs—transmitted between computer processes—must ideally function both as messages and certificates. When the recipient of some proof evidence is a computer program, that evidence should be enough to enable the program to reconstruct the proof object which that evidence purports to represent; otherwise, it has to be discarded. Of course, there are many precisions to make, and an incomplete proof, even an incorrect one, is not always without value. Nevertheless, ultimately a complete proof object is needed to establish trust, and the proof-as-certificate aspect guarantees and safeguards the utility of the proof-as-message.

The concept of proof as an object—materialized as some kind of document—that attests to the truth of a statement made about a certain, formally defined theory finds ample practical support in the ever-growing array of software tools that facilitate the production and verification of proofs: automated theorem provers (Alt-Ergo, Vampire, Z3), proof assistants (Coq, Isabelle, Abella), model checkers (NuSMV, PRISM, Bedwyr), programming languages with sophisticated type systems (Idris, F*), etc. These tools are generally not designed with communication in mind, and as their number and sophistication increases, so does fragmentation. Yet different tools have different strengths, and it is natural to

wonder how to combine them—and their proofs—gracefully. The issue is more
than a theoretical curiosity, as complex developments may involve proofs which
are more easily obtained through a mix of (compatible) formalisms. The ability
to understand and admit proofs irrespective of their provenance is a prerequisite
for this scenario, akin to the interoperation of computer programs.

One may draw inspiration from the progressive application of formal methods
in the related area of programming languages. The study and development of
programming languages have been aided by the use of at least two important the-
oretical frameworks: on the one hand, *context-free grammars*, or CFG (Hopcroft
et al., 2006), are used to define the structure of programs; on the other, *structural
operational semantics*, or SOS (Plotkin, 1981, 2004), are used to define the evalua-
tion and behavior of programming languages. Both of these frameworks make it
possible to define the syntax and semantics of a programming language in a way
that is independent of a particular parser and particular compiler. Specifications
in these frameworks are both mathematically rigorous and easily given prototype
implementations using the logic programming paradigm (Borras et al., 1988;
Hannan, 1993; Shieber et al., 1995; Miller, 2009). These techniques scale to the
definition of practical programming languages, as demonstrated by the formal
specification of ML (Milner et al., 1990).

Similarly, work on automated and interactive reasoning systems can benefit
from the introduction of frameworks that are capable of defining the meaning
of proof descriptions that are output by proving tools, and representing these
descriptions in a shared language—that of logic. Such formal semantics of proof
languages make it possible to establish a *separation of concerns* between the produc-
tion of proofs and the checking of proofs. On the one hand, production is carried
out by *theorem provers*: the various families of tools (such as listed above) capable
of reasoning about formal specifications. These are complex, evolving pieces of
software; they are potentially difficult to prove correct, and thus vulnerable to
programming errors which can endanger their logical soundness. On the other
hand, *proof checkers* could be small and persistent, as well as easy to trust and
prove correct. In such a setting, the provenance of a proof should not be critical
for trusting it—subject to its successful checking by a trusted tool.

The key, then, is to define a logical framework where the syntax and, criti-
cally, the semantics of proofs can be defined in a clean, declarative fashion that
is both universal and permanent. That is, the framework should be able to
represent a broad spectrum of proof structures, such as resolution refutations,

tableaux, (un)satisfiability proofs, superposition, etc. All these kinds of proof evidence—given a definition of their semantics— are modeled through a unified representation of proofs, which much satisfy certain properties.

3.1.1   **Definition**   The term *proof certificate* denotes a document that should elaborate into a full formal proof by means of a *proof checker*. Any general-purpose framework implementing these concepts should satisfy the following four requirements (Miller, 2014), which we shall call *Miller's desiderata*, referring to them by number:

1. A simple checker can, in principle, check if a proof certificate denotes a proof.

2. The format for proof certificates must support a wide range of proof options.

3. A proof certificate is intended to denote a proof in the sense of structural proof theory.

4. A proof certificate can simply leave out details of the intended proof.

Harnessing recent advances in structural proof theory, *Foundational Proof Certificates* (FPCs) have been proposed as a general framework for the expression of proof evidence (Miller, 2011; Chihani et al., 2013, 2016b). In this framework, focused sequent calculi (for which see Section 2.5) act as foundations of a unifying proof system, which—as its parallels in the programming language world—is easily implemented in proof checking kernels in a logic programming language, as we will see in Section 4.4. In this context, an FPC is a machine-readable document which expresses a proof in terms of a series of *synthetic inference rules*. Those inference rules, along with their logical interpretation, are given by a certificate definition, which operates as nexus between the two sides of a checker: (a) for the *kernel*, it is a small logic program loaded and run by the kernel, used to guide proof search; and (b) for the *client*, a small domain-specific language in which proofs (i.e., proof certificates) can be written. The payload that a client must provide to a proof checker consists of both a certificate and its definition, although FPC definitions are modular and it is in their vocation to be reusable.

The rest of the chapter is organized as follows. Section 3.2 introduces the extensions to the focused sequent calculus that form the theoretical basis for the framework. Section 3.3 presents the first of four simple FPC definitions which are used as recurring examples, here a simple decision procedure for propositional

logic. Section 3.4 continues with the second of these examples by by defining precise guidance information obtained from an oracle in the certificate. Section 3.5 complements previous certificate formats with a more flexible constraint on the size of proofs expressed as their depth in number of bipoles. Section 3.6 completes the series of examples with a certificate format for proofs by binary resolution, closer to the proofs produced by automated theorem proving tools. Section 3.7 discusses the relationships between the various parts and realizations of the framework, which will find concrete expression in Chapter 4. Section 3.8 concludes the chapter.

3.2   Augmented sequent calculus

The standard sequent calculus of Section 2.4, while undeniably interesting from a theoretical perspective, is not well suited for automation because it lacks structure. One modern criterion for what constitutes a well-behaved proof system is whether the logic admits a focused version, as illustrated in Section 2.5. Focused proofs substantially reduce the amount of nondeterminism by structuring the proof in an alternation of asynchronous and synchronous phases. Within each asynchronous phase, only invertible rules are used and their ordering is irrelevant; within each synchronous phase, the sequence of inference rules is fully determined. Although focused proofs exhibit a much larger degree of canonicity, the sources of *meaningful nondeterminism* remain: picking a formula to focus on in the decide rule, devising a lemma to cut into the proof, choosing a disjunct or an existential witness to instantiate the rules of the synchronous phase, etc. Focused phases are, in a sense, *synthetic inference rules*, but lack flexibility.

The solution adopted by the FPC framework is to *augment* the focused sequent calculus to allow fine control to the point of making it programmable. Consider Figure 3.1, which presents the augmented calculus LKF^a. Three main categories of changes are introduced:

1. Starting from a standard focused system (here, the *LKF* of Figure 2.4), every inference rules has all its premises and its conclusion enriched with *certificate terms*, denoted Ξ.

2. Moreover, an additional premise is added to every rule—except that for negative true, t^-. The new premise represents a predicate that relates the certificate terms in the conclusion and the premises, as well as every piece of

additional information required by the rule (disjunctive choices, existential witnesses, etc.). When fresh eigenvariables are involved (i.e., treating the universal quantifier) the certificate terms on the premises are parameterized over the new variable.

3. Finally, the storage zone Γ is transformed into a multiset of *indexed formulas*, and inference rules storing and pulling formulas from this zone are modified to reflect the new data structure.

The operational reading of the augmented calculus parallels bottom-up proof reconstruction. Under this discipline, a concrete certificate term is expected in the conclusion. The new (client-supplied) relational premises—which will be presently characterized as *clerks* and *experts*—would then take the conclusion certificate as "input" and use it to constrain all other elements as the "outputs" at the premises. If the output certificate is related to continuation certificates for the premises (and ancillary information: disjuncts, etc.), each possible combination of values offers an opportunity to continue, and possibly finish, the proof. In most practical FPC definitions, the relation behaves like a partial function from conclusion certificate to premise certificates. By contrast, some forms of ancillary information can be not only span several different values, but also be completely unconstrained and left to the checker to reconstruct—logic variables can be used to reflect these degrees of freedom. Similarly, clerk and expert relational premises refer to formulas in storage exclusively by index; the model does not enforce a functional mapping, so that an index may select an arbitrary number of formulas from storage.

3.2.1 Example Let us revisit the interesting cases of the introduction rules in Example 2.4.1—albeit in their focused versions. Consider the introduction rule for the positive disjunction. In bottom-up proof search—once the inference rule is augmented with an expert—, one of the charges of an FPC definition is to have the expert dictate the disjunctive choice. Based on a certificate term and its own defining clauses, the expert may stipulate that, at a given point in the derivation, the left or the right disjunct should be used to proceed with the proof. It may also decree that no choice is acceptable—and in consequence a derivation may not be obtained, even if one exists—or that either choice may be attempted (hence, an implementation should choose one of the disjuncts and continue the proof attempt, and return to try the second disjunct if the first fails).

ASYNCHRONOUS INTRODUCTION RULES

$$\frac{}{\Xi_0 \vdash \Gamma \Uparrow t^-, \Theta}$$

$$\frac{\Xi_1 \vdash \Gamma \Uparrow A, \Theta \quad \Xi_2 \vdash \Gamma \Uparrow B, \Theta \quad \wedge_c(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0 \vdash \Gamma \Uparrow A \wedge^- B, \Theta}$$

$$\frac{\Xi_1 \vdash \Gamma \Uparrow A, B, \Theta \quad \vee_c(\Xi_0, \Xi_1)}{\Xi_0 \vdash \Gamma \Uparrow A \vee^- B, \Theta}$$

$$\frac{(\Xi_1\, y) \vdash \Gamma \Uparrow (B\, y), \Theta \quad \forall_c(\Xi_0, \Xi_1)}{\Xi_0 \vdash \Gamma \Uparrow \forall x.B, \Theta} \dagger$$

$$\frac{\Xi_1 \vdash \Gamma \Uparrow \Theta \quad f_c(\Xi_0, \Xi_1)}{\Xi_0 \vdash \Gamma \Uparrow f^-, \Theta}$$

SYNCHRONOUS INTRODUCTION RULES

$$\frac{t_e(\Xi_0)}{\Xi_0 \vdash \Gamma \Downarrow t^+}$$

$$\frac{\Xi_1 \vdash \Gamma \Downarrow B_1 \quad \Xi_2 \vdash \Gamma \Downarrow B_2 \quad \wedge_e(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0 \vdash \Gamma \Downarrow B_1 \wedge^+ B_2}$$

$$\frac{\Xi_1 \vdash \Gamma \Downarrow B_i \quad \vee_e(\Xi_0, \Xi_1, i)}{\Xi_0 \vdash \Gamma \Downarrow B_1 \vee^+ B_2} \ i \in \{1, 2\}$$

$$\frac{\Xi_1 \vdash \Gamma \Downarrow (B\, t) \quad \exists_e(\Xi_0, \Xi_1, t)}{\Xi_0 \vdash \Gamma \Downarrow \exists B}$$

IDENTITY RULES

$$\frac{\langle l, \neg P_a \rangle \in \Gamma \quad init_e(\Xi_0, l)}{\Xi_0 \vdash \Gamma \Downarrow P_a} \ init$$

$$\frac{\Xi_1 \vdash \Gamma \Uparrow B \quad \Xi_2 \vdash \Gamma \Uparrow \neg B \quad cut_e(\Xi_0, \Xi_1, \Xi_2, B)}{\Xi_0 \vdash \Gamma \Uparrow \cdot} \ cut$$

STRUCTURAL RULES

$$\frac{\Xi_1 \vdash \Gamma, \langle l, C \rangle \Uparrow \Theta \quad store_c(\Xi_0, \Xi_1, l)}{\Xi_0 \vdash \Gamma \Uparrow C, \Theta} \ store$$

$$\frac{\Xi_1 \vdash \Gamma \Downarrow P \quad \langle l, P \rangle \in \Gamma \quad decide_e(\Xi_0, \Xi_1, l)}{\Xi_0 \vdash \Gamma \Uparrow \cdot} \ decide$$

$$\frac{\Xi_1 \vdash \Gamma \Uparrow N \quad release_e(\Xi_0, \Xi_1)}{\Xi_0 \vdash \Gamma \Downarrow N} \ release$$

3.1 Figure   The augmented *LKF^a* focused proof system for classical logic (Chihani et al., 2013). Presentation conventions are shared with Figure 2.4.

The treatment of the existential quantifier generalizes that of the positive disjunction in that the number of possible terms may be zero (if the type is uninhabited), finite or infinite (in inductively defined types like the natural numbers): more complex selection strategies are possible, but remain variations on those available in the (binary) disjunctive case. In a focused proof system, the decide rule is the primary source of proof structure. The choice of formula under focus is afforded the same flexibility with the peculiarity that selections are determined by the *indexing scheme* imposed by the certificate terms and the store clerks. A decision depends on both the indexes allowed by the decide expert and the sets of formulas filed under those indexes. Again in Example 2.4.1, if all formulas share a single index, the number of choices remains unchanged; if each formula is stored under a unique index, a certificate term can record the sequence of choices that lead to a single, directed path to success.

In brief, the augmentation of *LKF* by certificates, indexes, and clerk and expert relations modifies the original system merely by restricting when inference rules can be applied: it affects completeness and leaves soundness untouched.

3.2.2 **Theorem**   The system LKF^a is sound w.r.t. classical logic (Chihani et al., 2016b, Section 5).

Proof. The *LKF* system can be recovered from LKF^a by removing all the augmentations (marked in Figure 3.1), and therefore every proof of LKF^a is also a proof of *LKF*. The result follows from Theorem 2.5.1.   □

Theorem 3.2.2 implies that the soundness of the system cannot be compromised by the client. Note that the augmentations of LKF^a are completely generic and can be assigned arbitrary meanings by the client: this is the function of FPC definitions. By furnishing declarations and definitions (i.e., syntax and semantics) to define a set of augmentations, the sequent calculus gains support for *programmable proof reconstruction*. Effectively, such a description defines a family of certification strategies. There are five groups of elements that constitute an FPC definition, each described in the following paragraphs:

1. **Polarization.** A polarization strategy determines how to translate (standard) unpolarized formulas into polarized versions of them—the other direction is direct by polarity erasure. As noted in Section 2.5, the choice of polarities by itself does not affect provability, but its interactions with the other members of an FPC definition may. Despite this seeming leniency,

the role of polarities should not be underestimated: they have a strong impact on the proofs that can be found for a given theorem, and can represent the difference between brute force search (exponential in complexity) and purposeful navigation, guided by a certificate along a predetermined set of nondeterministic choices. For illustration, compare the certificate formats and corresponding examples in Sections 3.3 and 3.4.

2. **Certificate terms.** Conceptually, certificate terms represent the *state of a proof*, or the region of a proof in which we find ourselves at a given point in the derivation. State information can be arbitrarily complex and, as the proof evolves, so do the certificates, and the information contained in them can be used (by clerks and experts) to steer the derivation towards success.

3. **Indexes.** Index terms control two important aspects of guided proof reconstruction: *naming* and *storage*. As synchronous formulas are moved to the storage zone Γ, they are annotated with a data structure contained in an index term, supplied in the *store* rule. Like certificate terms, they may contain arbitrary information, and can later be called upon to selectively retrieve a subset of candidate formulas for selection by the *decide* rule (to treat during the positive phase inaugurated by the rule) or by the *init* rule (to attempt to finish a branch of the derivation). Lookup can be as loose or as tight as the designer chooses, offering great control over backtracking points, classification and selection of formulas to make the proof progress, etc. A good indexing scheme is critical to performant proof checking.

4. **Clerks.** These predicates, denoted by the $_c$ subscript in Figure 3.1, define the control semantics of asynchronous rules. While not explicitly responsible for the decisions that are characteristic of synchronous rules, they can nonetheless perform bookkeeping and record information that will eventually enable their synchronous counterparts to make their decisions when they are executed. These can be thought of as ordinary program clauses, making full use of the power of logic programming. Nontermination is allowed by the framework, as it concerns itself only with the soundness of successful derivations—for which termination of all instances of clerks and experts is necessary.

5. **Experts.** The synchronous counterparts of clerks, signified by the $_e$ subscript, are—like their asynchronous duals—arbitrary program clauses, but

in addition to possibly recording information while processing certificate terms they will be required to make the decisions demanded by their phase and supply information about those decitions to the checker. How much or how little they commit to one or several possible courses of action is not dictated by the framework and remains purely a design issue. For example, $\vee_e$ must select one of the disjuncts, B_1 or B_2, to continue the derivation. A possible, nondeterministic course of action is to try to use B_1 first, backtracking to try B_2 if a complete derivation based on B_1 cannot be found. Similarly, $\exists_e$ may provide a set of closed terms as witnesses or let an arbitrary term be instantiated over the course of the proof.

The phase of the inference rule for a given logical connective is immediate from the polarity of the connective. Structural rules are more nuanced: *store* occurs during the asynchronous phase and is assigned a clerk, although it is charged with the critical operation of filing formulas into storage, in the process assigning them indexes. In turn, *decide* arbitrates the transition from asynchronous to synchronous phase by using those very indexes, and its complement *release* mediates (albeit trivially) the other phase transition, from synchronous to asynchronous. Both these transitions can be seen as operating "between two worlds," and are here both declared as experts. The last group of identity rules is supervised by experts. *init* is a standard expert of the synchronous phase: it operates on a positive literal and involves selecting a stored complementary literal. On the other hand, *cut* acts as an alternative to *decide* at the boundary of an asynchronous phase, and instead of ending the phase, it prolongs it, but there is no doubt that its duties and position correspond to those of an expert.

In the next few sections, we present a number of FPC definitions that illustrate typical uses of the framework and will reoccur in subsequent chapters. The definitions are given in executable λProlog code. They are direct, declarative transcriptions of the mathematical relations they encode, comprising certificate and index term constructors, and clauses for clerks and experts—polarization conventions are implicitly defined from those. Only standard logic programming features are used; argument order is preserved from Figure 3.1. We refer to Chapter 4, specifically Section 4.4 for a study of the implementation of LKF^a as a proof checking *kernel* and its interface with client-defined clerks and experts, as given by the kernel's API.

3.3   Running example: CNF decision procedure

An extreme minimalist use of the FPC framework is studied as our first example. Consider the propositional fragment of classical logic, where all logical constants are negatively polarized. In this situation, there is no opportunity to offer guidance during the synchronous phase, and our only recourse is to exhaustive search. While completely blind and exponentially inefficient, it is clearly a sound, if empty, brute force "strategy," a fact that can be clearly represented with a simple FPC definition. Each of its five components can be explained as follows:

1. **Polarization** is purely negative across all allowed logical connectives (i.e., no quantifiers). By convention, formulas are in conjunctive normal form, and negations translate into atoms of complementary polarities, e.g., negated atoms are given negative polarity; non-negated atoms are given positive polarity. This is the only allowed use of positive polarity.

2. **Certificate terms** carry no information. A singleton, nullary constructor, `cnf`, leaves no option but to propagate it from conclusion to each premise, unchanged.

3. **Indexes** also carry no information. A singleton, nullary constructor, `idx`, is used to file all formulas in the *store* rule. Conversely, the *decide* rule can only use this unique index to specify which formula(s) to choose. Hence, the storage zone acts as a simple bucket of formulas; it contains no information to help us discriminate the right formula to focus on. Each instance of the *decide* rule attempts to find a proof for every formula currently stored in Γ.

4. **Clerks** are declared for the negative connectives to enable proof search to proceed through them and store positive formulas and negative literals, i.e., atoms, without distinction.

5. **Experts** are declared only for phase transitions and for the *init* rule, to allow an atom to be matched with its stored complement and close a branch of the derivation. It can be seen that, by design, the other inference rules cannot occur, and therefore their definition would have no effect in proof search.

The FPC definition is presented in full in Figure 3.2. This is, in fact, a decision procedure for classical propositional logic.

```
% Signature
type lit                 index.
type cnf                  cert.

% Implementation of clerks and experts
andC         cnf cnf cnf.
orC            cnf cnf.
falseC         cnf cnf.
releaseE       cnf cnf.
initialE          cnf lit.
decideE        cnf cnf lit.
storeC         cnf cnf lit.
```

3.2 **Figure**   The CNF decision procedure FPC (Chihani et al., 2016b, Section 7.1).

3.3.1 **Example**   With patience, the theoremhood of small propositional formulas can now be checked by using cnf with the entry point sequent. Let F be a formula of *LK*, and F^- be its negative polarization. Then F is a theorem of *LK* iff cnf ⊢ · ⇑ F^- is provable.

3.4 Running example: oracle strings

Still in the decidable setting of classical propositional logic, the polar opposite of the exhaustive exploration of the previous section—our sole recourse given the lack of meaningful decisions—is the concrete expression of those decisions and their representation of one instance of a successful search as the certificate term. Thus, the tree of decisions effectively constitutes an *oracle*. The shift finds reflection in the choice of polarities, which in turn affect the proofs that can be found—and, fundamentally, open an avenue of feeding the oracle via a proof certificate. We next define the components of the FPC definition, reproduced in Figure 3.3.

1. **Polarization** is purely positive across the propositional connectives; negative polarity is allowed only with atomic scope.

2. **Certificate terms** carry as information a tree representing an oracle of decisions represented by the kind oracle, with constructors for branching conjunction (c, with a continuation for each branch) and disjunction (l and r, respectively instructing to pick the left or the right disjunct, with a continuation for the choice), as well as a branch terminator (emp).

```
% Signature
kind oracle    type.
type emp       oracle.                          % empty
type l, r      oracle -> oracle.                % left, right
type c         oracle -> oracle -> oracle. % conjunction

type start, consume, restart    oracle -> cert.
type root, lit                   index.

% Implementation of clerks and experts
decideE  (start Oracle) (consume Oracle) root.
storeC   (start Oracle) (start Oracle) root.
decideE  (restart Oracle) (consume Oracle) root.
storeC   (restart Oracle) (restart Oracle) lit.
initialE (consume emp) lit.
trueE    (consume emp).
andE     (consume (c OracleL OracleR))
         (consume OracleL) (consume OracleR).
orE      (consume (l Oracle)) (consume Oracle) left.
orE      (consume (r Oracle)) (consume Oracle) right.
releaseE (consume Oracle) (restart Oracle).
```

3.3 Figure The oracle string FPC (Chihani et al., 2016b, Section 7.2).

Oracle information is wrapped in three different constructors: the principal consumer of decisions, consume; as well as start and restart, used to simulate the homonymous rules of the purely positive fragment of *LK* in the focused setting of full *LKF* via its structural rules.

3. **Indexes** carry little information: as in the previous section, lit is used to store negative literals for further discharge by the initial rule. A second index, root, marks the formula on which the (re)start rule proceeds to the regular checking (i.e., consumption) phase.

4. **Clerks** are defined exclusively for the store rule: at the outset, to store the positive theorem candidate as root; after a release that brings the state to await a restart, to store the negative literals that caused the release of focus as lit.

5. **Experts** are defined for each synchronous propositional introduction rule as consuming the oracle corresponding to that exact connective, i.e., disjunctive choices, conjunctive branchings, as well as closing branches by looking up literals under lit or treating true; all this occurs in the consume phase.

In addition, the decide rule implements the (re)start by focusing on the formula stored under `root`, and the release of focus (into `restart`) when a negative literal is encountered.

It is instructive to compare the purely negative proofs of Section 3.3 with the purely positive proofs of this section. By polarizing a classical formula with negative or positive bias, we obtain different proofs of the same (unpolarized) theorems, though these proofs have vastly differing structures and computational behaviors.

3.5 Running example: decide depth

Another useful FPC definition provides a simple restriction on the proofs it allows by placing a certain bound on their size and admitting only "small" proofs which satisfy that bound. While of limited use by itself, it is representative of a common pattern of certificates as resources. Let us examine its components group by group:

1. **Polarization** does not impose any restrictions on what connectives can be used: this FPC definition concerns itself solely with a measure of the size of an arbitrary proof.

2. **Certificate terms** are limited to a single constructor, `dd`, containing a single piece of information: the maximum allowed decide depth from the present point in the proof (represented by a natural number, here of type `nat` as the standard inductive type).

3. **Indexes** carry no information: as in the previous example, a singleton, nullary constructor (called `indx` in this case) is used for all formulas in the storage zone.

4. **Clerks** are defined for each asynchronous inference rule. Each clerk propagates the decide depth bound unchanged from conclusion to premises. When treating the universal quantifier, the abstraction over the continuation certificate is vacuous, i.e., the eigenvariable plays no role in the new certificate. Storage of formulas uses the unique index at our disposal, `indx`.

5. **Experts** are defined for each synchronous inference rule—except cut. Like their clerk duals, they propagate the decide depth bound from conclusion

```
% Signature
type indx    index.
type dd      nat -> cert.

% Implementation of clerks and experts
andC        (dd D) (dd D) (dd D).
andE        (dd D) (dd D) (dd D).
falseC      (dd D) (dd D).
releaseE    (dd D) (dd D).
orC         (dd D) (dd D).
allC        (dd D) (x\ dd D).
orE         (dd D) (dd D) Choice.
someE       (dd D) (dd D) T.
storeC      (dd D) (dd D) indx.
initialE    (dd D) indx.
trueE       (dd D).
decideE     (dd (succ D)) (dd D) indx.
```

3.4 Figure   The decide depth FPC (Blanco et al., 2017a).

to premises and select the singleton index indx whenever an index term
is solicited by the interface. There are two points of importance. First,
the decide depth bound is *decremented* on the decide rule while the cur-
rent allowance permits it (i.e., is greater than zero). Second, disjunctive
choices and existential witnesses are completely unconstrained by the FPC
definition, and fresh logic variables are returned to the kernel.

Figure 3.4 presents the FPC definition just described. An inductive definition
of natural numbers with zero and successor constructors (the latter written as
succ in the figure) is assumed present. These FPCs have fewer constraints and
in fact encompass the proofs covered by the definitions in Sections 3.3 and 3.4.
Decide depth bounds are less useful for proof search in isolation, but offer a useful
way to express the size of focused proofs in terms of their essential high-level
components—bipoles.

3.6 Running example: binary resolution

It is instructive to see how the FPC framework scales up to the core of a standard
proof technique. To this end we turn to *resolution refutations*. Suppose we want
to prove a formula of the form $\neg C_1 \vee \ldots \vee \neg C_n$. This is equivalent to refuting
the negation of the formula, i.e., $C_1 \wedge \ldots \wedge C_n$—as usual, assumed in conjunctive

normal form—where each clause C_i a disjunction of literals closed by universal
quantifiers. The key idea of the technique lies in the binary resolution rule (here,
subscripted as and bs are arbitrary literals).

$$\frac{a_1 \vee \cdots \vee a_{i-1} \vee P_a \vee a_{i+1} \vee \cdots \vee a_m \quad b_1 \vee \cdots \vee b_{j-1} \vee \neg P_a \vee b_{j+1} \vee \cdots \vee b_n}{a_1 \vee \cdots \vee a_{i-1} \vee a_{i+1} \vee \cdots \vee a_m \vee b_1 \vee \cdots \vee b_{j-1} \vee b_{j+1} \vee \cdots \vee b_n}$$

A proof by binary resolution is structured as a sequence of applications of
this rule. Each of these steps proceeds by attempting to apply the resolution rule
to a pair of clauses to generate a new clause, until the empty clause f is reached,
at which point the refutation succeeds. By assigning names to the clauses that
compose the formula to be proved, as well as to the new clauses that result from
applications of the resolution rule, it becomes possible to represent compactly a
proof by resolution by a list of triples, each denoting the two premises and the
conclusion of each application of the rule, respectively. A certificate representing
a proof by resolution must encode this information. Assuming clause names
are natural numbers assigned incrementally and starting from one, the following
triple of lists is a natural encoding of the information contained in the proof.

1. A list of clauses corresponding to the $C_1, \ldots, C_n$ of the formula whose
   proof will be attempted. These will receive identifiers 1 through n.

2. A list of clauses used in the proof but not included in the input clauses, i.e.,
   derived by applications of the resolution rule on previous clauses. These
   will receive identifiers starting from $n + 1$.

3. A list of triples of numeric indexes $\langle i, j, k \rangle$, where i and j are the indexes of
   the premises and k the index of the conclusion of an instance of the binary
   resolution rule.

In order to design the FPC definition in detail, we determine one possible
shape of a general proof by resolution in LKF^a, and work around to sculpt it
during proof reconstruction. A detailed description can be found, e.g., in Chihani
et al. (2016b). Conceptually, a proof is divided in three types of phases, or regions:

1. The first phase starts the proof and asynchronously stores the clauses of the
   goal formula, each under its respective index. After all clauses have been
   stored, the second phase commences.

2. The second phase translates each use of the resolution rule (represented by a triple of indexes in the certificate) into an instance of the cut rule, where the derived clause acts as cut formula. One of the branches of this cut continues on to the next instance of the resolution rule, forming a backbone of cuts that finishes with the empty clause.

3. The third phase (type) branches off from each instance of the cut rule in the second phase. It reconstructs a shallow proof that the triple of indexes specified by the certificate is, indeed, a correct application of the binary resolution rule.

The FPC definition is shown in Figure 3.5. The three phase types in a proof by resolution correspond, in order, to the three groups of clerk and expert clauses—separated by line comments—in Figure 3.6. (This structure is revisited and expanded upon in Chapter 7.)

3.6.1 **Example** Suppose we wish to prove the following formula:

$$r(z) \wedge (\forall x. \neg r(x) \vee t(x)) \wedge \neg t(z)$$

Here, z is a term of a certain type i, and r and t are relations of type $i \rightarrow o$. To prove the formula, we instead attempt to find a refutation of its negation:

$$\neg r(z) \vee (\exists x. r(x) \wedge \neg t(x)) \vee t(z)$$

We may do so by hand or by resorting to any of a number of automated theorem provers. A resolution-based prover should be able to provide evidence for the validity of the goal formula.

Instead of relying on an informally specified proof script, it would be easy to adapt a proving tool to emit the information contained in the proof by resolution as a formally defined FPC as per the definition in Figure 3.5. According to this encoding, a possible certificate will have the following shape.

```
[resol 1 2 4, resol 4 3 5]
[1 ↦ r(z),
 2 ↦ ∀x.¬r(x) ∨ t(x),
 3 ↦ ¬t(z)]
[4 ↦ t(z),
 5 ↦ f]
```

In this presentation, indexes and their mappings to subformulas are given explicitly, whereas the basic encoding detailed above relies on implicit numbering. Both are acceptable variations of the same family of certificates. An independent checker can use this certificate and the definition on which it is based to verify that the goal formula is, in fact, a theorem.

The binary resolution FPC just presented illustrates the intricacies of precise encodings and their close connection to proof reconstruction. The FPC definition is designed in such a way that some natural and mostly independent extensions—discussed throughout Chapter 8—are straightforward, but other, slightly different encodings of resolution refutations do not share this property. Experience shows that complex FPC definitions risk brittle behavior in the face of changes and additions, and tests need to be maintained and pre- and postconditions carefully documented. Naturally, simpler "proof scaffolds" are more robust—commonly at the cost of some loss in efficiency. (Following this line of thought, for example, the shallow proofs of the third phase could be given a much more compact, implicit representation.)

3.7 Checkers, kernels, clients and certificates

At a basic level, there is a juxtaposition between the two parts that form a proof checker. On the one hand there is a trusted, sound *kernel* that implements a focused sequent calculus as a logic program. On the other, there is a *client* that specializes the kernel by providing an untrusted FPC definition and certificates built on it. Thus, the combination of the kernel and an FPC definition results in a concrete instance of the proof checker. Nonetheless, note that the client side is divided in two parts: the FPC definition and the FPC proper, the latter itself the client of the former. While the implementation of the kernel does not concern the client side, its *semantics* is of direct concern to the author of an FPC definition. On the other side, the writer of FPCs based on a definition is only interested in the higher-level semantics given by that definition, and not by its implementation of the kernel's. Moreover, the writer of FPCs is seldom interested in polarities, as they do not affect provability and polarization is often fixed by the semantics of the FPC definition, so that *as a user* one can easily write, say, formulas of *LK* and FPCs, thus staying at a high level of abstraction.

From the time of the original FPC proposal by Miller (2011), a motivating analogy has been advanced in the form of what are called synthetic inference rules.

```
% Deduced clauses are listed via the lemma predicate.
type lemma              int -> form -> o.

% Label of clauses which are never literals.
type idx      int -> index.
% Labels for literals that enter the side context.
type lit               index.
% Label for the stored pivot literal.
type pivot             index.
% Used in small proofs
type immediate         index.

% Needed just for the initial clerks.
type start   int ->  list cert -> cert.
% List of resolution triples.
type rlist             list cert -> cert.
% Temporary linkage to share an index.
type rlisti  int ->  list cert -> cert.

% Introduce a resolvent subproof.
type resolve         int -> cert -> cert.
% Introduce an order-ambiguous resolvent subproof.
type resolveX        int -> cert -> cert.

% First index in resolvent
type res             int -> cert -> cert.
% Second index in resolvent
type rex             int -> cert -> cert.

type small                         cert.
type nsmall                        cert.
type ismall                        cert.
% Must do an initial rule immediately.
type rdone                         cert.
% End of the left premise of cut.
type done                          cert.
```

3.5 Figure   The binary resolution FPC: signature (Chihani et al., 2016b, Section 7.3).

```
orC      (start Ct Certs) (start Ct  Certs).
falseC  (start Ct Certs) (start Ct  Certs).
storeC  (start Ct Certs) (start Ct' Certs) (idx Ct) :-
        inc Ct Ct'.
cutE     (start _ Certs) C1 C2 Cut :-
        cutE (rlist Certs) C1 C2 Cut.
cutE     (rlist (resolve K Cert::Certs))
        Cert (rlisti K Certs) Cut :-
        lemma K Cut, Cert = (res _ _).

% Ambiguous order
cutE     (rlist (resolveX K (res I (rex J done))::Certs))
        (res I (rex J done)) (rlisti K Certs) Cut :-
        lemma K Cut.
cutE     (rlist (resolveX K (res I (rex J done))::Certs))
        (res J (rex I done)) (rlisti K Certs) Cut :-
        lemma K Cut.

falseC (rlist    Rs) (rlist Rs).
storeC  (rlisti K Rs) (rlist Rs) (idx K).
decideE (rlist []) rdone (idx I).
trueE   rdone.

%  Left premise
allC     (res I Cert) (x\ res I Cert).
orC      (res I Cert) (res I Cert).
falseC   (res I Cert) (res I Cert).
storeC   (res I Cert) (res I Cert) lit.
decideE  (res I (rex J Cert)) (rex J Cert) (idx I).
decideE  (res I (rex J Cert)) (rex I Cert) (idx J).
someE    (rex J Cert) (rex J Cert) T.
someE    done done T.

andE     (rex J Cert) small (rex J Cert).
andE     (rex J Cert) (rex J Cert) small.
releaseE (rex J Cert) (rex J Cert).
storeC   (rex J Cert) (rex J Cert) pivot.
decideE  (rex I Cert) Cert (idx I) :- Cert = done.

andE     done small done.
andE     done done small.
initialE done pivot.

andE     small  small  small.
trueE    small.
initialE small  lit.
releaseE small  nsmall.
storeC   nsmall nsmall immediate.
decideE  nsmall ismall lit.
initialE ismall immediate.
```

3.6 Figure   The binary resolution FPC (continued): implementation.

The addition of focusing to the sequent calculus establishes a first approximation in the form of alternating synchronous and asynchronous phases. Pursuing the standard analogy, the metatheory of focused proof systems becomes the "rules of chemistry" which allow us to take "atoms" of inference and compose them into more complex, higher-level "molecules" of inference—these correspond to the inference rules of the calculus and the phases of focusing, respectively. In the FPC framework, all certificates are ultimately expressed in terms of those rules of inference, and proof checkers implement and apply those rules.

An alternative and much closer interpretation of the FPC framework views, rather, in computational terms. We liken the framework, rather, to a fantastic new assembly language, one that programs a machine that is based on a certain logic and whose instruction set, or ISA, is the set of inference rules of its corresponding sequent calculus. The definition of the semantics of a certain FPC definition is essentially the definition of a *domain-specific language*, or DSL, in which proof evidence of the theoremhood of a formula can be expressed. A concrete certificate acts as a program written in the language of the FPC definition and interpreted on the checker that implements the assembly of the chosen logic, i.e., the architecture of the logic computer. The analogy is apt insofar as it reflects the deep ties between certificate definitions and the intricacies of proof systems: until now, programming these FPC definitions has been the delicate domain of the expert.

While some attention has been paid to the discipline of programming FPC definitions (Blanco and Chihani, 2016, 2017), the sole appeal remains to the underlying logic. Concrete definitions to certify a certain proof family or a tool cannot be easily extended to other, superficially similar tools, given that each employs its own ad hoc DSL. The task of writing what amounts to a compiler to a language based on logic remains a nontrivial task.

3.8 Notes

The increased complexity of modern automated theorem provers has brought with it a need for proof certification. Potential sources of errors in claimed proofs range from bugs in the code to inconsistencies in the object theory. To address this problematic, various tools for proof certification have been implemented that can improve our confidence that the output from theorem provers constitutes in fact a proof. These tools can be classified into two groups according to the *object of verification*:

1. A given *theorem prover* could itself be proved formally correct. See, for example, Ridge and Margetson (2005).

2. The *output of a theorem prover* can be verified independently from the tool that produced it. This possibility can be further subdivided in two types based on the *proof reconstruction strategy*:

   (a) Systems for replaying proofs using external theorem provers for the verification of specific proof steps. Among these, we count general-purpose tools like Sledgehammer (Paulson and Susanto, 2007; Böhme and Nipkow, 2010), PRocH (Kaliszyk and Urban, 2013), and GDV (Sutcliffe, 2006), as well as more specialized efforts such as the verification of proofs generated by the E prover using Metis (Paulson and Susanto, 2007).

   (b) Tools that comprise an encoding or a translation of the semantics of certain theorem provers, which is then used to replay proofs from those known provers. This group admits a further subdivision based on *specificity criteria*:

      i. Specific tools, such as Ivy (McCune and Shumsky, 2000) and the encodings of MESON (Loveland, 1968) in HOL Light, and of Metis (Hurd, 2003) in Isabelle.

      ii. General-purpose tools like Dedukti (Boespflug et al., 2012) and ProofCert (Miller, 2011). Our interest and our efforts concentrate in this last subcategory.

These various classes of tools represent different approaches to proof certification. While we can have a high level of trust in the correctness of formally verified provers in category 1, their performance cannot be compared to that of the leading theorem provers like E and Vampire (Riazanov and Voronkov, 2002). The remaining groups do not pose restrictions on the provers themselves but the generality and automation of those in category 2 group come with the cost of using an external theorem prover and translations, which might result in reduced confidence. The families of tools in subfamily 2b require an understanding of the semantics of a theorem prover so that one may guarantee the soundness of proofs by their reconstruction in a low-level formal logic. Working with an actual proof has several advantages—as one can apply procedures like proof transformations. Group 2(b)ii has additional advantages over its sibling 2(b)i: a single certifier can

be written that should be able to check proofs from a range of different systems and the existence of a common language for proofs allows for the creation of proof libraries and marketplaces (Miller, 2011).

The tools in this last target group have had, so far, only limited success in the theorem proving community at large. One reason for this is that understanding and specifying the semantics of proofs requires sophistication in the interplay between deduction and computation—whether via rewriting in functional style or by proof search. Separating theorem provers from proof checkers using a simple, declarative specification of proof certificates is not new: see Harrison et al. (2014) for a historical account. We give here a brief partial sketch.

A common starting point is the dependently typed λ-calculus LF (Harper et al., 1993), originally proposed as a framework for specifying natural deduction proofs; the Elf system (Pfenning, 1989) provided both type checking and inference for this framework; and the proof-carrying code project of Necula (1997) used LF as a target proof language. The LFSC system is an extension of The dependently typed λ-calculus has been extended with side conditions by the LFSC system; an implementation of it has been successfully used to check proofs coming from SMT solvers like CLSAT and CVC4 (Stump et al., 2013). Yet another extension to the dependently typed λ-calculus is Deduction Modulo (Cousineau and Dowek, 2007; Boespflug, 2011): in this calculus, rewriting is available.

The Dedukti proof checker (Boespflug et al., 2012), based on this latter extension, endeavors to answer similar questions as those posed in this chapter through different methods, adopting a more computational view of checking based a functional instead of a relational paradigm, and congruences generated by sets of rewrite rules in lieu of FPC definitions. Dedukti has been successfully used to check proofs from such systems as Coq (Boespflug and Burel, 2012) and HOL (Assaf and Burel, 2015) among other systems. In the domain of higher-order classical logic, the GAPT system (Ebner et al., 2016) is capable of proof checking in sequent calculus, resolution, and expansion trees—a generalization of Herbrand disjunctions—; it supports both checking and transformation among proofs expressed in those supported formats.

The Foundational Proof Certificate framework described in this chapter was recently proposed as a means of defining the semantics of a wide range of proof languages for first-order classical and intuitionistic logic (Chihani et al., 2013; Chihani, 2015; Chihani et al., 2016b). Instead of starting with a dependently typed λ-calculus, the FPC framework is based on Gentzen's lower-level notion

of sequent calculus proof. Previously, FPC definitions have been formulated to model diverse sources of proof evidence, among these resolution refutations (Robinson, 1965), expansion trees (Miller, 1984), Frege proof systems, matings (Andrews, 1976), simply typed and dependently typed λ-terms, equality reasoning (Chihani and Miller, 2016), tableau proofs for some modal logics (Miller and Volpe, 2015; Libal and Volpe, 2016b; Marin et al., 2016), and decision procedures based on conjunctive normal forms, truth table evaluation, and the G4ip calculus (Dyckhoff, 1992; Troelstra and Schwichtenberg, 2000). Some simple examples have been covered in this chapter. New applications are described throughout Parts II and III. As with other declarative and high-level frameworks, proof checkers for FPC specifications can be implemented using the logic programming model of computation (Chihani et al., 2015, 2016b; Miller, 2017).

In addition to those standard applications in theorem proving, FPCs have been applied to model checking (Heath and Miller, 2015, 2017) given a richer logic than used in the last paragraph. $\mu MALL$, i.e., multiplicative-additive linear logic with greatest and least fixed points as logical connectives instead of exponentials, is suitable for this purpose (Baelde and Miller, 2007). In a similar vein, the addition of fixed points to intuitionistic logic, μLJ, serves to reason about constructive proofs and their expression as outlines, in a note closer to the connection between theorem provers and certification (Baelde et al., 2010). In this extended logical framework, further developments will be studied in Part III.

As we have noted, the formula indexing mechanism of the FPC framework does not impose functionality, i.e., different formulas can have the same index. Previously, indexes have been identified with diverse structures, including de Bruijn numerals and formula occurrences (Chihani et al., 2013). It is possible to conceive very sophisticated indexing structures that assign sets of properties to the stored lemmas (e.g., "associativity lemmas" or "lemmas about natural number addition"). These rich indexing schemes could then be used to greatly increase the expressiveness of the decide rules.

The simple examples in this chapter serve already to showcase the existence of versions of proofs with very different properties and behaviors. Both extremes of very implicit proofs (as in Section 3.3, with constant certificate size and exponential checking time) and very explicit proofs (as in Section 3.4) can be expressed in the framework. Surely the nature and effectiveness of proof checkers can be greatly affected by the level of detail of a proof format. Well-designed

FPC definitions will exhibit the standard tradeoff between certificate space and checking time. Chapters 5 and 6 will pursue this line of inquiry.

Even as we guide users away from the cryptic assembly of the sequent calculus, letting them instead write abstract certificate terms based on an FPC definition, the establishment of a mapping between these abstract terms and their correspondence with the assembly level is an inevitable step of significant complexity that needs to be repeated, with variations, for each FPC definition. Despite this, the applicability of the framework to a number of representative and highly varied settings has been studied with satisfactory results. Some inroads have been made in the application of the FPC framework to certify the output of resolution-based, automated theorem provers like the E prover (Schulz, 2013; Chihani et al., 2015), and more recently and comprehensively for Prover9 (McCune, 2010; Blanco et al., 2017a). On this topic, see esp. Chapters 7 and 8, but also Chapters 11 and 13.

Certification applies to automated and interactive theorem provers alike—software with common foundations yet very different operating principles. There exist efforts to integrate exemplars across the various categories of tools. Typically, one starts from a proof assistant and calls an automated theorem prover through an interface (called a *proof hammer*) to try to finish parts of the proof on behalf of the user; see, for example, Blanchette et al. (2016). As integration grows, the lines separating these categories blur to the point that classification becomes unclear. For example, dependent types are commonly at the intersection between programming languages and proof assistants: for instance, Agda defines itself as both. Proof assistants may further shift towards automation by applying machine learning to the task of obtaining proof scripts starting from corpora of existing proofs (Kaliszyk and Urban, 2015).

Part II

Logics without fixed points

4 Logic programming in intuition-istic logic

4.1 Logic and computation

The mathematical study of computation encompasses a diverse range of abstract models, of which Turing machines and λ-calculi are among the best known; their notions of computable functions coincide with the concept of general recursion. Moreover, the Church-Turing thesis conjectures that every effectively calculable function is computable, say, by a Turing machine or an assimilable model. Hence, if a concrete computer or a programming language running on a computer can simulate a Turing machine (up to finite amounts of memory), it can compute any function computable by a Turing machine: in other words, it is Turing-complete. Many models of computation have this property, and some of these models are based directly on logical principles.

In fact, logic can be seen as playing two kinds of roles in computation (Miller, 1995). On the one hand, it can be used *externally* as a tool to reason about mathematical structures used to model programs and their behavior; that is, *computation-as-model*. On the other hand, logic can be used *internally*: that is, logical elements (formulas, etc.) can be used as the building blocks of computation; that is, *computation-as-deduction*. In this latter class, two different visions giving rise to two distinct paradigms exist. First, *proof normalization* models the state of computation as a proof term, and the act of computing as the reduction of that proof term to a normal form; this is the foundation of *functional programming*. Second, *proof search* sees the state of computation as a collection of hypotheses and a goal to prove from those, and the act of computing as the derivation of a proof of the goal; this is the foundation of *logic programming* as embodied by languages such as Prolog—and our primary interest.

The automation of proof search at the center of logic programming relies on the cut elimination property, which in suitable logics asserts the existence of cut-free, analytical proofs. The resulting programming style is not *functional*, but *relational*: pure logic programming, like pure functional programming, are free from side effects; logic programming generalizes functional programming by adding nondeterminism to the model of computation. The core concepts of logic programming share with functional programming core concepts like terms and types, while replacing functions with formulas, relations, and (explicit) proofs. This chapter is not meant to be a comprehensive introduction to logic programming: for this, consult among others Sterling and Shapiro (1986); O'Keefe (1990); Miller and Nadathur (2012); our presentation is based on the latter book. For a historical perspective on the development of λProlog, see also Nadathur and Miller (1988).

The rest of the chapter is organized as follows. Section 4.2 introduces some essential concepts of logic programming. Section 4.3 provides a succinct tutorial introduction to the higher-order logic programming language λProlog, which is used extensively throughout the document. Section 4.4 presents a proof checking kernel for the FPC framework (in particular, the *LKF^a* logic) as a representative application of logic programming ideas in a concrete language like λProlog. Section 4.5 concludes the chapter.

4.2   Logic programming

In what follows, we shall be interested in logic programming languages with support for rich types—this is in contrast with standard Prolog, which is untyped. Typed terms are interpreted in the usual sense of the Simple Theory of Types of Church (1940). Under this view, a logic programming language must provide syntactic support to write *signatures* Σ that permit us to write terms and formulas over their types and type constructors, *logic programs* $\mathcal{P}$ as collections of formulas over a given signature, and *goal formulas* G. In addition to those three syntactic elements, the language must implement the semantics of a proof calculus, hence enabling the construction (by searching) proofs of a goal G given a program $\mathcal{P}$, both over a signature Σ. The necessity to *construct* proofs points towards an intuitionistic interpretation whose informal interpretation is the following:

1. The proof of t succeeds regardless of signature and program.

2. The proof of a conjunctive goal $G_1 \wedge G_2$ proceeds by finding proofs of G_1 and G_2 based on the unaltered signature and program.

3. The proof of a disjunctive goal $G_1 \wedge G_2$ proceeds by finding a proof of G_1 or a proof of G_2 based on the unaltered signature and program.

4. The proof of an implicational goal $D \supset G$ proceeds by finding a proof of the consequent G based on the unaltered signature and the program extended with the program clause dictated by the antecedent, $\mathcal{P} \cup \{D\}$.

5. The proof of a universal goal $\forall x.G$ proceeds by finding a proof of $[y/x]G$—where x is replaced by a fresh constant y (an *eigenvariable*)—on the signature extended with the new constant $\Sigma \cup \{y\}$ and the unaltered program.

6. The proof of an existential goal $\exists x.G$ proceeds by finding a proof of $[t/x]G$—where x is replaced by a term t—on the unaltered signature and program. In implementation, a placeholder or *logic variable* for a concrete term (of which there may be infinitely many) will be generated for t and instantiated by solving problems of term *unification*.

The resulting operational semantics must correspond to the declarative reading of logic in the underlying sequent. Missing from this picture is the treatment of atomic goals, which will depend on the specific logic being implemented. Let us now consider the logical framework of *Horn clauses*, which—in their first-order variation—are the substrate of the Prolog programming language. The following recursive definition defines formulas for goals G and for program clauses D; A denotes atomic formulas:

$$G ::= t \mid A \mid G \wedge G \mid G \vee G \mid \exists x.G$$
$$D ::= A \mid G \supset D \mid D \wedge D \mid \forall x.D$$

This is one of several equivalent definitions of Horn clauses. Here, quantifiers are polymorphic at the type of the bound variable x. The resulting logic is said to be *first-order* if quantification is only allowed at types of order 0 or 1; it is *higher-order* if quantification is allowed at types of arbitrary order (while excluding the type of predicates, commonly written o). Equivalently, program clauses can be organized as formulas of the following form:

$$\forall x_1. \cdots \forall x_m.A_1 \wedge \cdots \wedge A_n \supset A_0$$

$$nat\ z$$
$$\forall N.nat\ N \supset nat\ (s\ N)$$
$$\forall N.plus\ z\ N\ N$$
$$\forall K.\forall M.\forall N.plus\ K\ M\ N \supset plus\ (s\ K)\ M\ (s\ N)$$

4.1 Figure   Logic specification of natural numbers and addition on them as Horn clauses. The specification is based on a type *nat* with two constructors representing the standard inductive definition of natural numbers: *z* of type *nat*, and *s* of type *nat* → *nat*.

That is, a prefix of universally quantified variables that bind an implication with a conjunctive antecedent of atoms and an atomic consequent. The definition of Horn clauses disallows implications and universal quantifiers in goals; in consequence, the signature and the program remain unaltered during proof search (and therefore during program execution). To complete the semantics of the resulting programming language, it needs to furnish the semantics of proof search on atomic goals. This process of *backchaining* analyzes the program to determine if the goal is a known fact (in which case the proof is completed) or may be the consequent of some other antecedent conditions, in which case proofs for those will be sought.

Horn clauses are a powerful framework for writing logical specifications, such as the program that contains clauses defining the construction and addition of natural numbers in Figure 4.1. Nevertheless, it is possible to generalize them by carefully allowing both signatures and programs to grow during proof search. *Hereditary Harrop formulas* extend the definition of Horn clauses as follows:

$$G ::= t \mid A \mid G \wedge G \mid G \vee G \mid \exists x.G \ \boxed{\mid D \supset G \mid \forall x.G}$$
$$D ::= A \mid G \supset D \mid D \wedge D \mid \forall x.D$$

In the extended logic, universal quantification is allowed in goals, as is implication subject to the restriction that the antecedent be a program clause. The syntax of program clauses remains unaltered, but is now mutually recursive with the definition of goals. This richer framework is one of the cornerstones of λProlog. The second enhancement is the replacement of first-order terms with higher-order λ-terms and quantification. This support for the application of abstractions to

bound variables enables a powerful form of abstract syntax—i.e., the representation of expressions not by strings, but by data structures—where constructs like names and variable bindings are reflected directly as binders in the representation meta-language. This approach is called *higher-order abstract syntax*, or HOAS (Pfenning and Elliott, 1988). In (higher-order) logic programming, *λ-tree syntax* (Miller and Palamidessi, 1999; Miller, 2000) incarnates the ideas of HOAS in a practical way; λProlog offers support for it by its use of dynamic higher-order pattern unification. These problems are especially relevant to Section 13.2; the difficulty of their application in a functional setting are discussed in Section 6.6.

Logic programming languages commonly implement some impure features, such as a "cut" operator that restricts backtracking search—not to be confused with the logical rule of cut—or a negation operator which purports to succeed of a given goal fails. These extra-logical features, which have no reflection in the logic that serves as the foundation of logic programming, may facilitate some programming tasks, but do so at the cost of risking soundness if they are not applied in very restricted cases and with great care. We will have little use for these and will avoid them whenever possible, instead strongly favoring pure, declarative programs with a clear mirror image in the underlying logic.

4.3 λ**Prolog**

Atomic types in λProlog are defined by the keyword **kind**. Typically, a kind thus defined represents a type like natural numbers. However, these kinds can also define families of types parameterized by other types, such is lists of elements of a given type (predefined by λProlog) or pairs of elements of two given types. Arrow types in kinds are used for these directives:

```
kind   nat    type.
kind   pair   type -> type -> type.
```

These *kind expressions* determine how *type expressions* may be constructed from the concrete kinds that they define. Concrete type expressions are derived by the usual mechanism of application: `nat` is already a complete type expression, whereas `pair` must be given arguments, for instance (`pair nat nat`) for pairs of naturals. In general, in a type expression $\tau_1 \to \cdots \to \tau_n \to \tau_0$ with $n \geq 0$, τ_0 is called the *target type* and the τ_i with $i > 0$ (if any) are called *argument types*. The *order* of a type expression $O(\tau_1 \to \tau_2)$ is defined recursively

as $\max(O(\tau_1) + 1, O(\tau_2))$, where the base case of types expressions without type arguments has order 0. A first-order language restricts types to be of order 0 or 1.

For kinds to be populated by terms, they need to be endowed with *type constructors*, defined by the keyword **type** and characterized by type expressions whose target type is the kind in question. For instance, natural numbers can be defined by zero and the successor of another natural number, and generic pairs by the two types of their elements. Polymorphism is supported by the introduction of *type variables* which can be unified with any concrete type. In concrete syntax, arrow notation is unambiguously overloaded in the two contexts of kind and type expressions:

```
type    z    nat.
type    s    nat -> nat.
type    pr   A -> B -> pair A B.
```

Based on this, we can write typed terms such as (s z) of type nat or (pr z (s z)) of type (pair nat nat). These examples illustrate application; the second operator of the λ-calculus, abstraction, is written \. For instance, (x\ (s x)) represents an abstraction which, applied to a natural number, return its successor. The name x is thus bound in the expression that follows the abstraction operator.

The type of formulas in λProlog is designated by o. It is populated by the *logical constants* that represent the connectives of the logic, therefore taking other formulas (i.e., terms of type o) as their arguments. In addition to these logical constants, the programmer defines *predicates* (also called *relations*) by writing type constructors with o as a *target type*: for example, a relation that takes two natural numbers and relates them to their addition can be typed as:

```
type    plus    nat -> nat -> nat -> o.
```

That is, plus is a relation between triples of inductively defined naturals that shall be defined to hold iff the sum of the two first arguments is equal to the third. Relation symbols stand in contrast to **type** constructors with target types other than i, which define *function symbols*. The full specification for plus specification is shown in Figure 4.2. Generally speaking, in writing the clauses for those relations, we will ordinarily resort to the logical constants. In λProlog, these are the following:

1. true of type o, for *t*.

```
% Signature: type
kind    nat      type.
type    z        nat.
type    s        nat -> nat.

% Signature: predicates
type    is_nat   nat -> o.
type    plus     nat -> nat -> nat -> o.

% Module
is_nat z.
is_nat (s N) :- is_nat N.

plus z N N.
plus (s K) M (s N) :- plus K M N.
```

4.2 **Figure**   Relational specification of natural numbers and addition on them in standard λProlog. Signature and module delimiters are omitted.

2. **,** of type o -> o -> o, for ∧, used in the bodies of clauses to sequence subgoals in the usual manner.

3. **&** of type o -> o -> o, also for ∧, used with the heads of clauses. It permits several heads to share a goal. For example, if we have a client-defined type of formulas with constructors and and or (as well as not) and we wish to recurse the structure of terms of this type, we could compact repetitious bodies by writing:

```
recurse (not A)   :- recurse A.
recurse (and A B) &
recurse (or A B)  :- recurse A, recurse B.
```

4. **;** of type o -> o -> o, for ∨.

5. **:-** of type o -> o -> o, for ⊂ in the sense of the implication ⊃ found in program clauses D, where the succedent (i.e., the head of a clause) is written before the antecedent (i.e., the body of the clause), following the standard syntax of proof search as implemented in logic programming languages.

6. **=>** of type o -> o -> o, for ⊃, the implication found in goals G, whose succedent is a program clause D which is used to extend the program for the purposes of finding a proof of the succedent under *hypothetical reasoning*.

7. **pi** of type (A -> o) -> o, for ∀. Quantification operates on a formula
   F abstracted over an arbitrary but defined type T, written **pi** (x:T)\F;
   the type can often be uniquely determined by type inference and the anno-
   tation dropped, thus writing simply **pi** x\F, where x\F is the argument
   of the connective. In proof search, a fresh eigenvariable of the appropriate
   type is applied to the abstracted formula.

8. **sigma** of type (A -> o) -> o, for ∃. In proof search, a fresh logic
   variable of the appropriate type is applied to the abstracted formula.

Existential variables are often implicitly quantified and represented by names
starting with an uppercase letter. However, the abstraction operator can bind any
name in an expression: (X\ (s X)) and (x\ (s x)) are equivalent. The
anonymous logic variable _ can be used to represent a bound variable which goes
unused in the body of the abstraction.

λProlog programs are structured into modules. A module is composed two
parts. First, a *signature file*, which opens with the keyword **sig** <name>. and
declares the interface for the module, namely kind and type operators and their
complementary definitions. Second, a *module file*, started with the keyword
module <name>., and which contains clauses for the declared relations and
other private declarations. Both signatures and modules can depend on others of
the same type by *signature accumulation* (accum_sig <name>.) and *module
accumulation* (accumulate <name>.), respectively. These programming lan-
guage concepts have a clear logical interpretation and interact harmoniously with
the other features of λProlog. One important practical point is to ensure that
each module used by a program be accumulated at exactly one point: multiple
accumulations of the same module correspond to the creation of multiple copies
of the contained clauses and a combinatorial explosion in the number of possible
backtracking points.

Like other logic programming languages, λProlog has (limited) support for
I/O, as well as support for some extra-logical features like built-in arithmetic,
a backtracking cut operator ! and a negation operator not used to exercise
negation-as-failure. All these will be used sparingly and only in situations when it
is logically sound to do so.

Nowadays, two major implementations of λProlog coexist: the Teyjus com-
piler (Nadathur and Mitchell, 1999) and the ELPI interpreter (Dunchev et al.,
2015). Both will be the subject of lengthy discussion in Chapter 8. These are in

addition to the declarative core of λProlog implemented at the specification level in the Abella theorem prover (Baelde et al., 2014).

4.4   FPC kernels

Logic programming is ideally suited to the implementation of proof search, where built-in mechanisms such as backtracking search and unification coincide with the requirements of many proof systems—among those the augmented sequent calculi that define the FPC framework (Miller and Nadathur, 2012, Chapter 9). A proof checking kernel, i.e., a program that implements one such calculus, is an interesting logic program that exhibits the characteristic features of λProlog while laying the technical foundations for subsequent chapters. In this section we concentrate on the FPC proof system for classical logic, *LKF*[a], introduced in Section 3.2 as an evolution of the sequent calculi presented throughout Chapter 2.

The kernel and its public interface are presented in full in Figure 4.3. For the client, the interest resides in the standard interfaces essentially shared by all kernel implementations for a given logic. The interface is divided in three parts:

1. The definition of the *object logic* of polarized classical formulas. The signature defines a type of atoms `atm` and a type of terms `i` which may appear in atoms; both must be defined by the client. In addition, the logical constants are defined: constructors to inject atoms into positive or negative literals, as well as the various logical connectives. Quantifier constructors build formulas from abstractions over formulas, resorting to the native support of λProlog for binding representation of manipulation and thus avoiding its thorny implementation. In addition to the standard connectives in both polarities, a pair of *delay connectives* that force a polarity on an arbitrary formula are introduced for practical purposes. Delays can be defined in terms of the standard connectives and their presence is therefore inessential. A number of related predicates are charged with the construction and deconstruction of formulas from and to their components, as well as various polarity checks used by the inference rules of the sequent calculus.

2. The client signature for the *FPC framework* defines the kinds of certificates and indexes (as well as disjunctive choices) and declares predicates for clerks and experts corresponding to the annotations for each inference rule in Figure 3.1. In order to instantiate the kernel into a proof checker, a client must provide type constructors for certificates and indexes—together with

```
%% Logic signature
kind    form, i, atm    type. % Formulas, terms, and atoms
type    n, p                  atm -> form. % Literals
type    f+, f-, t+, t-        form. % Units
type    d-, d+           form -> form. % Delays
type    &-&, &+&    form -> form -> form. % Conjunctions
type    !-!, !+!    form -> form -> form. % Disjunctions
type    all, some   (i -> form)  -> form. % Quantifiers
infixr  &-&, &+&   6.
infixr  !-!, !+!   5.

% Construction and deconstruction of formulas
type    true+, true-, false+, false-           form -> o.
type    conj+, conj-          form -> form -> form -> o.
type    disj+, disj-          form -> form -> form -> o.
type    lit-, lit+                      atm -> form -> o.
type    all-, some+           (i -> form)  -> form -> o.

type    isNegForm, isNegAtm, isPosForm,
        isPosAtm, isNeg, isPos          form -> o.
type    negate                          form -> form -> o.
type    ensure-, ensure+   form -> form -> o.

%% FPC signature
kind    cert, index    type.
kind    choice         type.
type    left, right    choice.
type    allC           cert -> (i -> cert) -> o.
type    andC           cert -> cert -> cert -> o.
type    andE           cert -> cert -> cert -> o.
type    cutE           cert -> cert -> cert -> form -> o.
type    decideE        cert -> cert -> index -> o.
type    falseC         cert -> cert -> o.
type    initialE       cert -> index -> o.
type    orC            cert -> cert -> o.
type    orE            cert -> cert -> choice -> o.
type    releaseE       cert -> cert -> o.
type    someE          cert -> cert -> i -> o.
type    storeC         cert -> cert -> index -> o.
type    trueE          cert -> o.

%% Kernel signature
type    lkf_entry   cert -> form -> o.
type    async       cert -> list form -> o.
type    sync        cert ->      form -> o.
type    storage     index ->     form -> o.
```

4.3 Figure The *LKF^a* kernel in λProlog: signatures.

```
lkf_entry Cert Form :- async Cert [Form].

async Cert nil :-
  cutE Cert CertA CertB F,
  negate F NF, async CertA [F], async CertB [NF].
async Cert nil :-
  decideE Cert Cert' I,
  storage I P, isPos P, sync Cert' P.
async Cert [t- | _].
async Cert [f- | Rest] :-
  falseC Cert Cert',
  async Cert' Rest.
async Cert [d- A | Rest] :-
  async Cert [A | Rest].
async Cert [(A !-! B) | Rest] :-
  orC Cert Cert',
  async Cert' [A, B | Rest].
async Cert [(A &-& B) | Rest] :-
  andC Cert CertA CertB,
  async CertA [A | Rest], async CertB [B | Rest].
async Cert [all B | Rest] :-
  term_to_string Cert _, % Teyjus bug: force normalization
  allC Cert Cert',
  pi w\ async (Cert' w) [B w | Rest].
async Cert [C|Rest] :- (isPos C ; isNegAtm C),
  storeC Cert Cert' I,
  storage I C => async Cert' Rest.

sync Cert t+ :-
  trueE Cert.
sync Cert (d+ A) :-
  sync Cert A.
sync Cert N :- isNeg N,
  releaseE Cert Cert',
  async Cert' [N].
sync Cert (p A) :-
  initialE Cert I, storage I (n A).
sync Cert (A &+& B) :-
  andE Cert CertA CertB,
  sync CertA A, sync CertB B.
sync Cert (A !+! B) :-
  orE Cert Cert' C,
  ((C = left, sync Cert' A); (C = right, sync Cert' B)).
sync Cert (some B) :-
  someE Cert Cert' T,
  sync Cert' (B T).
```

4.4   Figure   The *LKF^a* kernel in λProlog (continued): implementation.

any ancillary declarations on which those constructors depend—and must determine the semantics of those constructors by defining the behavior of clerks and experts through clauses of their predicates. In doing so, an *FPC definition* is integrated with the kernel.

3. The *kernel signature* proper defines both types of sequents, synchronous and asynchronous, expressing the proof search for a given conclusion sequent. Both corresponding `async` and `sync` relations define parameters for the certificate and the workbench; the indexed storage is maintained via hypothetical reasoning, adding facts to the logic program by filing formulas with their indexes as clauses of the `storage` predicate. Ordinarily, the end user is only interested in the elementary operation that takes a formula and a certificate, forms the initial entry sequent with both and performs proof search guided by the certificate term; this is what is represented by the interface relation to the kernel, `lkf_entry`.

The kernel module implements guided proof search as a direct encoding of the proof system of Figure 3.1. Each inference rule is turned sideways and written clause with the conclusion at the head and the (conjunctive) premises as the body. As a general rule, calls to clerks and experts precede recursive calls to the proof search predicates on asynchronous and synchronous sequents, which are only performed if the corresponding clerk or expert declares the inference rule as applicable according to the certificate term. As noted above, the store rule uses the λProlog implication to extend the clauses that define the storage zone, initially empty; this relation is then queried by the decide rule. The treatment of quantifiers is also of interest: the universal quantifier relies on λProlog to generate a fresh *term eigenvariable* used to instantiate the formula abstraction; the continuation certificate produced by the clerk is also abstracted over a term: the new eigenvariable is applied to this abstracted certificate to obtain a "plain" certificate. Besides these interesting techniques in very specific places, the kernel code is remarkably simple.

In addition to the polarized formulas used by the kernel, represented by the kind `form`, it is common to define a standard classical (unpolarized) logic with a full set of connectives including, say, implication and non-atomic negation. Let us call this type of unpolarized formulas `bool`. An unpolarized logic is useful on the client side to write formulas as they are commonly understood; the module that defines the unpolarized logic must also furnish predicates to translate unpolarized formulas into polarized formulas—including their conversion to

negation normal form. This facilities are commonly available without loss of generality, given that in common FPC definitions—such as those presented in Chapter 3—the polarization scheme is fixed and leaves no choices to the user. Typically, these polarization predicates project unpolarized atoms to a standard encoding of polarized atoms.

Over time, several kernels and programming environments for those kernels have been developed with varying degrees of client support. One such family is used in works such as Chihani et al. (2016b); with minimal changes, this is the basis of the kernel presented in Figure 4.3, which is used throughout Part II. A related effort is represented by the Checkers system by Chihani et al. (2015), which offers more (scripting) support for modular definition of problems and their use of FPC definitions. In principle, the structural differences between both principal families of kernels are primordially cosmetic: both are straightforward implementations of sequent calculi—with focusing and augmentations—with some differences in their concrete syntax. (In fact, some of the experiments in coming chapters have been adapted to run in Checkers without issue by means of a shallow mapping translation.)

Some versions of these systems use *hosting* to implement some kernels on top of others, deemed more canonical in some sense. Typically, this involves hosting classical logic in intuitionistic logic (namely, LKF^a in LJF^a) by double-negation translations—as discussed in Section 2.3, but also by appealing to the LKU system of Liang and Miller (2009); for the connection with focusing, see Chihani et al. (2016a). A similar effort has been recently undertaken by Libal and Volpe (2016a). In reality, the coexistence of multiple kernels, including computational models that diverge significantly from the direct implementation of the sequent calculus, is possible and desirable; the only requirement is that the client interface—chiefly the clerks and experts through which FPC definitions sculpt the semantics of proofs—remains unchanged.

4.5 Notes

Proof checking has been implemented many times over the past decades, ranging from Automath (de Bruijn, 1980) to the Edinburgh LCF system (Gordon et al., 1979) and, more recently, to Dedukti (Dedukti). Although logic programming engines have seldom been used for such purposes, they make for rather natural and direct implementations of proof checkers, as Section 4.4 has shown. Logic

programming foundations make it possible to naturally perform certain tasks that might be harder to do in these other proof checking frameworks. One such central task is to do "proof reconstruction" as an integral part of proof checking: in terms of the FPC framework, the designer of a proof certificate format can leave out details of a proof from a certificate if that designer feels confident that the missing details can be reconstructed with acceptable costs. In that way, there is an easy trade-off between the size of certificates and the costs of checking those certificates.

5 Certificate pairing

5.1 Implicit and explicit versions of proof

A central issue in designing a proof certificate format (i.e., an FPC definition) involves choosing the level of proof detail that is stored within a certificate. If a lot of details (e.g., complete substitution instances and complete computation traces) are recorded within certificates, simple programs can be used to check certificates. Of course, such certificates may also be large and impractical to communicate between prover and checker. On the other hand, if many details are left out, then proof checking would involve elements of *proof reconstruction* that can increase the time to perform proof checking—and reconstruction—as well as increase the sophistication of the proof checking mechanism.

One approach to this trade-off is to invoke the *Poincaré principle* (Barendregt and Barendsen, 2002), which states that traces of computations (such as that for 2 + 2 = 4) should be left out of a proof and reconstructed by the checker. This principle requires a checker to be complex enough to contain a—possibly small— programming language interpreter capable of filling the gaps in a proof skeleton and thus elaborating it into a full proof. In frameworks like LFSC (Oe et al., 2009) and the Dedukti checker (Cousineau and Dowek, 2007; Boespflug et al., 2012; Assaf et al., 2016), such computations are performed using deterministic functional programs.

The FPC framework goes a step beyond such systems by allowing nondeterministic computation carried out by a higher-order logic programming language. As in other settings like, say, finite state machines, nondeterministic specifications can be exponentially smaller than deterministic ones: such a possibility for shortening specifications is an interesting option to exploit in specifying proof certificates in particular. Of course, deterministic computations are instances of

nondeterministic computations: similarly, FPCs can be restricted to deterministic computation when desired.

5.1.1 **Example** The following example illustrates a difference between requiring all details to be present in a certificate and allowing a certificate to elide some details. A proof checker for first-order classical logic could be asked to establish that a given disjunctive collection of literals, say, $L_1 \vee \cdots \vee L_n$ is provable. An explicit certificate of such a proof could be an unordered pair $\{i, j\} \subseteq \{1, \ldots, n\}$ such that L_i and L_j are complementary. A proof certificate term for this could be written as (`complementary i j`).

If we allow nondeterminism, then the indexes i, j do not need to be provided: instead, we could simply confirm that there exist guesses for i and j such that literal L_i is the complement of L_j. Compactly, a certificate term may be written as, for example, `some_complementary`. Of course, there may be more than one such pair of guesses. The use of nondeterminism here is completely sensible since a systematic and naive procedure for attempting a proof of such a disjunction can reconstruct the missing details. The cost of this nondeterminism is, in this case, a quadratic number of guesses in the size of the number of literals.

Since the sequent calculus can be used as the foundation for both logic programming and theorem proving, the nature and structure of nondeterministic choices in the search for sequent calculus proofs have received a lot of attention. For example, the original *LK* and *LJ* sequent calculus proof systems by Gentzen (1935) contain so many choices that it is hard to imagine performing meaningful proof search directly in those proof systems. Instead, those original proof systems can be replaced by focused sequent calculus proof systems in order to help structure nondeterminism—this development was covered in Section 2.5.

In particular, recall the common dichotomy between *don't-care* and *don't-know* nondeterminism and how it gives rise to two different phases of focused proof construction. Don't-know nondeterminism is employed in the *positive* phase, where significant choices affecting the evolution of the proof—choices determined by, say, an oracle or a proof certificate—are chained together. Don't-care nondeterminism is employed in the *negative* phase and it is responsible for performing determinate (i.e., functional) computation. As we shall see, this second phase provides support for the Poincaré principle.

The rest of the chapter is organized as follows. Section 5.2 introduces the *pairing meta-FPC*. Section 5.3 illustrates how it can be used to *elaborate* proof certificates (introduce more details) and to *distil* proof certificates (remove some

details). Section 5.4 presents the *maximally elaborate FPC* as the limit of elaboration and introduces the discussion of how certificate transformations can be used to provide trust in proof checking, which is the subject of Chapter 6. Section 5.5 outlines some experimental transformations between proof formats enabled by pairing. Section 5.6 concludes the chapter.

5.2 Pairing of FPCs

Because FPC definitions of proof evidence are declarative (in contrast to procedural), some powerful, formal manipulations of proof certificates are easily enabled. In this section, we demonstrate how the formal combination of two certificates—their *pairing*—can be used to transform proof certificates into other certificates, either more or less explicit than the first.

5.2.1 Example   Consider checking a proof certificate for a resolution refutation that does not contain the substitutions used to compute a resolvent (as in Example 3.6.1). Since the checking process computes a detailed focused sequent in the background, that process must compute all the substitution terms required by sequent calculus proofs (in the above example, the bound variable x shall be instantiated with the concrete term z). If we could check *in parallel* a second certificate that allows for storing such substitution terms, then those instances could be inserted into the second, more explicit certificate.

For example, consider the existential expert is defined for two different certificate constructors: one we shall call `instan` (providing an explicit instantiation term for the existential quantifier) and a second one designated simply by `f` (which contains no relevant information). These two correspond to the following two expert clauses:

$$\exists_e((\texttt{instan}\ t\ \Xi), \Xi, t). \qquad \exists_e((\texttt{f}\ \Xi), (\texttt{f}\ \Xi), t).$$

A pairing constructor of certificates, $\langle \cdot, \cdot \rangle$, could be defined for all clerks and experts by simply invoking the same clerk and expert on the components of the pair. For example, the existential expert would be defined as:

$$\exists_e(\langle \Xi_1, \Xi_2 \rangle, \langle \Xi_1', \Xi_2' \rangle, t) \ \texttt{:-}\ \exists_e(\Xi_1, \Xi_1', t) \wedge \exists_e(\Xi_2, \Xi_2', t).$$

In this way, if we pair an implicit proof certificate with the more explicit version of that certificate, we can use the underlying logic programming engine to

record into the more explicit certificate information that was discovered during the proof reconstruction of the implicit certificate.

Fortunately, it is a simple matter to do just such parallel checking of two proof certificates. The full specification (using λProlog syntax) of such a process is given in Figure 5.1. In the figure, `<c>` is an infix constructor of type `cert -> cert -> cert` and `<i>` is an infix constructor of type `index -> index -> index`. This pairing operation allows for the parallel checking of two certificates: at each step, both certificates must *agree* to allow the proof to progress. Essentially, the pairing FPC can be seen as a *meta-FPC* or an *FPC combinator* that takes two FPC definitions and combines them into one: each clerk and expert calls the corresponding clerk and expert from each certificate in the pair. The pairing construct is *composable*: each half of a pairing can itself be another, nested pairing.

For a pairing to be useful, the two certificates included in it must eventually be able to expand into the same underlying sequent calculus proof, but those certificates could retain different amounts of detail from each other—or different kinds of information. At each application of an inference rule, the two halves of a pairing must agree to allow proof construction to proceed (through success of their respective clerks or experts). In addition, they must agree on the *positive choices* that certificates relay to the kernel in order to find a proof. This *concrete interface* manifests directly in three pieces of information: (a) substitution terms for existential quantifiers; (b) choices for (positive) disjunctions; and (c) cut formulas. A fourth piece is necessary through indirection: while paired certificates need not agree on the notion of index—the pairing constructor `<i>` is used to form an index out of two indexes—the pair of indexes must be able to agree on a single formula on which to focus.

5.2.2 Example In Example 5.2.1, two variants of a binary resolution certificate are used, one that does not include substitution information and one that does. The key difference is their treatment of the existential expert, where the more explicit certificate provides a witness term t—in the example, the single addition of z for the bound x in clause 2—and the more implicit certificate leaves a hole in the form of a logic variable to be instantiated at a later point by unification. The definition of pairing for the existential expert ensures that both certificates agree on the witness. Agreement in this case is trivial: the logic variable in the implicit certificate gets immediately instantiated with the substitution term when both are unified by the pairing expert. Thus, it is possible to transform a proof certificate

```
type    <i>        index -> index -> index.
type    <c>        cert -> cert -> cert.
infix   <i>, <c>   5.

cutE     (A <c> B) (C <c> D) (E <c> F) Cut  :- cutE     A C E Cut,  cutE     B D F Cut.
allC     (A <c> B) (x\ (C x) <c> (D x))     :- allC     A C,        allC     B D.
andNegC  (A <c> B) (C <c> D) (E <c> F)      :- andNegC  A C E,      andNegC  B D F.
andPosE  (A <c> B) (C <c> D) (E <c> F)      :- andPosE  A C E,      andPosE  B D F.
decideE  (A <c> B) (C <c> D) (I <i> J)      :- decideE  A C I,      decideE  B D J.
falseC   (A <c> B) (C <c> D)                :- falseC   A C,        falseC   B D.
initialE (C <c> B) (I <i> J)                :- initialE C I,        initialE B J.
orNegC   (A <c> B) (C <c> D)                :- orNegC   A C,        orNegC   B D.
orPosE   (A <c> B) (C <c> D) E              :- orPosE   A C E,      orPosE   B D E.
releaseE (A <c> B) (C <c> D)                :- releaseE A C,        releaseE B D.
someE    (A <c> B) (C <c> D) W              :- someE    A C W,      someE    B D W.
storeC   (A <c> B) (C <c> D) (I <i> J)      :- storeC   A C I,      storeC   B D J.
trueE    (A <c> B)                          :- trueE    A,          trueE    B.
```

5.1 Figure The pairing meta-FPC. Signature and module names and accumulations of kernel signatures are omitted. Core declarations are assumed.

encoding resolution that does not contain substitution terms to one that does contain substitution terms. The reverse is also possible.

In a pairing construct, the more *restrictive* choice, or subset of choices, dominates each step of proof checking. Nonetheless, uses of pairing are not limited to zooming in and out on more implicits or explicit versions of the "same" proof certificate, as in the last example. Proof certificates can also be thought of as more general *proof search tactics* implementing more or less general strategies, and pairing these certificates as a modular method of tactic composition.

5.2.3 Example Consider the FPC definition that limits proof search by decide depth only, defined in Section 3.5. This constraint can be easily combined with other, independent strategies, as these two applications show:

1. The CNF search procedure for propositional logic from Section 3.3 can generate very large proofs. To find out whether a relatively shallow and quick proof exists, `cnf` and `dd`—at the desired decide depth—can be paired and checked together. This pairing works because the clerks and experts of both FPCs are compatible: they allow the same connectives to proceed without restrictions, and the indexes used to store and decide on formulas are also compatible.

2. In combination with the semantics of the kernel, `dd` performs depth-first search, which can lead to much longer search times when shallow proofs avail. It is a simple matter to obtain *iterative deepening* by pairing `dd` with a pseudo-FPC that generates integers in increasing order at the root of the proof and communicates those integers to `dd` through a logic variable.

With some support from the logic and the programming language, along with a growing library of FPC definitions, a sophisticated treatment of FPCs as a general-purpose formalism arises. This perspective is explored in Chapter 13.

While the transformations between proof certificates that can take place using the pairing FPC are useful—as we argue throughout the chapter—the extent of such transformations is also limited. For example, pairing cannot be used to transform a proof certificate based on, say, conjunctive normal forms, into one based on resolution, since the former makes no use of cut and the latter contains cuts. The pairing of two such certificates will (almost) always fail to succeed. The fundamental limitation of pairing as a means of transforming proofs lays within the spectrum of "many details, fewer details" and not between two different styles of proof. This is the topic of the next section.

5.3 Elaboration and distillation

When it checks a certificate, a kernel is building a formal sequent calculus proof which is not explicitly in the certificate, stored but is, in a very real sense, performed by the kernel. It is the fact that such a sequent calculus proof is being built that helps to provide trust in the kernel. If a certificate lacks necessary details for building such a sequent calculus proof—for example, substitution instances as in Example 5.2.2—a kernel could attempt to reconstruct those details. (Indeed, the kernel does exactly that in the case of binary resolution where substitution instances are not part of the certificate.)

The formal pairing of certificates described in the previous section connects two certificates that lead to the performance of the same sequent calculus proof. In the logic programming setting, it is completely possible to see such linking of certificates as a means to transform one certificate into another certificate. We use the term *elaboration* to refer to the process of transforming an implicit proof certificate into a more explicit proof certificate. The converse operation, which we call *distillation*, can also be performed: during such an operation, certain proof details can be discarded and a more implicit proof is produced.

Since a given proof certificate can be elaborated into a number of different sequent calculus proofs, certificates can be used to provide high-level descriptions of *classes* of proofs: `cnf` and `dd` are examples of such classes—informed, of course, by the formulas to be proved—and other important classes are discussed in Chapter 11.

5.3.1 Example   Following Part (2) of Example 5.2.3, we can illustrate the concept of classes of proofs by the following pairing:

$$\texttt{cnf <c> (dd N)}$$

If such a combined reconstruction is possible for a given formula, pairing the proof checking of the CNF decision procedure with a more explicit form of FPC, here decide depth, would mean that the missing proof details—namely, the decide depth of the proof, signified by the logic variable `N`—could be recorded.

In a similar fashion, the notion of *obvious logical inference* by Davis (1991) can be described easily as an FPC: here, an inference is "obvious" if all quantifiers are instantiated at most once; thus, using a kernel to attempt to check such an FPC against a specific formula essentially implements the check of whether or not an "obvious inference" can complete the proof.

Distillation, the complement of elaboration, also plays an important role in the practical manipulation of proof certificates. Consider, for example, a proof certificate that contains substitution instances for all quantifiers that appear within a proof (such as the resolution certificate with substitution information in Example 5.2.1). In some situations, such terms might be large and their occurrences within a certificate could make the size of this certificate explode—but the *usefulness* of this substitution information may vary. Namely, in the first-order logic setting, if a certificate stores instead linkage or mating information between literals in a proof, then the implied unification problems can be used to infer the missing substitutions—assuming that the kernel contains a trusted implementation of unification (for a discussion of related matters, see Chapter 6). The resulting certificates, where derivable substitutions are omitted, could be much smaller: checking these compressed certificates could, however, involve possibly large unification problems to be performed. The usual space-time tradeoffs apply; experimental coverage can be found in Section 8.5.

Besides applications to proof compression such as those, distilling can provide an elegant way to answer questions such as: What lemmas have been used in this proof? How deep (counting decide rules) is a proof? What substitution terms are used in a certain subproof? That is, it can be used as a framework to formulate *proof queries* to extract information from a proof. Certificates that retain only some coarse information, like the ones studied so far, can be used to provide some high-level insights into the structure of a given proof. Moreover, the next section provides a means of composing general queries which may involve information that is missing from a given proof certificate.

5.4 Maximally explicit FPCs

We can define a *maximally explicit* FPC that contains all the information that is explicitly needed to fill in all details in the augmented inference rules of a sequent calculus that implements the Foundational Proof Certificate framework. For the *LKF*^{a} proof system, the corresponding maximally explicit FPC (sometimes referred to as *maximally elaborate*) is given in Figure 5.2. Such an FPC represents an exhaustive trace of a proof tree.

The FPC definition is structured as follows. A top-level constructor, max, pairs a natural index with the symbolic representation of a proof tree. This wrapper is propagated to all continuation proof sub-trees and serves to assign

```
kind max type.
type ix                    nat -> index.
type max           nat -> max -> cert.
type max0                        max.
type max1                 max -> max.
type max2          max -> max -> max.
type maxa               index -> max.
type maxi       index -> max -> max.
type maxv           (tm -> max) -> max.
type maxt             tm -> max -> max.
type maxf  form -> max -> max -> max.
type maxc        choice -> max -> max.

allC     (max N (maxv C ))        (x\ max N (C x)).
andNegC  (max N (max2 A B))       (max N A) (max N B).
andPosE  (max N (max2 A B))       (max N A) (max N B).
cutE     (max N (maxf F A B))     (max N A) (max N B) F.
decideE  (max N (maxi I A))       (max N A) I.
storeC   (max N (maxi (ix N) A))  (max (s N) A) (ix N).
falseC   (max N (max1 A))         (max N A).
orNegC   (max N (max1 A))         (max N A).
releaseE (max N (max1 A))         (max N A).
orPosE   (max N (maxc C A))       (max N A) C.
someE    (max N (maxt T A))       (max N A) T.
trueE    (max N  max0).
initialE (max N (maxa I)) I.
```

5.2 **Figure** A maximally elaborate FPC. Signature and module names and accumulations of kernel signatures are omitted. Core declarations are assumed. Indexes are drawn from the usual inductive definition of natural numbers, nat here, where s denotes the successor.

increasing indexes to formulas as they are stored, so that the storage zone is (along each branch from the root of the tree) a *functional mapping* from naturals to formulas, making decide rules unambiguous. max is also the name of the type of *symbolic proof trees*, each of whose constructors represent different types of nodes, holding all information needed by the clerks and experts without recording the actual proof derivation. Each constructor represents a type of node in a proof tree: max0 is a leaf node; max1 is a simple unary node; max2 is a simple binary node; maxv is a unary node used to bind an eigenvariable to the rest of the tree; maxt is a unary node annotated with a term; maxf is a binary node annotated with a cut formula; maxc is a unary node annotated with with a (disjunctive) choice; and maxi is a unary node annotated with an index.

The encoding of the maximally elaborate FPC mirrors exactly the structure of the sequent calculus in Figure 3.1: thus, there is a one-to-one correspondence between proofs of LKF^a and these maximally elaborate certificates, modulo indexing conventions. The operation of maximal elaboration can also be seen as *injecting* or *recording* a *trace* of an *LKF* proof in the FPC framework. Typical usage will have a certificate variable paired with another certificate Ξ driving the proof proper via the standard idiom:

$$\Xi \texttt{<c>} \texttt{(max 0 Max)}$$

Here, the first index is given as zero—which we take the liberty of writing in standard numeric notation—, but this choice is inconsequential: unique indexes will be asigned to each stored formula starting from this value. It is fundamental that the proof tree to be recorded (represented by the logic variable `Max`) be injected into the family of certificate constructors for the maximally elaborate FPC via the top-level injector `max`, so that the appropriate clerks and experts may be selected. In short, such a fully explicit proof certificate can be automatically obtained through elaboration of any other proof certificate and the use of the pairing of certificates.

An important second use of maximally explicit FPCs is the *checking* of proof certificates, here really representing full proofs. In exchange for a comparatively large certificate size, checking becomes a determinate operation that can be performed very efficiently. Since all choices are stored in the certificate, proof checking becomes a *purely functional computation*. This has significant implications on the trust model of proof checkers as well as their implementation; these aspects are discussed at length in Chapter 6.

At the beginning of the section, we proposed the definition of "a" maximally explicit FPC. Indeed, figreffpc-maximal is not the only possible definition that constitutes an accurate trace of a proof tree. A more direct, slightly less compact definition (in terms of declarations) will replace the conflated node constructors of type `max` with one constructor of the appropriate type for each logical connective. For example, instead of `max2`—shared by both conjunctions—we will have:

```
type    maxAndNegC, maxAndPosE    max -> max -> max.
```

All other conflated encodings are similarly unfolded. The resulting definition is equivalent to the original, maximally elaborate FPC. The former, while more

pedantic in its recording of logical connectives in the tree, does not contain more information than the compact definition because the inference rule at a node is uniquely determined by the its conclusion. Other determinate FPCs can be given, say, recording only synchronous choices while leaving the asynchronous phase implicit. For a discussion of determinacy in FPC definitions, see Chapter 6.

A third use of maximally elaborate FPCs is *querying through distillation*, introduced in the previous section. A certificate where all proof information is explicitly available allows the extraction of arbitrary information about the proof object which said certificate represents. A general workflow may involve elaborating the source certificate into its maximal form and distilling the desired information from this intermediate representation, where said information is guaranteed to be available.

Maximally explicit certificates are trivially defined for other logics with minor variations based on the types of nodes in the proof tree, each annotated with each piece of information output by clerks and experts.

5.5 Experiments

We have experimented with various uses of certificate pairing and we report briefly on some of those experiments here. In particular, we have used pairing in our λProlog checker in order to distil and elaborate a number of matrix-style, i.e., cut-free, proofs. A few representative instances based on our case studies are:

- Propositional CNFs elaborate to matings (Andrews, 1981).

- Decision depth bounds elaborate to oracles (Chihani et al., 2016b, Section 7).

- Propositional CNFs, matings, decision depth bounds and oracles elaborate to maximal certificates.

All these pairings work in reverse as distillations, with the proviso that a maximally elaborate certificate distills to certificate formats which are compatible with the proof from which they were generated. Thus, the maximal elaboration of an oracle certificate (which operates on the positive phase) cannot be distilled as a mating tree (which operates on the negative phase) even if proofs in both formats exist. This illustrates the fundamental limitation of pairing.

The ensemble of operations works as expected, but the formats mentioned in this section cannot easily be employed beyond small numbers of moderately sized

examples, since these formats are seldom used in actual theorem provers. More extensive experiments must involve proof certificates where the cut rule is allowed, which furthermore will allow us to check the output of real software provers, both automated and interactive. For more details, see Chapters 7, 8, and 11.

5.6 Notes

The developments in this chapter have been presented in Blanco et al. (2017a), though earlier work, including Blanco and Miller (2015), anticipates the interest of combining and extracting information from proofs—expressed as certificates.

An application of the pairing FPC involves the reconstruction of a certain focused proof by two separate proof certificates. The definition in Figure 5.1 creates a pairing from two separate terms of the uniform certificate type `cert`. Contingent on the definition of the paired FPC definitions, this confusion can create ambiguity when attempting to elaborate a certificate of a certain kind, say A, to another one, say B, represented by a logic variable. To avoid mixed certificates combining parts from separate definitions, the clerks and experts in the B family must unambiguously constrain continuation certificates to their own family. A more type-based result could be achieved by defining the pairing constructor `<c>` as combining two disjoint certificate types: its type would be `certA -> certB -> cert`. The language should offer the option to alias families of certificates to either of the paired types—recent versions of Teyjus implement this experimental feature. Of course, both `certA` and `certB` could themselves be pairs of other FPCs.

The matings referred to in Section 5.3 are related to expansion trees, representations of proofs studied by Miller (1987); Chaudhuri et al. (2016). This paradigm has also been given expressin in the FPC framework.

As we noted already in Chapter 2, the advent of computer-aided verification enables proofs which by their sheer complexity are far beyond what manual calculation can reasonably achieve. In particular, automated theorem provers—commonly based on resolution calculi (for which see Section 3.6) and encoding theorems as instances of satisfiability problems (discussed at length in Chapter 7)—are prone to generate enormous, low-level proof descriptions. The current largest reported proof was obtained by such methods by Heule et al. (2016) and consists of an unsatisfiability certificate almost 200 TB in size, itself not a full elaboration

of a proof object. Substantial effort goes not only into producing, but into mechanically checking the correctness of such claims (Cruz-Filipe et al., 2017b).

While the discussion in this chapter is limited to classical first-order logic, the FPC framework is applied to other frameworks, such as intuitionistic logic and logics extended with least and greatest fixed points. Certificate pairing and maximally elaborate certificates are equally applicable to all these—and will be applied in various settings in successive chapters.

6 Determinate checkers

6.1 Trust and determinate FPCs

The FPC framework was originally proposed and designed to give answer to two needs: *communicating and trusting proofs* (Miller, 2014). The separation between untrusted provers and trusted checkers plays a vital role in this endeavor—as a matter of fact, it is the subject of the very first requirement in Miller's desiderata (Definition 3.1.1). It has been convincingly argued that a (higher-order) logic programming language like λProlog is a good choice to implement the augmented focused sequent calculi that embody the FPC framework. The resulting checkers are both elegant and powerful, and encode the proof systems they implement so faithfully and tersely that it is a simple matter to sanction them as *correct by construction*, and thus trustworthy. The present chapter is dedicated to studying and sharpening this assertion and the very concept of trusted checkers.

Firstly, we consider the implications of the general FPC architecture on its necessary *trusted computing base* (TCB). In a wide sense, the TCB comprises such varied elements as a hardware platform, an operating system and a compiler or interpreter. In the present day, it is not yet feasible to assemble a system where every component is formally verified: exemplars are few and fairly specific. What is more, conformance to a specification does it itself guarantee that the specification constitutes a *correct* and *complete* description of the system: that is, *verification* does not entail *validation*. Yet even admitting all these components into the TCB—a practical necessity—, we may question the choice of programming environment and its impositions on the TCB.

As a matter of fact, higher-order logic programming comes with a very characteristic set of features: sophisticated operations like variable binding and substitution, higher-order unification, and backtracking search, are all available as language primitives. One may eye these complex operations with suspicion and

wonder whether these must be admitted into the primeval TCB as the unavoidable price to pay, or whether, say, the simpler computational model of pure functional programming offers a viable alternative whose implementation is easier to come to trust. At the same time, we may note that it is precisely those features of logic programming that make the encoding of kernels so straightforward and easy to trust—provided that the implementation of said features in a logic programming language are, indeed, correct. Furthermore, Miller's desiderata appear to mandate the use of logic-like features in any *full* implementation of the FPC framework.

Section 5.4 anticipated a solution to this problem in the form of *determinate proof certificates*, which contain enough information about the proof they represent so as to render proof checking determinate. Such certificates could be passed as inputs to a *simplified checker* implemented in a functional programming language where none of the complicating signature features of logic programming are present. A priori, such a simplified checker should be even simpler to trust while implementing the determinate fragment of the FPC framework. Full generality could be restored by checking an arbitrary certificate with a general, logic programming-based checker, while using pairing to elaborate a determinate certificate, which could be then exported and re-checked against the same formula using a determinate checker. An architecture like this will be discussed in Chapter 8. Before that, the construction of such simplified checkers needs to be addressed.

Secondly—and orthogonally to the aforementioned minimization of the TCB of the FPC framework—the informal *correctness argument* by which an implementation of a checker may be declared sound and "obviously correct" may be brought under scrutiny. Given that the integrity of the entire framework rests on the correctness of this critical piece of software, close examination is warranted. While it has been posited that proof checkers are amenable to formalization, the informal argument has been treated as satisfactory, and no verification efforts have been undertaken. A hidden source of complexity lies precisely in the qualities of logic programming languages that would need to be modeled and proven correct inside a proof assistant. Nevertheless, a determinate checker would not face these obligations and could offer an approachable starting point.

Finally, we note by way of illustration that the notion of determinacy has considerable depth, beyond the proof traces that are its clearest expression. It is possible—at least in some logical settings—to leave out certain details from a proof certificate while still providing for determinate proof checking. For example, consider the variant of the maximally explicit FPC in which no substitution terms

```
type maxv    max -> max.
type maxt    max -> max.

allC  (max N (maxv C)) (max N C).
someE (max N (maxt A)) (max N A) T.
```

6.1 **Figure**   A variation on the maximally explicit FPC of Figure 5.2. The types and clauses given here replace the uses of `maxv` and `maxt` in that figure. In this version, first-order unification provides determinate computation of witness terms, and eigenvariables are managed fully by the kernel.

are stored in the certificate. Specifically, we redefine two pieces of information: (a) the certificate constructors previously assigned to eigenvariable bindings and substitution terms, used respectively to treat the universal and the existential quantifier; and (b) the clerk and expert predicates that linked those constructors with their corresponding connectives. The changes with respect to Figure 5.2 are shown in Figure 6.1.

Certificates of this modified format will not contain any reference to eigenvariables or to substitution terms (existential witnesses). A proof checker for such certificates can, however, use so-called *logic variables* instead of explicit witness terms and then perform unification during the implementation of the initial rule. Since the unification of first-order terms (even in the presence of eigenvariables and their associated constraints) is determinate, such proof checking will not involve the need to perform backtracking search. The main downside for this variant of the maximally explicit certificate is that checking will involve the somewhat more complex operation of unification. Of course, such unification must deal with either Skolem functions or eigenvariables in order to address quantifier alternation—λProlog treats eigenvariables directly since it implements unification under a mixed quantifier prefix (Miller, 1992).

The rest of the chapter is organized as follows. Section 6.2 presents the implementation of a functional checker for determinate certificates in the OCaml programming language. Section 6.3 addresses the verification of a determinate checker specialized for the maximally elaborate certificates of Section 5.4 in the Coq proof assistant. Section 6.4 considers the interface of these checkers with outside tools. Section 6.5 discusses the use of proof checkers and proof certificates as an extension of the repertoire of tactics available to proof assistants. Section 6.6 concludes the chapter.

6.2 Functional checkers in OCaml

As noted in Section 5.4, a sufficiently detailed certificate turns certificate checking into a behaviorally determinate process, which can be performed by a kernel programmed in a functional language without side effects. Such a checker will be simpler and potentially easier to analyze and trust. A maximally explicit certificate contains all the information needed to build a proof in its focused sequent calculus. In particular, all don't-know nondeterminism is given a definite answer in the certificate, so that unification and backtracking search are not utilized by the checker.

   To demonstrate this possibility, we implement a determinate proof checker as an OCaml program, called MaxChecker. It can be used with any determinate certificate definition, for which the following module implementations need to be given by the client:

1. A *certificate type* defining the certificate constructors.

2. An *index type* defining the index constructors, together with a comparison function (used by the context module to file formulas into storage).

3. An implementation of clerks and experts according to the specification of a *bureau module*. Each clerk and expert takes a certificate as input and returns an option type formed by continuation certificates and all necessary information: indexes, existential witness terms, etc. (As a special case, the true expert returns a boolean instead of an option type of unit.)

   The maximally explicit certificate of Figure 5.2 is one such definition. In addition to an FPC definition, a concrete instance of the kernel—and its concrete certificates—rely on a problem specification on which formulas are parameterized, as well as certificates, insofar as they can include formulas or some of their components. A problem signature is characterized by the following pair of client-side definitinions:

1. A *term type*, which could be empty in propositional formulas, where only atoms are used. The term type can be recursive, and term constructors are only expected to have term parameters.

2. An *atom type*, parametric on a term type. Atom constructors are only expected to have parameters of this term type.

The public interface of the checker is simply an entry function that takes a certificate and a formula as arguments and returns a boolean indicating whether the certificate represents a determinate proof of the formula. This is translated into a call of the main function, which takes a certificate and a sequent and returns a boolean if and only if the certificate is able to prove the sequent; Figure 6.2 shows this function. The structure of the checker closely resembles λProlog-based kernels, with the difference that all pattern matching operations are structured in the same function instead of divided in program clauses.

Each inference rule is delegated to its associated auxiliary function (clerk or expert), which is charged with calling the corresponding clerk or expert from the FPC definition used to instantiate the kernel and, if successful, performs the requisite recursive calls to the main function (with which all these helpers are mutually recursive). Figure 6.3 presents an representative selection of some of the more interesting functions. These illustrate in more detail a number of representative design constraints:

1. In the logic programming encoding, universal and existential quantifiers envelop a *formula abstraction* that becomes a formula after a term is applied to it: an eigenvariable for the universal quantifier and a term (fully unified or not) for the existential quantifier. In OCaml, the argument of both quantifiers is a *function* that takes a term argument and returns a formula; moreover, term arguments are always fully defined.

2. Terms at the kernel level can be either client-side terms, defined by the user, or eigenvariables. The latter are a separate kind of term, created fresh when the kernel encounters a universal quantifier, then passed to the formula function. In this kernel, they are never unified. Elements provided by the user (i.e., formulas and certificates) cannot contain eigenvarible terms.

3. The storage zone becomes a functional context where each index may occur once. An attempt to store a formula under an already claimed index results in an error. It is the responsibility of the certificate to ensure functionality of the indexing scheme.

Finally, an example function from the bureau that implements the maximally explicit FPC definition is given in Figure 6.4; all clerk and expert functions have simple definitions in the same style.

There is an important precision to make: at the end of the asynchronous phase, the decide and cut rules are in *conflict*: that is, the sequent does not

```ocaml
let rec check certificate = function
  | Unfocused(storage, workbench) ->
    (match workbench with
    | [] ->
      if decide_cut_conflict certificate then
        failwith "Decide_and_cut_are_in_conflict"
      else
        let check_decide = decide_expert certificate storage
        and check_cut = cut_expert certificate storage in
        check_decide || check_cut
    | hd :: tl ->
      (match hd with
      | NegativeFalse ->
        false_clerk certificate storage tl
      | NegativeAnd(left, right) ->
        and_clerk left right certificate storage tl
      | NegativeOr(left, right) ->
        or_clerk left right certificate storage tl
      | NegativeTrue ->
        true
      | ForAll(formula) ->
        forall_clerk formula certificate storage tl
      | _ ->
        store_clerk hd certificate storage tl
      )
    )
  | Focused(storage, workbench) ->
    (match workbench with
    | PositiveTrue ->
      true_expert certificate
    | PositiveAnd(left, right) ->
      and_expert left right certificate storage
    | PositiveOr(left, right) ->
      or_expert left right certificate storage
    | Exists(formula) ->
      exists_expert formula certificate storage
    | PositiveAtom(atom) ->
      init_expert atom certificate storage
    | PositiveFalse ->
      false
    | _ ->
      release_expert workbench certificate storage
    )
```

6.2 **Figure**   The MaxChecker kernel written in OCaml. For display purposes, the store and release rules make use of catch-all matches instead of exhaustive listings. check is defined with mutually recursive clerk and expert handlers.

```
and forall_clerk formula certificate storage workbench =
  match Lkf_bureau.all_clerk certificate with
    | None -> false
    | Some certificate' ->
      let eigenvariable = Lkf_term.eigenvariable () in
      let certificate_filled = certificate' eigenvariable
      and formula_filled = formula eigenvariable in
      check certificate_filled
        (Unfocused(storage, formula_filled :: workbench))

and store_clerk formula certificate storage workbench =
  match Lkf_bureau.store_clerk certificate with
    | None -> false
    | Some (certificate', index) ->
      let storage' = Lkf_context.add index formula storage
      in check certificate' (Unfocused(storage', workbench))

and exists_expert formula certificate storage =
  match Lkf_bureau.exists_expert certificate with
    | None -> false
    | Some (certificate', term) ->
      let formula_filled = formula term in
      check certificate' (Focused(storage, formula_filled))

and init_expert atom certificate storage =
  match Lkf_bureau.init_expert certificate with
    | None -> false
    | Some index ->
      (match Lkf_context.find index storage with
      | None -> false
      | Some formula ->
        formula = NegativeAtom(atom)
      )
```

6.3 **Figure** The MaxChecker interface to FPCs in OCaml. Shown here are some especially interesting cases. All definitions are defined in mutual recursion with the kernel as defined in Figure 6.2.

```
let decide_expert = function
  | Index(index, next) -> Some (next, index)
  | _ -> None
```

6.4 **Figure** Definition of clerks and experts of the maximally elaborate FPC against the MaxChecker bureau interface, as called in Figure 6.3. All definitions follow the pattern exemplified here.

uniquely determine a unique inference rule that can be applied—subject to the certificate term and the clerks and experts. In general, the kernel must look at both bureau functions and ascertain that at most one allows the proof to proceed with the present certificate and, if one such function avails, use it, otherwise the proof cannot continue. If the kernel is specialized to work with a fixed FPC definition, it is possible that the two rules are never in conflict, and the specialized checker integrating such an FPC definition could be simplified accordingly. The maximally explicit FPC satisfies this property of absence of conflict.

The resulting program is remarkably compact: the complete kernel is slightly over 150 lines of code, and the bureau for the maximally explicit FPC definition about 50 lines of code, both formatted for legibility, not compactness. The handful of supporting type modules referred to in the discussion (formulas, contexts, sequents, etc.) are equally succinct. The ensemble of modules and usage examples adds up to 350 lines of code. This is an auspicious starting point for formalization efforts, given that (in particular) the maximally explicit certificate contains all the information needed to build a focused sequent calculus proof, and the proof checker is a terminating program performing purely functional computation without any complex operations (like unification and search). The next section continues this line of research by performing the verification of a checker for propositional classical logic.

6.3 Verified checkers in Coq

The Coq proof assistant (Bertot and Castéran, 2004) can be used to program a determinate proof checker as a purely functional, terminating program. This program can then be verified by proving the appropriate theorems on it, in particular the *soundness theorem*. Since we will only be utilizing one particular FPC format with this checker (i.e., the maximally explicit FPC), we shall consider the presentation of the specialized checker where the FPC definition is embedded in the kernel. This will remove the obligation to prove the soundness of the kernel for any conceivable FPC definition; this choice will remove some inessential reasoning clutter from the development. In this section, we present a formalization from first principles with an aim to uncovering the fundamental complexity of the problem, without any dependencies on libraries and complex external results. The port from OCaml is generally straightforward; two specific design choices merit discussion.

To commence, consider the inductive type of polarized formulas for the propositional fragment of *LKF* where non-logical constants, i.e., atoms, are represented by the type of propositions in Coq, `Prop`. The negation of formulas and both unfocused and focused sequents are defined in the usual manner. The choice of `Prop` as the type of atoms is the most flexible and it is advantageous from an internal perspective: if the checker proves a formula, its depolarization trivially yields a Coq proposition which can then be considered proven—as a matter of fact, *certified*—and used normally inside Coq. The tradeoff is that the checker can no longer yield a simple boolean as an answer, because the initial rule involves equality between atoms (i.e., Coq propositions) which is not decidable in general. Therefore, the checker must operate modulo equality between `Prop`s and return a proposition consisting of these terminal equalities. In successful runs, those equalities will be identities, and the proposition trivially true.

The second important choice is the representation of the *indexed storage* in both types of sequents. As we know from the OCaml implementation in Section 6.2 and, further still, from the definition of the maximally elaborate FPC in Section 5.4, unique indexes can be drawn from an incrementing counter from the root to each point in the proof tree—that is, unicity is enforced along each branch. Instead of using a partial map library, we exploit these observations by modeling the storage as a list. The index of an element is its position in the list, so that a fresh index is assigned by appending a formula at the end of the list. This representation, while inefficient, can be easily swapped—as the proofs will make clear, the few lemmas required by the main result are all easy, a fact that the simple list representation makes particularly clear.

Given these considerations, we can encode the determinate checker as a simple fixed point definition, presented in Figures 6.5 and 6.6. Unlike in the OCaml checker, the maximally elaborate FPC is fixed and embedded in the checker: there are no helper functions; instead, the pattern matching of each clerk and expert and subsequent processing are all inlined. The resulting code is remarkably succinct and predictably decreasing on the size of the certificate.

To begin to prove the soundness of the checker, we need to relate the polarized formulas of the FPC framework with logical connectives of propositions in Coq, both in isolation and as part of a sequent. As we follow the one-sided presentation of *LKF*, the latter case will correspond to the disjunction of the parts—in any case, the proofs will guarantee the correctness of this connection.

```
Fixpoint check (cert : certificate) (seq : sequent) : Prop
  := match seq with
    (* Decide and cut. *)
    | store // [] ⇒
      match cert with
        | Index1 (Idx idx) next ⇒ (* Decide. *)
          match nth_error store idx with
            | Some form ⇒ next ⊢ (store \\ form)
            | None ⇒ False
          end
        | Cut form nextp nextn ⇒
          (nextp ⊢ (store // [form])) ∧
          (nextn ⊢ (store // [negate form]))
        | _ ⇒ False
      end
    (* Negative phase. *)
    | store // (False- :: work) ⇒
      match cert with
        | Cert1 next ⇒ next ⊢ (store // work)
        | _ ⇒ False
      end
    | store // ((forml ∧- formr) :: work) ⇒
      match cert with
        | Cert2 nextl nextr ⇒
          (nextl ⊢ (store // (forml :: work))) ∧
          (nextr ⊢ (store // (formr :: work)))
        | _ ⇒ False
      end
    | store // ((forml ∨- formr) :: work) ⇒
      match cert with
        | Cert1 next ⇒
          next ⊢ (store // (forml :: formr :: work))
        | _ ⇒ False
      end
    | _ // (True- :: _) ⇒ True
    (* Store. *)
    | store // ([_]- as form :: work)
    | store // (True+ as form :: work)
    | store // (False+ as form :: work)
    | store // ([_]+ as form :: work)
    | store // ((_ ∧+ _) as form :: work)
    | store // ((_ ∨+ _) as form :: work)
      ⇒
```

6.5 Figure   The classical propositional MaxChecker kernel and FPC in Coq. Po-
larized logical connectives, as well as atoms (here, arbitrary propositions, in
brackets), follow Coq's standard notation with + and − suffixes. Sequents are
one-sided, ⇑ is marked // and ⇓ is marked \\.

```
      match cert with
        | Index1 (Idx idx) next ⇒
          if beq_nat (length store) idx
          then next ⊢ ((store ++ [form]) // work)
          else False
        | _ ⇒ False
      end
    (* Positive phase. *)
    | store \\ True+ ⇒
      match cert with
        | Cert0 ⇒ True
        | _ ⇒ False
      end
    | store \\ (forml ∧+ formr) ⇒
      match cert with
        | Cert2 nextl nextr ⇒
          (nextl ⊢ (store \\ forml)) ∧
          (nextr ⊢ (store \\ formr))
        | _ ⇒ False
      end
    | store \\ (forml ∨+ formr) ⇒
      match cert with
        | Choice Left next ⇒ next ⊢ (store \\ forml)
        | Choice Right next ⇒ next ⊢ (store \\ formr)
        | _ ⇒ False
      end
    | store \\ [atomp]+ ⇒
      match cert with
        | Index0 (Idx idx) ⇒
          match nth_error store idx with
            | Some [atomn]- ⇒ atomp = atomn
            | _ ⇒ False
          end
        | _ ⇒ False
      end
    | _ \\ False+ ⇒ False
    (* Release. *)
    | store \\ True- as form | store \\ False- as form
    | store \\ [_]- as form | store \\ (_ ∧- _) as form
    | store \\ (_ ∨- _) as form
      ⇒
      match cert with
        | Cert1 next ⇒ next ⊢ (store // [form])
        | _ ⇒ False
      end
  end
```

6.6 Figure The classical propositional MaxChecker kernel and FPC in Coq (continued). Presentation conventions are shared with Figure 6.5.

6.3.1   **Definition**   Let $[\![\cdot]\!]^0$ be a *depolarization function* that maps polarized to unpolarized formulas. It assumes that the negative polarity is reserved for negated atoms. It is defined as follows:

$$[\![t^+]\!]^0 = [\![t^-]\!]^0 = t$$
$$[\![f^+]\!]^0 = [\![f^-]\!]^0 = f$$
$$[\![P_a]\!]^0 = a$$
$$[\![N_a]\!]^0 = \neg a$$
$$[\![A \wedge^+ B]\!]^0 = [\![A \wedge^- B]\!]^0 = [\![A]\!]^0 \wedge [\![B]\!]^0$$
$$[\![A \vee^+ B]\!]^0 = [\![A \vee^- B]\!]^0 = [\![A]\!]^0 \vee [\![B]\!]^0$$

Here, P_a is the atom a positively polarized, N_a is the atom a negatively polarized, and A and B are arbitrary polarized formulas. The notion of depolarization function is easily extended to focused sequents. In *LKF*, one-sided sequents are interpreted as classical disjunctions, and their depolarization is defined as:

$$[\![\vdash \Gamma \Uparrow \Theta]\!]^0 = \left(\bigvee_{A \in \Gamma} [\![A]\!]^0 \right) \vee \left(\bigvee_{B \in \Theta} [\![B]\!]^0 \right)$$

$$[\![\vdash \Gamma \Downarrow B]\!]^0 = \left(\bigvee_{A \in \Gamma} [\![A]\!]^0 \right) \vee [\![B]\!]^0$$

The same scheme is valid for *LKF*^{a}, where the index assigned to each formula in the storage area Γ is quietly discarded.

An implementation of a checker must adapt the depolarization function on sequents to operate on what data structures are used to implement the various zones. In the present case, only lists are used.

In order to prove the main result of the soundness of the checker, we make use of a small number of helper lemmas. The first property is a sort of pseudo-excluded middle applied to the depolarization function.

6.3.2   **Lemma**   Let B be a polarized formula and $\neg B$ its negation. It cannot be the case that both their depolarizations, $[\![B]\!]^0$ and $[\![\neg B]\!]^0$, hold.

Proof.   By a simple case analysis on the structure of B.   □

The remaining auxiliary results are technical lemmas used to relate a particular implementation of the storage zone, its interpretation as a set of formulas at the

sequent level, and the depolarization of this fraction of the sequent as a disjunction of (unpolarized) formulas. While these lemmas are formulated in terms of lists in the present treatment, they are easily swappable with homologues for the indexed data structures of choice, as the abstract properties we require are simple.

6.3.3 Lemma   Given an indexed storage zone Γ and an index i, if the index addresses a formula B, $(i, B) \in \Gamma$, then the depolarization of the storage $[\![\Gamma]\!]^0$ is logically equivalent to the depolarization where the formula B is added as a disjunct: $[\![\Gamma]\!]^0 \vee [\![B]\!]^0$.

Proof. In our encoding, the index lookup corresponds to indexed access. The proof proceeds a simple induction on the structure of the storage zone Γ and the logical properties of disjunction. □

6.3.4 Lemma   Given an indexed storage zone Γ and an index i, if the index addresses a formula B, $(i, B) \in \Gamma$, and if the depolarization of the formula, $[\![B]\!]^0$, holds, then so does the depolarization of the entire zone, $[\![\Gamma]\!]^0$.

Proof. In our encoding, the index lookup corresponds to indexed access. The proof proceeds a simple induction on the structure of the storage zone Γ. □

6.3.5 Lemma   Given an indexed storage zone Γ and a formula B, if the depolarization of the storage zone augmented with the formula $[\![\Gamma \cup \{B\}]\!]^0$ holds, then so does the disjunction of the depolarization of the parts: $[\![\Gamma]\!]^0 \vee [\![B]\!]^0$.

Proof. In our encoding, storage corresponds to the append operation at the end of the list modeling Γ. The proof proceeds by a simple induction on the structure of the storage zone Γ. □

Armed with these results, we are ready to prove the principal theorem, which establishes the connection (the *implication*) between the checkability of LKF^a sequents and their corresponding depolarization as members of Coq's `Prop` type of propositions, `Prop`.

6.3.6 Theorem   Let Ξ be a maximally elaborate certificate and S be a sequent. If Ξ successfully certifies the sequent S via `checker`, then the depolarization of the sequent $[\![S]\!]^0$ holds.

Proof. The proof proceeds by structural induction on the certificate Ξ. Most cases follow directly from the induction hypotheses. The remaining cases, in addition, make use of the auxiliary results:

1. The case of cut makes use of Lemma 6.3.2.

2. The case of the initial rule, where the focus is on a positive atom, makes use of Lemma 6.3.3 along with the logical properties of disjunction and the excluded middle.

3. The case of the decide rule uses Lemma 6.3.4.

4. All variations of the store rule—applied to the various storable formulas—use Lemma 6.3.5.

This concludes the proof.   □

In trying to prove properties of a classical proof system (like *LKF*) in a constructive system (like Coq), the appeal to the axioms of classical logic is to be expected. Predictably, the axiom of the excluded middle makes a single appearance in the proof of the initial rule, where we look at an atom in both positive and negative polarities: back in Coq terms, a proposition and its negation.

In practice, a checker exposes a limited interface where a certificate is used to check a single formula, from which the initial sequent (for which recall Theorem 2.5.1) is derived. This property is a specialization from the general result:

6.3.7   Corollary   Let Ξ be a maximally elaborate certificate and B be a formula. If Ξ certifies the initial sequent $\vdash \cdot \Uparrow [B]$, then the unpolarized formula $[\![B]\!]^0$ holds.

Proof. Immediate from Theorem 6.3.6.   □

In the special case of certificates as *proof traces*, if the proof system is complete and if every proof has a trace in certificate form, a complementary theorem could be stated that, if a formula is provable, there must exist a certificate that checks it. This property goes beyond the full scope of the FPC framework and is not considered here.

One can imagine two possibilities to make practical use of a formally verified checker. First, a second standalone checker—like that in the previous section—could be extracted from Coq into, say, OCaml. Second, the checker could be used natively inside Coq as an alternative method for proof building. The next two sections discuss each of these possibilities in turn.

6.4 Extraction of verified checkers

The development of the proof checker in Coq constitutes an instance of *verified programming*, in which the code has been proven to satisfy its specification. To move beyond the boundaries of the proof assistant and become an independent executable program, code can be *extracted* to a functional programming language like OCaml (Letouzey, 2008). In terms of trust, we obtain correctness guarantees about the code by admitting additional systems into the trusted computing base. Specifically, those guarantees rely on the correctness of the proof assistant and on the procedure of code extraction, which themselves are not formally verified.

Moreover, the process of extraction comes with certain restrictions that interact with the design choices made in the previous section. In the first place, the use of Coq's native `Prop` renders code extraction inapplicable. In order to obtain extractable code, we need to, say, move from the world of propositions to the world of booleans—as in the original OCaml checker—, thus replacing logical connectives with boolean operations throughout the development. Furthermore, propositions-as-atoms must be replaced with a general model of atoms that can be extracted, say, based on strings (as is the case in some kernels written in λProlog). Such a model easily allows an extracted checker to be adapted to general problem signatures without having to translate those signatures into Coq code (the definition of an atom type in the native OCaml checker for each instance of the kernel is a representative of this approach).

Secondly, the possible interaction between any axioms of classical logic and the generation of purely functional, constructive code must be assessed. To begin with, code cannot be extracted from theorems that involve classical axioms, although in a `Prop`-based formalization this possibility is also precluded by the encoding of atoms. In a checker based on boolean return values where atoms are represented by a type with *decidable equality*, it is possible for the proofs to progress in a weaker setting. Even if some properties of classical logic are used to simplify reasoning in the final stages of the proof, extracting compilable code from the fixed point definitions instead of the theorems remains possible; this last option is employed by other verified systems such as the CompCert compiler.

Besides previous considerations—given a suitable target for extraction—for the extracted program to be usable, it remains to determine how the user is to interact with it: how formulas and certificates will be input and output. Separate from the kernel there must be a parser that reads from an input stream that contains

three families of items: (a) a collection of non-logical constants and their types (the *signature* discussed above); (b) a polarized version of a formula (the proposed theorem); and (c) a proof certificate expressed as a maximally explicit FPC. The kernel is then asked to check whether or not the given certificate yields a proof of the proposed theorem, calling to this end the verified `checker` function. If this check is successful, the kernel depolarizes the theorem and prints it out as a means to confirm what formula it has actually checked.

As Pollack (1998) has argued, the printer and parser of our system must be trusted to be faithfully representing the formulas that they input and output. This concern can be addressed in standard ways: by using standard parser generating tools in order to link trust in the checker with trust in a well-engineered and frequently used tool. Further refinements would come from an obvious direction, by crafting and employing verified parsers and printers. Strictly speaking, only the printer needs to be trusted, in that whatever formula was checked, and whether this differs from the input, can be ascertained by inspecting the trusted output of the checker, be this a simple "yes/no" answer or the declared theorem.

6.5   FPCs by reflection in Coq

The second use case for a verified checker takes place directly within the proof assistant where verification is carried out. In this environment, we can push this issue of trust another step. Since the MaxChecker is a simple terminating functional program, it is—as has been demonstrated—a simple matter to implement it within Coq. Moving on, one could formally prove that a successful check leads to a formal proof in, say, Gentzen's *LK* and *LJ* proof systems. By reflecting (Boutin, 1997; Harrison, 1995) these weaker proof systems into Coq—including the axiom of the excluded middle for classical logic proofs—, the chaining of a flexible (logic programming-based) certificate elaborator with the Coq-based checker can then be used to get the proof assistant to accept proofs from a range of other proof systems.

A more direct avenue exploits propositions-as-atoms to produce a similar result. Given a polarized formula, we know by application of Theorem 6.3.6 that if the "output proposition" from the checker is provable, so is the depolarization of the input formula. In the case of a successful check, the output proposition is trivially provable, as it only involves identity equalities where both sides are the same proposition. This usage scheme works as follows:

1. Given a (polarized) formula, secure a proof of theoremhood by an external prover, say, a resolution proof, for which an FPC definition exists.

2. Check the formula against the proof certificate and elaborate it to its maximally explicit form in a logic programming-based kernel.

3. Check the formula against the maximally explicit certificate in Coq.

4. Apply the soundness theorem to obtain the depolarization of the formula as a proved proposition to be used in a Coq development.

To apply this style of proof, it is necessary to move between Coq propositions and their polarized versions, to use the checker to prove the latter and then recover the original propositions. This process of *reification* corresponds to that of *polarization* in focused sequent calculus. The subject will be taken up and treated more at length in Section 13.3.

6.6   Notes

The original development of MaxChecker in OCaml was first introduced in Blanco et al. (2017a) as a consequence of the development of determinate FPC definitions, in particular the maximally elaborate FPC of Section 5.4.

The natural role of logic programming in the implementation of proof calculi in general and the FPC framework in particular is well established in the literature. Among others, it is discussed by Felty (1993); Miller and Nadathur (2012); Chihani (2015); Chihani et al. (2016b).

Functional programming serves as a reasonable simplification in the implementation mechanisms required—at the level of programming languages—in the TCB of a proof checker. To go further, we must turn to the layers of software below the (possibly verified) checker: compiler, operating system, hardware. At present, an end-to-end verification of the entirety of the systems on which a program (here, the proof checker) relies is impracticable; efforts like DeepSpec (DeepSpec) aim at exactly such ubiquitous application of the techniques described in this chapter. Still, questions about security and about the correctness of verified specifications abound. Nonetheless, the production of a formal proof of the ascribed properties of the checker represents a significant increase in the level of trust we may accord to a proof checker—whose trustworthiness is, indeed, the ultimate measure of its value.

In the discussion of extracting an executable functional program from a specification written in Coq, we must note that the concrete proof checker that has been the object of the chapter implements classical logic; in consequence, the associated proofs of correctness rely (in some, very limited, measure) on non-constructive reasoning, namely the axiom of the excluded middle. We may therefore be suspicious of the results of constructively extracting a program from such a development. However, subject to less stringent requirements, such an extraction—performed on the definition of the checker about which the theorems reason, and not on the theorems themselves—can be performed successfully. Indeed, classical reasoning does not by itself invalidate the extraction of a verified specification. For example, the CompCert verified compiler (Leroy, 2009) employs the excluded middle to derive some of its corollaries while its executables are extracted from the fixed point definitions. A similar situation is seen in certain specialized checkers for unsatisfiability proofs—the subject of Chapter 7—, a standard problem in classical logic (Cruz-Filipe et al., 2017a).

In this chapter, we have presented a determinate checker written in OCaml side by side with a formalization this MaxChecker in Coq which was restricted to the propositional fragment. In extending this treatment to the quantifiers, and with them to full first-order logic, the handling of bindings predictably becomes the principal point of interest. In Coq, bindings are not first-class constructs of the language and must therefore be explicitly modeled and their metatheory proved; several Coq libraries facilitate facilitate work with bindings and mitigate the increase in the complexity of proofs. Our use of `Prop` as the type of atoms is a further complication that needs to be addressed. A simplifying factor lifted from the OCaml checker consists of fixing a single type of terms—over which quantification may occur—mimicking the kernel in Figure 4.3 and those to come in Part III. An aspect of the OCaml code which resists easy formalization is the representation of bindings by function spaces in the encoding of higher-order abstract syntax. Adopting functions leads to so-called *exotic terms* and are far more general than the limited operation of substitution they are expected to represent (Despeyroux et al., 1995).

An appealing alternative is to specify the checker and prove it correct not in Coq, but in another proof assistant with rich metaprogramming support, where we can reason directly variable bindings, eigenvariables, etc., using λ-tree syntax, therefore obtaining simple proofs like those we have come to expect from our exposition of the propositional fragment. Abella, used extensively in Part III, is

one such system; an introduction to it is included in Chapter 10. In any case, it should be noted that the propositional checker alone covers many common sources of proofs, including the satisfiability refutations that are the topic of the next chapter.

A complementary development of reduced complexity is the adaptation of MaxChecker in both unverified and verified forms to function as a checker for intuitionistic logic; the changes required for this closely related kernel are few and predictably simple. A more ambitious undertaking may seek to obtain a verified checker for the general FPC framework, and in so doing would avoid the intermediate state of imposing determinacy—performed to increase trust in the underlying model of computation. Such a development is substantially more complex than the one undertaken in this chapter. Fundamentally, it relies on a certified implementation of logic programming and consequently of the interesting and complex problems of unification and backtracking search.

7 Unsatisfiability certificates

7.1 Boolean satisfiability

The *boolean satisfiability problem*, or SAT, is one of the quintessential problems of logic and computer science. Given a classical propositional (i.e., boolean) formula, the decision problem asks whether there exists a boolean interpretation that satisfies the formula—i.e., an assignment of truth values to the finite set of variables in the formula such that the formula evaluates to true. SAT is the first decision problem to be proved NP-complete (Cook, 1971) and in the intervening half-century has been instrumental in the study and classification of the complexity of decision problems. While it might seem that theoretical intractability precludes practical application, recent advances in heuristic algorithms have heralded spectacular progress in the size and sophistication of the problems that specialized programs, called *SAT solvers*, can process, pushing automated theorem provers based on these techniques out of the ivory tower of academia and into industrial practice.

We are mainly interested in proof evidence of the satisfiability properties of boolean expressions. A *positive proof* of satisfiability is simple: the *existence* of truth values that satisfy a propositional formula is expressed in first-order logic by binding the set of variables with existential quantifiers; a proof certificate needs only to give witnesses (true or false) for each of those variables. Such a positive certificate is easy to write and trivial to check. In contrast, a *negative proof* of *unsatisfiability*, or UNSAT, is more challenging. In proving the negation of the SAT property, the existential variables turn into universals, and the onus is to show that no combination of values given to those variables satisfy the formula—brute force being clearly impracticable. Similarly to resolution (treated in Section 3.6), unsatisfiability is of interest in theorem proving as a potential means of providing a *refutation*: to prove a theorem, obtain instead a refutation

of its negation. Our interest will be in these latter unsatisfiability certificates and their formal checking.

7.1.1   **Example**   Let variables be chosen from the set $x_1, x_2, \ldots$. The boolean expression $x_1 \wedge \neg x_2$ is satisfiable because there exist values for x_1 (true) and x_2 (false) that make the expression, and its first-order formulation $\exists x_1. \exists x_2. x_1 \wedge \neg x_2$, true.

Conversely, the boolean expression $x_1 \wedge \neg x_1$ is unsatisfiable because the negation of its first-order formulation $\forall x_1. \neg x_1 \vee x_1$ is a tautology.

The question of correctness of SAT solvers is particularly apt: they are complex *prover* programs—engineered for efficiency and heavily optimized—which carry out critical verification tasks. In order to trust their results, it is imperative to ensure that no unsound reasoning may occur—but maintaining a proof of soundness of such a program as it evolves may be unfeasible. Fortunately, the separation of concerns between prover and checker is recognized and enforced through the definition of standard *UNSAT certificate formats* to provide evidence of a proof of unsatisfiability. Most modern software is based on *Conflict-Driven Clause Learning*, or CDCL (Marques-Silva and Sakallah, 1999; Moskewicz et al., 2001)—a refinement of the classical DPLL search algorithm for complete satisfiability checking (Davis and Putnam, 1960; Davis et al., 1962)—, and support for those certificate formats is easily added to the base algorithm. Hence, instead of checking the tools, a specialized checker checks the results emitted by the tools.

Those specialized proof checkers are simpler, more stable programs, more amenable to a formal proof of their correctness. In fact, a series of recent formalization efforts has produced a number of *verified checkers* to cement the trust in a theorem by way of an UNSAT certificate that purports to refute the negation of said theorem. Similarly to our checker in Chapter 6, these tools are built in a proof assistant like Coq or Isabelle and proven correct against their specification, and then extracted as executable code. To aid efficiency, certificate formats are commonly extended with additional information to make checking more deterministic; lacking support from the SAT solvers, an intermediate processing step is then needed to enrich the standard certificates before they are passed to the verified checker. Though the certificates change, the objective is always the same: verifying that the input formula is unsatisfiable.

Instead of building a checker and laboriously proving it correct—(i.e., verifying that it is sound and "complete enough" for the domain of interest)—, the FPC framework provides a direct and foundational attack on the problem. Now, by first understanding how a proof of the unsatisfiability of a formula is built from

the information in an UNSAT certificate, we will craft FPC definitions to carve such proofs directly into the logic. An obvious advantage will be that, although (as with any other checker) we may want to prove a metatheorem about the relative completeness of the procedure, the framework guarantees its soundness, and in consequence, the concrete checker—run on top of a trusted kernel—cannot ever declare the theoremhood of an impostor formula.

The rest of the chapter is organized as follows. Section 7.2 introduces the standard formats and the properties on which UNSAT certificates are based. Section 7.3 studies the connection between resolution proofs and a primitive form of UNSAT certificate based on resolution traces. Section 7.4 undertakes the certification in the FPC framework of the current family of UNSAT proof evidence. Section 7.5 continues the developments of the previous section while relaxing its reliance in the cut rule. Section 7.6 concludes the chapter.

7.2 Redundancy properties and shallow certificates

A SAT (or UNSAT) problem consists of a boolean expression, generally written in DIMACS CNF format. A text file represents a boolean expression in clausal normal form as a series of lines. The first line includes two numbers: the number of variables v present in the formula (represented implicitly by the numbers $1, 2, \ldots, v$, and the number of clauses c of the formula in CNF form. The first line is followed by c lines, each representing a clause. A clause is given as a subset of literals drawn from the variable set $1, 2, \ldots, v$ (unsigned if they are positive, prefixed by a minus sign if they are negated) and terminated by 0. An example is given in Figure 7.1.

The key insight of the DPLL backtracking algorithm is its introduction of simplifications based on the properties of boolean variables while preserving the completeness of backtracking search. In particular if a clause contains exactly one literal, its satisfaction forces a fixed truth value, which can be used to simplify other occurrences of the involved variable. Formally:

7.2.1   Definition   A *unit clause* is a clause with a single literal. *Unit propagation* (also called *boolean constraint propagation*, or BCP) is a simplification procedure for formulas in conjunctive normal form which, given a literal clause l, performs the following operations:

1. It removes every clause containing the literal l except the unit clause.

```
p cnf 4 8
 1  2 -3  0
-1 -2  3  0
 2  3 -4  0
-2 -3  4  0
 1  3  4  0
-1 -3 -4  0
-1  2  4  0
 1 -2 -4  0
```

7.1 **Figure**   Standard example of propositional formula in the DIMACS CNF format. It comprises 4 variables, say x_1, x_2, x_3, x_4, and 8 clauses. The represented formula is: $(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor x_3) \land \cdots \land (x_1 \lor \neg x_2 \lor \neg x_4)$.

2. It removes every negated instance of the literal, $\neg l$, across all clauses.

The removal of negated instances may cascade and lead to the formation of new unit clauses; the process is repeated to saturation. Unit propagation *derives a conflict* if it results in a pair of complementary unit clauses l' and $\neg l'$.

This operation is also related to the inference rule of *unit resolution*, which simplifies the general rule of binary resolution (discussed in Section 3.6) by imposing that one of the resolvents be a unit clause.

Current UNSAT certificate formats are designed around the use of *redundant clauses*, that is, clauses which added to a formula in CNF preserve properties like unsatisfiability. If we want to prove that a CNF formula is unsatisfiable, a simple procedure involves adding clauses that preserve unsatisfiability and hopefully assist in adding further redundant clauses until arriving at the empty clause—at which point the formula can be declared trivially unsatisfiable. For example, accumulating a tautological formula to a conjunction of clauses is always redundant, though also uninformative. Let us consider the following, more interesting property:

7.2.2 **Definition**   A clause $C = l_1 \lor \cdots \lor l_n$ is a *reverse unit propagation* (RUP) clause w.r.t. a CNF formula F if unit propagation on an assignment that falsifies C (that is, the addition to F of the unit clauses $\neg l_1, \ldots, \neg l_n$) derives a conflict.

A RUP clause is said to have the *asymmetric tautology* property, or AT. Addition of RUP clauses to a CNF formula preserves logical equivalence. RUP is the strongest redundancy property that preserves equivalence.

```
1 2 0
1 0
2 0
0
```

7.2 Figure Certificate of unsatisfiability for the formula in Figure 7.1 in the RUP certificate format.

The simplest among the UNSAT certificate formats we will study is that called *reverse unit propagation* (RUP) certificate format. A RUP certificate consists of a series of clauses, called "lemmas," each of which has the RUP property with respect to the accumulation of the initial formula and previous RUP lemmas. The certificate ends with the empty clause. Figure 7.2 presents an example of a RUP certificate, with clauses represented in the same DIMACS CNF format as the input formula.

7.2.3 Example Starting from the problem in Figure 7.1, take the first lemma in Figure 7.2, negate it and attempt to derive a conflict by unit propagation on the resulting formula:

```
 1  2 -3 |     2 -3 |        -3 |        -3
-1 -2  3 |          |          |
 2  3 -4 |  2  3 -4 |     3 -4 |        -4
-2 -3  4 | -2 -3  4 |          |
 1  3  4 |     3  4 |     3  4 |         4
-1 -3 -4 |          |          |
-1  2  4 |          |          |
 1 -2 -4 |    -2 -4 |          |
-1       | -1       | -1       | -1
-2       | -2       | -2       | -2
```

First, the unit clauses -1 and -2 are propagated. This generates a new unit clause, -3, whose propagation derives a conflict between 4 and -4. Therefore, the lemma is a RUP clause that can be added while preserving logical equivalence. The process is repeated with each subsequent lemma. Once all lemmas have been added, only the empty clause remains. That is, to finish the proof, unit propagation on the set of original clauses and added lemmas must derive a conflict.

```
 1   2  -3 |            |            |
-1  -2   3 |     -2   3 |         3  |          3
 2   3  -4 |  2   3  -4 |            |
-2  -3   4 | -2  -3   4 |     -3   4 |          4
 1   3   4 |            |            |
-1  -3  -4 |     -3  -4 |     -3  -4 |         -4
-1   2   4 |      2   4 |            |
 1  -2  -4 |            |            |
 1   2    |            |            |
 1        | 1          | 1          | 1
 2        | 2          | 2          | 2
```

Thus, the original formula is unsatisfiable.

The RUP format is reasonably compact and easy to implement (both its production and its checking), but rather costly to verify. The main contributor to the inefficiency of checking is the accumulation and persistence of lemmas, even as they become unnecessary by the addition of newer lemmas.

In fact, a defining characteristic of the more general CDCL algorithm that lies at the foundations of modern SAT solvers is its ability to both *add* and *remove* clauses by conflict analysis techniques. Clause elimination techniques, like their additive counterparts, can delete clauses whose removal preserves properties like satisfiability. The solver must provide this information in the certificate, which motivates a simple extension to the RUP format. DRUP certificates add the option to mark extant clauses (or lemmas) as deleted by prefixing them with a d marker; they are otherwise identical to RUP certificates. Figure 7.3 shows an example certificate. Intuitively, when we attempt to derive a conflict by unit propagation, it is irrelevant that we ignore a subset of deleted lemmas, as long as we can find the conflict with less information.

Clause deletion alone results in significant performance improvements, but progress does not stop there. More sophisticated redundancy properties, powerful enough to express in terms of them all the processing techniques employed by standard SAT solvers. Namely, the following property:

7.2.4   Definition   A clause C has the *resolution asymmetric tautology* property (RAT) w.r.t. a CNF formula $F = C_1 \wedge \cdots \wedge C_n$ if either:

    1. C has the AT property w.r.t. F; or

```
    1   2   0
d   1   2  -3   0
    1   0
d   1   2   0
d   1   3   4   0
d   1  -2  -4   0
    2   0
    0
```

7.3 Figure Certificate of unsatisfiability for the formula in Figure 7.1 in the DRUP certificate format.

```
   -1   0
d  -1  -2   3   0
d  -1  -3  -4   0
d  -1   2   4   0
    2   0
d   1   2  -3   0
d   2   3  -4   0
    0
```

7.4 Figure Certificate of unsatisfiability for the formula in Figure 7.1 in the DRAT certificate format.

2. C contains a literal l such that all the clauses that result from resolving C with a clause C_i of F on l (i.e., C_i contains the literal $\neg l$) have the AT property w.r.t. F.

Addition of RAT clauses to a CNF formula preserves satisfiability (and therefore its complement, unsatisfiability). It is the strongest redundancy property that preserves SAT (and UNSAT).

It is clear that RAT generalizes AT (i.e., RUP). We consider directly the certificate format that combines the addition of RAT lemmas with clause deletion: the result is called DRAT. Syntactically, it coincides with DRUP. In order to simplify operation, each lemma to be added either has the RUP property or it has the RAT property *on its first literal*. An example is given in Figure 7.4.

7.2.5 Example Again starting from the problem in Figure 7.1, we take the first lemma in the certificate, this time from Figure 7.4. By applying unit propagation, we note that its negation does not derive a conflict. Thus, it does not have the

RUP property, so we select the first (here, only) literal of the lemma and generate a new set of clauses by resolving it with all applicable current clauses:

```
 1  2 -3 |      2 -3
-1 -2  3 |
 2  3 -4 |
-2 -3  4 |
 1  3  4 |      3  4
-1 -3 -4 |
-1  2  4 |
 1 -2 -4 |     -2 -4
```

Each of the three resulting clauses can then be checked for the RUP property with respect to the current clause set. The checks succeed, and so the original lemma has the RAT property and can be added to the set of clauses. The addition and deletion of lemmas continues until the end, when only the empty clause remains and unit propagation should derive a conflict without assistance of any more lemmas.

DRAT is the current de facto standard for UNSAT certificates, partly owing to its adequate expressive power: the standard toolbox of processing techniques at the disposal of modern SAT solvers can be formulated in terms of sequences of RAT lemmas—although not all techniques can be given short translations; for an overview and recent developments see Järvisalo et al. (2012); Heule et al. (2015). We say that all these formats are "shallow" because they do not incorporate a definition of their semantics in the sense that an FPC definition does. Our objective will be to interpret these formats as proper foundational certificates and furnish the missing proof theoretical pieces.

7.3 Resolution FPCs and traces

We commence with a further exploration of the connections between resolution refutations and UNSAT certificates, which are another form of refuting a formula, and consequently proving the theoremhood of its negation. Both families of proofs operate on the same principle: starting from a formula in clausal form, prove and accumulate a series of lemmas (themselves clauses) until arriving at an extended collection of clauses from which unsatisfiability is immediate. It is only in the properties and proofs of those lemmas that the two methods differ.

First, let us recall the binary resolution FPC of Section 3.6. The heart of a general proof by binary resolution, as presented by Chihani et al. (2016b), is a

backbone of cuts. Each cut corresponds to an application of the resolution rule, which adds a new, derived clause on one cut branch and proves that the clause indeed follows by binary resolution on the other cut branch; these auxiliary proofs are simple. It keeps adding derived clauses until it can derive the empty clause, thus finishing the proof. The following general *proof pattern* is inspired by this style of reasoning.

7.3.1 **Definition** Let $F = C_1 \wedge \cdots \wedge C_m$ a CNF formula, and $L_1, \ldots, L_n$ a sequence of lemmas, each of them a clause, that lead to a refutation of F by a certain rule of inference (e.g., binary resolution, RUP, etc.).

If F is unsatisfiable, then the DNF formula $\neg F = \neg C_1 \vee \cdots \vee \neg C_m$, where negated clauses are conjuncts, is a tautology. If lemmas preserve satisfiability when added to F, so do their conjunctive negations when added to $\neg F$.

Given a procedure, Λ, to build proofs of redundancy of a new (negated) lemma with respect to a set of assumptions (clauses and prior lemmas, all negated), an *LKF* proof by *lemma backbone* proceeds by:

1. Storing the negated clauses as initial assumptions.

2. Cutting each negated lemma into the proof, building a derivation Λ that it follows from the current set of assumptions (and adding it as an assumption in the opposite branch).

3. The final lemma is the empty clause (negated, true). The final proof, Λ_0, shows the tautology of the complete set of negated clauses and lemmas by its redundancy property, while the opposite branch is closed by true.

The proof schema is shown in Figure 7.5. Disjunctions are polarized negatively and conjunctions are polarized positively. To ensure all negated clauses are positive, including unit clauses, they can be paired with the conjunctive unit, true.

The resolution certificate in Figure 3.5 is an example of this pattern—and, as we begin to corroborate in the next section, the same organization applies to redundancy properties like RUP. Before doing that, in complement to proofs which explicitly exploit redundancy properties, we further our study of resolution by adapting one of its main variants:

7.3.2 **Definition** Let $A \bowtie B$ designate a clause that results from resolving clauses A and B. *Hyperresolution* is a generalization of the binary resolution rule that takes a sequence of clauses $C_1, C_2, \ldots, C_n$ ($n \geq 2$) and yields a clause that results

$$
\cfrac{
  \cfrac{
    \cfrac{\ \ }{\vdash \ldots, t^{+} \Downarrow t^{+}}\ decide
  }{\vdash \ldots, t^{+} \Uparrow}\ store
}{\vdash \ldots \Uparrow t^{+}}\ cut
$$

$$
\Lambda_0
$$

$$
\cfrac{
  \cfrac{\vdash \neg C_1,\ldots,\neg C_m,\neg L_1,\ldots,\neg L_n \Uparrow}
       {\vdash \neg C_1,\ldots,\neg C_m,\neg L_1,\ldots,\neg L_n \Uparrow f^{-}}
  \qquad
  \cfrac{\vdash \neg C_1,\ldots,\neg C_m,\neg L_1,\ldots,\neg L_n \Uparrow}
       {\vdash \neg C_1,\ldots,\neg C_m,\neg L_1,\ldots,\neg L_{n-1} \Uparrow \neg L_n}\ store \; cut
}{\vdash \neg C_1,\ldots,\neg C_m,\neg L_1,\ldots L_{n-1} \Uparrow}
$$

$$
\cfrac{
  \cfrac{\vdash \neg C_1,\ldots,\neg C_m,\neg L_1 \Uparrow}
       {\vdash \neg C_1,\ldots,\neg C_m \Uparrow \neg L_1}\ store \; cut
}{\vdash \neg C_1,\ldots,\neg C_m \Uparrow}
$$

$$
\Lambda_n
$$

$$
\vdash \neg C_1,\ldots,\neg C_m,\neg L_1,\ldots,\neg L_{n-1} \Uparrow L_n
$$

$$
\vdash \neg C_1,\ldots,\neg C_m \Uparrow
$$

$$
\cfrac{
  \cfrac{
    \vdash \neg C_1 \Uparrow \neg C_2 \vee \cdots \vee \neg C_m
  }{\vdash \Uparrow \neg C_1, \neg C_2 \vee \cdots \vee \neg C_m}\ store
}{\vdash \Uparrow \neg C_1 \vee \neg C_2 \vee \cdots \vee \neg C_m}
$$

$$
\Lambda_1
$$

$$
\vdash \neg C_1,\ldots,\neg C_m \Uparrow L_1
$$

$$
\vdash \neg C_1,\ldots,\neg C_m
$$

7.5 Figure The lemma backbone proof pattern for CNF refutations.

```
 1   1   2  -3  0  0
 2  -1  -2   3  0  0
 3   2   3  -4  0  0
 4  -2  -3   4  0  0
 5   1   3   4  0  0
 6  -1  -3  -4  0  0
 7  -1   2   4  0  0
 8   1  -2  -4  0  0
 9   1   2   0  1  3  5  0
10   1   0   9  4  5  8  0
11   2   0   9  3  6  7  0
12   0  10  11  2  4  6  0
```

7.6 Figure   Certificate of unsatisfiability for the formula in Figure 7.1 in the TraceCheck certificate format.

from the application of a left fold on the list with the binary resolution relation:
$(\cdots (C_1 \bowtie C_2) \bowtie \cdots) \bowtie C_n$.

Hyperresolution avoids the creation of intermediate clauses and leads to more compact proofs. SAT solvers are easily adapted to emit proof certificates based on conflict analysis and redundancy criteria like RUP, but they cannot be so easily modified to produce proofs by resolution. Resolution certificates contain large amounts of proof evidence which also makes them much larger, but faster to check. It is important to note that a sequence of resolution steps may result in zero, one, or several possible solutions. In the context of a correct certificate, such a sequence will yield at least one solution, but backtracking is needed to ensure the one needed for the proof is eventually inspected.

Hyperresolution derivations are used in a legacy UNSAT certificate format, TraceCheck. A certificate file contains two kinds of lines: original clauses and derived lemmas—as in any proof by resolution. Each line starts with a unique number which names each clause. First, the clauses of the original form are given—always following DIMACS conventions—terminated by two zeroes. They are followed by a sequence of lemmas, terminated with a zero, followed by a sequence of existing clause identifiers which derive the clause by hyperresolution, finished with the second zero. As always, the empty clause concludes the proof evidence. An example is shown in Figure 7.6.

7.3.3   **Example**   Consider the resolution proof given in TraceCheck format in Figure 7.6. Clauses 1 through 8 reproduce the formula to refute (namely, that of Figure 7.1), and lemmas 9 through 12 constitute a proof by (hyper)resolution.

Clause 9 states that the lemma $x_1 \vee x_2$ follows by resolving clauses 1 and 3 ($x_1 \vee x_2 \vee \neg x_3 \bowtie x_2 \vee x_3 \vee \neg x_4 = x_1 \vee x_2 \vee \neg x_4$); then resolving the result with clause 5 ($x_1 \vee x_2 \vee \neg x_4 \bowtie x_1 \vee x_3 \vee x_4 = x_1 \vee x_2 \vee x_3$); and, finally, resolving with 1 again ($x_1 \vee x_2 \vee x_3 \bowtie x_1 \vee x_2 \vee \neg x_3 = x_1 \vee x_2$). In all these cases, there is only one literal on which to resolve; literals repeated in both resolvents are not duplicated. The rest of the lemmas proceed similarly.

To define an FPC for hyperresolution proofs (for example, expressed in a transcription of the TraceCheck format), we shall assume use of the lemma backbone pattern. It will then suffice to exhibit a proof procedure, Λ, that builds a proof of a lemma, $\vdash \neg C_1, \ldots, \neg C_k \Uparrow C_{k+1}$ by making use of the hyperresolution information associated to the lemma in the certificate.

7.3.4   **Theorem**   Let $\neg C_1, \ldots, \neg C_k$ be a set of negated clauses. If a negated lemma $L = \neg l_1 \wedge \cdots \wedge \neg l_n$ follows by a hyperresolution sequence $[i_1, \ldots, i_m]$ on the negated clauses, Figure 7.7 builds an LKF^a proof that the lemma $\neg L$ follows from the assumed set of clauses.

Proof. The proof evidence (i.e., the certificate) is the hyperresolution sequence. Assume for brevity that each negated clause $\neg C_i$ is stored under a matching numeric index i; the mapping is not shown in the figure.

Initially, the lemma L is seen as a disjunction of literals, all of which are stored. All literals are stored under a unique index for the literals of the lemma; also for brevity, the figure does not show indexes explicitly.

Once all literals are stored, the proof creates a backbone of $m - 1$ cuts unrolling the sequence of applications of binary resolution steps in the hyperresolution sequence. The two first indexes in the sequence are used to select the clauses indexed by them; the cut formula is the result of resolving both clauses. For the j-th cut, the formula $\neg C_{k+j}$ is equal to $\neg C_k \bowtie \neg C_{j+1}$ (except $\neg C_{k+1} = \neg C_{i_1} \bowtie \neg C_{i_2}$). Therefore, $\neg C_{k+j} = (\cdots (\neg C_{i_1} \bowtie \neg C_{i_2} \bowtie \cdots) \bowtie \neg C_{i_j+1}$. Then:

1. In the positive branch, we have to prove that the derived clause C_{k+j} follows from binary resolution. This is exactly what the FPC definition in Section 3.6 does. The task is delegated to one such certificate, abbreviated Ξ in the figure.

2. In the negative branch, the new negated clause $\neg C_{k+j}$ is stored and the construction of the backbone continues.

At the end of the backbone, when the last formula has been stored, we have precisely the result of the hyperresolution sequence, namely the expansion: $\neg C_{k+m-1} = (\cdots (\neg C_{i_1} \bowtie \neg C_{i_2} \bowtie \cdots) \bowtie \neg C_{i_m}$. If the hyperresolution sequence yields the negated clause that was expected, this is $\neg l_1 \wedge \cdots \wedge \neg l_n$. Since the complementary literals are all stored from the beginning of the proof, focusing on the C_{k+m-1} yields n literal branches, each of which can be closed, and with it the proof. $\square$

That is, the full proof of a TraceCheck-style UNSAT certificate proceeds by a *nested lemma backbone* whose main procedure for the proofs of lemmas, Γ, is that given by Theorem 7.3.4. This, in turn, builds a binary resolution proof from the information contained in a hyperresolution sequence, and delegates the proofs of lemmas to the procedure given by the binary resolution FPC. The only difference is that Theorem 7.3.4 does not derive the empty clause, but the context of literals of the lemma derived by hyperresolution.

The FPC definition follows directly from this result and the structure in Figure 3.5 through the insertion of the nested backbone, which mimics the structure of the main backbone. The salient feature of the hyperresolution backbone is that the cut formulas are not contained in the lean certificate, but rather they are extracted from the context of the sequent, namely the storage zone—compare this with Example 3.6.1, where all derived clauses are explicitly provided. There are two technical solutions to this necessity:

1. Allow the cut expert to inspect the storage zone Γ to assist in the programmatic composition of a cut formula B.

2. Add information in the certificate (and the clerks and experts) to replicate the relevant sections of the storage zone in the certificate, so as to derive the cut formulas without access to the state of the kernel.

In standard kernel designs the state is not only unwriteable, but also unreadable by clerks and experts. However, there is an argument to be made that experts can occasionally be more "expertly" if they can read parts of the state; this is especially true of cut. The second possibility of a certificate reflecting parts of the kernel in *lockstep* is also a recurring pattern that is observed in Section 11.6 and Section 12.4, where it will be discussed at length.

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{
        \overline{[] \vdash l_1, \ldots, l_n, \ldots \Downarrow \neg l_1} \quad \cdots \quad \overline{[] \vdash l_1, \ldots, l_n, \ldots \Downarrow \neg l_n}}{[] \vdash l_1, \ldots, l_n, \neg C_1, \ldots, \neg C_{k+m-1} \Downarrow \neg C_{k+m-1}} \wedge^+ \; decide
      }{[] \vdash l_1, \ldots, l_n, \neg C_1, \ldots, \neg C_{k+m-2} \Uparrow \neg C_{k+m-1}} \; store
    }{[k+m-2, i_m] \vdash l_1, \ldots, l_n, \neg C_1, \ldots, \neg C_{k+m-2} \Uparrow} \; cut
    \qquad
    \Lambda_{m-1}
  }{\Xi \vdash l_1, \ldots, l_n, \neg C_1, \ldots, \neg C_{k+m-2} \Uparrow C_{k+m-1}}
}{\quad}
$$

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{[] \vdash l_1, \ldots, l_n, \neg C_1, \ldots, \neg C_k \Uparrow \neg C_{k+1}}{[] \vdash l_1, \ldots, l_n, \neg C_1, \ldots, \neg C_k \Uparrow \neg C_{k+1}} \; store
    }{[k+1, i_3, \ldots, i_m] \vdash l_1, \ldots, l_n, \neg C_1, \ldots, \neg C_k \Uparrow} \; cut
    \qquad \Lambda_1
  }{\Xi \vdash l_1, \ldots, l_n, \neg C_1, \ldots, \neg C_k \Uparrow C_{k+1}}
}{\quad}
$$

$$
\cfrac{
  \cfrac{
    \cfrac{[i_1, \ldots, i_m] \vdash l_1, \neg C_1, \ldots, \neg C_k \Uparrow l_2 \vee^- \cdots \vee^- l_n}{[i_1, \ldots, i_m] \vdash \neg C_1, \ldots, \neg C_k \Uparrow l_1, l_2 \vee^- \cdots \vee^- l_n} \; store
  }{[i_1, \ldots, i_m] \vdash \neg C_1, \ldots, \neg C_k \Uparrow l_1 \vee^- \cdots \vee^- l_n}
}{\quad}
$$

7.7 Figure   Proof schema of a hyperresolution proof step in LKF^a.

Study of resolution will resume in Chapter 8. For the rest of this chapter, we concentrate on redundancy-based UNSAT certificates.

7.4 Unsatisfiability FPCs with cut

In this section we adapt the lemma backbone pattern to the families of UNSAT certificates that are actually used in practice. As in the previous section for hyper-resolution proofs and the TraceCheck format, the sole metatheoretic obligation is to prove the redundancy of a lemma with respect to the set of clauses and previously added lemmas (both negated). We start by considering how to obtain LKF^a proofs of RUP certificates.

7.4.1 Theorem   Let $\neg C_1, \ldots, \neg C_k$ be a set of negated clauses. Given a negated clausal lemma $L = \neg l_1 \wedge \cdots \wedge \neg l_n$, if the lemma has the RUP property w.r.t. the formula formed by the set of clauses, then there is a simple LKF^a proof of the lemma.

Proof. The *base context* in a RUP proof step (after the literals of the lemma have been stored) is the union of the set of previous clauses and lemmas (both negated), $\neg C_1, \ldots, \neg C_k$, and the literals of the RUP lemma, $l_1, \ldots, l_n$. Those literals are essentially unit clauses which will serve as sources of RUP reductions, i.e., unit propagations. These operations will be modeled as a backbone of cuts.

The proof proceeds by *levels*. Initially, at level 0, all the stored formulas are active: clauses can be addressed by number; literals can be addressed by a common index. Both index types are qualified by level.

Each level consists of an application of unit propagation on an active literal. To progress from level i to level $i + 1$, a cut is used. To obtain the cut formula, select an arbitrary i-level literal (i.e., currently active), say l^i. Assume its negation $\neg l^i$ holds, and for each i-level negated clause, $\neg C_j^i$:

1. If the negated clause contains $\neg l^i$, its true occurrence is removed at the next level: $\neg C_j^{i+1} = \neg C_j^i - \{\neg l^i\}$.

2. If the negated clause contains l^i, it is removed—there is no $\neg C_j^{i+1}$.

3. All other clauses are copied unchanged at level $i + 1$: $\neg C_j^{i+1} = \neg C_j^i$.

The cut formula is the disjunction of all these clauses. Then:

1. In the positive branch, the revised negated clauses are stored independently reflecting their new level. All unit clauses at level i except l^i are stored at

level $i + 1$ together with any new unit clauses contained in the cut formula. Having stored all (negated) clauses, we proceed with the next cut in the backbone on a new unit clause.

2. In the negative side, the negated cut formula becomes a (positive) CNF formula, $\neg B = \bigwedge_j C_j^{i+1}$. It will be stored and immediately focused upon. In the positive phase, as many branches as there are clauses in $\neg B$ appear.

   For each branch, there is a release on a $C_j^{i+1} = m_1 \vee \cdots \vee m_k$, and in the negative phase its literals are stored. Immediately, we focus on the i-level predecessor $\neg C_j^i$, which contains at least the complementary literals $\neg m_1 \wedge \cdots \wedge \neg m_k$.

   The positive phase results, again, in a number of branches on literals whose complement is in storage, and which are therefore easily closed. If the assumed literal for unit propagation, $\neg l^i$, appears as well, l^i is available in storage to close the branch, as well.

The backbone ends when all unit clauses have been processed—and with it unit propagation. If the lemma did satisfy the RUP property, unit propagation must result in a pair of complementary literals, which will be available and can be used to finish the proof.   □

As the case of TraceCheck, the full proof of the RUP UNSAT certificate is an exemplar of the nested backbone pattern. Also as with TraceCheck, lemma derivations rely on a smart choice of formulas based on the context by the cut expert. In addition, indexing is used heavily to keep track of the provenance of formulas in storage—which never deletes data. Besides the sequence of RUP lemmas, the certificate only needs to keep track of the current level in each subproof.

7.4.2 **Example**   Consider the problem in Figure 7.1. The claim is that the following formula is unsatisfiable. For compactness, variables x_i are represented by their numeric identifiers, as in DIMACS format, and negated variables are barred:

$$(1 \vee 2 \vee \bar{3}) \wedge (\bar{1} \vee \bar{2} \vee 3) \wedge (2 \vee 3 \vee \bar{4}) \wedge (\bar{2} \vee \bar{3} \vee 4) \wedge (1 \vee 3 \vee 4) \wedge (\bar{1} \vee \bar{3} \vee \bar{4}) \wedge (\bar{1} \vee 2 \vee 4) \wedge (1 \vee \bar{2} \vee \bar{4})$$

Equivalently, the following formula is a tautology:

$$(\bar{1} \wedge \bar{2} \wedge 3) \vee (1 \wedge 2 \wedge \bar{3}) \vee (\bar{2} \wedge \bar{3} \wedge 4) \vee (2 \wedge 3 \wedge \bar{4}) \vee (\bar{1} \wedge \bar{3} \wedge \bar{4}) \vee (1 \wedge 3 \wedge 4) \vee (1 \wedge \bar{2} \wedge \bar{4}) \vee (\bar{1} \wedge 2 \wedge 4)$$

We want to prove that $(1 \lor 2)$ is a RUP lemma. If that is the case, adding the negated disjunct $(\bar{1} \land \bar{2})$ to the presumed theorem preserves its tautology. The interesting part of the proof is the side of the cut that shows that the lemma follows from the assumptions. The negated lemma is precisely the original RUP clause, $(1 \lor 2)$. Once everything is stored, the goal looks like this at level 0:

$$\bar{1} \land \bar{2} \land 3, 1 \land 2 \land \bar{3}, \bar{2} \land \bar{3} \land 4, 2 \land 3 \land \bar{4}, \bar{1} \land \bar{3} \land \bar{4}, 1 \land 3 \land 4, 1 \land \bar{2} \land \bar{4}, \bar{1} \land 2 \land 4, 1, 2$$

Again, for succinctness, indexes and sequent notation—including polarities—are abbreviated. We observe that there are unit clauses to propagate, so we pick one, say 2, and assume its negation $\bar{2}$ holds. We use this as the basis for unit propagation and its effect on the currently considered set of stored (negated) clauses to define the cut formula:

$$(\bar{1} \land 3) \lor (\bar{3} \land 4) \lor (\bar{1} \land \bar{3} \land \bar{4}) \lor (1 \land 3 \land 4) \lor (1 \land \bar{4})$$

Let us first consider what happens on the negative branch, where the negation of the cut formula will be stored and immediately focused upon:

$$(1 \lor \bar{3}) \land (3 \lor \bar{4}) \land (1 \lor 3 \lor 4) \land (\bar{1} \lor \bar{3} \lor \bar{4}) \land (\bar{1} \lor 4)$$

Synchronous treatment results in as many branches are there are clauses. Let us consider the first of these, $(1 \lor \bar{3})$. As a negative formula, it will be released and its literals $1, \bar{3}$ stored in the context, signaling the end of the negative phase. Noting that the clause comes from the negation of a negated clause at the current level, namely $\bar{1} \land \bar{2} \land 3$, we decide on the origin clause. Observe that for every literal in the formula we just stored its complement—and if the clause contains $\bar{2}$, that is the unit clause we are using for unit propagation. Hence, all branches can be closed. This extends to all clauses in the negative branch of cut.

In the positive branch of cut, we store the revised set of negated clauses and mark the used unit clause, so that the effective contents at level 1 are:

$$\bar{1} \land 3, \bar{3} \land 4, \bar{1} \land \bar{3} \land \bar{4}, 1 \land 3 \land 4, 1 \land \bar{4}, 1$$

We now perform unit propagation on the still untreated unit clause 1, assuming $\bar{1}$ and proceeding in the same manner. The new cut lemma, once stored—which corresponds with the level 2 state, is the following. We obviate the negative

side of cut as immediate by the same procedure exhibited above.

$$3, \bar{3} \wedge 4, \wedge \bar{3} \wedge \bar{4}$$

Unit propagation generates a new unit clause, 3. If the same process is applied to obtain level 3, we end up with a pair of unit clauses, 4 and $\bar{4}$, from which the positive backbone of lemmas ends.

The scheme just presented is easily extended to support DRUP certificates simply by adding deletion instructions to the certificate; these will update levels normally, and will additionally remove from considerations those clauses marked for deletion.

7.5   Cut-free unsatisfiability FPCs

In Section 7.3 and Section 7.4, we have constructed LKF^a proofs based on a double backbone pattern: a primary spine of cuts adding a series of lemmas, and in the proof of each of these lemmas a secondary spine of cuts attending to hyperresolution or unit propagation criteria, respectively. Now we turn to a more general proof scheme that for proving lemmas based on redundancy criteria, like RUP certificates. The overall proof still follows the lemma backbone pattern, of course, but the derivations of the lemmas will not rely on cuts as it did in previous proofs.

7.5.1   **Theorem**   Let $\neg C_1, \ldots, \neg C_k$ be a set of negated clauses. Given a negated clausal lemma $L = \neg l_1 \wedge \cdots \wedge \neg l_n$, if the lemma is redundant w.r.t. the formula formed by the set of clauses, then there is a cut-free LKF^a proof of the lemma.

Proof. The base context in a redundancy proof step (after the literals of the lemma have been stored) is the union of the set of previous clauses and lemmas (both negated), $\neg C_1, \ldots, \neg C_k$, and the literals of the current lemma, $l_1, \ldots, l_n$. Those literals are essentially unit clauses.

The proof proceeds as a tree of decides on negated clauses from the set $\neg C_1, \ldots, \neg C_k$. A negated clause is *inactive* if it contains a positive literal whose negation is not among the $l_1, \ldots, l_n$: a focused proof on a positive literal can only end by an *init* rule, for which the negated literal needs to be in the storage area.

Clauses all of whose positive literals have negative counterparts in storage are *active*. The decide rule may focus on any active clause, say, $\neg C_i = m_1 \wedge^+ \cdots \wedge^+ m_k$. Processing the conjunctions results in k branches, as many as literals in $\neg C_i$. By

definition of active clause, all branches on positive literals are closed by *init* on their negative complements.

Branches focused on negative literals m_j proceed by releasing the focus and storing the negative literal, thus potentially growing the set of available negative literals—and with them the set of active (negated) clauses. For each branch, the proof can continue by selecting a locally active, still unused (negated) clause until all branches are closed, in which case the proof succeeds.   □

In this proof scheme, as opposed to previous ones, the expert in charge of orchestrating the proof is the decide expert. Like in previous proof schemes, it benefits greatly from having read access to the storage area—or otherwise replicating the necessary information as part of the certificate term, at the cost of additional complexity.

7.5.2   **Example**   Consider once again the problem defined by Figure 7.1. The DRAT certificate in Figure 7.4 asserts that $\bar{1}$ is a RAT lemma. Indeed we saw in Example 7.2.5 that the clause has the RAT but not the RUP property. On the negative side of the main cut, we will have the following goal, where presentation conventions will be shared with Example 7.4.2 throughout the example:

$$\bar{1} \wedge \bar{2} \wedge 3, 1 \wedge 2 \wedge \bar{3}, \bar{2} \wedge \bar{3} \wedge 4, 2 \wedge 3 \wedge \bar{4}, \bar{1} \wedge \bar{3} \wedge \bar{4}, 1 \wedge 3 \wedge 4, 1 \wedge \bar{2} \wedge \bar{4}, \bar{1} \wedge 2 \wedge 4, \bar{1}$$

Now, instead of cutting on a formula, we decide on one of the clauses to operate on. The clause has to be such that all positive literals can be closed with currently available negative literals; otherwise, it is impossible to continue the proof. Here, from the stock of $\bar{1}$, we have two possible choices: $(\bar{1} \wedge \bar{3} \wedge \bar{4})$ and $(1 \wedge \bar{2} \wedge \bar{4})$. Say we focus on the latter. This yields three branches, one for each literal. 1 is closed immediately and two remain. One of them (after release and store) is:

$$\bar{1} \wedge \bar{2} \wedge 3, 1 \wedge 2 \wedge \bar{3}, \bar{2} \wedge \bar{3} \wedge 4, 2 \wedge 3 \wedge \bar{4}, \bar{1} \wedge \bar{3} \wedge \bar{4}, 1 \wedge 3 \wedge 4, 1 \wedge \bar{2} \wedge \bar{4}, \bar{1} \wedge 2 \wedge 4, \bar{1}, \bar{2}$$

Now we have a larger stock to close positive literals. If we next decide on $(1 \wedge 2 \wedge \bar{3})$, two of the branches are immediately closed, and the single remaining branch adds $\bar{3}$ to the set of negative literals: $\bar{1}, \bar{2}, \bar{3}$. For the next step, we can decide on $(2 \wedge 3 \wedge \bar{4})$, which like its predecessor results in a single continuation branch, to which the negative literal $\bar{4}$ is added. Finally, by deciding on $(1 \wedge 3 \wedge 4)$, all branches can be closed, and with it the subproof.

The second sequent that remained open from the first decide is:

$$\bar{1} \wedge \bar{2} \wedge 3, 1 \wedge 2 \wedge \bar{3}, \bar{2} \wedge \bar{3} \wedge 4, 2 \wedge 3 \wedge \bar{4}, \bar{1} \wedge \bar{3} \wedge \bar{4}, 1 \wedge 3 \wedge 4, 1 \wedge \bar{2} \wedge \bar{4}, \bar{1} \wedge 2 \wedge 4, \bar{1}, \bar{4}$$

This allows a new negated clause to be decided on effectively, $(\bar{2} \wedge \bar{3} \wedge 4)$, but all the (unused) applicable clauses allow us to close at most one branch out of three. Let us say that we decide on this last choice, which at least can close one, and proceed as follows:

1. $\bar{2}$ is added to the list of negative literals: $\bar{1}, \bar{2}, \bar{4}$. By deciding on $(1 \wedge 2 \wedge \bar{3})$, the 1 and 2 branches are closed, and $\bar{3}$ is added to the list of negative literals: $\bar{1}, \bar{2}, \bar{3}, \bar{4}$. By deciding on $(1 \wedge 3 \wedge 4)$, all branches are closed.

2. $\bar{3}$ is added to the list of negative literals: $\bar{1}, \bar{3}, \bar{4}$. By deciding on $(1 \wedge 3 \wedge 4)$, all branches are closed.

3. 4 is closed immediately with $\bar{4}$.

The proof of the lemma is thus finished.

We have glossed over the indexing schemes which are used to make sure that we do not decide several times on the same negated clause along the same branch, as well as the maintenance of the available negated literals, but these are both easily encoded. In addition to that, the decide expert can make use of this information to implement sophisticated heuristics to decide in which order to try the available clauses, for example, based on the number of branches that will be immediately closed—so as to reduce the branching factor.

7.6 Notes

The SAT problem is one of the most representative problems in computer science, not only because of its theoretical significance, but also due to the breadth of practical applications of modern SAT solvers. Good general treatments of the problem can be found in the monographs Biere et al. (2009); Knuth (2015). Heule and Biere (2015) present the state of the art of what is broadly understood of a proof of (un)satisfiability.

The idea of using unsatisfiability proofs as certificates for use by an independent checker goes back at least to Goldberg and Novikov (2003), shortly after the rise of the current generation of SAT solvers and, with them, the large-scale

applications of the problem; their ascent coincided with the appearance of the modern series of SAT solving competitions (SatComp). In contrast to an early line of proof formats more directly based on resolution, the first of the current family of proof formats, RUP, was proposed by Gelder (2008) and quickly made its appearance in a dedicated track in the SAT competitions; the trimming of clauses that leads to the refinement of DRUP appeared some years later (Heule et al., 2013b) along with the DRUP-trim checker.

The RAT property is of great importance because all proof techniques utilized by modern SAT solvers can be expressed in terms of sequences of RAT lemmas (Järvisalo et al., 2012)—even though this does not imply that it can express everything *efficiently*. Its use coupled with clause deletion led to the definition of the DRAT format (Heule et al., 2013a; Wetzler et al., 2014) and its associated checker, DRAT-trim, which remain the reigning standards to this day. Like its predecessor, DRAT-trim is a highly optimized C program in which bugs are occasionally found. Indeed, such a well-defined and stable checker is a prime target for verification as undertaken by a number of recent formalization efforts using proof assistants—as opposed to our pure logic-based approach.

As it stands, UNSAT certificate checking is an expensive operation: common figures report checking times in the same order of magnitude as proving times. These costs are compounded by the penalty of modeling and extracting checkers in higher-level languages and assistants, which has motivated the definition of specialized proof formats that refine a DRAT source certificate. The GRIT format (Cruz-Filipe et al., 2017b) identifies the costliest operation in DRAT checking—namely, finding the unit clauses used during unit propagation—and includes this information in the certificate; it is formalized in Coq, but does not cover the RAT property in its entirety. The LRAT format (Cruz-Filipe et al., 2017a) extends GRIT to verify all of RAT by choosing a pivot element and checking the RAT property across all clauses containing the negation of that element; certified checkers are implemented in Coq and ACL2 that are roughly as fast as the unverified DRAT-trim. Independently, the GRAT format (Lammich, 2017) also extends the insight of GRIT from DRUP to DRAT by more explicit information about a particular way to execute the checking operation; a verified checker is implemented in Isabelle.

All these refinements are geared towards the production reasonably performant, specialized checkers. Since solvers do not emit any of these extended certificate formats, an additional unverified stage transforms DRAT certificates to

each new format; only the checker operating on these augmented certificates is verified. However, some form of completeness property on the mapping from a DRAT certificate to one of the new formats—stating, say, that the original certificate is checkable iff its map is—is only empirically validated.

It should be noted that the standard boolean satisfiability problem is limited to classical propositional logic. The generalization of SAT to include quantifiers is called the *quantified boolean formula* problem, or QBF. An important generalization replaces variables with predicate and function symbols drawn from various background theories—themselves expressed in first-order logic with equality—: this is the *satisfiability modulo theories* problem, or SMT. However, the lively activity in certification standards and checkers for SAT has as of yet no close parallel in either extended setting, although a certificate format for QBF, called QRAT, has been proposed (Heule et al., 2014). For overviews of both problems, refer to the appropriate chapters in Biere et al. (2009).

In the context of the FPC framework, the proof formats proposed in this chapter are very easily implemented—provided that the kernel allows clerks and experts to inspect its state without modifying it, otherwise relatively extensive bookkeeping is required. In standard presentations, including those in Chapters 4 and 10, the kernel is completely opaque to all clerks and experts, which are simply called by the kernel without any information about their context. Some older kernel implementations have provided limited context information to clerks and experts, in particular the one formula that is being operated upon in an inference rule. A somewhat more generous kernel (which remains functionally sealed) is a reasonable possibility and well suited to the generation of smart cut formulas and to sophisticated bookkeeping based on the contents of the context. While everything can be simulated at the level of clerks and experts, the result is less modular and more cumbersome. Section 11.6 illustrates this alternative approach, which recurs later in Section 12.4.

8 Certification of theorem provers

8.1 Towards FPCs in the large

In previous chapters, we have developed techniques to add and subtract detail to proof certificates while representing the same proof (Chapter 5). If sufficient detail is added, the process of checking becomes determinate and can be delegated to a functional checker that is even simpler, faster and more reliable than standard proof checkers based on logic programming (Chapter 6). Until this point, a number of realistic if somewhat academic FPC definitions have been presented (Chapter 3), but previous related publications (Chihani, 2015; Chihani et al., 2016b; Libal and Volpe, 2016b) have yet to address the transition from small, handcrafted examples and idealized calculi to one of the purported goals of ProofCert: the emission of proof certificates by provers and their independent validation by trusted proof checkers in realistic scenarios. A stepping stone was the adaptation as foundational proof certificates of well defined, standard proof formats (Chapter 7). Completing this opening move towards proof certificates "in the large" is the subject of the present chapter.

In Section 6.1, we anticipated a general architecture to support a spectrum of trust levels, which we exercise over the course of the chapter. Truly, a determinate checker can reduce the trusted computing base needed to come to trust the theoremhood of a formula, but the proof traces that constitute tractable proof certificates for MaxChecker are as artificial as their interest is purely mechanistic. A practical solution must involve both non-determinate and determinate checkers, combined in such a way that the resulting system enjoys the expressiveness of the former and the trustworthiness of the latter.

With the components at our disposal, it is now a relatively easy matter to describe the architecture of a *composite proof checker* that we can use to check *any* proof certificate defined by the FPC framework while only needing to trust

MaxChecker. First, use the general, say, λProlog-based checker to perform the formal checking of a formula based on an arbitrary proof certificate accompanied by its FPC definition; this checking operation must pair the certificate with a maximally explicit certificate that will result from the elaboration of the first (as explained in Example 5.3.1 and Section 5.4). Second, independently run MaxChecker on this explicit certificate and the same formula. Of course, we only need to trust the second of these checkers—with the proviso that the trusted checker must contain a trusted printer to output successfully checked formulas. (An independent matter is how to obtain confidence in the correctness of the trusted checker, but—as noted in Chapter 6—a verified implementation of the full checker is not yet within reach, thus motivating the two-tier architecture.)

Our goal is to certify the proof evidence produced by a *bona fide*, complex theorem proving tool, that is sufficiently powerful to provide us with realistic, reasonably sized and publicly available *proof corpora*. To that end, we have selected Prover9 (McCune, 2010), a legacy, automated theorem prover of modest capabilities. An important feature for our experiments is that the output from the software exposes a relatively simple and well-documented resolution calculus, perhaps the simplest deductive model used in practice. We will refine and elaborate the original resolution FPC presented in Section 3.6, and study some sources of nondeterminism and their effects on checking.

The rest of the chapter is organized as follows. Section 8.2 introduces Prover9 and describes a significant subset that is used to model the proofs selected for certification. Section 8.3 revisits the binary resolution certificates and presents various extensions and elaborations addressing significant sources of nondeterministic behavior. Section 8.4 describes in detail the elaboration workflow and the practical composition of the general and determinate provers, along with the requirements of each. Section 8.5 analyzes the results of the certification of all publicly available (non-equational) Prover9 proofs in the TPTP library, the effect of the various formats and checkers, and the systems on which they run. Section 8.6 discusses the extension of the techniques employed throughout the chapter to certify the results of more tools and achieve some level of interoperation across tools. Section 8.7 concludes the chapter.

8.2 The automated theorem prover Prover9

Prover9, developed by McCune (2010) is an automated theorem prover for first-order and equational logic based on a simple resolution calculus, successor to the Otter theorem prover. While no longer actively developed—its last version was released in November 2009—, it remains (in spite of its simplicity) a moderately competent prover to this day and a benchmark for newer tools, as evidenced by its continued placement in the CASC system competition for automated theorem provers (CASC): it remains a good baseline for new developments. Yet in that simplicity we find one of its virtues. Prover9 reports proofs in a well-documented format, close to its calculus, and offers tools to manipulate (and even verify) those proofs. This rare luxury among automated theorem provers paves the way to certification without excessive efforts in reverse engineering.

The output format of Prover9 represents proofs by a sequence of steps, each of which is derived from previous steps by one of 17 primary tactics (of which 14 are used in standard proofs), followed by a sequence drawn from 5 secondary tactics. Proofs in standard format may contain non-clausal assumptions and goals together with clauses. These standard proofs can be transformed by external programs, notably Prover9's own `Prooftrans`, which can simplify justifications and produce more verbose proofs in a subset of the grammar of tactics (i.e., instances of the hyperresolution rule can be transformed into sequences of applications of the binary resolution rule). As a consequence, Prover9 proofs can be brought to close proximity with our model binary resolution FPC described in Section 3.6. In fact, it will suffice to cover only the following five tactics from Prover9's vocabulary:

1. `assumption`: primary tactic which annotates the input formula.

2. `clausify`: primary tactic which annotates the result of translating a non-clausal assumption to CNF.

3. `resolve`: primary tactic which performs binary resolution on two clauses.

4. `factor`: primary tactic which performs factoring on two literals of a clause.

5. `merge`: secondary tactic which removes a literal that is identical to a previous literal in the clause that results from the previous tactic.

```
%% SIGNATURE

% A pair for factoring.
type factor          int -> int  -> cert.

% List of (the) clause on which to factor.
type factr                       int -> cert.

% Used to start the sync phase in factor checking.
type fsmall                            cert.

%% MODULE

cutE    (rlist (factor I K ::Certs))
        (factr I) (rlisti K Certs) Cut :- lemma K Cut.

% Describe the meaning of the factoring subproof.
allC      (factr I) (x\ factr I).
orC       (factr I) (factr I).
falseC    (factr I) (factr I).
storeC    (factr I) (factr I) lit.
decideE   (factr I) fsmall (idx I).
someE     fsmall fsmall T.
andE      fsmall small small.
trueE     fsmall.
initialE  fsmall lit.
```

8.1   Figure   Additions made to the binary resolution FPC (Figure 3.5) to support factoring. A new top-level proof step constructor is added to the resolution step, along with constructors and clauses for the (small) factoring subproof.

Support for factoring requires simple additions to the binary resolution FPC listed in Figure 8.1. These additions are completely modular and permit certification of the non-equational fragment of Prover9; the remaining equational fragment makes use of *paramodulation* and a small number of ancillary tactics. Such additions have been coded as proof certificates by (Chihani et al., 2015) and applied to the certification of a (very small) fragment of proofs produced by the E prover. We will instead concentrate on what the seemingly simple binary resolution FPC can achieve with no or small changes when applied to proofs generated by an automated theorem prover like Prover9.

8.3   Resolution certificate elaboration

In addition to certifying a sizeable number of resolution proofs produced by a real theorem prover, we will explore the effects of adding and removing details from proofs through the pairing combinator introduced in Section 5.2. We note that the binary resolution FPC given in Section 3.6—which closely reflects the semantics of Prover9—leaves two optional aspects of a resolution proof implicit:

1. The order in which two clauses are resolved to yield a third is unspecified.

2. Substitution terms used to instantiate quantifiers are not given.

The fact that the order of the resolvents is left unspecified has by itself a potentially enormous impact on proof checking if standard techniques are used, given the exponential number of backtracking points it can generate in a degenerate proof.

A more explicit proof could provide one or both kinds of information, thereby making checking more deterministic. These proof formats can be easily encoded as FPC definitions in the style of Figure 3.5. In fact, they are simple variations that can be formulated as simple changes on the original FPC. Figure 8.2 shows the minimal change required to impose a fixed ordering of the resolvents, replacing two possible schemes for the decide rule in the left premise with one, slighlty simplified clause. Figure 8.3 presents the limited additions made to support explicit substitution information; these changes extend the resolution rule to the new constructors and simplify the treatment of the decide rule and the existential quantifier. Both sets of changes, together with the original FPC, can be described as modular additions to a common template for resolution FPCs.

The preceding notes have addressed the addition of more information to a proof certificate capable of modeling proof evidence produced by Prover9 (i.e., its *elaboration*). Conversely, Prover9 tactics specify not only the names of the affected clauses, but also (the indexes of) the involved literals. This information is lost in the encoding of the binary resolution FPC: strictly speaking, it is a *distillation* of the—in some respects—more complete proof produced by the tool.

8.3.1   Example   Consider the resolution example in Example 3.6.1. In Prover9, an equivalent proof is expressed by the following script, once preprocessed and then simplified by `Prooftrans`. Clause numbering is slightly beautified to reflect the fact that the `clausify` tactic is used to replace universal quantifiers with

```
% Left premise
allC      (res I Cert) (x\ res I Cert).
orC       (res I Cert) (res I Cert).
falseC    (res I Cert) (res I Cert).
storeC    (res I Cert) (res I Cert) lit.
%decideE  (res I (rex J Cert)) (rex J Cert) (idx I).
%decideE  (res I (rex J Cert)) (rex I Cert) (idx J).
decideE   (res I Cert) Cert (idx I) :- Cert = (rex _ _).
someE     (rex J Cert) (rex J Cert) T.
someE     done done T.
```

8.2 Figure   Addition of ordering of resolvents to the (unordered) binary resolution
FPC (Figure 3.5). The only necessary change occurs in decide rule of the left
premise, where instead of two clauses, one for each possible ordering of the
resolvents, a single clause fixes said ordering. Changes are noted by showing, for
each affected rule, the old clauses commented and immediately followed by their
replacements.

fresh eigenvariables, so that said quantifiers are not directly visible in processed
proof scripts. Thus, we get:

```
1 r(z).  [assumption].
2 -r(c1) | t(c1).  [clausify(0)].
3 -t(z).  [assumption].
4 t(z).  [resolve(1,a,2,a)].
5 \$F.  [resolve(3,a,4,a)].
```

In practice, the signatures of the various flavors of the binary resolution FPC
must use disjunct sets of constructors (e.g., `resolve`, `factor`,...for the "base"
FPC in Figure 8.1; `resolve'`, `factor'`, ...for the variation in Figure 8.2,
etc.). This requirement allows the top-level constructor of a proof certificate—and
any constructor at any point in a certificate—to determine the certificate family
to which it belongs. With this information, a missing certificate (represented
by a logic variable) can be reconstructed via pairing drawing exactly from the
correct set of constructors. Otherwise, mixed certificates could be constructed
and checked—at the cost of an explosion in the number of choices and an erosion
of the separation of semantics we endeavor to dictate.

8.3.2 Example   Continuing Example 8.3.1, we look at the various representations
under consideration. The full standard payload (unordered, without substitution
information) strongly closely resembles the description in Example 3.6.1:

```
% Introduce an order-ambiguous resolvent subproof.
%type resolveX    int -> cert -> cert.

% Eigenvariables allowed into certificate here.
type rquant       (i -> cert) -> cert.
% Use the following substitution term.
type subst        i -> cert -> cert.
```

8.3 **Figure**   The ordered binary resolution FPC with added subtitution informa-
tion. Changes to the signature are limited to the removal of resolvent sufproofs
with ambiguous ordering (cut clauses involving the `resolveX` constructor are
likewise removed) and the introduction of constructors for quantification and
substitution. Changes in the module (see Figure 8.3) add constructor condutions
to the rules, which further constraint the system and allow for more natural
elaboration. Presentation conventions are shared with Figure 8.2.

```
[(r z),                          % 1
 (forall x\ or (ng (r x)) (t x)), % 2
 (ng (t z))]                     % 3
[(t z),                          % 4
 ff]                             % 5
[resolve 4 (res 1 (rex 2 done)),
 resolve 5 (res 3 (rex 4 done))]
```

Here, clause numbering is implicit in both clause lists. The certificate is
properly the third list of inference (resolution) rules; any ordering of the resolvent
clauses is allowed. As a first refinement, the order of the resolvents is fixed, so
that only one combination of the exponentially many reorderings is accepted:

```
[resolve' 4 (res' 1 (rex' 2 done')),
 resolve' 5 (res' 4 (rex' 3 done'))]
```

The clause lists remain unaltered; only the certificate fraction is adapted.
Finally, substitution information can be added to the relevant rules:

```
[resolve'' 4 (res'' 1 (rex'' 2 (subst'' z done''))),
 resolve'' 5 (res'' 4 (rex'' 3 done''))]
```

While there is no standard feature to perform renaming while allowing for
modular definitions and name reuse in λProlog, an experimental branch of Teyjus
enables this kind of operations. In what follows, the requisite renamings will be
assumed regardless of the technical means used to achieve them.

```
orC      (start Ct Certs) (start Ct  Certs).
falseC   (start Ct Certs) (start Ct Certs).
storeC   (start Ct Certs) (start Ct' Certs) (idx Ct) :- inc Ct Ct'.
cutE     (start _ Certs) C1 C2 Cut :- cutE (rlist Certs) C1 C2 Cut.
%cutE    (rlist (resolve K Cert::Certs)) Cert (rlisti K Certs) Cut :-
% lemma K Cut, Cert = (res _ _).
cutE     (rlist (resolve K Cert::Certs)) Cert (rlisti K Certs) Cut :-
  lemma K Cut, (Cert = (res _ _); Cert = (rquant _)).
cutE     (rlist (factor I K ::Certs)) (factr I) (rlisti K Certs) Cut :- lemma K Cut.

% Left premise
allC     (rquant Rs) Rs :- Rs = (x\ rquant (_ x)) ; Rs = (x\ res (_ x) (_ x)).
orC      (res I Cert) (res I Cert).
falseC   (res I Cert) (res I Cert).
storeC   (res I Cert) (res I Cert) lit.
%decideE (res I (rex J Cert)) (rex J Cert) (idx I).
%decideE (res I (rex J Cert)) (rex I Cert) (idx J).
decideE  (res I Cert) Cert (idx I) :- Cert = (subst _ _) ; Cert = (rex _ _).
%someE   (rex J Cert) (rex J Cert) T.
%someE   done done T.
someE    (subst T Cert) Cert T        :- Cert = (subst _ _) ; Cert = (rex _ _) ; Cert = done.

andE     (rex J Cert) small (rex J Cert).
andE     (rex J Cert) (rex J Cert) small.
releaseE (rex J Cert) (rex J Cert).
storeC   (rex J Cert) (rex J Cert) pivot.
%decideE (rex I Cert) Cert (idx I) :- Cert = done.
decideE  (rex I Cert) Cert (idx I) :- Cert = (subst _ _) ; Cert = done.
```

8.4 **Figure** The ordered binary resolution FPC with added subtitution information (continued). Presentation conventions are shared with Figure 8.2.

8.4 Certification workflow

In this section, we describe the certification of resolution proofs produced by an automated theorem prover based on a simple resolution calculus—namely, the non-equational fragment of Prover9. The most permissive of the binary resolution FPCs that have been discussed (with unordered resolution and no substitution information) subsumes the inference rules used by the fragment of interest of Prover9.

In order to organize the experiments, we compose a module `resolution-elab` which accumulates the various versions of the resolution FPC and defines instances of pairing between the unordered binary resolution certificate without substitution information (Figure 8.1) and other, more explicit certificate placeholders (Figures 8.2 and 8.3), which are completed by means of elaboration. These form a framework for our experiments in certification of Prover9 proofs and "translating between implicit and explicit versions of proof." The module is actually a schema for other modules, that is, it does not define any non-logical constants or any proofs by resolution based on those constants: it is syntactically correct and can be readily compiled, but is inert. To define resolution problems and their proofs, the following elements can be plugged in.

1. In the signature file, constructors for atoms (of type `bool`) and terms (of type `i`), both of which may have term arguments. Each of these must be complemented in the module definition by a clause of `pred_pname` or `fun_pname`, respectively, used by the polarized to translate formulas into the NNF format required by the kernel. This constitutes the user signature.

2. In the module file, problem definitions based on the signature. These are the combination of a problem identifier, and a triple of lists:

    (a) A list of the clauses that constitute the input formula (therefore given implicitly in clausal normal form).

    (b) A list of clauses derived from applications of proof steps.

    (c) A list of proof steps providing justifications for the derived clauses representing. This list is properly a schematic representation of a proof by binary resolution with factoring.

8.4.1 Example   Continuing from Example 8.3.2, the translation of the problem to λProlog results in a series of term and atom declarations in the signature:

```
type    r_p9, t_p9    i -> bool.
type    z_p9, c1_p9   i.
```

These declarations are given a prover-specific prefix to avoid clashes with other names. The module contains the payload given in Example 8.3.2—adapted to reflect the prefixed identifiers—alongside auxiliary declarations for printing and counting.

```
pred_pname (r_p9 X1) "r_p9" [X1].
size_bool (r_p9 X1) Size :- size_term X1 SizeX1,
                            Size is SizeX1 + 1.
pred_pname (t_p9 X1) "t_p9" [X1].
size_bool (t_p9 X1) Size :- size_term X1 SizeX1,
                            Size is SizeX1 + 1.
fun_pname z_p9 "z_p9" [].
size_term z_p9 Size :- Size is 1, !.
fun_pname c1_p9 "z_p9" [].
size_term c1_p9 Size :- Size is 1, !.
```

The module defines three possible pairings, all starting from the Prover9-like FPC in Figure 8.1 and elaborating to: (a) binary resolution with ordering (Figure 8.2); (b) binary resolution with ordering and substitutions (Figure 8.3); and (c) maximally explicit elaborations (Figure 5.2). For all three pairings, predicates are given in triads of: (a) elaboration; (b) elaboration followed by checking of the new certificate; and (c) elaboration followed by reporting. The decomposition in steps is geared towards producing accurate measurements, from which exact checking times can be derived. This approach has two practical advantages. First, experiments can be carried out from a single λProlog program, without exporting and importing intermediate results. Second, it circumvents certain limitations in Teyjus; as a result, a direct comparison with other implementations of the language, like ELPI, is possible. To this end, utility predicates are defined in the module in the following fashion:

1. `check_unordered`: check the Figure 8.1 certificate specified by a problem description without any additional operations.

2. `elab_to_sans`: pair the Figure 8.1 certificate with, and elaborate into, a Figure 8.2 certificate through checking.

3. `check_sans`: perform the elaboration in `elab_to_sans`, followed by an independent check of the resulting Figure 8.2 certificate.

4. `print_sans`: perform the elaboration in `elab_to_sans`, followed by printing of problem size statistics.

5. `elab_to_subst`, `check_subst`, `print_subst`: like the `*_sans` predicates above, but pairing and elaborating the example with a Figure 8.3 certificate.

6. `elab_to_max`, `check_max`, `print_max`: like both the `*_sans` and `*_subst` predicates above, but pairing and elaborating the example with a Figure 5.2 certificate.

7. `elab_and_export`: pair the Figure 8.1 certificate specified by the example with a Figure 5.2 placeholder, as `elab_to_max`, and output the certified formula and maximally explicit certificate as OCaml code, ready to be imported in MaxChecker.

Each of these predicates takes a problem identifier as argument. The `check_*` and `elab_to_*` predicates work without any further additions. Reporting predicates rely on auxiliary operations on user-defined atom and term constructors as follows.

1. `print_*` require a `size_bool` clause for each atom constructor and a cut-terminated `size_term` clause for each term constructor. The size of a constructor is defined as 1 (the constructor itself), plus the sum of the sizes of its term arguments.

2. `elab_and_export` requires a `print_name` clause for each atom constructor and a cut-terminated `print_term` clause for each term constructor. Atom names relate the string representation of atoms given in `pred_pname` and a valid OCaml identifier. Terms relate the constructor to a valid OCaml identifier and to the printed representations of its arguments for use by MaxChecker.

All these additions can be easily generated automatically. To import a formula and certificate into MaxChecker, the atom and term signatures must agree with the identifiers given by the translation.

8.4.2   Example   Finalizing the sequence from Example 8.4.1, the MaxChecker instantiation relies on the definition of isomorphic, native OCaml types for terms:

```
type 'a t =
| R_p9 of 'a
| T_p9 of 'a
```

As well as atoms:

```
type t =
  | Eigenvariable of int
  | Z_p9
  | C1_p9
```

MaxChecker then takes the translation of the original formula as well as the maximally elaborate certificate in OCaml syntax and checks the combination of both; this final piece of information is omitted for brevity.

The process of certifying a Prover9 proof comprises three main steps:

1. Extract the problem signature from the Prover9 proof script. From this, constructors and auxiliary clauses are extracted. Each proof step is mapped into the sets of clauses and justifications that define a problem. Having done this, the `resolution-elab` module is instantiated. If MaxChecker is used, identifiers and constructors for its atom and term types are derived.

2. Certify the extracted formula with the extracted certificate on the λProlog kernel. The baseline goal `check_unordered` checks the certificate as given in the problem description; further exploration is possible.

3. If MaxChecker is used, solve goal `elab_and_export`, export the translated formula and certificate to OCaml, and run the functional checker on the maximally explicit elaboration.

The sequence can be fully automated. A translation from Prover9 to either kernel should generate valid identifiers for each language, and particularly in λProlog avoid clashes between names, say, by using a dedicated namespace. Note that renumbering of clauses with respect to the original clause numbers of Prover9 may be necessary, given that the FPC numbers its clauses by order as they are given in the certificate, first the base clauses and after those the derived clauses; contrast this with Prover9 proofs, in which assumptions can be introduced at any point in the proof.

8.5 Analysis of results

The study, optimization and comparison of theorem provers relies on the existence of widespread benchmarks. In the automated theorem proving community, the TPTP library—short for *Thousands of Problems for Theorem Provers*—fulfills this role (Sutcliffe, 2009). Its success is cemented in its syntactic conventions, easy to understand by mathematicians and to parse and process by programs alike, and close to the syntax of Prolog. An important part of TPTP is its syntactic support for the expression not only of problems but of proofs of those problems. For a large number of theorem prover, the library contains a collection of solved theorems along with the generated proofs, TSTP—short for *Thousands of Solutions for Theorem Provers*—(Sutcliffe et al., 2004). Those proofs are expressed as sequences of inference rules whose dependencies satisfy a DAG structure (in essence, a form of Frege proof, for which see Section 11.1). The ready availability of a widely recognized corpus of proofs forms the basis of the experimental study.

We have collected the full set of Prover9 refutations in the TPTP library—a total of 2668 in version 6.4.0—and excluded 52 files with irregular formatting (the resulting set of examples is precisely that of version 6.3.0). Of these, 978 fall in the fragment supported by the resolution FPCs; 27 are empty proofs that refute false. The two largest problems are extreme outliers, also excluded since they would be of limited utility to establish or confirm trends. Each problem is expanded into a detailed proof in the simplified binary resolution calculus via the homonymous `expand` option of Prover9's built-in `Prooftrans` tool.

A further preprocessing step is required. Prover9 accepts as input arbitrary first-order formulas (i.e., they need not be given in clausal normal form), and transforms those non-clausal assumptions as necessary by way of the `clausify` tactic; the resulting translated clauses are added and the original assumption given as their provenance. This has two consequences. First, non-clausal assumptions are made redundant by this "clausification" process and will be removed; note that resolution certificates only describes problems problems expressed in clausal normal form. Second, suppose that not all clauses of the input formula (or its CNF translation) are used in the proof. There is no guarantee that every single clause—including unused ones—will be part of the proof script produced by Prover9, and thus in the proof certificate derived from the script. In consequence, such a certificate may indeed represent a stronger theorem than the original
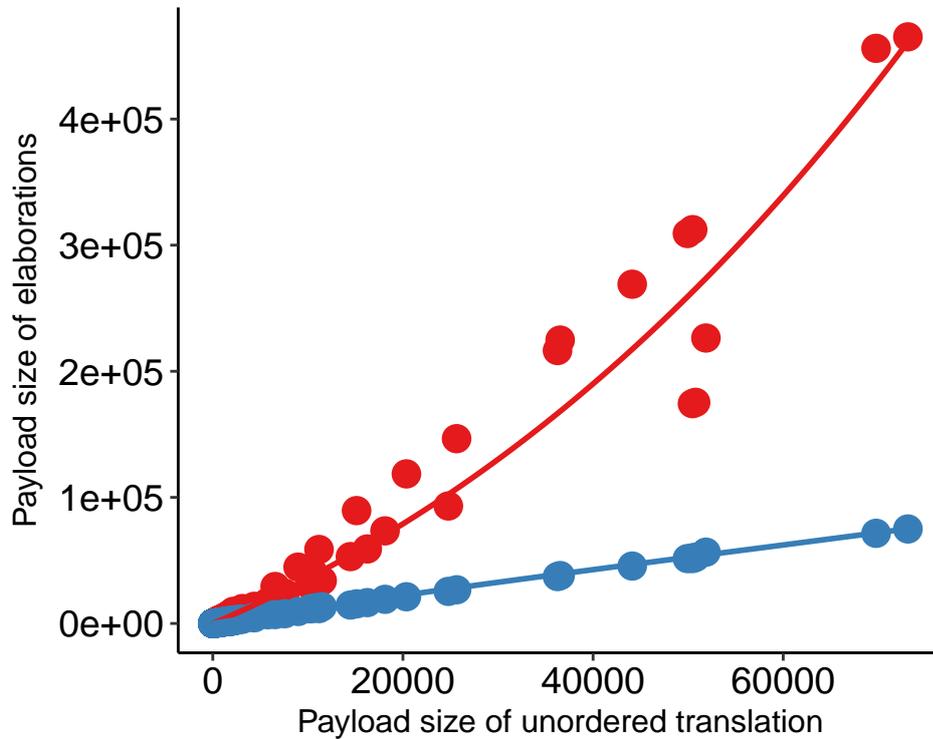
8.5 **Figure** Space complexity of resolution certificate elaborations. Payload sizes are defined as the sum of components of a problem: formula and certificate. The size is computed as the sum of constructors, where natural numbers (serving as indexes) are represented natively and counted as single constructors. In this and following figures, data series are represented by the following color codes: `unordered-without` (Figure 8.1), `ordered-without` (Figure 8.2), `ordered-with` (Figure 8.3), `maximal` (Figure 5.2).

formulation used by the theorem prover, albeit the derived theorem statement can be easily shown to imply original theorem statement.

Having obtained the data and performed this preprocessing, the general workflow described in the previous section is applied. As part of the experiments, we run the λProlog-based checker on two separate implementations of the language: the more mature compiler Teyjus (Nadathur and Mitchell, 1999) and the more modern interpreter ELPI (Dunchev et al., 2015). The dataset presents us with ample amounts of problems encoded as logic programs, in sizes and numbers rarely seen in the language.
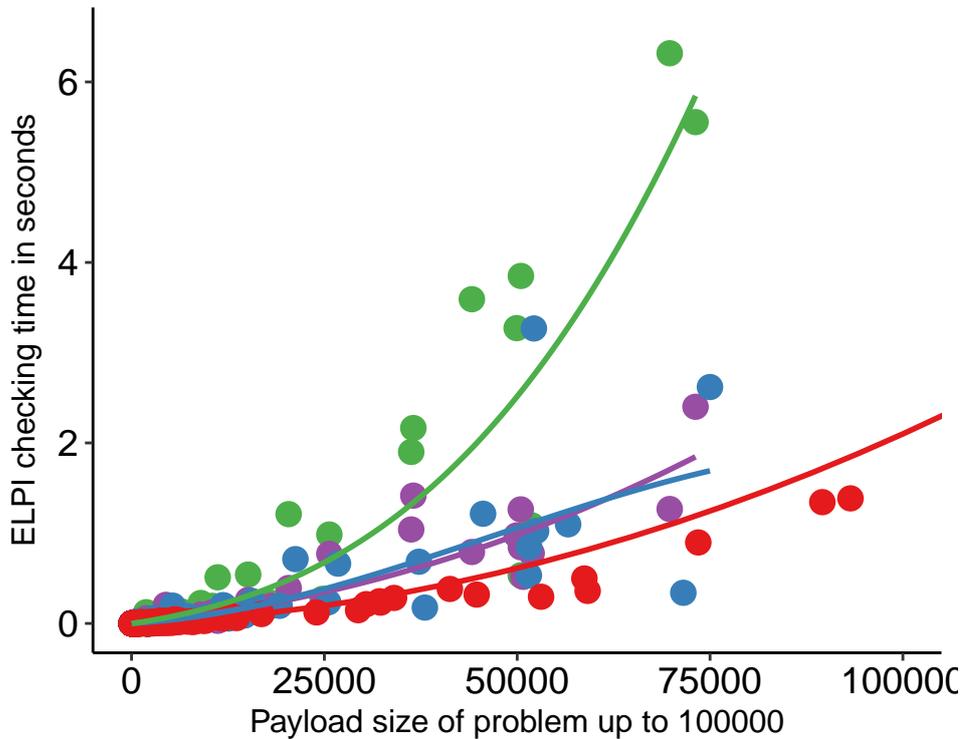
8.6 Figure Time complexity of resolution certificate elaborations on ELPI. Presentation conventions are shared with Figure 8.5.

We have successfully checked all resolution refutations produced by Prover9 involving binary resolution and factoring; no errors have been found in this set of Prover9 proofs. Quantitative information emanates naturally from the data. A first point of interest concerns the size of certificates and how it is defined. The natural approach is to defined the size of a resolution certificate to be the sum of the sizes of the initial and derived clauses along with their justifications; and the size of a maximally explicit certificate as the size of the actual certificate term plus the size of the original set of clauses. In this way, we compare different certificate formats by the size of their full payload: how much it costs to express a problem and provide proof evidence for it—the theorem is implicit in, say, a resolution certificate, but not in a maximally elaborate certificate.

In both binary resolution and maximally explicit certificates, natural numbers are used extensively as indexes and have a great bearing on their overall sizes. Here we consider various possible representations. First, we may use *machine integers* and count a natural number as one constructor. Second, we can use the standard
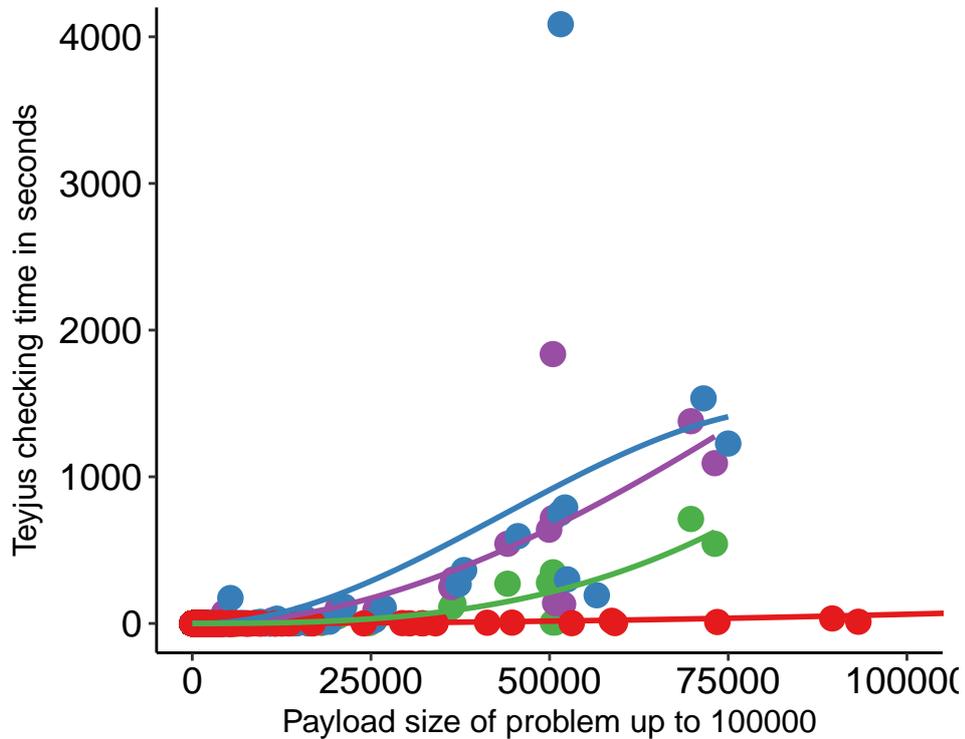
8.7 **Figure**   Time complexity of resolution certificate elaborations on Teyjus. Presentation conventions are shared with Figure 8.5.

inductive definition and extend the count of the number of constructors to the inductively defined naturals; this results is a size increase, possibly important, which can then be measured. Other reasonable *alternative inductive encodings* (like the ternary definition that mimics a binary representation, where a number is either zero, twice the value of a number, or twice the value of a number plus one) will fall somewhere between the two extremes considered here.

Figure 8.5 presents the effect of elaboration on certificate size. Adding ordering information (from `unordered-without` to `ordered-without`) does not affect certificate size, and therefore that first uninformative data series it not shown in the figure. Certificate sizes grow as they are made more explicit, though the blowup here is bounded by small constants. Elaborating from the original `unordered-without` to `ordered-with` adds a linear multiplicative constant to the payload; sizes grow by 16% on average. Finally, elaborating to the maximally explicit certificate causes an increase by an average factor of 2.8, ranging between 1.02 and 6.54. We do not expect hard trends since results depend

on the coverage of a wide space of problems by available data, and in particular the corpus of larger examples at our disposal is relatively limited.

If we adopt the simple inductive definition of natural numbers and adjust counts accordingly, the increase in size is considerable. On average, this representation causes certificates to grow by an average factor of 5.8 as they are elaborated to their maximally explicit form. However, there is much greater variability, ranging between factors of 1.2 to 361.

Concentrating on the more natural approximation of natural indexes as constants, Figure 8.6 presents the evolution of checking times under the various certificate formats as the size of said certificates grows. As a matter of fact, and as may be expected, the more detailed a certificate is, the faster it is to check. Overall, progress is fairly rapid: for example, a sizeable certificate in the `unordered-without` format about 75000 symbols large can be checked in approximately 6 seconds; a similarly sized certificate in the maximally explicit format can be checked in about a second. However, due to size blowup it cannot be asserted that a maximally explicit certificate will always check faster than its resolution equivalent, and in fact for some of the larger problems we notice an inversion of this naive hypothesis. Both extremes (i.e., unordered binary resolution without substitutions and maximally explicit elaboration) appear to exhibit behavior that is fitted well by a quadratic regression curve, although the proof corpus becomes sparser as problem sizes grow, and more data would be necessary to establish definite trends—if any avail. In addition, it should be noted that the more explicit resolution certificates gain a large part of the efficiency of the much simpler maximal elaborations by a very moderate increase in size and complexity; ordering of the resolvents, in particular, is the determining factor in avoiding backtracking points.

The use of Teyjus as λProlog runtime, as depicted in Figure 8.7 yields overall qualitatively similar results. However, performance significantly slower and more asymmetrical. Outliers are more frequent and more extreme, and the overhead of elaboration is substantial with respect to the much faster and more consistent ELPI; the intermediate formats show particularly erratic behavior. Conversely, the checking times for OCaml-based MaxChecker on the large, maximally explicit certificates running are completely negligible compared to both elaboration and checking times in λProlog: in particular, MaxChecker checks every example in less than 0.01 seconds.

Taken as a benchmark, the Prover9 corpus enables comparison of the two principal implementations of λProlog. Functionally, both have behaved equivalently with some minimal rough edges in the ELPI parser being uncovered by our experiments. The principal differences are summarized in the following points:

1. There are moderate limits to the size of the terms Teyjus can parse, both in the compiler `tjcc` and in the simulator `tjsim`. While these have not impeded the list-based formulation of resolution certificates, exporting and importing large proofs and formulas is problematic. This motivates the additive composition of elaboration and checking steps given in the `resolution-elab` module, in which once the unordered certificate is read all computation is performed in-memory.

2. The intermediate compilation step in Teyjus, absent from the ELPI interpreter, has scalability issues of its own. Compilation times are seen to grow substantially once a certain threshold in the size of the proof translation is reached. For the very largest examples in the corpus, this grows to make Teyjus unusable, to the point of compilation possibly failing to terminate (and certainly not doing so in any reasonable amount of time).

3. In some of the larger examples, the process of elaboration has been observed to surpass the capacity of Teyjus' internal data structures, causing a premature stack overflow and a termination of checking. This can be observed first when combining elaboration and checking in the `check_*` predicates.

4. Teyjus does not implement a predicate to measure execution times inside the language, whereas ELPI reports the execution time of goals by default. Therefore Teyjus must rely on external tools and we need find a way to separate the time taken to load the program from the proper user time required by elaboration and checking operations.

There are observable performance differences between the two systems. Generally speaking, ELPI runs as fast or faster and scales better, although the two systems show different patterns of behavior, especially in the relative cost of running elaboration through paired certificates, compared with the checking time of each individual certificate. In its favor, the interface of Teyjus is more complete and more amenable to scripting. Although workarounds can be found for ELPI, batch reporting in Teyjus remains more informative.

8.6 The next 700 certificate formats

In this and the previous chapter, we have successfully extended the areas of application of the FPC framework from small handcrafted examples to the practical domain of automated theorem provers. We have done so by designing certificates for the proof evidence produced by actual theorem provers, and then converting and checking the proofs produced by those tools. In Chapter 7, the objects of formalization were part of a series of well defined proof formats shared by a large family of software in wide use—still a rarer fortune. In previous sections of this chapter, we considered a relatively simple and well documented proof format which could be reduced to a standard calculus with minimal changes. In light of these encouraging results, it remains to consider how to extend them—and their adoption—further. There are two principal aspects of this push towards widespread use: first, the definition of certificate formats supporting additional proving tools; second, the recognition of FPCs by these tools and the import of proofs and interoperability between compatible provers.

The first aspect is intricately related to the mechanisms offered by the FPC framework to program proof search in the "logical computer" of the augmented sequent calculus. This, the critical step, remains the domain of the specialist logician. Indeed, the main difficulty lies in translating the proof evidence produced by a certain tool into formal terms. Each format constitutes a sort of domain-specific language for the writing of proofs, which must be compiled into the assembly language of the underlying proof system: an FPC definition fulfills the role of such a compiler. Currently, there is no systematic methodology to move from an arbitrary description of a proof semantics into a lower-level description built upon the sequent calculus; the capability to embed high-level descriptions of semantics as part of an FPC would greatly simplify this task. Some advances have been made in streamlining the use of augmented sequent calculi as a bona fide programming environment, therefore offering some guarantees of completeness (in a loose sense) and of continuous integration of tests—since programming in this exotic logical assembly, where instructions are inference rules, involves behaviors that exceed the complexity of, say, the more conventional imperative and functional languages (Blanco and Chihani, 2016, 2017). Metatheoretical results—e.g., soundness and completeness of provability by an FPC definition with respect to a proof system and a mapping between proofs in that source

system and concrete FPCs expressed in that FPC definition—must be established separately.

The second aspect of interoperability through proof certificates is fundamentally one of integration. A prover may not only write its output as a proof certificate, but also read certificates in an understood definition, in essence acting as a checker (or delegating the task to a dedicated checker) and reconstructing a proof for a given formula, which can then upon success be accepted as proved. The expression of proof semantics is self-contained in FPC definitions and provides support for the use of proofs generated by external tools. (A more challenging sort of interoperability involves translating a proof from one certificate format to another a given tool can understand; as we observed in Chapter 5, this is in general not possible.) On the whole, this second aspect follows easily from the first, which is in that sense its prerequisite.

The preceding two aspects are fundamentally technical in nature: they may involve a certain amount of work to specify and give proof theoretical readings of the various proof calculi, as well as translations as FPC definitions and integration of proof checkers with the provers themselves. These questions are best solved by establishing a dialog between the authors of theorem provers and the authors of FPC definitions (if they differ); the work presented here makes a convincing case that such a dialog is not only possible but deeply beneficial. The principal challenge is social: making the authors of theorem provers aware of the existence of these Foundational Proof Certificates and collaborating with them to add support for certification and proof checking. The desirability of such formats is acknowledged by the theorem proving community, as the recent first ARCADE workshop—celebrated as part of CADE-26 in August 2017—made manifest (ARCADE). While "every problem is a people problem," the FPC framework has a strong claim towards becoming a canonical solution, if it is not the only one.

8.7 Notes

The original certification experiments on Prover9 were published in Blanco et al. (2017a). Two additional topics are mentioned here in passing and are elaborated elsewhere. In related work in Blanco et al. (2016), we proposed an extension of the TPTP format that integrates the semantics of inference rules as logic programming specifications as an intermediate step towards filling the semantic void of free-form proof output formats. On the side of the FPC framework, a

programming methodology and assistant tools that aim to make the use of proof certificates more approachable is proposed in Blanco and Chihani (2016).

In order to capture all of Prover9's proofs in the TPTP repository we need to add support for paramodulation: the FPC for paramodulation given in Chihani et al. (2015) is a starting point, and can be adapted with few modifications to work with other, related implementations of resolution calculi like the one employed in this chapter. The development in that paper of the paramodulation FPC follows that of of the proof certifier Checkers. This system implements a pair of kernels for the standard calculi *LKF^a* and *LJF^a* in the same tradition of the kernel used throughout the present Part II and defined in Section 3.7. To these kernels it adds a structured module system for the definition of problem signatures and their composition with FPC definitions to yield complete instances of the checker. The plan was to use Checkers to certify the proofs produced by the E prover, which is based on a superposition calculus—itself a variant of resolution. However, only a minimal sample of proofs could be checked owing to difficulties in modeling the semantics of the relatively rich inference rules of the theorem prover out of limited documentation. In our view, this experiment showcases the necessity for communication and collaboration between the authors of provers and checkers, and a greater need for documentation.

Until now, the process for certifying the output of a theorem prover has been for a independent effort on the checking side to understand the semantics of each inference rule of the object calculus. This approach generally suffers from missing documentation, naming conventions, changes across software versions, etc. One way to overcome this gap is to supply the implementers of theorem provers with an easy to use format in which to describe the semantics of their inference rules. This format should be general enough to allow specifications to range from precise, determinate definitions to more implicit, less specific hints that would instruct a checker on how to reconstruct a full proof of the object calculus—even if left partly (albeit inessentially) unspecified. The insight of the proposal in Blanco et al. (2016) is to reduce the gap mentioned above by employing a format that is already known to implementers of theorem provers, namely the TPTP format, enriching a syntactic base with semantic information expressed in a logic programming style. The resulting model is closer to the proof checkers of the FPC framework, although there remains to establish the connection between the logic programs that embed the semantics of a proof system and the expanded proof in an underlying sequent calculus.

The computational model of the related project Dedukti, based on the λ-calculus, is closer to the more type-driven paradigm of common proof assistants, themselves routinely founded upon the precepts of functional programming (rather than logic programming, as in the FPC framework). Libraries have been developed to translate proof evidence originating in the common OpenTheory format shared by the HOL family of proof assistants (which includes the standard configuration of Isabelle), fragments of Coq and its close relative Matita (both based on the calculus of inductive constructors), and theorem provers like iProver and Zenon (both extended from first-order logic to Deduction Modulo). In this setting, work towards interoperability has already been initiated (Assaf and Cauderlier, 2015; Cauderlier and Dubois, 2017).

In practice, when such frameworks as FPCs are not directly understood by the source provers, it is necessary to convert the proof evidence output by the provers into suitable, equivalent proof certificates. This translation need not be trusted as the result of checking is tied to the formula being checked.

Part III

Logics with fixed points

9 Fixed points in logic

9.1 Fixed points and equality as logical connectives

Logical frameworks based on the theory of intuitionistic logic (outlined in Chapter 4)—as well as linear logic—have been adopted as the foundations of higher-order logic programming and used to specify many aspects of programming languages and theorem provers. An important limitation of the logical frameworks used until now—notably λProlog—is that they do not offer a natural treatment of inductive definitions and proofs under the paradigm of higher-order abstract syntax that those frameworks employ widely and fruitfully (for reference on these topics, see Section 4.2). As inductive reasoning is the bread-and-butter of functional and logic programming languages, and of their metatheory, extending the logic with a clean treatment of these concepts is of the essence for the formal study of those paradigms and reason about them effectively.

McDowell and Miller (1997, 2000, 2002) developed in response a logic that incorporated the concept of *definitions*, whose use allows certain declarations to be treated as closed, i.e., as fixed point expressions representing (mutually) recursive predicates—along with rules for unfolding (similar to backchaining) and induction on definitions. These connections were matured through research efforts by Miller and Tiu (2003, 2005)—with Tiu and Momigliano (2003) and further work by Tiu (2004, 2006)—and Gacek et al. (2008b, 2011). Baelde and Miller (2007), followed by work by Baelde (2008b, 2009, 2012), extended these developments to general least and greatest fixed points expressions as logical connectives in various logics. The systems Bedwyr and Abella have their roots in this line of work, which forms also the proof theoretical basis of the developments in this Part III.

From the perspective of the sequent calculus, the two principal criteria that invest a connective with logical legitimacy are: (a) the definition of introduction

$$nat \equiv \mu(\lambda Nat. \qquad \lambda n.n = 0\, \vee$$
$$\exists n'.n = S\ n' \wedge^+ Nat\ n')$$
$$plus \equiv \mu(\lambda Plus. \quad \lambda k.\lambda m.\lambda n.(k = 0 \wedge^+ m = n)\, \vee$$
$$\exists k'.\exists n'.n = S\ n' \wedge^+ p = S\ p' \wedge^+ Plus\ k'\ m\ n')$$

9.1 **Figure**   Logic specification of natural numbers and addition on them as a least fixed points in μLJF. The specification is based on a type *nat* with two constructors representing the standard inductive definition of natural numbers: 0 of type *nat*, and S of type $nat \rightarrow nat$.

rules for the connective; (b) the preservation of the cut elimination property once those rules are folded into the proof system. Let us first consider a generic *fixed point* operator, μ, say, for now, in a simple, one-sided calculus reminiscent of *LK*. The introduction rule for such a connective would be:

$$\frac{\vdash B(\mu B)\bar{t}, \Gamma}{\vdash \mu B\bar{t}, \Gamma}$$

Here, B is a formula, abstracted over a recursive predicate and an arbitrary number n of variables, called the *body* of the fixed point. The fixed point operators take the body and exposes the abstracted variables. To this combination can be applied a list of n terms, $\bar{t}$, acting as the *arguments* to the fixed point. The unfolding operation applies to the body its own definition wrapped in the fixed point (recursion) and the list of variables. Two example fixed point expressions— the inductive definition of natural numbers and the addition relation on these—are shown in Figure 9.1. A predicate thus becomes a name for a fixed point expression.

This illustration motivates the need for *equality* as a logical connective. Fixed point definitions can very naturally encode recursive specifications such as those written in a programming language like Prolog—compare the encoding of the same relations in λProlog, in Figure 4.2. In a relational specification expressed in pure logic, it becomes necessary to relate the abstracted variables that act as parameters of a predicate with the values passed them as arguments. In a basic sense, the pattern matching at the head of the clause must take place inside the logic. This is achieved by defining the introduction rules for equality. In the same

one-sided setting as above, these are:

$$\frac{}{\vdash t = t, \Gamma} \qquad \frac{\vdash \{\Gamma\sigma : \sigma \in CSU(s = t)\}}{\vdash s \neq t, \Gamma} \; \dagger$$

The rule for equality applied on two instances of the same term t has the same effect as the initial rule: it finishes the current branch of the proof. The rule for inequality inspects two terms, s and t, and determines the conditions under which they are equal; these conditions are expressed by a set of substitutions σ, each representing conditions (the *complete set of unifiers*, or CSU) which make the two terms equal. If there are no such solutions, the set of premises is empty and the rule succeeds immediately (this endows the system with a notion of *negation-as-failure*); otherwise, each possible solution to the equation becomes a premise whose context is the same context of the conclusion, Γ, once the substitutions have been applied—this proviso is represented by $\dagger$. The treatment of equality marks also the introduction of *unification*—which computes the substitutions that make two terms equal—not only in the implementation details of a logic programming language, but as part and parcel of the logic proper. In first-order logic, the complete set of unifiers coincides with the more familiar *most general unifier*, or MGU, to which we shall return in Section 13.2.

9.1.1 Example Consider the inductive definition of the type of natural numbers in Figure 9.1 and compact notation for constants. The sequent $\vdash 2 = 2$ is immediately provable the application of the rule of equality, whereas $\vdash 1 = 2$ is not. Conversely, $\vdash 1 \neq 2$ is proved by the inequality rule: because no unifiers of $1 = 2$ exist, the empty set of premises is trivially satisfied. Finally, $\vdash S\, a \neq S\, b, \Gamma$ (where a and b are variables) is provable if $(\vdash \Gamma)\sigma$ is provable subject to the substitution σ which allows $a = b$ to unify.

Coming back to the fixed point connectives, we may advance that with the addition of focusing the operation of the introduction rule for μ, also called the *unfolding rule*, would not split in two, but be identical in both asynchronous and synchronous splits of the original introduction rule. The differences that materialize the division between *least and greatest fixed points* (resp. μ and ν) arise from the addition of (respectively) *induction and coinduction*: without these principles, both connectives are indistinguishable. If induction and coinduction are added, the symmetry between the least and greatest fixed points is restored. Returning to the two-sided sequent calculus, the least fixed point μB is characterized by the

following two introduction rules:

$$\frac{\Gamma, S\bar{t} \vdash \mathcal{R} \quad BS\bar{x} \vdash S\bar{x}}{\Gamma, \mu B\bar{t} \vdash \mathcal{R}} \; induct \qquad\qquad \frac{\Gamma \vdash B(\mu B)\bar{t}}{\Gamma \vdash \mu B\bar{t}} \; unfold$$

The right introduction rule is just an *unfolding* of the fixed point, which expresses that $B(\mu B)\bar{t} \supset \mu B\bar{t}$. The left introduction rule is the *induction principle*, where S is the inductive invariant; its right premise shows that the invariant is a prefixed point (i.e., $BS \subseteq S$), whereas the left premise shows that the invariant can be used in lieu of the fixed point to prove the base goal. Note that the right premise operates on a fresh set of universal variables used as arguments, $\bar{x}$. In addition, from *induct* it is possible to derive a left unfolding rule as a particular case:

$$\frac{\Gamma, B(\mu B)\bar{t} \vdash \mathcal{R}}{\Gamma, \mu B\bar{t} \vdash \mathcal{R}}$$

The greatest fixed point with its introduction rules is the dual of the least fixed point, and completely symmetric with it:

$$\frac{\Gamma \vdash S\bar{t} \quad S\bar{x} \vdash BS\bar{x}}{\Gamma \vdash \nu B\bar{t}} \; coinduct \qquad\qquad \frac{\Gamma, B(\nu B)\bar{t} \vdash \mathcal{R}}{\Gamma, \nu B\bar{t} \vdash \mathcal{R}} \; unfold$$

And with them, the right unfolding rule as a particular case of coinduction:

$$\frac{\Gamma \vdash B(\nu B)\bar{t}}{\Gamma \vdash \nu B\bar{t}}$$

The rest of the chapter is organized as follows. Section 9.2 structures the developments outlined in this section as part of an intuitionistic sequent calculus. Section 9.3 extends the FPC framework with kernels based on those rich intuitionistic proof systems. Section 9.4 presents the concept of nominal abstraction and integrates it in the proof system. Section 9.5 concludes the chapter.

9.2 Focused sequent calculus

Section 2.4 presented the foundations of sequent calculus proof systems applied to classical logic—the dominant paradigm in automated theorem proving and in all of Part II. It was noted then that an intuitionistic sequent calculus results from a simple restriction on standard two-side sequents: namely, that at most one formula appear on the right-hand side. The pervasive symmetry of the classical

setting allows a presentation based on one-sided sequents, on which the discipline of focusing was presented. This addition entailed a redesign of the structural rules of the traditional *LK* which preserved the soundness and completeness of the resulting *LKF* with respect to classical logic.

This chapter lays the necessary proof theoretical foundations for the remaining chapters. The present Part III is closer to the world of inductive definitions, proof scripts, etc., which is characteristic of proof assistants, themselves typically built upon constructive logics. From the point of view of the sequent calculus, it is a simple matter to streamline the presentation of *LK* into that of *LJ* by enforcing the intuitionistic restriction at the level of inference rules. An important detail concerns the division between additive and multiplicative connectives. In the classical sequent calculus, the two sets of rules are interadmissible—in fact, the focused calculus *LKF* integrates both—but as can be seen from Figure 2.2, the right rule for multiplicative disjunction violates the intuitionistic restriction. Ergo, only additive disjunction is intuitionistically valid. As focusing is added to *LJ* to obtain *LJF*, only the positive disjunction (written ∨ instead of ∨⁺) is present in the system; both positive and negative conjunctions continue to coexist.

Figure 9.2 shows the focused sequent calculus *LJF*. In contrast with Figure 2.4, it presents a two-sided development of a focused calculus; in addition, it integrates the constructive constraints in the rules instead of maintaining them as side conditions. Like Figure 2.4 (as opposed to, say, Figure 2.1), it features directly the structural rules adapted to the focusing discipline (as opposed to the more traditional weakening, contraction, etc.). Both factors (intuitionistic, two-sided) contribute to a more complex taxonomy of sequents:

1. **Unfocused sequents** $\Gamma \Uparrow \Theta \vdash \mathcal{R}$ divide their left-hand side into two zones: storage, Γ, and workbench, Θ. The right-hand side $\mathcal{R}$ is more interesting, because it contains exactly one formula and must model a limited single-place, storage-or-workbench division: the formula must be assigned to one of these "areas." Thus: (a) $\Gamma \Uparrow \Theta \vdash B \Uparrow$ has the RHS formula B in the workbench; and (b) $\Gamma \Uparrow \Theta \vdash \Uparrow B$ has B in storage.

2. **Focused sequents on the left** $\Gamma \Downarrow B \vdash \mathcal{R}$, where the left workbench contains exactly the formula under focus. On the RHS, the goal must be in storage, because otherwise the asynchronous phase would not have finished giving way to a focus, in this case on the left.

3. **Focused sequents on the right** $\Gamma \vdash B \Downarrow$, where the right workbench contains the formula under focus, and the LHS workbench must be empty.

The two-sided focused sequent calculus further imposes a deterministic order of evaluation of formulas across all workbenches: first the proper workbench on the left-hand side, then the right-hand side formula if it is in "workbench mode." In spite of the greater superficial variety, the division between asynchronous and synchronous phases is identical to that in the one-sided calculus. However, two-sided calculi are bigger (in number of inference rules) than one-sided calculi, all situations—except the self-symmetrical cut—have to be treated on both sides of the sequent. In exchange, the two-sided presentation emphasizes the symmetries in the calculus and allows for more uniform presentation across logics.

Note that (two-sided) sequents in the asynchronous phase have two storage-or-workbench divisions represented by two up-arrows. In the asynchronous phase, a sequent has a formula under focus in the workbench of one of the sides (marked by the usual down-arrow) while the other side is storage-only, and the second arrow that separates this zone from the empty workbench is omitted.

The *LJF* proof system with added fixed points and equality will become μLJF, the basis of our subsequent study. In *LJF*, the connectives $\wedge^+$, $\vee$, t^+, f^+, and $\exists$ are positive; the connectives $\wedge^-$, $\supset$, t^-, f^-, and $\forall$ are negative. As for the new connectives, equality, $=$, and the least fixed point, μ, are defined as positive; the greatest fixed point, ν, is defined as negative. These polarities are natural choices for the semantics of the fixed points, though it may be possible to assign them differently (Baelde, 2008b, Chapter 4). The set of focused inference rules that expand *LJF* into μLJF are given in Figure 9.3. There are now essentially three operations (and their corresponding inference rules) that can be used to treat a least fixed point formula on the left-hand side of the sequent:

1. The most substantial inference rule on least fixed points is the *induction* rule. In its general form, its premises involve an induction *invariant* (as we will see in Chapter 11, there exist common simplifications that apply in most situations). Like the cut rule, induction is non-analytic in the sense that its inference rule does not have the subformula property.

2. The least fixed point can be *unfolded* on the left as a direct consequence of the induction rule.

3. The fixed point can be *frozen* in the sense that when a dedicated version of the store-left rule is applied to it, the resulting occurrence of the fixed

point in storage will never be unfolded again, nor will it be the site of an induction. Such frozen fixed points can only be used later in proof construction within an instance of the initial rule.

All three rules take place in the asynchronous phase. They are completed by the right-unfold rule on least fixed points in the synchronous phase. Dually, greatest fixed points feature synchronous unfolding on the left and three asynchronous rules on the right: coinduction (also non-analytic), right unfolding as a consequence of coinduction, and freezing.

The freezing rules deserve special mention. As we process a fixed point asynchronously (say, μ on the left), at a certain point in the proof we decide to fix it and never modify it again (i.e., by inductive rules). As the fixed point is effectively negative, the storage rules would never operate on in. The freezing rule moves the fixed point to a region of storage reserved for fixed points, which one stored (i.e., frozen) are never decided upon. Following Blanco and Miller (2015), we model this behavior through a dedicated *frozen zone*, written Φ. In the case of greatest fixed points, the homologous process takes place with one important distinction: the single-formula slot on the RHS functions as a multi-purpose zone. Once a greatest fixed point is moved to right storage (i.e., frozen), the whole RHS is blocked until proof's end.

Frozen fixed points come into play in the revised initial rules. These rule now search not an atom of complementary polarity on the opposite side, but an identical (i.e., unifiable, see above) and unalterable (i.e., frozen) atom on the opposite frozen zone. In short, fixed points play the role of atoms and replace the "undefined atoms"—not definitional—with fixed points. The other connective that intervenes in the finalization of branches is equality. Interestingly, both new types of connectives—fixed points and equality—are used as initial-style rules, and both involve unification problems at the logic level.

Critically, the addition of focusing must preserve the set of theorems of the original unfocused system:

9.2.1 Theorem   The system μLJF is sound and complete w.r.t. μLJ.

Proof. Proved in Baelde (2008b). $\square$

9.3 Augmentations and kernels

In the same way LJF is extended to μLJF—and, before them, LJ is extended to μLJ—by the addition of fixed points and equality, the inference rules in Figure 9.3,

ASYNCHRONOUS INTRODUCTION RULES

$$\frac{\Gamma \Uparrow A, B, \Theta \vdash \mathcal{R}}{\Gamma \Uparrow A \wedge^+ B, \Theta \vdash \mathcal{R}} \qquad\qquad \frac{\Gamma \Uparrow \Theta \vdash \mathcal{R}}{\Gamma \Uparrow t^+, \Theta \vdash \mathcal{R}}$$

$$\frac{\Gamma \Uparrow \vdash A \Uparrow \quad \Gamma \Uparrow \vdash B \Uparrow}{\Gamma \Uparrow \vdash A \wedge^- B \Uparrow} \qquad\qquad \frac{}{\Gamma \Uparrow \vdash t^- \Uparrow}$$

$$\frac{\Gamma \Uparrow A, \Theta \vdash \mathcal{R} \quad \Gamma \Uparrow B, \Theta \vdash \mathcal{R}}{\Gamma \Uparrow A \vee B, \Theta \vdash \mathcal{R}} \qquad\qquad \frac{}{\Gamma \Uparrow f, \Theta \vdash \mathcal{R}}$$

$$\frac{\Gamma \Uparrow A \vdash B \Uparrow}{\Gamma \Uparrow \vdash A \supset B \Uparrow} \qquad \frac{\Gamma \Uparrow \vdash [y/x]B \Uparrow}{\Gamma \Uparrow \vdash \forall x.B \Uparrow} \dagger \qquad \frac{\Gamma \Uparrow [y/x]B, \Theta \vdash \mathcal{R}}{\Gamma \Uparrow \exists x.B, \Theta \vdash \mathcal{R}} \dagger$$

SYNCHRONOUS INTRODUCTION RULES

$$\frac{\Gamma \vdash A \Downarrow \quad \Gamma \vdash B \Downarrow}{\Gamma \vdash A \wedge^+ B \Downarrow} \qquad\qquad \frac{}{\Gamma \vdash t^+ \Downarrow}$$

$$\frac{\Gamma \Downarrow A_i \vdash \mathcal{R}}{\Gamma \Downarrow A_1 \wedge^- A_2 \vdash \mathcal{R}} \qquad\qquad \frac{\Gamma \vdash A_i \Downarrow}{\Gamma \vdash A_1 \vee A_2 \Downarrow}$$

$$\frac{\Gamma \vdash A \Downarrow \quad \Gamma \Downarrow B \vdash \mathcal{R}}{\Gamma \Downarrow A \supset B \vdash \mathcal{R}} \qquad \frac{\Gamma \Downarrow [t/x]B \vdash \mathcal{R}}{\Gamma \Downarrow \forall x.B \vdash \mathcal{R}} \qquad \frac{\Gamma \vdash [t/x]B \Downarrow}{\Gamma \vdash \exists x.B \Downarrow}$$

IDENTITY RULES

$$\frac{}{\Gamma \Downarrow N_a \vdash N_a} \; init_l \qquad \frac{}{\Gamma, P_a \vdash P_a \Downarrow} \; init_r \qquad \frac{\Gamma \Uparrow \vdash B \Uparrow \quad \Gamma \Uparrow B \vdash \Uparrow \mathcal{R}}{\Gamma \Uparrow \vdash \Uparrow \mathcal{R}} \; cut$$

STRUCTURAL RULES

$$\frac{\Gamma, N \Downarrow N \vdash \mathcal{R}}{\Gamma, N \Uparrow \vdash \Uparrow \mathcal{R}} \; decide_l \qquad\qquad \frac{\Gamma \vdash P \Downarrow}{\Gamma \Uparrow \vdash \Uparrow P} \; decide_r$$

$$\frac{\Gamma \Uparrow P \vdash \Uparrow \mathcal{R}}{\Gamma \Downarrow P \vdash \mathcal{R}} \; release_l \qquad\qquad \frac{\Gamma \Uparrow \vdash N \Uparrow}{\Gamma \vdash N \Downarrow} \; release_r$$

$$\frac{C, \Gamma \Uparrow \Theta \vdash \mathcal{R}}{\Gamma \Uparrow C, \Theta \vdash \mathcal{R}} \; store_l \qquad\qquad \frac{\Gamma \Uparrow \vdash \Uparrow D}{\Gamma \Uparrow \vdash D \Uparrow} \; store_r$$

9.2 Figure The *LJF* focused proof system for intuitionistic logic (Liang and Miller, 2009). Here, P is a positive formula; N is a negative formula; P_a is a positive literal; N_a is a negative literal; A and B are arbitrary formulas; C is a negative formula or a positive literal; and D is a positive formula or a negative literal. $\mathcal{R}$ represents an arbitrary right-hand side; in structural and initial rules, a mixed style that combines this generic symbol and some sequent arrows is used. The proviso marked as † is the usual eigenvariable restriction.

ASYNCHRONOUS INTRODUCTION RULES

$$\frac{\Phi;\Gamma \Uparrow S\,\bar{t},\Theta \vdash \mathcal{R} \quad \Phi;\Gamma \Uparrow B\,S\,\bar{y} \vdash S\,\bar{y}\Uparrow}{\Phi;\Gamma \Uparrow \mu B\,\bar{t},\Theta \vdash \mathcal{R}}\ inductL\ \dagger$$

$$\frac{\Phi;\Gamma \Uparrow\ \vdash S\bar{t}\Uparrow \quad \Phi;\Uparrow S\bar{y} \vdash BS\bar{y}\Uparrow}{\Phi;\Gamma \Uparrow\ \vdash \nu B\bar{t}\Uparrow}\ inductR\ \dagger$$

$$\frac{\Phi;\Gamma \Uparrow\ \vdash B(\nu B)\,\bar{t}\Uparrow}{\Phi;\Gamma \Uparrow\ \vdash \nu B\,\bar{t}\Uparrow}\ unfoldL \qquad\qquad \frac{\Phi;\Gamma \Uparrow B(\mu B)\,\bar{t},\Theta \vdash \mathcal{R}}{\Phi;\Gamma \Uparrow \mu B\,\bar{t},\Theta \vdash \mathcal{R}}\ unfoldR$$

$$\frac{\mu B\,\bar{t},\Phi;\Gamma \Uparrow \Theta \vdash \mathcal{R}}{\Phi;\Gamma \Uparrow \mu B\,\bar{t},\Theta \vdash \mathcal{R}}\ freezeL \qquad \frac{\Phi;\Gamma \Uparrow\ \vdash\ \Uparrow^{\Phi} \nu B\,\bar{t}}{\Phi;\Gamma \Uparrow\ \vdash \nu B\,\bar{t}\Uparrow}\ freezeR$$

$$\frac{\{\Phi;\Gamma \Uparrow \Theta \vdash \mathcal{R}\}\sigma \quad \sigma \in CSU(s=t)}{\Phi;\Gamma \Uparrow s=t,\Theta \vdash \mathcal{R}}$$

SYNCHRONOUS INTRODUCTION RULES

$$\frac{\Phi;\Gamma \vdash B(\mu B)\,\bar{t}\Downarrow}{\Phi;\Gamma \vdash \mu B\,\bar{t}\Downarrow}\ unfoldR \qquad \frac{\Phi;\Gamma \Downarrow B(\nu B)\,\bar{t} \vdash \mathcal{R}}{\Phi;\Gamma \Downarrow \nu B\,\bar{t} \vdash \mathcal{R}}\ unfoldL \qquad \frac{}{\Phi;\Gamma \Downarrow t=t \vdash \mathcal{R}}$$

IDENTITY RULES

$$\frac{\mu B\,\bar{t} \in \Phi}{\Phi;\Gamma \vdash \mu B\,\bar{t}\Downarrow}\ initR \qquad\qquad \frac{}{\Phi;\Gamma \Downarrow \nu B\,\bar{t} \vdash\ \Uparrow^{\Phi}\nu B\,\bar{t}}\ initL$$

9.3 **Figure**   The *µLJF* focused proof system for intuitionistic logic with fixed points (Baelde et al., 2010). This figure contains the new inference rules for least and greatest fixed points and equality which are added to the proof system in Figure 9.2. The style of encoding follows Blanco and Miller (2015). A new storage zone for frozen (least) fixed points, Φ, is added to all sequents and threaded throughout all existing inference rules. When a greatest fixed point is frozen, the entire right-hand side becomes frozen and can no longer be manipulated; this new restriction is represented by the storage-only annotation $\Uparrow^{\Phi}$. Two new initial rules replace atoms with fixed points: an initial rule applies to a fixed point under focus if a frozen copy of the same fixed point is available on the opposite side of the sequent. Besides these changes, presentation conventions are shared with Figure 9.2.

ASYNCHRONOUS INTRODUCTION RULES

$$\frac{\Xi_1:\ \Gamma \Uparrow A, B, \Theta \vdash \mathcal{R} \qquad \wedge_c^+(\Xi_0, \Xi_1)}{\Xi_0:\ \Gamma \Uparrow A \wedge^+ B, \Theta \vdash \mathcal{R}} \qquad\qquad \frac{\Xi_1:\ \Gamma \Uparrow \Theta \vdash \mathcal{R} \qquad t_c^+(\Xi_0, \Xi_1)}{\Xi_0:\ \Gamma \Uparrow t^+, \Theta \vdash \mathcal{R}}$$

$$\frac{\Xi_1:\ \Gamma \Uparrow\ \vdash A \Uparrow \qquad \Xi_1:\ \Gamma \Uparrow\ \vdash B \Uparrow \qquad \wedge_c^-(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0:\ \Gamma \Uparrow\ \vdash A \wedge^- B \Uparrow} \qquad\qquad \frac{t_c^-(\Xi_0)}{\Xi_0:\ \Gamma \Uparrow\ \vdash t^- \Uparrow}$$

$$\frac{\Xi_1:\ \Gamma \Uparrow A, \Theta \vdash \mathcal{R} \qquad \Xi_2:\ \Gamma \Uparrow B, \Theta \vdash \mathcal{R} \qquad \vee_c(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0:\ \Gamma \Uparrow A \vee B, \Theta \vdash \mathcal{R}} \qquad\qquad \frac{f_c^+(\Xi_0)}{\Xi_0:\ \Gamma \Uparrow f, \Theta \vdash \mathcal{R}}$$

$$\frac{\Xi_1:\ \Gamma \Uparrow A \vdash B \Uparrow \qquad \supset_c(\Xi_0, \Xi_1)}{\Xi_0:\ \Gamma \Uparrow\ \vdash A \supset B \Uparrow}$$

$$\frac{(\Xi_1 y):\ \Gamma \Uparrow\ \vdash [y/x]B \Uparrow \qquad \forall_c(\Xi_0, \Xi_1)}{\Xi_0:\ \Gamma \Uparrow\ \vdash \forall x.B \Uparrow}$$

$$\frac{(\Xi_1 y):\ \Gamma \Uparrow [y/x]B, \Theta \vdash \mathcal{R} \qquad \exists_c(\Xi_0, \Xi_1)}{\Xi_0:\ \Gamma \Uparrow \exists x.B, \Theta \vdash \mathcal{R}}$$

SYNCHRONOUS INTRODUCTION RULES

$$\frac{\Xi_1:\ \Gamma \vdash A \Downarrow \qquad \Xi_2:\ \Gamma \vdash B \Downarrow \qquad \wedge_e^+(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0:\ \Gamma \vdash A \wedge^+ B \Downarrow} \qquad\qquad \frac{t_e^+(\Xi_0)}{\Xi_0:\ \Gamma \vdash t^+ \Downarrow}$$

$$\frac{\Xi_1:\ \Gamma \Downarrow A_i \vdash \mathcal{R} \qquad \wedge_e^-(\Xi_0, \Xi_1, i)}{\Xi_0:\ \Gamma \Downarrow A_1 \wedge^- A_2 \vdash \mathcal{R}} \qquad\qquad \frac{\Xi_1:\ \Gamma \vdash A_i \Downarrow \qquad \vee_e(\Xi_0, \Xi_1, i)}{\Xi_0:\ \Gamma \vdash A_1 \vee A_2 \Downarrow}$$

$$\frac{\Xi_1:\ \Gamma \vdash A \Downarrow \qquad \Xi_2:\ \Gamma \Downarrow B \vdash \mathcal{R} \qquad \supset_e(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0:\ \Gamma \Downarrow A \supset B \vdash \mathcal{R}}$$

$$\frac{\Xi_1:\ \Gamma \Downarrow [t/x]B \vdash \mathcal{R} \qquad \forall_e(\Xi_0, \Xi_1, t)}{\Xi_0:\ \Gamma \Downarrow \forall x.B \vdash \mathcal{R}}$$

$$\frac{\Xi_1:\ \Gamma \vdash [t/x]B \Downarrow \qquad \exists_e(\Xi_0, \Xi_1, t)}{\Xi_0:\ \Gamma \vdash \exists x.B \Downarrow}$$

9.4　**Figure**　The augmented *LJF^a* focused proof system for intuitionistic logic (Chihani et al., 2016b). Presentation conventions are shared with Figure 9.2.

IDENTITY RULES

$$\frac{initL_e(\Xi_0)}{\Xi_0:\ \Gamma \Downarrow N_a \vdash N_a} \qquad \frac{(l,\ P_a\ ) \in \Gamma \quad initR_e(\Xi_0, l)}{\Xi_0:\ \Gamma \vdash P_a \Downarrow}$$

$$\frac{\Xi_1:\ \Gamma \Uparrow \vdash F \Uparrow \quad \Xi_2:\ \Gamma \Uparrow F \vdash \Uparrow R \quad cut_e(\Xi_0, \Xi_1, \Xi_2, F)}{\Xi_0:\ \Gamma \Uparrow \vdash \Uparrow R}$$

STRUCTURAL RULES

$$\frac{\langle l, N \rangle \in \Gamma \quad \Xi_1:\ \Gamma \Downarrow N \vdash R \quad decideL_e(\Xi_0, \Xi_1, l)}{\Xi_0:\ \Gamma \Uparrow \vdash \Uparrow R}$$

$$\frac{\Xi_1:\ \Gamma \vdash P \Downarrow \quad decideR_e(\Xi_0, \Xi_1)}{\Xi_0:\ \Gamma \Uparrow \vdash \Uparrow P}$$

$$\frac{\Xi_1:\ \Gamma \Uparrow P \vdash \Uparrow R \quad releaseL_e(\Xi_0, \Xi_1)}{\Xi_0:\ \Gamma \Downarrow P \vdash R}$$

$$\frac{\Xi_1:\ \Gamma \Uparrow \vdash N \Uparrow \quad releaseR_e(\Xi_0, \Xi_1)}{\Xi_0:\ \Gamma \vdash N \Downarrow}$$

$$\frac{\Xi_1:\ \langle l, C \rangle, \Gamma \Uparrow \Theta \vdash \mathcal{R} \quad storeL_c(\Xi_0, \Xi_1, l)}{\Xi_0:\ \Gamma \Uparrow C, \Theta \vdash \mathcal{R}}$$

$$\frac{\Xi_1:\ \Gamma \Uparrow \vdash \Uparrow D \quad storeR_c(\Xi_0, \Xi_1)}{\Xi_0:\ \Gamma \Uparrow \vdash D \Uparrow}$$

9.5 Figure The augmented *LJF^a* focused proof system for intuitionistic logic (continued). Presentation conventions are shared with Figure 9.2. The identity of each inference rule is immediate from its corresponding clerk or expert; names are therefore omitted.

ASYNCHRONOUS INTRODUCTION RULES

$$\frac{\Xi_1: \quad \Phi; \Gamma \Uparrow S\,\bar{t}, \Theta \vdash \mathcal{R} \quad (\Xi_2\,\bar{y}): \quad \Phi; \Gamma \Uparrow B\,S\,\bar{y} \vdash S\,\bar{y} \Uparrow \quad inductL_c\,(\Xi_0, \Xi_1, \Xi_2, S)}{\Xi_0: \quad \Phi; \Gamma \Uparrow \mu B\,\bar{t}, \Theta \vdash \mathcal{R}} \dagger$$

$$\frac{\Xi_1: \quad \Phi; \Gamma \Uparrow B(\mu B)\,\bar{t}, \Theta \vdash \mathcal{R} \quad unfoldL_c\,(\Xi_0, \Xi_1)}{\Xi_0: \quad \Phi; \Gamma \Uparrow \mu B\,\bar{t}, \Theta \vdash \mathcal{R}}$$

$$\frac{\Xi_1: \quad \langle l, \mu B\,\bar{t} \rangle, \Phi; \Gamma \Uparrow \Theta \vdash \mathcal{R} \quad freezeL_c\,(\Xi_0, \Xi_1, l)}{\Xi_0: \quad \Phi; \Gamma \Uparrow \mu B\,\bar{t}, \Theta \vdash \mathcal{R}}$$

$$\frac{\Xi_1: \quad \Phi; \Gamma \Uparrow \,\vdash S\bar{t} \Uparrow \quad (\Xi_2\,\bar{y}): \quad \Phi; \Uparrow S\bar{y} \vdash B S\bar{y} \Uparrow \quad inductR_c\,(\Xi_0, \Xi_1, \Xi_2, S)}{\Xi_0: \quad \Phi; \Gamma \Uparrow \,\vdash \nu B\bar{t} \Uparrow} \dagger$$

$$\frac{\Xi_1: \quad \Phi; \Gamma \Uparrow \,\vdash B(\nu B)\,\bar{t} \Uparrow \quad unfoldR_c\,(\Xi_0, \Xi_1)}{\Xi_0: \quad \Phi; \Gamma \Uparrow \,\vdash \nu B\,\bar{t} \Uparrow}$$

$$\frac{\Xi_1: \quad \Phi; \Gamma \Uparrow \,\vdash \Uparrow^\Phi \nu B\,\bar{t} \quad freezeR_c\,(\Xi_0, \Xi_1)}{\Xi_0: \quad \Phi; \Gamma \Uparrow \,\vdash \nu B\,\bar{t} \Uparrow}$$

$$\frac{\{\Xi_1: \quad \Phi; \Gamma \Uparrow \Theta \vdash \mathcal{R}\}\sigma \quad \sigma \in CSU(s = t) \quad =_c\,(\Xi_0, \Xi_1)}{\Xi_0: \quad \Phi; \Gamma \Uparrow s = t, \Theta \vdash \mathcal{R}}$$

9.6 Figure   The augmented μLJF^a focused proof system for intuitionistic logic with fixed points (Blanco and Miller, 2015). Presentation conventions are shared with Figure 9.3.

SYNCHRONOUS INTRODUCTION RULES

$$\frac{\Xi_1: \ \Phi; \Gamma \vdash B(\mu B)\, \bar{t} \Downarrow \quad \boxed{unfoldR_e(\Xi_0, \Xi_1)}}{\Xi_0: \ \Phi; \Gamma \vdash \mu B\, \bar{t} \Downarrow}$$

$$\frac{\Xi_1: \ \Phi; \Gamma \Downarrow B(\nu B)\, \bar{t} \vdash \mathcal{R} \quad \boxed{unfoldL_e(\Xi_0, \Xi_1)}}{\Xi_0: \ \Phi; \Gamma \Downarrow \nu B\, \bar{t} \vdash \mathcal{R}}$$

$$\frac{\boxed{=_e (\Xi_0)}}{\Xi_0: \ \Phi; \Gamma \Downarrow t = t \vdash \mathcal{R}}$$

IDENTITY RULES

$$\frac{\langle l,\, \mu B\, \bar{t}\, \rangle \in \Phi \quad \boxed{initR_e(\Xi_0, l)}}{\Xi_0: \ \Phi; \Gamma \vdash \mu B\, \bar{t} \Downarrow}$$

$$\frac{\boxed{initL_e(\Xi_0)}}{\Xi_0: \ \Phi; \Gamma \Downarrow \nu B\, \bar{t} \vdash \Uparrow^{\Phi} \nu B\, \bar{t}}$$

9.7 Figure   The augmented μLJF^a focused proof system for intuitionistic logic with fixed points (continued). Presentation conventions are shared with Figure 9.3.

extended with clerks and experts by the usual method, are added to the augmented *LJF^a* to form the full augmented system *μLJF^a*: intuitionistic logic with fixed points (and equality) augmented with clerks and experts. The standard intuitionistic logic is augmented in Figure 9.4, and the new rules in the system for the fragment comprising fixed points and equality are shown in Figure 9.6.

The extension follows along the lines of Section 3.2. Notice that the new frozen storage, Φ, is indexed like Γ, but each is independent from the other. In particular, frozen fixed points are only selected from storage by the new initial-right rule—which, like decide-left on stored, non-frozen formulas, uses a non-deterministic index in the FPC framework. The single formula that may be frozen on the right requires no index because there is at most one choice.

Implementing the FPC framework for intuitionistic logics as kernels also proceeds as a natural extesion of the treatment of *LKF^a* in Section 4.4, although new technical considerations will come into play. They will be discussed in Section 10.3. A point of interest in using these kernels is the profusion of sources of nonterminating behavior—notably, by the inductive rules (including unfolding on both sides) of least and greatest fixed points. Relatedly, many more inference rules are in *conflict* with respect to *LKF^a*—where only decide and cut could be applied under the same conditions—; again, fixed points are the new source of conflict with the set of three rules which can be applied asynchronously on each kind of fixed point. An FPC definition will need to carefully orchestrate the operation on fixed points and their operations or risk copious backtracking and even nontermination.

As in the case of *LKF^a* by Theorem 3.2.2, usage of the augmented system is justified by a simple soundness guarantee formulated in terms of erasure of the augmentations of the FPC framework. This protects the system from anomalous behavior in client-supplied FPC definitions, as in the example of nonterminating unfolding above.

9.3.1 Theorem The system *μLJF^a* is sound w.r.t. intuitionistic logic with fixed points (*μLJ*).

Proof. The *μLJF* system can be recovered from *μLJF^a* by removing all the augmentations (marked in Figures 9.4 and 9.6), and therefore every proof of *μLJF^a* is also a proof of *μLJF*: this is the soundness guarantee. The result follows from Theorem 9.2.1.   □

9.4 Nominal abstraction

The final extension to the logic, complementary to the addition of fixed points and equality, is *nominal abstraction*. In specifying and reasoning about structures, it is common to rely on a recursive traversal on inductive types; in many interesting cases, these constructs involve a notion of *binding*—pervasive, for example, in the naming effects of quantifiers in logic, and in foundational aspects programming language syntax such as variables and their scope. When inspecting terms of such types, we recurse inside the binders of a (globally) closed terms and consequently need to consider (locally) open terms. A standard technique to model the *dynamic* behavior of binders (as opposed to their *static* structure, reflected in the terms proper), involves the addition of an *evaluation context* recording open binders as the structure is recursed.

At the level of the meta-logic used to write the aforementioned specifications, quantifiers are the primitives that model binding. Indeed, universal quantification displays some desirable traits to perform this kind of reasoning. In the *intensional interpretation* adopted in the sequent calculus since Gentzen's original designs (e.g., Figure 2.3), the corresponding introduction rule quantifier—read bottom-up— states that to prove the universal quantifier it suffices to prove the formula where the bound variable has been substituted with a new *eigenvariable*: a fresh, unused variable at the appropriate type, unused elsewhere in the proof, which represents a *generic* instance of the type. For Gentzen, eigenvariables are immutable and unaffected by variable substitutions. However, the introduction of fixed points and equality—and the style of direct reasoning on logic specifications—turns eigenvariables into sites for substitution (in particular, equality on the left-hand side involves unification of eigenvariables).

9.4.1 Example Suppose there is a property P that takes two arguments. Under the reading of eigenvariables as fresh names, a proof of $\forall x.\forall y.Pxy$ involves two *different* names, x and y. Furthermore, a proof of $\forall z.Pz$ involves just one name in the proof of P. However, consider the following implication: $\forall x.\forall y.Pxy \supset \forall z.Pz$. Although this is logically valid, it can be interpreted under the conception of eigenvariables as fresh names as stating that if there is a proof of P using two different names there is a proof of P with a single name—which strays from the intended meaning of the specification.

Hence, the treatment of logic until this point conflates the concepts of universal quantification and generic judgment—by having the universal quantifier

assume these two sets of incompatible competencies. The distinction between the
two can be addressed directly within the logic by various means. Among these,
we use the concept of *nominal abstraction* embodied by a new *nabla* quantifier
(signified ∇) developed by Miller and Tiu (2002, 2005) to represent generic judg-
ments, i.e., statements relying on the declaration of fresh local variables. The
nabla quantifier extends sequents with an explicit representation of *local context*.
Consider the standard sequent notation from Section 2.4 (with the intuitionistic
restriction integrated):

$$\Sigma : A_1, \dots, A_n \vdash A_0$$

Here we have made explicit the *eigenvariable signature* Σ containing the set
of eigenvariables introduced, say, by the asynchronous rules on quantifiers in
Figure 9.2 (routinely signaled by the proviso † used to represent the eigenvariable
restriction)—Section 11.4 discusses an explicit encoding of this signature and
its limitations. Each formula in the sequent must now be decorated with a
local context σ, similar to the global context Σ but with scope limited to its
corresponding formula and containing the set of locally fresh variables (in effect,
turning sequents into binding structures):

$$\Sigma : \sigma_1 \triangleright A_1, \dots, \sigma_n \triangleright A_n \vdash \sigma_0 \triangleright A_0$$

The introduction rules for nabla are shown, for intuitionistic sequent calculus,
in Figure 9.8. Note that there is one introduction rule on the left and one on the
right, both identical in their treatment of the affected formula and its context—and
leaving the rest of the sequent intact. When focusing is applied, nabla exhibits
a self-duality which is reflected in the duplication of both introduction rules
in both asynchronous and synchronous phases. That is, nabla is unaffected by
focusing, a fundamentally neutral connective. The additions are compatible with
the extension to *μLJF* made in Figure 9.3.

The same Figure 9.8 represents the additions made in *LJF^a* (and *μLJF^a*). In its
dedicated purpose of reasoning about generic judgments, the nominal quantifier
is treated eagerly by the kernel. No clerks and experts are used, and its treatment
always succeeds. In an implementation of the FPC framework, this process is
carried out by the kernel, unbeknownst to the client side. Implementation issues
are studied in Section 10.3.

ASYNCHRONOUS INTRODUCTION RULES

$$\frac{\Gamma \Uparrow \;\vdash\; \sigma,(y:\tau) \rhd [y/x]B \Uparrow}{\Gamma \Uparrow \;\vdash\; \sigma \rhd \nabla(x:\tau).B \Uparrow} \;\dagger \qquad\qquad \frac{\Gamma \Uparrow \sigma,(y:\tau) \rhd [y/x]B, \Theta \vdash \mathcal{R}}{\Gamma \Uparrow \sigma \rhd \nabla(x:\tau).B, \Theta \vdash \mathcal{R}} \;\dagger$$

SYNCHRONOUS INTRODUCTION RULES

$$\frac{\Gamma \Downarrow \sigma,(y:\tau) \rhd [y/x]B \vdash \mathcal{R}}{\Gamma \Downarrow \sigma \rhd \nabla(x:\tau).B \vdash \mathcal{R}} \;\dagger \qquad\qquad \frac{\Gamma \vdash \sigma,(y:\tau) \rhd [y/x]B \Downarrow}{\Gamma \vdash \sigma \rhd \nabla(x:\tau).B \Downarrow} \;\dagger$$

9.8 **Figure** The *LJF* focused proof system for intuitionistic logic augmented with nominal quantification. In this presentation, the global context Σ is explicitly maintained, and local contexts are extended to all formulas. These changes are threaded throughout all other inference rules, although local contexts are only manipulated by the rules in this figure. All zones previously involving formulas are likewise augmented so that their members are pairs of formulas and their contexts. The freshness restriction $\dagger$ applies to all inference rules with respect to the freshly introduced and locally scoped *nominal constant*, y. Presentation conventions are shared with Figure 9.2.

9.5 Notes

The development of the proof theory of fixed points springs from a line of research that studies the use of definitions and induction (McDowell and Miller, 2000, 2002; Momigliano and Tiu, 2003; Miller and Tiu, 2005; Tiu and Momigliano, 2012). The use of fixed points was originally studied in the context of linear logic, where it constitutes an alternative to the exponentials for the modeling of unbounded behavior (Baelde and Miller, 2007; Baelde, 2008b, 2009, 2012). These results—extended to intuitionistic logic— provide the justification for designing proof systems in this way, and form the proof theoretical core of the development of reasoning systems like Bedwyr and Abella, to which the next chapter is devoted. An interesting question is whether results such as Theorem 9.3.1 extend to classical logic as well. This point is unknown: it is certainly possible to obtain a classical system with added fixed points, say, μLK, but the extension of focusing to a tentative μLKF remains an open problem.

The addition of least and greatest fixed points brings a pair of new initial rules: such formulas can function as "defined" atoms and eliminate the need for a separate category of proper ("undefined") atoms, such as has been used

assumed throughout Part II. Nevertheless, undefined atoms can be associated a more natural notion of genericity. Both defined and undefined atoms with their corresponding sets of inference rules can coexist in the same proof system instead of discarding one for the other.

The common kinds of sequents encountered in proofs involving undefined and defined atoms differ: in the former case, sequents with large numbers of small formulas abound; in the latter, sequents have fewer, substantially larger formulas. At the root is the definitional nature of fixed points: for example, suppose we define multiplication in terms of iterated addition. Figure 9.1 presents a least fixed point expression for addition; the corresponding least fixed point for multiplication follows a similar structure, but it moreover *inlines* the fixed point for addition. This growth in the size of individual expressions continues as more complex definitions are composed from existing ones.

Purely positive fixed points, where every connective is positive, occur commonly in logic specifications—such as that in Figure 9.1. In this context, focusing on a purely positive (least) fixed point on the right typically corresponds to the concept of performing a computation as part of a proof. When this is the case, the focus is never released and the proof succeeds or fails based on the ability of the specification to perform the required computation. For example, according to the aforementioned specification, $\Gamma \vdash plus\ 2\ 2\ 4\Downarrow$ will succeed—and in logic programming, since the third argument is functionally determined by the other two, the result of the addition operation can be computed via proof search—; conversely, $\Gamma \vdash plus\ 2\ 2\ 5\Downarrow$ will fail. This relationship has been studied in closer detail by Gérard and Miller (2017).

The nominal abstraction presented in this chapter is the full development of the nabla quantifier, which culminates in the work of Miller and Tiu (2005); Gacek et al. (2008b, 2011): this is the theory used in systems like Abella. A minimal presentation, which removes some of the standard properties of the connective, is developed in Baelde (2008a); we shall not consider it further here. Binding can be modeled by other, alternative means, but nominal abstraction is both powerful and convenient; Baelde et al. (2014, Section 6) provides a practical overview. We revisit the problem in Section 12.3.

Section 12.5 also returns to the question of negation-as-failure first observed in the rules for equality as a logical connective. The effects of equality in proof search (Viel and Miller, 2010) are closely related to unification, which itself will be a subject of further discussion in Section 13.2.

10 Proof search with fixed points

10.1 Automating logic

Traditionally, model checking has been seen as separate from theorem proving. Where theorem proving revolves around the concept of *provability*, model checking (Clarke et al., 1999; Baier and Katoen, 2008; Grumberg and Veith, 2008) considers *satisfiability* under a certain model. However, the extension of standard logics with fixed points unifies both views and allows model checking to be interpreted in terms of deduction, i.e., as a specific kind of theorem proving activity. It does so by observing that the exploration of fixed points captures both *finite success* and *finite failure*. In Part II, predicates were formally undefined (i.e., they did not have an associated introduction rule); instead, atoms were defined by a theory which specified how to derive conclusions from them. The introduction of fixed points enables the definition of recursive definitions directly within the logic. The definitions thus embedded are constant throughout proof search; this fact marks the move from the *open-world* to the *closed-world assumption*.

The Bedwyr system generalizes standard logic programming through the implementation of a fragment of the logics described in Chapter 9 that is nonetheless amenable to automation (Tiu et al., 2005; Baelde et al., 2007). The logic is organized in two levels, in such a way that all rules on the left are invertible and, in consequence, proof search alternates between the left and the right sides while giving preference to the left side, whose nondeterminism is by construction of the don't-care variety. The resulting language can be expressed compactly by the following grammar:

$$L_0 ::= t \mid A \mid L_0 \wedge L_0 \mid L_0 \vee L_0 \mid \exists x.L_0 \mid \nabla x.L_0$$
$$L_1 ::= t \mid A \mid L_1 \wedge L_1 \mid L_1 \vee L_1 \mid \exists x.L_1 \mid \nabla x.L_1 \mid \forall x.L_1 \mid L_0 \supset L_1$$

Here, each predicate A (taking the place of atomic formulas in Chapter 4) is classified as belonging to either L_0 or L_1. Moreover, predicates—encoded as fixed points—are stratified so that a definition may refer only to predicates at lower levels (of stratification, not to be confused with the two levels of the logic). Goals can be drawn from both L_0 and L_1. The restriction that forbids nested implications splits proof reconstruction in two levels and consequently two specialized provers, one for each class of formulas. The L_0 prover corresponds corresponds to a simplified version of λProlog extended to allow nabla in the body of clauses. In addition to adopting the observations about the organization of proof search in phases made above, the L_1 prover must treat the case of the implication $G_0 \supset D_1$. It does so in two steps:

1. First, attempt to prove the L_0 goal—where L_1 eigenvariables are treated as sites for substitution, i.e., L_0 logic variables, and L_1 logic variables are disallowed.

2. For every solution to G_0, apply its set of substitutions to D_1 and proceed to find a proof under those. As with similar rules, and empty set of solutions vacuously results in success.

Thus, in this logic, failure to prove a goal corresponds to a proof of the negation of the goal. Formulated in terms of standard, depth-first proof search, the state exploration associated to model checking properties can derive inefficient search and redundant treatment of goals. These particularities can be accommodated by the addition of *tabling* of proved goals reflecting the provability relation between entries in the tables (Miller and Nigam, 2007; Miller and Tiu, 2013). Both finite success and finite failure can be tabled.

Other, more expressive logics may not lend themselves well to full automation, but their automation is regardless of great interest. In front of programs such as automated theorem provers and model checkers is the group of tools called *proof assistants* (also, *interactive theorem provers*). Ultimately, the objective of both families of tools is the same—proving theorems by building formal proofs in a given logic—, but proof assistants rely on their users for instructions on how to build proofs (and may attempt to discharge simple goals by automated proving techniques). Their uses range from the formal verification of software—as undertaken in Section 6.3—to the rigorous proof of mathematical results so complex that they resist manual analysis, among these famously the four color theorem (Gonthier, 2005) and the Kepler conjecture (Hales et al., 2015).

A more powerful logic than the one implemented by Bedwyr is $\mathcal{G}$ (Gacek et al., 2011), which is the core of the Abella proof assistant (Gacek, 2008; Baelde et al., 2014). It shares with Bedwyr the two-level approach, which distinguishes a *reasoning level* (presented in the next section) and a *specification level*, which is implemented as a subset of λProlog. The rich logic with support for the λ-tree syntax approach makes it well suited to model and study the metatheory of programming languages (Gacek et al., 2008a, 2012; Wang et al., 2013). In fact, the $\mathcal{G}$ logic is very close to μLJF and its aggregated extensions from Chapter 9. Kernels and typical programs we will write in the reasoning level are compatible with the common subset of Abella and Bedwyr, which we call Bedwyr^0, and can be written in such a way that they are valid specifications in both systems simultaneously.

The rest of the chapter is organized as follows. Section 10.2 provides a tutorial introduction to Abella and Bedwyr and outlines a common dialect that can be used to write compatible specifications for both systems. Section 10.3 expounds the kernels that implement the FPC framework for the logics used in this part (namely, μLJF^a). Section 10.4 provides some additional examples. Section 10.5 concludes the chapter.

10.2 Abella

For the most part, the reasoning level of Abella coincides syntactically with Bedwyr—and semantically in the common fragment shared by both. Atomic types are defined by the keyword **Kind** and type constructors by the keyword **Type**; note that kind expressions make use of a lowercase **type**; currently, more complex kind expressions are not supported. For example:

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

The type of formulas at the reasoning level is called `prop` (as opposed to `o` at the specification level). The following logical constants are given:

1. `true` of type `prop`, for t.

2. `false` of type `prop`, for f.

3. `/\` of type `prop -> prop -> prop`, for $\wedge$.

4. `\/` of type `prop -> prop -> prop`, for ∨.

5. `->` of type `prop -> prop -> prop`, for ⊃.

6. `forall` of type `(A -> prop) -> prop`, for ∀.

7. `exists` of type `(A -> prop) -> prop`, for ∃.

8. `nabla` of type `(A -> prop) -> prop`, for ∇.

9. `=` of type `A -> A -> prop`, for =.

Note that Abella's implication is the implication at L_1 in the previous section and not the hypothetical implication of Section 4.3. A key difference with respect to λProlog is that predicates in Abella are defined as definitions with target type `prop`, all of whose clauses must be given at definition time, with clauses being separated with **;** and terminated with **.**, and the head and the body of a clause being separated with **:=**. Inductive definitions are given by **Define** and coinductive definitions are given by **CoDefine** declarations. Other syntactic conventions resemble those of λProlog. For example, for the inductive definition of natural numbers:

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

In λProlog, a predicate always fails if the theory defines no clauses for it. In Abella, this behavior must be made explicit through a clause in the definition:

```
Define undefined : nat -> nat -> prop by
  undefined X Y := false.
```

Inductive and coinductive definitions correspond to fixed points, respectively least and greatest (indeed, we will make use of an explicit correspondence in Section 13.4). Theorems and proofs are introduced by the **Theorem** environment, which gives name to a formula and follows by the description of a proof by a script written in the language of tactics of Abella. These are not too relevant to the present discussion; refer to the tutorial (Baelde et al., 2014) for details.

Modular support in Abella is very limited. At the beginning of a development, a specification written in a slightly modified λProlog can be loaded by the keyword `Specification` and specification-level predicates referenced by enclosing them in curly brackets. Although the specification is written in λProlog, the closed-world assumption is enforced on this level once Abella finishes loading

it. However, composition of λProlog modules is supported by the standard mechanism of accumulation. For example, if the code in Figure 4.2 is contained in a module named `test`, it could be used as follows:

```
Specification "test".
```

```
Theorem zero_is_natural : { is_nat z }.
search.
```

The equivalent specification at the reasoning level is given in Figure 11.1. The programming languages defined by Abella and Bedwyr are very closely related. In order to write code that is valid under both systems, a few important differences—mostly additional restrictions in Abella—must be noted:

1. Existential variables that do not appear in the head of a clause must be explicitly declared in Abella by `exists`. Such variables can be left implicit in Bedwyr, as they commonly are in λProlog, by resorting to the syntactic convention of uppercase identifiers.

2. Anonymous variables are not supported in Abella, and instead explicit named identifiers must be given. In Bedwyr, they can be written `_`.

3. Polymorphic types are not supported in Abella beyond those predefined by the logical constants (quantifers and equality). Hence, for example, separate list types and constructors must be defined for each type of list—as defined by the type of its elements. In Bedwyr they are written as in full λProlog.

4. Type constructors in Abella are always prefix. Bedwyr has conventions to define infix operators out of sequences of special characters, such as `|=` and `++`, which also determine associativity.

5. The predefined signature in Abella is essentially empty: everything needs to be built from scratch. Bedwyr defines string and natural literals with equality only.

6. Some meta-commands available in Bedwyr only support operations like file inclusion and assertions about finite success and finite failure—which have no direct correspondence in the stronger logic of Abella. These features need to be mimicked by external preprocessors and other Abella constructs.

Of this list, only the lack of polymorphic types is profound, although an experimental extension by Yuting Wang is currently in development. These

restrictions also extend to the λProlog interpreter in predictable ways, which does not at the moment implement the language in full—extended to the standard library of predefined types and their predicates, which is absent from the interpreter: everything in a development must be explicitly defined. We should also note that Bedwyr provides an incomplete implementation of higher-order pattern unification, which can lead to unexpected failures in correct code. Throughout Part III we shall sometimes, for the sake of conciseness in presentation, resort to the Bedwyr flavor of syntax, which more closely resembles the logic programs presented in λProlog.

10.3 FPC kernels

As an original development, adding to the existing kernels in λProlog, we implemented a family of kernels based on μLJF^a and its extensions, on which to base further experimentation. As in Section 4.4, it also serves to present an interesting use case for Abella as a logic programming language. Here we present the basic version of the kernel.

Figure 10.1 shows the encoding of the logic in Abella. There is no separation between the signature and the definition of the module, nor built-in mechanisms for module composition. Unlike in λProlog, composing a logic program from its constituent "modules" must carefully track the order of the dependencies between them—a more primitive endeavor. Based on the logic thus defined, Figure 10.2 shows how the fixed point definitions in Figure 9.1 can be encoded in Abella. These fixed points are here named as `Definitions` of Abella, which allows us to refer to them symbolically, say, when defining multiplication in terms of addition without having to write all definition strata every time. Thus, we could write:

```
Define times : (i -> bool) -> prop by times
(mu Pred\Args\ or
  (some N\
    (eq Args (zero ++ N ++ zero ++ argv)))
  (some K\ some M\ some N\ and
    (eq Args ((succ K) ++ M ++ N ++ argv))
    (some N'\ and
      (Pred (K ++ M ++ N' ++ argv))
      (Plus (N' ++ M ++ N ++ argv)))))
:=
plus Plus.
```

```
% Base types
Kind    bool, i    type.

% List types
Kind    list_bool    type.
Type    nil_bool     list_bool.
Type    cons_bool    bool -> list_bool -> list_bool.

Kind    list_i    type.
Type    nil_i     list_i.
Type    cons_i    i -> list_i -> list_i.

% Fixed point argument lists
Type    argv    i.
Type    arg@    i -> i -> i.

% Logical constants
Type    tt, ff          bool.
Type    and, or, imp    bool -> bool -> bool.
Type    all, some       (i -> bool) -> bool.
Type    eq              i -> i -> bool.
Type    mu, nu          ((i -> bool) -> i -> bool) -> i -> bool.

% Polarities
Define negative : bool -> prop by
  negative (imp P Q) ;
  negative (all P) ;
  negative (nu B T).

Define positive : bool -> prop by
  positive tt ; positive ff ;
  positive (and P Q) ; positive (or P Q) ;
  positive (some P) ;
  positive (eq P Q) ;
  positive (mu B T).
```

10.1   **Figure**   The logic *μLJF* encoded in Abella. The types of formulas and terms, `bool` and `i`, are given; term constructors must be given by the signature of the full proof checker. When list types are needed, dedicated constructors need to be declared as well, possibly together with a `member` predicate. The fixed point connectives are declared as taking a single argument, by convention representing a list of arguments encoded as a term of the logic by the reserved constructors `arg@` and `argv`.

```
Define is_nat : (i -> bool) -> prop by is_nat
(mu Pred\Args\
  (some N\ and
    (eq Args (N ++ argv))
    (or
      (eq N zero)
      (some N'\ and
        (eq N (succ N'))
        (Pred (N' ++ argv)))))).

Define plus : (i -> bool) -> prop by plus
(mu Pred\Args\
  (some K\ some M\ some N\ and
    (eq Args (K ++ M ++ N ++ argv))
    (or
      (and
        (eq K zero)
        (eq M N))
      (some K'\ some N'\ and (and
        (eq K (succ K'))
        (eq N (succ N')))
        (Pred (K' ++ M ++ N' ++ argv)))))).

Define is_nat' : (i -> bool) -> prop by is_nat'
(mu Pred\Args\ or
  (eq Args (zero ++ argv))
  (some N\ and
    (eq Args ((succ N) ++ argv))
    (Pred (N ++ argv)))).

Define plus' : (i -> bool) -> prop by plus'
(mu Pred\Args\ or
  (some N\
    (eq Args (zero ++ N ++ N ++ argv)))
  (some K\ some M\ some N\ and
    (eq Args ((succ K) ++ M ++ (succ N) ++ argv))
    (Pred (K ++ M ++ N ++ argv)))).
```

10.2 Figure   Logic specification of natural numbers and addition on them as a least
fixed points in *μLJF* encoded in Abella. For clarity, arg@ is written as infix
++, i.e., (M ++ N ++ argv) instead of (arg@ M (arg@ N argv)). Two
versions of each fixed point are given: first (unprimed), with a global pattern
match of the argument list and specializations in each clause; second (primed),
with pattern matching entirely contained within each clause.

Needless to say, this style is unwieldy and we would ideally like to avoid it altogether, relying instead on writting something like Abella definitions and having the fixed points generated automatically. Section 13.4 discusses how to use Abella to program μLJF.

Figure 10.3 shows the centerpiece of the development: the μLJF^a kernel in its basic, declarative version; it relies on standard declarations and logic programmings while presenting some features of interest. First, note that—unlike in λProlog—storage zones must be explicitly represented as part of the sequents, since hypothetical judgments do not allow us to grow the model at runtime under the open-world assumption. Second, there is a single instance of implication in the kernel (as there is in the LKF^a kernel); its role is not to file formulas in storage, but to treat equality on the left. In kernels with and without fixed points alike, these are the only instances of implication we have observed: practical clerks and experts are operationally simple programs which make no use of such advanced features—in fact, clerks and experts in Abella can be equivalently programmed in λProlog at the specification level with minimal changes, none profound. Unlike in λProlog, because of the closed-world assumption, a signature for clerks and experts independent from their definitions cannot be given, though it is implied by the kernel. If it were, or if clerks and experts were written in λProlog, it would be a variation on Figure 10.6.

A small number of secondary decisions are more dependent on the design of the particular implementation. For example, the development version of the kernel extends sequents with bookkeeping structures and spy harnesses to facilitate debugging. The ordering of the inference rules, in particular those that are in conflict, has a potentially large performance impact on proof search—in the kernel of Figure 10.3 fixed points operations have the following priorities: initial, unfold, induction, and freezing (subject to concrete FPC restrictions); when lemmas are added in Section 11.4, they will be placed after the standard decide rule. An interesting detail is that the implication on the left is treated by two distinct experts, each recursing on the premises in a different order; we used these to assess their impact in proof search, commonly finding advantageous to treat the antecedent first—under Bedwyr's incomplete handling of unification, this was often the only feasible option.

To conclude the section, we explore two extensions to the declarative kernel. In first place, in our encoding of the μLJF logic in Figure 10.1, we were able to devise a clever encoding that allowed us to express fixed points with arbitrary

```
Kind    cert, idx    type.
Kind    choice       type.
Type    left, right  choice.

Kind    goal  type.
Type    unk, sto, frz   bool -> goal.

Kind    ctx  type.
Type    kvp   idx -> bool -> ctx.

%                  Xi          Phi        Gamma
%                                         Delta     Goal
Define async : cert -> list_ctx -> list_ctx ->
                                   list_bool -> goal -> prop,
       syncL : cert -> list_ctx -> list_ctx ->
                                       bool -> goal -> prop,
       syncR : cert -> list_ctx -> list_ctx
                                       -> bool -> prop by

async Xi Phi Gamma (cons_bool (and P Q) Delta) G :=
  exists Xi', andClerk Xi Xi' /\
  async Xi' Phi Gamma (cons_bool P (cons_bool Q Delta)) G ;
async Xi Phi Gamma (cons_bool (or P Q) Delta) G :=
  exists Xi' Xi'', orClerk Xi Xi' Xi'' /\
  async Xi'  Phi Gamma (cons_bool P Delta) G /\
  async Xi'' Phi Gamma (cons_bool Q Delta) G ;
async Xi Phi Gamma nil_bool (unk (imp P Q)) :=
  exists Xi', impClerk Xi Xi' /\
  async Xi' Phi Gamma (cons_bool P nil_bool) (unk Q) ;
async Xi Phi Gamma (cons_bool ff Delta) G :=
  ffClerk Xi ;
async Xi Phi Gamma (cons_bool tt Delta) G :=
  exists Xi', ttClerk Xi Xi' /\
  async Xi' Phi Gamma Delta G ;
async Xi Phi Gamma nil_bool (unk (all P)) :=
  exists Xi', allClerk Xi Xi' /\
  forall x, async (Xi' x) Phi Gamma nil_bool (unk (P x)) ;
async Xi Phi Gamma (cons_bool (some P) Delta) G :=
  exists Xi', someClerk Xi Xi' /\ forall x,
  async (Xi' x) Phi Gamma (cons_bool (P x) Delta) G ;
async Xi Phi Gamma (cons_bool (eq P Q) Delta) G :=
  exists Xi', eqClerk Xi Xi' /\
  ((P = Q) -> async Xi' Phi Gamma Delta G) ;
```

10.3   Figure   The *μLJF^a* kernel in Abella.

```
async Xi Phi Gamma (cons_bool (mu B T) Delta) G :=
  exists Xi', unfoldLClerk Xi Xi' /\
  async Xi' Phi Gamma (cons_bool (B (mu B) T) Delta) G ;
async Xi Phi Gamma (cons_bool (mu B T) Delta) G :=
  exists Xi' Xi'' S, indClerk Xi Xi' Xi'' S /\
  async  Xi'     Phi Gamma (cons_bool (S T)   Delta   ) G /\
  forall x,
  async (Xi'' x) Phi Gamma (cons_bool (B S x) nil_bool)
                                                (unk (S x)) ;
async Xi Phi Gamma (cons_bool (mu B T) Delta) G :=
  exists Xi' Idx, freezeLClerk Xi Xi' Idx /\
  async
     Xi' (cons_ctx (kvp Idx (mu B T)) Phi) Gamma Delta G ;
async Xi Phi Gamma nil_bool (unk (nu B T)) :=
  exists Xi', unfoldRClerk Xi Xi' /\
  async Xi' Phi Gamma nil_bool (unk (B (nu B) T)) ;
async Xi Phi Gamma nil_bool (unk (nu B T)) :=
  exists Xi' Xi'' S, coindClerk Xi Xi' Xi'' S /\
  async  Xi'     Phi Gamma   nil_bool
                                              (unk (S T)) /\
  forall x,
  async (Xi'' x) Phi nil_ctx (cons_bool (S x) nil_bool)
                                              (unk (B S x)) ;
async Xi Phi Gamma nil_bool (unk (nu B T)) :=
  exists Xi', freezeRClerk Xi Xi' /\
  async Xi' Phi Gamma nil_bool (frz (nu B T)) ;
syncR Xi Phi Gamma (and P Q) :=
  exists Xi' Xi'', andExpert Xi Xi' Xi'' /\
  syncR Xi'  Phi Gamma P /\
  syncR Xi'' Phi Gamma Q ;
syncR Xi Phi Gamma (or P Q) :=
  exists Xi' C, orExpert Xi Xi' C /\ (
  (C = left  /\ syncR Xi' Phi Gamma P) \/
  (C = right /\ syncR Xi' Phi Gamma Q) ) ;
syncL Xi Phi Gamma (imp P Q) G :=
  exists Xi' Xi'', impExpert Xi Xi' Xi'' /\
  syncL Xi'  Phi Gamma Q G /\
  syncR Xi'' Phi Gamma   P ;
syncL Xi Phi Gamma (imp P Q) G :=
  exists Xi' Xi'', impExpert' Xi Xi' Xi'' /\
  syncR Xi'' Phi Gamma   P /\
  syncL Xi'  Phi Gamma Q G ;
syncR Xi Phi Gamma tt :=
  ttExpert Xi ;
```

10.4 **Figure** The *μLJF^a* kernel in Abella (continued).

```
syncL Xi Phi Gamma (all P) G :=
  exists Xi' T, allExpert Xi Xi' T /\
  syncL Xi' Phi Gamma (P T) G ;
syncR Xi Phi Gamma (some P) :=
  exists Xi' T, someExpert Xi Xi' T /\
  syncR Xi' Phi Gamma (P T) ;
syncR Xi Phi Gamma (eq T T) :=
  eqExpert Xi ;
syncL Xi Phi Gamma (nu B T) (frz (nu B T)) :=
  initLExpert Xi ;
syncL Xi Phi Gamma (nu B T) G :=
  exists Xi', unfoldLExpert Xi Xi' /\
  syncL Xi' Phi Gamma (B (nu B) T) G ;
syncR Xi Phi Gamma (mu B T) :=
  exists Idx, initRExpert Xi Idx /\
  member_ctx (kvp Idx (mu B T)) Phi ;
syncR Xi Phi Gamma (mu B T) :=
  exists Xi', unfoldRExpert Xi Xi' /\
  syncR Xi' Phi Gamma (B (mu B) T) ;
async Xi Phi Gamma (cons_bool C Delta) G :=
  exists Xi' Idx, negative C /\
  storeLClerk Xi Xi' Idx /\
  async Xi' Phi (cons_ctx (kvp Idx C) Gamma) Delta G ;
async Xi Phi Gamma nil_bool (unk G) :=
  exists Xi', positive G /\
  storeRClerk Xi Xi' /\
  async Xi' Phi Gamma nil_bool (sto G) ;
async Xi Phi Gamma nil_bool G :=
  exists Xi' Idx C ?1, (G = (sto ?1) \/ G = (frz ?1)) /\
  decideLClerk Xi Xi' Idx /\
  member_ctx (kvp Idx C) Gamma /\
  syncL Xi' Phi Gamma C G ;
async Xi Phi Gamma nil_bool (sto G) :=
  exists Xi', decideRClerk Xi Xi' /\
  syncR Xi' Phi Gamma G ;
syncL Xi Phi Gamma C G :=
  exists Xi', positive C /\
  releaseLExpert Xi Xi' /\
  async Xi' Phi Gamma (cons_bool C nil_bool) G ;
syncR Xi Phi Gamma G :=
  exists Xi', negative G /\
  releaseRExpert Xi Xi' /\
  async Xi' Phi Gamma nil_bool (unk G).
```

10.5 Figure   The *μLJF^a* kernel in Abella (finished).

```
Type andClerk     cert ->       cert  -> prop.
Type impClerk     cert ->       cert  -> prop.
Type ffClerk      cert                 -> prop.
Type ttClerk      cert ->       cert  -> prop.
Type allClerk     cert -> (i -> cert) -> prop.
Type someClerk    cert -> (i -> cert) -> prop.
Type eqClerk      cert ->       cert  -> prop.
Type unfoldLClerk cert ->       cert  -> prop.
Type unfoldRClerk cert ->       cert  -> prop.
Type freezeRClerk cert ->       cert  -> prop.
Type orClerk      cert -> cert  -> cert -> prop.
Type freezeLClerk cert -> cert  -> idx -> prop.
Type indClerk     cert -> cert  -> (i -> cert)
                                  -> (i -> bool) -> prop.
Type coindClerk   cert -> cert  -> (i -> cert)
                                  -> (i -> bool) -> prop.

Type andExpert     cert -> cert -> cert   -> prop.
Type orExpert      cert -> cert -> choice -> prop.
Type impExpert     cert -> cert -> cert   -> prop.
Type impExpert'    cert -> cert -> cert   -> prop.
Type ttExpert      cert                   -> prop.
Type allExpert     cert -> cert -> i      -> prop.
Type someExpert    cert -> cert -> i      -> prop.
Type eqExpert      cert                   -> prop.
Type initLExpert   cert                   -> prop.
Type unfoldLExpert cert -> cert           -> prop.
Type initRExpert   cert         -> idx    -> prop.
Type unfoldRExpert cert -> cert           -> prop.

Type decideLClerk   cert -> cert -> idx -> prop.
Type decideRClerk   cert -> cert        -> prop.
Type releaseLExpert cert -> cert        -> prop.
Type releaseRExpert cert -> cert        -> prop.
Type storeLClerk    cert -> cert -> idx -> prop.
Type storeRClerk    cert -> cert        -> prop.
```

10.6 Figure   The FPC signature of *μLJF^a* in Abella. Clerk and expert predicates must adhere to this hypothetical specification. If these predicates are given in λProlog, it suffices to change the type constructors to **type** and the target types to o; in this case, the separate signature can be given.

numbers of arguments, therefore obtaining a universal encoding of the connectives. When it comes to quantification, we are not so lucky. In the kernel for the *μLJF^a* logic—as with *LKF^a* before it—the object logic features quantification over the type of terms only. Even if all primitive kinds are reflected on the type i, arrow types cannot be modeled by this device alone—in fact, simple types are required to study the metatheory of interesting languages, as well as formalisms like the *π*-calculus. In the absence of polymorphism in Abella, a succinct encoding is not possible. A workable if unsatisfying solution involves the definition of separate sets of quantifiers at different types (using exclusively i and the arrow), encoding the inference rules for quantifiers in the kernel once for each type at which quantification is supported, and likewise cloning clerk and expert definitions. A summary of changes is given in Figure 10.7.

In second and last place, the addition of nominal abstraction to the logic, discussed in Section 9.4, must also be reflected in the kernel. For this we use a technique through which nominal variables are explicitly represented as local context with scope at the level of formulas. The encoding we used is based on based on the explicit representation of sequents developed by Miller and Tiu (2002); McDowell and Miller (2002)—we retain the terminology even though in our context it is no longer used to record eigenvariables. The resulting kernel is shown in Figure 10.8.

Under this regime, formulas are abstracted over a type which represents nominal variables as projections over the term of types of a counter stack: `fst rst`, `fst rst rst`, etc. When pattern matching a formula at the head (i.e., conclusion) of an inference rule, its components are themselves formula abstractions to which the abstraction variable is propagated. This extends to every inference rule in the system, with some representative examples detailed in Figure 10.9. All four inference rules for nabla, presented in Figure 9.8, are treated identically and transparently—in the sense that their are invisible to the FPC framework. In each case, a new nominal variable is generated by projecting the stack of `rst` to the type of terms through i, and a new `rst` is added to the stack for the continuation. However, pattern matching in this encoding generates unification problems that fall outside the fragment of pattern unification supported by Abella and Bedwyr—which, failing to find a solution, will be unable to perform any checking with this kernel. Despite this difficulty, the problems have simple solutions, which an extension of the unification framework (presented in Section 13.2) restores.

```
%% Logic

% Argument coercions for argument lists
Type    arg_ii    (i -> i) -> i.

% Logical constants
Type    all_ii, some_ii    ((i -> i) -> bool) -> bool.

% Polarity snippets
  negative (all_ii P) ;
  positive (some_ii P) ;

%% Mocked FPC signature

Type    allExpert_ii    cert -> cert -> (i -> i) -> prop.
Type    someExpert_ii   cert -> cert -> (i -> i) -> prop.
Type    allClerk_ii     cert -> ((i -> i) -> cert) -> prop.
Type    someClerk_ii    cert -> ((i -> i) -> cert) -> prop.

%% Kernel

syncL Xi Phi Gamma (all_ii P) G := exists Xi' T,
        allExpert_ii Xi Xi' T /\
        syncL Xi' Phi Gamma (P T) G ;

syncR Xi Phi Gamma (some_ii P) := exists Xi' T,
        someExpert_ii Xi Xi' T /\
        syncR Xi' Phi Gamma (P T) ;

syncL Xi Phi Gamma (all_ii P) G := exists Xi' T,
        allExpert_ii Xi Xi' T /\
        syncL Xi' Phi Gamma (P T) G ;

syncR Xi Phi Gamma (some_ii P) := exists Xi' T,
        someExpert_ii Xi Xi' T /\
        syncR Xi' Phi Gamma (P T) ;
```

10.7 Figure Extensions to the encoding of the μLJF^a logic to support polymorphic quantification in Abella. For each supported type, new connectives with their polarities, argument list contents, clerks and experts, and type-specific instances of the general inference rules are cloned. Here we show the additions for quantification over the arrow type $i \rightarrow i$.

```
% Explicit eigenvariable encoding
Kind    evs    type.
Type    fst    evs -> i.
Type    rst    evs -> evs.

% List types, with built-in abstraction over evs
Kind    list_bool    type.
Type    nil_bool     list_bool.
Type    cons_bool    (evs -> bool) -> list_bool -> list_bool.

% Object-level encoding of nabla, without polarity
Type    nabl   (i -> bool) -> bool.

% Kernel with treatment of nabla
Define async : cert -> list_ctx -> list_ctx ->
               list_bool      -> goal           -> prop,
       syncL : cert -> list_ctx -> list_ctx ->
               (evs -> bool) -> goal           -> prop,
       syncR : cert -> list_ctx -> list_ctx
                            -> (evs -> bool) -> prop by

async Xi Phi Gamma (cons_bool (l\ nabl (P l)) Delta) G :=
  async Xi Phi Gamma
                (cons_bool (l\ P (rst l) (fst l)) Delta) G ;

async Xi Phi Gamma nil_bool (unk (l\ nabl (P l))) :=
  async Xi Phi Gamma nil_bool (unk (l\ P (rst l) (fst l))) ;

syncL Xi Phi Gamma (l\ nabl (P l)) G :=
  syncL Xi Phi Gamma (l\ P (rst l) (fst l)) G ;

syncR Xi Phi Gamma (l\ nabl (P l)) :=
  syncR Xi Phi Gamma (l\ P (rst l) (fst l)) ;

% Kernel interface
Define prove : cert -> bool -> prop by
  prove Cert Form := exists Cert',
    unmarshal Cert Cert' /\
    async Cert' nil_ctx nil_ctx nil_bool (unk (l\ Form)).
```

10.8 Figure   Extensions to the μLJF^a kernel written in Abella to support nabla.

```
% Some standard and interesting cases
syncR Xi Phi Gamma (l\ and (P l) (Q l)) := exists Xi' Xi'',
  andExpert Xi Xi' Xi'' /\
  syncR Xi'  Phi Gamma P /\
  syncR Xi'' Phi Gamma Q ;

async Xi Phi Gamma nil_bool (unk (l\ all (P l))) :=
  exists Xi', allClerk Xi Xi' /\ forall x,
  async (Xi' x) Phi Gamma nil_bool (unk (l\ P l x)) ;

syncR Xi Phi Gamma (l\ some (P l)) := exists Xi' T,
  someExpert Xi Xi' T /\
  syncR Xi' Phi Gamma (l\ P l T) ;

async Xi Phi Gamma (cons_bool (l\ mu (B l) (T l)) Delta) G
  := exists Xi',
  unfoldLClerk Xi Xi' /\
  async Xi' Phi Gamma
          (cons_bool (l\ (B l) (mu (B l)) (T l)) Delta) G ;

async Xi Phi Gamma (cons_bool (l\ mu (B l) (T l)) Delta) G
  := exists Xi' Xi'' S,
  indClerk Xi Xi' Xi'' S /\
            async Xi'     Phi Gamma
        (cons_bool (l\ S (T l))   Delta   ) G /\
  forall x, async (Xi'' x) Phi Gamma
        (cons_bool (l\ (B l) S x) nil_bool) (unk (l\ S x)) ;
```

10.9 **Figure**   Extensions to the *μLJF^a* kernel written in Abella to support nabla (continued).

10.4 Examples

An interesting example adapts the pairing combinator introduced in Section 5.2 for the *LKF^a* system, to the present *μLJF^a* in Figure 10.10. The port is completely straightforward and presents no difficulties. It is important to note that a definition of the pairing clerks and experts in Abella is, by itself, useless—it requires other certificate definitions upon which to operate. In this closed world, then, it is not possible to write a self-contained definition of the FPC that is at the same time capable of interacting with the FPCs that use it. Consequently FPC definitions in Abella appear more (syntactically) complex than they (semantically) are. The solution to this problem is to use the specification level and write and compose definitions as λProlog modules by one of two means:

1. Modifying the kernel to accept FPC definitions at the specification level instead of at the reasoning level.

2. Generating FPC definitions at the reasoning level from definitions at the specification level by a preprocessor.

Either way, the specification in λProlog is not only much shorter, but also more legible and modular. Because *μLJF^a* is a two-sided calculus, the size of an FPC definition roughly doubles that of a similar one-sided calculus; this point is taken up again in Section 12.4.

10.5 Notes

Some of our developments on FPC kernels originate in work on Bedwyr and then ported to Abella, which is the one system that remains in active development. Bedwyr supports automation of proof search to a greater degree; aspects of this behavior could be built into Abella, to which end Chapter 13 offers an advanced preliminary study. The *μLJF^a* kernel was originally developed for the work presented in Blanco and Miller (2015) and subsequently refined. Support for nabla was added with a view towards the work presented in Blanco et al. (2017b).

In our discussion of quantifier polymorphism and the applicable workarounds, the battery of changes is limited to the *μLJF^a* system. When we exercise the proof system indirectly by writing Abella programs and reifying them into *μLJF^a*, this last step—for which refer to Section 13.4—will also need to be extended. Some versions of *LKF^a* and *LJF^a* kernels written in λProlog (Chihani et al.,

2016b) make use of this language's polymorphic features, though all the versions contemplated in this work are monomorphic.

A point by which we have set little store is the definition of a certificate transformer at the entry point of the kernel which is charged with performing *marshaling*. This functionality is supplied to streamline the definition of compact, initial forms of certificates, which are expanded to their full form before the start of checking proper. This is little more than a convenience, but removes from the user the burden of initializing bookkeeping structures in which they as clients have no direct interest. Marshaling will be used to define compact outline formats in Chapter 11. By default, the marshaling predicate can simply be taken to be the identity relation.

Proof scripts in Abella apply *tactics* to indirectly invoke the inference rules of the underlying logic $\mathcal{G}$. Upon success, a proof script generates a *witness* that records a trace of information that morally resembles the elaborations of Chapter 5—specifically Section 5.4—and could serve as the bedrock of thorough certification of Abella proofs. Abella lacks a language of *tacticals* to compose tactics from other tactics. All these aspects can be handled inside the FPC framework and are treated in further detail in Chapter 13.

If model checking can be seen as deduction, its proof evidence may also be expressed as proof certificates, as Heath and Miller (2015, 2017) have shown. Previous applications of model checking include the work of Mundhenk and Weiß (2010). Tabling has been studied, among others, by Ramakrishna et al. (1997); Yang et al. (2004); Tiu (2005).

At the beginning we compared model checking and theorem proving in terms of the relation between satisfiability and provability. It is worth observing that a similar connection has been observed in Chapter 7 between satisfiability and unsatisfiability—and, more generally, the notion of refutation.

```
% Signature
type   idx2   idx -> idx -> idx.
type   pair#  cert -> cert -> cert.

% Module.
ffClerk (pair# L0 R0) :-
  ffClerk L0, ffClerk R0.
ttClerk (pair# L0 R0) (pair# L1 R1) :-
  ttClerk L0 L1, ttClerk R0 R1.
andClerk (pair# L0 R0) (pair# L1 R1) :-
  andClerk L0 L1, andClerk R0 R1.
orClerk (pair# L0 R0) (pair# L1 R1) (pair# L2 R2) :-
  orClerk L0 L1 L2, orClerk R0 R1 R2.
impClerk (pair# L0 R0) (pair# L1 R1) :-
  impClerk L0 L1, impClerk R0 R1.
eqClerk (pair# L0 R0) (pair# L1 R1) :-
  eqClerk L0 L1, eqClerk R0 R1.
ttExpert (pair# L0 R0) :-
  ttExpert L0, ttExpert R0.
andExpert (pair# L0 R0) (pair# L1 R1) (pair# L2 R2) :-
  andExpert L0 L1 L2, andExpert R0 R1 R2.
orExpert (pair# L0 R0) (pair# L1 R1) C :-
  orExpert L0 L1 C, orExpert R0 R1 C.
impExpert (pair# L0 R0) (pair# L1 R1) (pair# L2 R2) :-
  impExpert L0 L1 L2, impExpert R0 R1 R2.
impExpert' (pair# L0 R0) (pair# L1 R1) (pair# L2 R2) :-
  impExpert' L0 L1 L2, impExpert' R0 R1 R2.
eqExpert (pair# L0 R0) :-
  eqExpert L0, eqExpert R0.
allClerk (pair# L0 R0) (x\ pair# (L1 x) (R1 x)) :-
  allClerk L0 L1, allClerk R0 R1.
someClerk (pair# L0 R0) (x\ pair# (L1 x) (R1 x)) :-
  someClerk L0 L1, someClerk R0 R1.
allExpert (pair# L0 R0) (pair# L1 R1) T :-
  allExpert L0 L1 T, allExpert R0 R1 T.
someExpert (pair# L0 R0) (pair# L1 R1) T :-
  someExpert L0 L1 T, someExpert R0 R1 T.
```

10.10 Figure   The pairing meta-FPC in Abella implemented at the specification level.

```
indClerk (pair# L0 R0)
         (pair# L1 R1) (x\ pair# (L2 x) (R2 x)) S :-
  indClerk L0 L1 L2 S, indClerk R0 R1 R2 S.
coindClerk (pair# L0 R0)
           (pair# L1 R1) (x\ pair# (L2 x) (R2 x)) S :-
  coindClerk L0 L1 L2 S, coindClerk R0 R1 R2 S.
unfoldLClerk (pair# L0 R0) (pair# L1 R1) :-
  unfoldLClerk L0 L1, unfoldLClerk R0 R1.
unfoldRExpert (pair# L0 R0) (pair# L1 R1) :-
  unfoldRExpert L0 L1, unfoldRExpert R0 R1.
unfoldLExpert (pair# L0 R0) (pair# L1 R1) :-
  unfoldLExpert L0 L1, unfoldLExpert R0 R1.
unfoldRClerk (pair# L0 R0) (pair# L1 R1) :-
  unfoldRClerk L0 L1, unfoldRClerk R0 R1.
freezeLClerk (pair# L0 R0) (pair# L1 R1) (idx2 IL IR) :-
  freezeLClerk L0 L1 IL, freezeLClerk R0 R1 IR.
initRExpert (pair# L0 R0) (idx2 IL IR) :-
  initRExpert L0 IL, initRExpert R0 IR.
freezeRClerk (pair# L0 R0) (pair# L1 R1) :-
  freezeRClerk L0 L1, freezeRClerk R0 R1.
initLExpert (pair# L0 R0) :-
  initLExpert L0, initLExpert R0.
storeLClerk (pair# L0 R0) (pair# L1 R1) (idx2 IL IR) :-
  storeLClerk L0 L1 IL, storeLClerk R0 R1 IR.
decideLClerk (pair# L0 R0) (pair# L1 R1) (idx2 IL IR) :-
  decideLClerk L0 L1 IL, decideLClerk R0 R1 IR.
storeRClerk (pair# L0 R0) (pair# L1 R1) :-
  storeRClerk L0 L1, storeRClerk R0 R1.
decideRClerk (pair# L0 R0) (pair# L1 R1) :-
  decideRClerk L0 L1, decideRClerk R0 R1.
releaseLExpert (pair# L0 R0) (pair# L1 R1) :-
  releaseLExpert L0 L1, releaseLExpert R0 R1.
releaseRExpert (pair# L0 R0) (pair# L1 R1) :-
  releaseRExpert L0 L1, releaseRExpert R0 R1.
```

10.11 **Figure** The pairing meta-FPC in Abella implemented at the specification level (continued).

11 Proof outlines

11.1 Frege proofs

Consider the familiar notion of *Frege proofs*—also known as *Hilbert proofs*—: lists of formulas such that every formula in that list is either an axiom or follows from previously listed formulas using an inference rule. This notion of *inference rule*, as used in this and other styles of proof, is, usually, greatly restricted by limitations of human psychology, and by what skeptics are willing to *trust*. Typically, checking the application of inference rules involves simple syntactic checks.

11.1.1 Example   Take the following rule for set inclusion, which states that, given a set A and a strict subset B, we may conclude B:

$$\frac{A \quad A \supset B}{B}$$

The applicability of this rule requires deciding on whether or not two premises have the structure A and $A \supset B$, and the conclusion has the structure B.

The introduction of automation into theorem proving has allowed us to engineer inference steps that are significantly more substantial, and can comprise both *computation* and *deduction*. As we note extensively, recent proof theoretic results allow us to extend the literature of theorem proving from being a study of minuscule inference rules—such as *modus ponens* in Hilbert-style systems, or Gentzen-style introduction rules—to a study of large-scale, formally defined, *synthetic inference rules*. In this chapter, we describe a particular way to specify and check such synthetic inference rules as a way to inductively prove lemmas from previous lemmas, in a style close to that in which proofs are written inside proof assistants like Coq, Isabelle or Abella.

Let us return to the world of Frege proofs. In what follows, we will not speak of axioms, as the concept is redundant and can be subsumed by inference rules:

an axiom can be generally described as an inference rule that depends on zero previous lemmas. Moreover, when we speak of a formula that is a member of a list of formulas comprising a Frege proof, we shall usually refer to it simply as a lemma. Presently, we will consider the relationship between this linearized style and more structured, arborescent schemes—closer to the proof trees of the sequent calculus—and its effects on proof checking.

The rest of the chapter is structured as follows. Section 11.2 presents a motivating example of the kind of proof development used throughout the chapter to guide out designs. Section 11.3 introduces the concept of *proof outlines* as high-level descriptions of proofs and discusses their logical interpretation. Section 11.4 translates that interpretation into the logic, augmenting the μLJF^a proof system and the checkers that implement it. Section 11.5 describes the first of two families of proof outlines, a lightweight yet flexible collection of proof descriptors. Section 11.6 elaborates on the previous section in a second family of outlines that offers finer control akin to that found in the proof scripts of proof assistants. Section 11.7 revisits the cases study of Section 11.2 and reviews a number of interesting applications of proof outlines. Section 11.8 concludes the chapter.

11.2   Case study

In this section, we will present a fundamental motivating example from which many other use cases follow. Consider defining the addition of natural numbers using the standard inductive relational specification in Abella, shown in Figure 11.1. Here, the expected inductive definitions are given together with a predicate for *typing judgments* about naturals, whose utility will immediately become apparent.

Once these definitions are introduced, routinely we will find that we need to establish several properties of the addition relation immediately before progressing to more interesting work, e.g., that addition is determinate and total. Anyone familiar with proving such theorems knows that their proofs are simple: basically, the obvious induction leads quickly to a final proof. Figure 11.2 shows how this is done in Abella. The necessity for typing judgments now becomes clear: unlike systems like Coq, Abella can only induct on hypotheses, even if typed variables avail in the context. The direct Abella equivalent involves applying induction on a predicate that follows the full inductive structure of the kind in question, i.e., a

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

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

Define plus : nat -> nat -> nat -> prop by
  plus z N N ;
  plus (s N) M (s P) := plus N M P.
```

11.1 Figure   Relational specification of natural numbers and addition on them in standard Abella at the reasoning level. Compare this with the specification level of λProlog in Figure 4.2.

typing judgment that takes an arbitrary member of its typed, validates its structure by exhaustive recursion through its inductively defined constructors, and succeeds. (Clearly, it is possible to derive these predicates mechanically, though Abella does not provide this facility.)

Of course, if we wish to prove more facts about addition, we may need to come up with and prove some lemma before simple inductions will work. For example, proving the commutativity of addition makes use of two additional lemmas, as shown in Figure 11.3. These three theorems, as well as those in Figure 11.2, all have the same high-level proof outline: apply induction with the *obvious invariant*, apply some previously proved lemmas and the inductive hypothesis, and deal with any remaining branches by case analysis.

The fact that many theorems can be proved by resorting to this pattern of *induction-lemmas-cases*—and, indeed, whole developments are routinely organized around it—is well known and built into existing theorem provers. For example, the waterfall model of the Boyer-Moore theorem prover (Boyer and Moore, 1979) proves such theorems in a similar fashion, but operating on inductive definitions of functions. In a similar relational style as that of Abella, the Twelf system (Pfenning and Schürmann, 1999) can often prove automatically certain properties, such as the statements that some relations are total and functional, using a series of similar steps to those described here (Schürmann and Pfenning, 2003). The tactics and tacticals of LCF have also been used to implement procedures that attempt to find proofs using this kind of process (Wilson et al., 2010). Finally, and closer to the present approach, the TAC procedure of Baelde et al. (2010) attempts

```
Theorem plustotal :
  forall N, nat N -> forall M, nat M ->
  exists S, plus N M S.
induction on 1. intros. case H1.
  % Base case
  search.
  % Inductive case
  apply IH to H3. apply H4 to H2. search.

Theorem plusdeterm :
  forall N, nat N -> forall M, nat M ->
  forall S, plus N M S -> forall T, plus N M T -> S = T.
induction on 1. intros. case H1.
  % Base case
  case H3. case H4. search.
  % Inductive case
  case H3. case H4. apply IH to H5.
  apply H8 to H2. apply H9 to H6. apply H10 to H7. search.
```

11.2 **Figure** Simple properties of addition in Abella. Each of totality and determinism follow directly from an induction on the first argument of the plus relation and routine case analysis and applications on hypotheses in the context.

to apply precisely such a scheme, although in a rather fixed and inflexible fashion, which has not continued beyond its original development.

Here, we endeavor to show how to describe the simple rules that can be used to prove a given lemma based on previously proved lemmas. Specifically, we will define proof certificates that describe the structure of the intended proof outlines that we expect and then run a proof checker on those certificates to see whether the certificate can be elaborated into a full proof of the candidate theorem, or not. Since the design of the certificate language is based on the proof theory of synthetic connectives and since the proof checker we use employs both unification and backtracking search, this approach to describing high-level inference rules is both highly flexible and natural.

11.3 Certificate design

Imagine telling a colleague: "The proof of this theorem follows by a simple induction and the three lemmas we just proved." You may or may not be correct in such an assertion since: (a) the proposed theorem may not be provable; and

```
Theorem plus0com :
  forall N, nat N ->  plus N z N.
induction on 1. intros. case H1.
  % Base case
  search.
  % Inductive case
  apply IH to H2. search.

Theorem plusscom :
  forall M, nat M -> forall N, nat N ->
  forall P, plus M N P -> plus M (s N) (s P).
induction on 1. intros. case H1.
  % Base case
  case H3. search.
  % Inductive case
  case H3. apply IH to H4. apply H6 to H2. apply H7 to H5.
  search.

Theorem pluscom :
  forall N, nat N -> forall M, nat M ->
  forall S, plus N M S  -> plus M N S.
induction on 1. intros. case H1.
  % Base case
  case H3. apply plus0com to H2. search.
  % Inductive case
  case H3. apply IH to H4. apply H6 to H2. apply H7 to H5.
  apply plusscom to H2. apply H9 to H4. apply H10 to H8.
  search.
```

11.3 **Figure** Commutativity of addition in Abella. A proof by simple induction relies on two auxiliary lemmas, one for each case of the induction on natural numbers (zero and successor). Each of the two lemmas is proved by simple induction, as is the main theorem—with the proviso that now not only hypotheses in the context, but also the auxiliary lemmas may be applied.

(b) the simple proof you describe may not exist. In any case, it is clear that there is a rather simple, high-level algorithm to follow that will search for such a proof.

Moreover, this corresponds with standard mathematical practice, in which "simple proofs" may be described schematically and with minimum clutter, concentrating on interesting cases and application of useful lemmas. In this respect, proof outlines are not only of utility to machine checkers, but also to mathematicians. The pedantry inherent to formal developments spawns a large number of shallow, mostly unremarkable proof obligations, most of which can be mechanically dispatched by a small number of common proof patterns.

In this section, we show how the FPC framework can formally specify such an algorithm. Following the paradigm of focused proof systems for first-order logic, there is a clear, high-level outline to follow for doing proof search for cut-free proofs: first do all invertible inference rules and then select a formula on which to do a series of non-invertible choices. This latter phase ends when one encounters invertible inference rules again or the proof ends. In the setting we describe here, there are two significant complicating features with which to be concerned.

1. **Treating the induction rule.** The invertible phase is generally treated as a place where no important choices in the search for a proof appear. When dealing with a formula that is a fixed point, however, this is no longer true. As described in Section 9.2, we treat a fixed point expression either by *freezing*—for which see also Baelde (2012)—, *unfolding*, or using an invariant to perform an *induction*—here, this will be extended with the possibility of deriving the "obvious" inductive invariant. These options are directly connected to the rules introduced in Figure 9.3. In particular:

   (a) We can choose to "freeze" the fixed point, meaning that we choose not to induct on it.

   (b) We can set up an inductive step. This second choice is in turn divided into three sub-choices:

      i. We can choose to simply unfold a fixed point definition. In fact, the concept of unfolding follows as a direct consequence of applying induction.

      ii. We can take an explicit induction offered by the author in the certificate. In the context of this discussion, a human actor will seldom need to make use of this option—very often, the obvious invariant (for which see below) is all one needs. However, the

author of a certificate can also be a theorem prover or a proof checker. In this latter case of machine-generated proofs, an explicit invariant may be routinely inserted into a certificate. These questions are treated in Section 11.4.

iii. We can select the surrounding sequent context to be the actual inductive invariant. This corresponds to the notion of *obvious* or *immediate invariant*.

2. **Lemmas must be invoked.** The application of lemmas into a proof outline is critical to the kind of linear proof development we have in mind. Although the focusing framework does not restrict the shape of lemmas, we consider here the effect of focused proof construction with a lemma that is a Horn clause. For example, the three lemmas addressing the commutativity of addition in Figure 11.2 are Horn clauses.

11.3.1   Example   Consider applying a Horn lemma of the form $\forall \bar{x}.[A_1 \supset A_2 \supset A_3]$ in proving the sequent $\Gamma \vdash E$. Since the formulas A_1, A_2, and A_3 are polarized positively, we can design the proof outline (simply by only allowing fixed points to be frozen during this part of the proof) so that $\Gamma \Downarrow \forall \bar{x}.[A_1 \supset A_2 \supset A_3] \vdash E$ is provable if and only if there is a substitution θ for the variables in the list of variables $\bar{x}$ such that θA_1 and θA_2 are in Γ and the sequent $\Gamma, \theta A_3 \vdash E$ is provable. The application of such a lemma is then seen as forward chaining: if the context Γ contains two atoms (i.e., frozen fixed points), then add a third.

The main issue that a certificate-as-proof-outline therefore needs to provide is some indication of what lemmas should be used during the construction of a proof. The following natural specifications of collections of supporting lemmas— starting from the least explicit to the most explicit—are easily written within our framework:

1. A bound on the number of lemmas that can be used to finish the proof, chosen freely from the collection of previously proven and known lemmas.

2. A list of possible lemmas to use in finishing the proof. These can be assumed as hypotheses during the proof; a separate proof for each lemma is required as well.

3. A tree of lemmas, indicating which lemmas are applied following the conjunctive structure of the remaining proof.

Each of these three categories refine the previous one with additional information on which lemmas to use and where, thereby reducing the amount of nondeterminism and enabling faster proofs—or disproofs *of the proof outline*. Additional refinements are possible and can bring outlines even closer to the *proof scripts* that are commonly written as avatars for proofs in interactive theorem provers. Before we study the encoding of these procedures as proof certificates, we need to consider the extensions imposed on the proof system—and its implementation—by the two distinguishing features of this proposal: lemmas and obvious inductions. Augmenting contexts as illustrated in Example 11.3.1 is critical for eventually enabling obvious inductions to succeed in completing a proof. In this way, the focused proof system can easily be used to apply lemmas. All this will be the subject of the next section.

11.4 Logic support

In general, it appears that (co)inductive invariants are often complex, large, and tedious structures to build and use. Thus, it is most likely that we need to develop a number of techniques by which invariants are not built directly but are rather implied by alternative reasoning principles. For example, Abella allows the user to do induction not by explicitly entering an invariant but rather by performing a certain kind of guarded, circular reasoning. Closer to our approach, Coq automatically derives induction principles from inductive definitions, but also allows users to define their own custom inductions.

In the present context, we consider a single approach to specifying invariants. Let us consider the case of induction on a least fixed point, i.e., on the right-hand side of the sequent during the asynchronous phase. Recall the associated inference rules in Figures 9.3 and 9.6, namely the μLJF rule for induction (i.e., on least fixed points; we omit the frozen zones for succinctness):

$$\frac{\Gamma \Uparrow S\,\bar{t}, \Theta \vdash \mathcal{R} \quad \Gamma \Uparrow B\,S\,\bar{y} \vdash S\,\bar{y} \Uparrow}{\Gamma \Uparrow \mu B\,\bar{t}, \Theta \vdash \mathcal{R}} \; inductL$$

The principle we now introduce involves taking the conclusion of that rule $\Gamma \Uparrow \mu B\,\bar{t}, \Theta \vdash \mathcal{R}$ and abstracting out the fixed point expression to yield the *obvious invariant*, which we write $\hat{S}$. This invariant is extracted from the conclusion in such a way that one of the premises, $\Gamma \Uparrow S\,\bar{t}, \Theta \vdash \mathcal{R}$, has an easy proof—in fact, it is made trivial by definition of $\hat{S}$. As a result, only the second premise related to

the induction rule needs to be properly proved. The following augmented rule is used to generate and check whether or not the obvious induction invariant can be used. The resulting extensions are presented in Figure 11.4. Sequents are strenghtened with a "zone" Σ which represents the list of eigenvariables in the sequent, required to compute obvious invariants.

The development of *obvious coinduction* is completely symmetric. In both cases, there is the choice of whether to omit or maintain the branch that becomes redundant upon application of the obvious invariant. Pruning said branch from the sequent calculus requires (a) enriching the calculus with the computation of said obvious invariants (thereby becoming part of any kernel implementing this logic); and (b) furnishing evidence of the provability of the obvious branch in the general case (given in Figure 11.5). Figure 11.4 shows both possible formulations of the resulting calculus, and their augmentations with the corresponding clerks are given in Figure 11.6. From the point of view of the writer of proof certificates, the fact that the left premises of the full rules follow directly from the obvious invariants allows us to confuse both presentations. That is, *without* pruning the trivial branch, the kernel can perform proof search to check this premise— which is guaranteed to succeed—while clerks need only produce a continuation certificate for the non-trivial branch, effectively as if the obvious premise was discharged as a proof obligation.

After developing the necessary logic support, all that remains is to integrate them in the proof checker developed in Section 9.3. The extensions are neatly divided in two groups of changes paralleling the discussion in Section 11.3:

1. The obvious (co)induction rules are added as variants of, respectively, standard (co)induction. The kernel must be extended with the Σ zone for eigenvariables, and the new rules come with specialized code to compute obvious (co)invariants. There are two technical disadvantages to this last addition. First, we inject relatively complex, non-declarative code into the trusted kernel—although we are under no obligation to trust it if we verify both premises for each involved rule, i.e., if we rule out the single-premise rules in Figure 11.6. Second, and more serious, there is no clean, declarative way to maintain a list of eigenvariables in the presence of eigenvariable unification: under this discipline, the obvious rules are only meaningful in the initial stage of a proof, before any such operations (triggered by the equality rules) pollute the Σ zone. In particular, the scheme does not support nested obvious inductions.

ASYNCHRONOUS INTRODUCTION RULES

$$\frac{\{y\} \cup \Sigma;\, \Gamma \Uparrow\, \vdash [y/x]B \Uparrow}{\Sigma;\, \Gamma \Uparrow\, \vdash \forall x.B \Uparrow} \qquad\qquad \frac{\{y\} \cup \Sigma;\, \Gamma \Uparrow [y/x]B, \Theta \vdash \mathcal{R}}{\Sigma;\, \Gamma \Uparrow \exists x.B, \Theta \vdash \mathcal{R}}$$

$$\frac{\Sigma;\, \Gamma \Uparrow S\,\bar{t}, \Theta \vdash \mathcal{R} \qquad \{\bar{y}\};\, \Gamma \Uparrow B\, S\, \bar{y} \vdash S\, \bar{y} \Uparrow}{\Sigma;\, \Gamma \Uparrow \mu B\, \bar{t}, \Theta \vdash \mathcal{R}}\; inductL$$

$$\frac{\Sigma;\, \Gamma \Uparrow \hat{S}\,\bar{t}, \Theta \vdash \mathcal{R} \qquad \{\bar{y}\};\, \Gamma \Uparrow B\, \hat{S}\, \bar{y} \vdash \hat{S}\, \bar{y} \Uparrow \qquad \dagger}{\Sigma;\, \Gamma \Uparrow \mu B\, \bar{t}, \Theta \vdash \mathcal{R}}\; obviousL$$

$$\frac{\blacksquare \qquad \{\bar{y}\};\, \Gamma \Uparrow B\, \hat{S}\, \bar{y} \vdash \hat{S}\, \bar{y} \Uparrow \qquad \dagger}{\Sigma;\, \Gamma \Uparrow \mu B\, \bar{t}, \Theta \vdash \mathcal{R}}\; obviousL'$$

$$\dagger \quad \hat{S} := \lambda x.\forall \Sigma.\bar{x} = \bar{t} \supset \left(\bigwedge \Theta\right) \supset \mathcal{R}$$

$$\frac{\Sigma;\, \Gamma \Uparrow\, \vdash S\bar{t} \Uparrow \qquad \{\bar{y}\};\, \Uparrow S\bar{y} \vdash BS\bar{y} \Uparrow}{\Sigma;\, \Gamma \Uparrow\, \vdash \nu B\bar{t} \Uparrow}\; inductR$$

$$\frac{\Sigma;\, \Gamma \Uparrow\, \vdash \hat{S}\bar{t} \Uparrow \qquad \{\bar{y}\};\, \Uparrow \hat{S}\bar{y} \vdash B\hat{S}\bar{y} \Uparrow \qquad \ddagger}{\Sigma;\, \Gamma \Uparrow\, \vdash \nu B\bar{t} \Uparrow}\; obviousR$$

$$\frac{\blacksquare \qquad \{\bar{y}\};\, \Uparrow \hat{S}\bar{y} \vdash B\hat{S}\bar{y} \Uparrow \qquad \ddagger}{\Sigma;\, \Gamma \Uparrow\, \vdash \nu B\bar{t} \Uparrow}\; obviousR'$$

$$\ddagger \quad \hat{S} := \lambda x.\exists \Sigma.\bar{x} = \bar{t} \wedge \left(\bigwedge \Gamma\right)$$

11.4 **Figure**   Extensions to the *μLJF* focused proof system needed to support obvious inductions in proof outlines, introduced informally in Baelde et al. (2010). A sequent is now prefixed by Σ, its set of eigenvariables. Asynchronous quantifiers add new eigenvariables and (not shown here) asynchronous equality (possibly) unifies them. Fixed points are endowed with a new obvious (co)induction rule, each with two variants: with or without checking of the obvious premise. The provisos define the obvious invariant that trivially satisfies the obvious branch.

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{
        \ldots, \Theta \vdash \Theta_1 \Downarrow \quad \cdots \quad \ldots, \Theta \vdash \Theta_n \Downarrow
      }{
        \Gamma, \hat{S}\bar{t}, \Theta \vdash \bigwedge\Theta \Downarrow
      }\;\wedge_e
      \qquad
      \cfrac{\ldots \Downarrow \mathcal{R} \vdash \mathcal{R}}{}\;\supset_e
    }{
      \Gamma, \hat{S}\bar{t}, \Theta \Downarrow (\bigwedge\Theta) \supset \mathcal{R} \vdash \mathcal{R}
    }\;\supset_e
  }{
    \cfrac{
      \cfrac{
        \ldots \vdash \bar{t} = \bar{t} \Downarrow
      }{}\;=_e
      \qquad
      \Gamma, \hat{S}\bar{t}, \Theta \Downarrow \bar{t} = \bar{t} \supset (\bigwedge\Theta) \supset \mathcal{R} \vdash \mathcal{R}
    }{
      \cfrac{
        \Gamma, \hat{S}\bar{t}, \Theta \Downarrow \forall\Sigma.\bar{t} = \bar{t} \supset (\bigwedge\Theta) \supset \mathcal{R} \vdash \mathcal{R}
      }{
        \cfrac{
          \Gamma, \hat{S}\bar{t}, \Theta \Downarrow \hat{S}\bar{t} \vdash \mathcal{R}
        }{
          \cfrac{
            \Gamma, \hat{S}\bar{t}, \Theta \Uparrow \vdash \mathcal{R}
          }{
            \Gamma \Uparrow \hat{S}\bar{t}, \Theta \vdash \mathcal{R}
          }\;store
        }\;decide
      }\;\beta
    }\;\forall_e
  }
}{
  \cfrac{
    \Gamma \Uparrow \mu B\bar{t}, \Theta \vdash \mathcal{R}
    \qquad
    \cfrac{
      \Pi \\ \vdots \\ \Gamma \Uparrow B\hat{S}\bar{y} \vdash \hat{S}\bar{y} \Uparrow
    }{}\;obviousL
  }{}
}
$$

11.5 Figure Proof schema of the obvious induction showing how the obvious branch follows directly from the obvious invariant, i.e., the goal follows from the obvious invariant and the remaining hypotheses. Because all the components of the obvious invariant are present in the sequent in complementary positions, all the branches in the subproof suggest immediate applications of the initial rules. A proof obligation for the non-trivial branch remains to be satisfied by a proof Π.

ASYNCHRONOUS INTRODUCTION RULES

$$\dfrac{\Xi_1: \Sigma; \Gamma \Uparrow \hat{S}\,\bar{t}, \Theta \vdash \mathcal{R} \qquad (\Xi_2\bar{y}): \{\bar{y}\}; \Gamma \Uparrow B\hat{S}\bar{y} \vdash \hat{S}\bar{y} \Uparrow}{\Xi_0: \Sigma; \Gamma \Uparrow \mu B\,\bar{t}, \Theta \vdash \mathcal{R}} \; obviousL_c(\Xi_0, \Xi_1, \Xi_2) \quad \dagger$$

$$\dfrac{(\Xi_1\bar{y}): \{\bar{y}\}; \Gamma \Uparrow B\hat{S}\bar{y} \vdash \hat{S}\bar{y} \Uparrow}{\Xi_0: \Sigma; \Gamma \Uparrow \mu B\,\bar{t}, \Theta \vdash \mathcal{R}} \; obviousL'_c(\Xi_0, \Xi_1) \quad \dagger$$

$$\dfrac{\Xi_1: \Sigma; \Gamma \Uparrow \vdash \hat{S}\bar{t} \Uparrow \qquad (\Xi_2\bar{y}): \{\bar{y}\}; \Uparrow \hat{S}\bar{y} \vdash B\hat{S}\bar{y} \Uparrow}{\Xi_0: \Sigma; \Gamma \Uparrow \vdash \nu B\bar{t} \Uparrow} \; obviousR_c(\Xi_0, \Xi_1, \Xi_2) \quad \ddagger$$

$$\dfrac{(\Xi_1\bar{y}): \{\bar{y}\}; \Uparrow \hat{S}\bar{y} \vdash B\hat{S}\bar{y} \Uparrow}{\Xi_0: \Sigma; \Gamma \Uparrow \vdash \nu B\bar{t} \Uparrow} \; obviousR'_c(\Xi_0, \Xi_1) \quad \ddagger$$

11.6   **Figure**   Extensions to the augmented μLJF^a focused proof system needed to support obvious inductions in proof outlines (Blanco and Miller, 2015). The new inference rules introduced in Figure 11.4 are augmented with clerks in the usual way. Note that the premises that involve the body B of the fixed point take a continuation certificate abstracted over the fresh variable(s) $\bar{y}$. The side conditions † and ‡ that define the obvious (co)invariants are unchanged from Figure 11.4.

2. The treatment of lemmas presents us with two orthogonal choices. First, do we treat them as assumptions within a proof? Second, do we consider them separate from the standard storage zones on which the decide rules operate? Here, the answer to these two questions will be "yes." A separate zone Λ, i.e., a map from lemma names to formulas, is threaded throughout the kernel, and the checker's interface is enriched with an additional list of lemmas to populate Λ. The consequence of the first question is that lemma application will be modeled as a decide rule, and not as a cut rule (which must come with an inlined proof of the lemma). The consequence of the second section is the definition of *lemma decide rules* which operate like *local decide rules*, limited to the zone Λ, leaving the old rules unchanged.

It is straightforward to see that all these additions preserve the soundness of the enriched μLJF^a system—by the same elementary reasoning used, say, to prove the soundness of LKF^a with respect to LKF in Theorem 3.2.2. The extensions to the Abella kernel are summarized in Figure 11.7. The new zones are threaded through existing code, new inference rules are added and an interface function composes the initial sequence and populates the zone of lemmas; all other rules remain otherwise unaltered. The new zones are populated only when new eigenvariables are generated (Σ) and when the kernel is called with a set of lemmas (Λ). Obvious inductions rely for the generation of invariants in a significant amount of non-declarative code to implement the computation of $\hat{S}$ in Figure 11.4, which is not shown here. The rules depicted here are the brief, one-premise rules, which do not separately check the branch made trivial by the obvious invariants, given that these are proven to be built correctly, otherwise soundness is endangered. Of course, in the context of a development, say, in an interactive theorem prover, we will wish to enforce that a proof be supplied for every lemma used as a hypothesis at any point in the development. This work of integration will be discussed in Chapter 13.

11.5 Certificate families: simple outlines

With the client-side requirements for proof outlines established and reflected in the logic, we turn our attention the actual FPC definitions in two related families of outline certificates. The first of these families defines what we call *simple outlines*. This format aims to be simple to read and write without limiting the proofs that it can find compared to other, denser representations. Simple outlines do not

```
Kind    lemma    type.
Type    lemma    idx -> bool -> lemma.

Kind    list_lemma    type.
Type    nil_lemma      list_lemma.
Type    cons_lemma     lemma -> list_lemma -> list_lemma.

Define member_lemma : lemma -> list_lemma -> prop by
        member_lemma Lemma (cons_lemma Lemma  Rest) ;
        member_lemma Lemma (cons_lemma Lemma' Rest) :=
                member_lemma Lemma Rest.

Define async : list_lemma -> cert -> list_i -> list_ctx ->
                list_ctx -> list_bool -> goal -> prop,
        syncL : list_lemma -> cert -> list_i -> list_ctx ->
                list_ctx ->       bool -> goal -> prop,
        syncR : list_lemma -> cert -> list_i -> list_ctx ->
                list_ctx                -> bool -> prop     by

async Lambda Xi Sigma Phi Gamma
  (cons_bool (mu B T) Delta) G := exists Xi' S,
    indClerk' Xi Xi' /\
    indInvariant' Sigma Delta G T S /\ forall x,
    async Lambda (Xi'  x) (cons_i x nil_i) Phi Gamma
      (cons_bool (B S x) nil_bool) (unk (S x)) ;

async Lambda Xi Sigma Phi Gamma
  nil_bool (unk (nu B T)) := exists Xi' S,
    coindClerk' Xi Xi' /\
    coindInvariant' Sigma Gamma T S /\ forall x,
    async Lambda (Xi'  x) (cons_i x nil_i) Phi nil_ctx
      (cons_bool (S x) nil_bool) (unk (B S x)) ;

async Lambda Xi Sigma Phi Gamma nil_bool G :=
  exists Xi' Idx C ?1, (G = (sto ?1) \/ G = (frz ?1)) /\
  decideLClerk' Xi Xi' Idx /\
  member_lemma (lemma Idx C) Lambda /\
  syncL Lambda Xi' Sigma Phi Gamma C G ;

Define prove_with_lemmas :
  cert -> bool -> list_lemma -> prop by
    prove_with_lemmas Cert Form Lemmas := exists Cert',
      unmarshal Cert Cert' /\
      async Lemmas Cert' nil_i nil_ctx
        nil_ctx nil_bool (unk Form).
```

11.7 Figure   Extensions to the *μLJF^a* kernel written in Abella to support both non-local decides on lemmas and obvious induction and coinduction.

restrict polarities and use a simple indexing scheme for referring to formulas by simple, unstructured names, i.e., individual *indexes for lemmas* can be defined as nullary constructors of kind `idx` as needed. Here we are especially interested in the naming and selection of lemmas, as managed by the client side. The kernel handles its internal indexing of formulas generated during proof reconstruction by two related mechanisms:

1. Atoms, i.e., frozen fixed points, are stored on the left—upon freezing— under a special index `idxatom`. The initial rule on the left always selects from this bucket. (Recall that freezing on the right-hand side is a singleton, and so freezing it does not require an index.)

2. All other formulas are stored on the left under a second dedicated index `idxlocal`. The decide rule on the left can either select a "local formula" or a lemma by any of the mechanisms described below. (Again, storing and deciding on the right do not require any indexing.)

If greater granularity is required, it is a simple matter to implement it modularly and add it to simple outlines via *certificate pairing*. For this, see Chapter 5 as well as its adaptation in Section 10.4. The latter which would need to be extended with the decide rule on lemmas from Section 11.4; this external piece of information requires agreement between the two halves of a certificate pair—unlike local decides, made independent by paired indexes.

The bulk of the complexity of the FPC definition rests on certificate constructors and their limited manipulations by clerks and experts. A first group of certificate constructors is meant to represent *whole proof subtrees*, from the root of the branch down to every leaf of the subtree. In this sense, they are self-contained and fully manage their own (limited) bookkeeping needs. The following four constructors are provided:

1. `(induction B AU SU AC SC)`: asynchronously look for the closest fixed point, apply obvious (co)induction to it and perform bounded search through `apply` (as explained next).

2. `(inductionS B AU SU AC SC S)`: asynchronously look for the closest fixed point, apply (co)induction with the supplied invariant and perform bounded search through `apply`.

3. `(apply B AU SU AC SC)`: perform bounded search and try to finish the proof by `search`.

4. `search`: attempt to finish the proof by simple exploration of the tree and application of initial rules to all branches.

These certificate constructors form a hierarchy that represents the evolution of the state of the proof: from induction to a number of bipoles—which comprise the application of lemmas—up to the posited end of the proof. All parameters are natural numbers representing decreasing counters. Those counters are used to bound the possible sources of non-terminating behavior in proof search: (a) decide rules, and (b) fixed point operations. Among the latter, freezing is bounded by definition and induction is controlled singly by `induction` and `inductionS`, leaving unfolding as the sole related source of unbounded behavior. Bounded proof search depends on the following set of parameters:

1. *Bipole bound:* after the current bipole, proof search may proceed up to B additional bipoles deeper. The release rule marks the end of a bipole; the rule is only enabled when the natural number of bipoles can be decremented.

2. *Asynchronous unfolding:* each bipole after the current one is allowed to perform up to AU unfolding operations along each branch during the asynchronous phase.

3. *Synchronous unfolding:* each bipole after the current one is allowed to perform up to SU unfolding operations along each branch during the synchronous phase.

4. *Asynchronous unfolding, current bipole:* until the end of the current bipole, up to AC asynchronous unfolding operations may be performed along each branch during the asynchronous phase.

5. *Synchronous unfolding, current bipole:* until the end of the current bipole, up to SC synchronous unfolding operations may be performed along each branch during the synchronous phase.

The "current bipole" counters AC and SC are technical bookkeeping devices, initialized respectively to AU and SU at the beginning of each bipole (i.e., on the release rules). The only reason to write a certificate where the initial current bipole counters are unequal to the per-bipole bounds would be as a very limited form of optimization. This unnecessary verbosity is averted by defining marshalled forms of the certificate constructors where only per-bipole counters are given.

1. `(induction! B AU SU)` is expanded to the unmarshalled certificate `(induction B AU SU AU SU)`, and similarly for `inductionS`.

2. `(apply! B AU SU)` is expanded to `(apply B AU SU AU SU)`.

These shorthand forms are closer to the minimal outlines the user may want to write, but may involve substantial backtracking search. In the first place, these certificates representing full subtrees leave the choice of which formula to focus on at each decide rule—lemma or local formula—to the kernel, which in turn must explore all possibly combinations allowed by its local context and the list of lemmas given to it. A second source of inefficiency comes from the unfolding bounds: these must be as large or greater than the needs of the costliest phases, even if in most cases much smaller bounds suffice.

To address the inefficiencies that arise from the extreme conciseness of the certificates above—and the laxity they impose on costly backtracking search throughout the entire proof tree—, a second family of certificate constructors is presented. The new constructors are written in a *continuation-passing style*, or CPS, which can be used to describe the interesting features of a *section of a proof*: asynchronous branching, order and choice of lemmas, and local unfolding bounds. A local description of a region of a proof tree is followed by continuation certificates. The following constructors are defined:

1. `(induction? C)`: in the asynchronous phase, look for the first available fixed point and do the obvious (co)induction, then continue the proof using the continuation certificate `C`.

2. `(inductionS? CL CR S)`: do induction on the first asynchronously available fixed point using `S` as invariant, and continue the two resulting branches of the proof using continuation certificates `CL` and `CR`.

3. `(case? A CL CR)`: in the current asynchronous phase, look for the first (branching) left disjunction. Apply the asynchronous unfolding rule at most `A` times to get to one such connective. To continue proof search, use continuation certificates `CL` and `CR` for the left and right branches.

4. `(apply? A S I C)`: finish the current bipole, performing at most `A` asynchronous unfoldings and `S` synchronous unfoldings until the end of the bipole. To decide on a formula at the boundary of the asynchronous and synchronous phases, use index `I`. When the bipole ends at the release rule, use certificate `C` to continue proof search.

```
ffClerk (induction? _).
ffClerk (inductionS? _ _ _).
ffClerk (case? _ _ _).
ffClerk (apply? _ _ _ _).

ttClerk (induction? C) (induction? C).
ttClerk (inductionS? L R I) (inductionS? L R I).
ttClerk (case? A L R) (case? A L R).
ttClerk (apply? A S I C) (apply? A S I C).

andClerk (induction? C) (induction? C).
andClerk (inductionS? L R I) (inductionS? L R I).
andClerk (case? A L R) (case? A L R).
andClerk (apply? A S I C) (apply? A S I C).

orClerk (induction? C) (induction? C) (induction? C).
orClerk (inductionS? L R I)
        (inductionS? L R I) (inductionS? L R I).
orClerk (case? _ L R) L R.
orClerk (apply? A S I C) (apply? A S I C) (apply? A S I C).

impClerk (induction? C) (induction? C).
impClerk (inductionS? L R I) (inductionS? L R I).
impClerk (case? A L R) (case? A L R).
impClerk (apply? A S I C) (apply? A S I C).

eqClerk (induction? C) (induction? C).
eqClerk (inductionS? L R I) (inductionS? L R I).
eqClerk (case? A L R) (case? A L R).
eqClerk (apply? A S I C) (apply? A S I C).

ttExpert (induction? _).
ttExpert (inductionS? _ _ _).
ttExpert (case? _ _ _).
ttExpert (apply? _ _ _ _).

andExpert (induction? C) (induction? C) (induction? C).
andExpert (inductionS? L R I)
          (inductionS? L R I) (inductionS? L R I).
andExpert (case? A L R) (case? A L R) (case? A L R).
andExpert (apply? A S I C)
          (apply? A S I C) (apply? A S I C).

orExpert (induction? C) (induction? C) left.
orExpert (induction? C) (induction? C) right.
orExpert (inductionS? L R I) (inductionS? L R I) left.
orExpert (inductionS? L R I) (inductionS? L R I) right.
```

11.8 Figure   The CPS fragment of the simple outline FPC at the specification level.

```
orExpert (case? A L R) (case? A L R) left.
orExpert (case? A L R) (case? A L R) right.
orExpert (apply? A S I C) (apply? A S I C) left.
orExpert (apply? A S I C) (apply? A S I C) right.

impExpert' (induction? C) (induction? C) (induction? C).
impExpert' (inductionS? L R I)
           (inductionS? L R I) (inductionS? L R I).
impExpert' (case? A L R) (case? A L R) (case? A L R).
impExpert' (apply? A S I C)
           (apply? A S I C) (apply? A S I C).

eqExpert (induction? _).
eqExpert (inductionS? _ _ _).
eqExpert (case? _ _ _).
eqExpert (apply? _ _ _ _).

allClerk (induction? C) (x\ induction? C).
allClerk (inductionS? L R I) (x\ inductionS? L R I).
allClerk (case? A L R) (x\ case? A L R).
allClerk (apply? A S I C) (x\ apply? A S I C).

someClerk (induction? C) (x\ induction? C).
someClerk (inductionS? L R I) (x\ inductionS? L R I).
someClerk (case? A L R) (x\ case? A L R).
someClerk (apply? A S I C) (x\ apply? A S I C).

allExpert (induction? C) (induction? C) _.
allExpert (inductionS? L R I) (inductionS? L R I) _.
allExpert (case? A L R) (case? A L R) _.
allExpert (apply? A S I C) (apply? A S I C) _.

someExpert (induction? C) (induction? C) _.
someExpert (inductionS? L R I) (inductionS? L R I) _.
someExpert (case? A L R) (case? A L R) _.
someExpert (apply? A S I C) (apply? A S I C) _.

indClerk (inductionS? L R I) L R I.

indClerk' (induction? C) (x\ C).

coindClerk (inductionS? L R I) L R I.

coindClerk' (induction? C) (x\ C).

unfoldLClerk (case? (s A) L R) (case? A L R).
unfoldLClerk (apply? (s A) S I C) (apply? A S I C).
```

11.9 Figure   The CPS fragment of the simple outline FPC at the specification level (continued).

```
unfoldRExpert (apply? A (s S) I C) (apply? A S I C).

unfoldLExpert (apply? A (s S) I C) (apply? A S I C).

unfoldRClerk (case? (s A) L R) (case? A L R).
unfoldRClerk (apply? (s A) S I C) (apply? A S I C).

freezeLClerk (induction? C) (induction? C) idxatom.
freezeLClerk (inductionS? L R I)
             (inductionS? L R I) idxatom.
freezeLClerk (case? A L R) (case? A L R) idxatom.
freezeLClerk (apply? A S I C) (apply? A S I C) idxatom.

initRExpert (induction? _) idxatom.
initRExpert (inductionS? _ _ _) idxatom.
initRExpert (case? _ _ _) idxatom.
initRExpert (apply? _ _ _ _) idxatom.

freezeRClerk (induction? C) (induction? C).
freezeRClerk (inductionS? L R I) (inductionS? L R I).
freezeRClerk (case? A L R) (case? A L R).
freezeRClerk (apply? A S I C) (apply? A S I C).

initLExpert (induction? _).
initLExpert (inductionS? _ _ _).
initLExpert (case? _ _ _).
initLExpert (apply? _ _ _ _).

storeLClerk (induction? C) (induction? C) idxlocal.
storeLClerk (inductionS? L R I)
            (inductionS? L R I) idxlocal.
storeLClerk (case? A L R) (case? A L R) idxlocal.
storeLClerk (apply? A S I C) (apply? A S I C) idxlocal.

decideLClerk (apply? A S I C) (apply? A S I C) I.

decideLClerk' (apply? A S I C) (apply? A S I C) I.

storeRClerk (induction? C) (induction? C).
storeRClerk (inductionS? L R I) (inductionS? L R I).
storeRClerk (case? A L R) (case? A L R).
storeRClerk (apply? A S I C) (apply? A S I C).

decideRClerk (apply? A S I C) (apply? A S I C).

releaseLExpert (apply? A S I C) C.

releaseRExpert (apply? A S I C) C.
```

11.10 Figure The CPS fragment of the simple outline FPC at the specification level (finished).

This set of constructors represents the structure of a proof more faithfully. Their definition is completely local and their bookkeeping is self-contained in their own counters, which have on effect on continuation certificates. To finish a branch of the proof in this format, a certificate constructor of the first group is used. If a certificate reproduces the bipole structure of a proof closely enough, it will generally suffice to use the shallow `search` to complete each branch of the proof scaffold, or a limited form of `apply` in a slightly more general setting, after the general high-level structure of the proof has been established.

Figure 11.8 presents the continuation-passing style parts of the FPC definition in the more compact notation of λProlog—i.e., at the specification level of Abella—where some of the more verbose idioms of Abella are elided for compactness. The code is presented in pure pattern matching style. This level of presentation affords a modular organization of the development; thus, the counterpart definitions for whole proof subtrees (similar to those shown here for the CPS fragment) can be contained in a separate file and are combined only when a specification is loaded—instead of accumulating as definitions at the reasoning level, more massive and less organized.

11.6 Certificate families: administrative outlines

The second family of high-level FPC definitions is what we call *administrative outlines*—in reference to the greater amount of bookkeeping information they contain, geared towards more precise descriptions of proofs of large families of commonplace theorems. Specifically, simple outlines offer means of describing the high-level structure of a proof and specifying what lemmas it may employ; they do not offer a comparable level of control over the structure of formulas and local hypotheses, and how to refer to them—for example, all local decisions are treated through a singleton index `idxlocal`, even though knowing how to make use of which hypotheses, and when, is at the heart of the proof scripts of assistants like Abella. Administrative outlines will provide mechanisms to express this information in the FPC framework.

The certificate constructors of administrative outlines allow for a significant amount of guidance to steer proof search more accurately, especially with respect to disambiguating choice points instead of relying on backtracking search—potentially very costly, especially as proofs grow. In describing the general structure of a proof, the kind of structures found in simple outlines reoccur, with

some important differences. These advanced features will be chiefly involved with refined forms of *indexing*. More specifically, we define a general-purpose index type for the storage of formulas, consisting of two parts which correspond to the two kinds of formulas we need to store—lemmas and local formulas:

1. A *numeric index* used as a unique identifier for formulas stored through the decide rule, and unused in the case of externally supplied lemmas.

2. A *boolean index* used to label and describe a formula. It holds a name through which to refer to lemmas, and maintains bookkeeping information in the case of locally stored formulas.

Certificate constructors in this FPC definition form a single family of tactics in continuation-passing style, with a single terminal constructor. While they look similar to the simple outlines of the previous section, their control flow differs significantly. A common *control structure*, `Ctrl`, constrains the operation of all of them. The defined constructors are the following:

1. (`induction Ctrl NamesB Cert`): apply obvious (co)induction on the first (asynchronously) available fixed point, and continue the proof using `Cert`. Use `NamesB` to give names to the components of the fixed point, as explained below.

2. (`inductionS Ctrl S NamesB NamesS Cert`): a variant of the previous constructor, apply (co)induction on the first available fixed point using `S` as invariant, then continue the proof using bounded `search` constrained by `Ctrl` on the base case and `Cert` on the inductive case. Use `NamesB` to give names to the components of the fixed point and `NamesS` to give names to the components of `S`.

3. (`case Ctrl CertL CertR`): apply asynchronous case analysis, locating the first disjunction on the left. Continue the proof using `CertL` and `CertR` for the left and right branches, respectively.

4. (`apply Ctrl Idx Names Cert`): perform bounded search, using `Idx` to decide on the next lemma and use `Names` to give names to the components of the selected lemma.Continue the proof using `Cert`.

5. (`search Ctrl`): perform bounded search without applying induction or deciding on externally provided lemmas.

All certificates share a common control structure that maintains bounds and bookkeeping parameters to constrain proof search. There is some variability in the way these parameters are used by the FPC, as will be seen. The structure has the form `(ctrl Limits Names)`, where `Names` is the current naming structure and `Limits` contains the following bounds and bookkeeping information. The five parameters given (less uniformly) for the various constructors of simple outlines are always present through the control structure. The design is reminiscent of TAC (Baelde et al., 2010) and some of the terminology is biased towards least fixed points, so that asynchronous and synchronous unfolding are associated with the left-hand and the right-hand side rules, respectively. The additional control fields are the following:

1. *Release right:* a boolean flag that enables the release rule when active. In common proofs of "administrative lemmas" involving least fixed points, once we reach a focus on the right, it is often meant to represent a purely positive computation which either terminates or fails. In these situations, a release is considered a dead end, and thus ruled out.

2. *Next local index:* an incrementing counter that contains the next available unique index to store formulas along the current branch of the proof, starting from zero.

3. *Current local index:* a bookkeeping parameter that maintains information about the progress of local indexes for the decide rule as the range of possible choices cycles through the range of indexes in local storage (given by the previous parameter).

We now turn our attention to the second component of the control structure. Consider the proof scripts written in a proof assistant like Abella. It is remarkable that a script consists of a sequence of decisions, and these can conceivably be turned into certificate constructors. In addition to selecting formulas and lemmas, an important part of the instructions in a proof script is the set of hypotheses that are used to instantiate each formula that the proof operates upon. This information has been absent from our certificates so far, which means backtracking search must be applied to find a right combination of values, if one indeed exists. In some cases this will be easy, but in others much time will be wasted applying, say, sequences of lemmas that make no sense, or using the wrong parameters, leading in the end to complex proof attempts that must be discarded.

Syntactically, a formula can be seen as a tree whose leaves are drawn from the nullary logical constants, the fixed point operators and the equality operator; and its nodes from the remaining, recursive logical constants. (Interestingly, unfolding operations grow a fixed point leaf into a subtree.) Under this view, a naming structure associated to a formula is another tree that replicates the branching structure of the formula down to the fixed points contained in the formula, and attaches names to them.

11.6.1 **Example**   Consider the fixed point definition of the addition relation on natural numbers given in Figure 10.2. Suppose we want to refer to the recursive call to plus in its body as "H1." Mimicking the branching structure of the formula by `split` constructors and labeling nodes by `name` constructors, we may get:

```
(split _ (split _ (split _ (name "H1"))))
```

Note that linear branching (i.e., quantifiers) is omitted from the syntax. Nodes representing subtrees that do not contain any fixed points are in effect don't-care identifiers (because they will never be used to constrain the inference rules that need to name fixed points-as-atoms), here represented by anonymous variables `_` (but see the discussion after Example 11.6.2).

The object of naming structures is labeling the "atoms" in a formula so that they can be matched with the contents of the context, especially frozen fixed points acting as hypotheses. Formulas as commonly defined in the FPC framework are terms of a simple inductive type; they are not annotated and are mostly or completely opaque to clerks and experts—see Section 4.4 for a general discussion, and Section 7.4 for a use case that benefits from reflective inspection. In consequence, name annotations have to be provided separately and in parallel to formulas and sequents. At the cost of some redundancy, it is possible to furnish this information in the certificate without any changes to the kernel being needed.

To this end, we use the second member of the control structure. This piece of information must shadow the structure of the sequent with a level of detail that allows us to give names to the components of interest. *Boolean indexes*, defined earlier, will hold this information, of which a simple name (such as is used to tag lemmas) is a particular case. Since formulas in storage already hold this information, the certificate must maintain name maps for the *workbench zones* on each side of the sequent turnstile, respectively: `(names Delta Goal)`.

The *data mirroring* just described creates a new requirement for *code mirroring* in the FPC definition: clerks and experts must be aware of the changes

made in the workbench by the kernel on each inference rule and reflect those operations exactly in the naming structures for them to remain accurate. This *lockstep requirement* is a fairly strong dependency which, while incapable of compromising soundness, might lead to mangled information that impedes progress of the proofs. The issue is mitigated by the fact that the kernel's operation is stable and well documented, although this lockstep pattern reoccurs in other scenarios, including applications to test data generation described in Section 12.6. Example 11.6.2 illustrates the mimicry of the kernel by the clerks and experts.

11.6.2 Example   Observe the implementation of the inference rules of the system *μLJF^a* in Figure 9.6 and companion figures as the Abella kernel in Figure 10.3. Figure 11.11 presents the implementation of the lockstep pattern for two representative rules: the introduction rule for negative conjunction and the full induction rule on least fixed points.

The clerks and experts are essentially implemented by auxiliary predicates which generate the continuations from the designated input certificates. They achieve this by manipulating the control structure through other, modular auxiliary predicates. For example, the conjunctive clerk looks for a split in the naming structure at the head of the list of names that replicate the LHS of the sequent and replaces it with its two sub-components—exactly as the kernel decomposes the connective. If a split naming structure is not found at that position, the default is copied on both positions.

The inductive clerk is somewhat more complicated—reflecting the manipulations that turn the conclusion of the inference rule into its premises. In particular, it must extract and combine naming structures for the fixed point and the invariant. The guiding principle is the same: the naming structure mirrors at all times the sequent to which the certificate is associated.

However, the burden of naming must not be an obligation, and restoring the lost flexibility is indeed simple. A `name` leaf will give name to all relevant sub-formulas covered by it. In this way we can define "buckets" of homonymous formulas which can be stored and decided upon indifferently. If we do not wish to make any use of this functionality, it suffices to write a naming structure for the initial sequent where a single name is used—like `idxatom` is in simple outlines: `(names nil (name "Dummy"))`. (Note that definite identifiers, and not anonymous variables as in Example 11.6.1, must be used if there are fixed points to be stored and decided upon.)

```
Define andClerkNames : cert -> cert -> prop by
  andClerkNames Cert Cert' :=
    getControl Cert
      (ctrl Limits
        (names ((split NameL    NameR) :: Delta) Goal)) /\
    setControl Cert
      (ctrl Limits
        (names (        NameL :: NameR  :: Delta) Goal))
      Cert' ;
  andClerkNames Cert Cert' :=
    getControl Cert (ctrl _ (names ((name Name) :: _) _)) /\
    pushDelta Cert (name Name) Cert'.

Define andClerk : cert -> cert -> prop by
  andClerk Cert Cert' :=
    andClerkNames Cert Cert'.

Define indClerkNames : cert -> cert -> (i -> cert) -> prop
by
  indClerkNames Cert CertSt CertBSx := forall x,
    % Pop current and decompose base certificate
    popDelta Cert _ Cert' /\
    Cert' = (inductionS Ctrl S NamesB NamesS SubCert) /\
    % Compose first sub-certificate
    pushDelta (search Ctrl) NamesS CertSt /\
    % Compose second sub-certificate
    replaceName NamesB NamesS NamesBSx /\
    SubCert' = (SubCert x) /\
    pushDelta SubCert'  NamesBSx SubCert'' /\ % Empty Delta
    setGoal   SubCert'' NamesS   SubCert''' /\
    CertBSx = (_\ SubCert''').

Define indClerk :
cert -> cert -> (i -> cert) -> (i -> bool) -> prop by
  indClerk Cert Cert' Cert'' S :=
    Cert = (inductionS _ S _ _ _) /\
    indClerkNames Cert Cert' Cert''.
```

11.11 Figure Lockstep operation in the administrative FPC. Two example clerks are shown with references to supporting relations used to manipulate certificates and their components. This general organization is extended to the remaining clerks and experts.

11.6.3   Example   Consider a theorem statement of the form $A \supset B \supset C$. Suppose we want to write a proof outline where we refer to the formula A as a hypothesis named "H1," to the formula B as hypothesis "H2," and to the goal C as "G." In the top-level certificate constructor of the administrative outline, we will provide the following naming structure:

```
(names
  nil
  (split (name "H1") (split (name "H2") (name "G")))
)
```

Similarly, suppose there is in our collection of lemmas one of the form $D \supset E$, such that, if we can equate A with D, we may infer E from it. We can guide the kernel towards this selection selection by a certificate that supplies the name of this lemma *and* establishes the correspondence between the hypothesis D and its match in the context, A, based the symbolic names given them above. Thus, we would pick "H1" and generate a new name for the conclusion E, say, "H3." We would write:

```
(split (name "H1") (name "H3"))
```

The last distinct role of naming structures concerns their role in inductive schemes (i.e., involving certificate constructors `induction` and `inductionS`). Two dedicated structures are needed for this:

1. Both `induction` and `inductionS` require a naming structure that assigns labels to the unfolding of the fixed point being induced upon. Because induction is a carefully guided "tactic," it is reasonable to assume that we know what our desired induction is—as well as the names of its parts.

2. In the general induction scheme of `inductionS`, the certificate must supply (along with the invariant) a matching structure naming the parts of said invariant. The FPC definition will then construct the naming structure that results from applying the induction rule. Therefore, the first naming structure (i.e., that of the unfolding, above) must confer a special annotation to the leaf corresponding to the place where the invariant will be injected.

On a final note, continuation certificates inherit the state of the naming structures of their predecessors at the continuation point. Ultimately, all this scaffolding replicates at the FPC level what proof assistants do to generate and manage names. Much of it could be automated, although the problem of choosing and using good names is difficult and best left to the client.

```
Theorem plus0com :
  forall N, nat N ->  plus N z N.
certify (induction! 1 0 1).

Theorem plusscom :
  forall M, nat M -> forall N, nat N ->
  forall P, plus M N P -> plus M (s N) (s P).
certify (induction! 1 0 1).

Theorem pluscom :
  forall N, nat N -> forall M, nat M ->
  forall S, plus N M S  -> plus M N S.
certify (induction! 2 1 0).
```

11.12   **Figure**   (Simple) proof outlines for the commutativity of addition in Abella.
For each lemma, a self-contained induction with adequate allowances for decide
depth and unfolding will arrive at the same proof as the scripts in Figure 11.3.
Here, the most compact, single-constructor forms are given with tight bounds
and in marshaled form.

11.7   Experiments

Let us return to the case study in Section 11.2. Suppose for the moment that
Abella has been extended with a tactic, `certify`, which takes a proof outline
and uses it to attempt to prove the current theorem. It is patently clear from,
say, Figure 11.3 that the proofs expressed therein can be written as outlines.
Inspection of the proofs or experimentation quickly lead to the simple outlines of
Figure 11.12—or to the essentially identical administrative outlines. (The interface
between Abella and the checker is detailed in Chapter 13.)

11.7.1   **Example**   We shall now revisit the case study in Section 11.2 and detail how
such proofs can be expressed in terms of proof outlines, leading the way to the
expression of proof scripts as proof certificates.

Let us analyze in detail the proof script of `plustotal` in Figure 11.2. It
begin by the common induction pattern:

```
induction on #. intros. case H#.
```

This corresponds to an application of the obvious induction, which implicitly
includes a case analysis—which derives as many goals as the predicate has clauses:
two in the case of addition, the emphzero case and the *successor case*. (Here, we
assume a model in which the obvious induction does not require a certificate for

the trivial branch.) Note that the induction certificate greedily inducts on the first available fixed point; inducing on others requires an easy generalization in the FPC definition, or reordering the hypotheses—so that H# becomes H1—in the theorem statement to account for that constraint. In certificate terms, this translates into the following certificate pattern:

```
(induction?
  (case? 0
    ...
    ...
  )
)
```

In it, there are two holes for the continuation certificates in each case. We now turn to consider each subgoal in turn. In the *zero case*, we have a simple goal, `exists S, plus z M S`, solved by a simple use of the `search` tactic. It is easy to see this involves focusing on the right for the positive phase, unfolding once on the right and applying initial. In certificate terms:

```
(apply? 0 1 (idx "local") search)
```

In the *successor case*, the process is a bit more complex. The goal starts from the following sequent:

```
Variables: M N1
IH : forall N, nat N * ->
       (forall M, nat M -> (exists S, plus N M S))
H2 : nat M
H3 : nat N1 *
============================
 exists S, plus (s N1) M S
```

Whereas in Abella we need to appeal to the induction hypothesis explicitly, in the certificate outline it becomes a natural consequence of the asynchronous phase, and it need not be handled explicitly here. In Abella, a new hypothesis results from `apply IH to H3`:

```
H4 : forall M, nat M -> (exists S, plus N1 M S)
```

The rest of the sequent remains unchanged. At this point we have a collection of "atoms" and negative formulas on the left, and a positive goal on the right: if nothing is done to the atoms, they will be frozen, and we will have reached the end of asynchrony. The proof script instructs to operate on H4 using other

hypotheses for assumptions. Therefore, fixed are indeed frozen and we move to the synchronous phase, and a bipole must be signaled through a local decide:

```
(apply? 0 0 (idxlocal) ...)
```

This local decide will reproduce the choice of H2. In Abella, at the end of the synchronous phase, a new hypothesis has been generated:

```
H5 : plus N1 M S
```

Finally, the search tactic finishes the goal, and with it the proof. This corresponds, as in the zero case, to a focus on the right hand side, with unfolding and initial. The final certificate is thus:

```
(induction?
  (case? 0
    (apply? 0 1 (idxlocal) search)
    (apply? 0 0 (idxlocal) (apply? 0 1 (idxlocal) search))
  )
)
```

By a similar process, we—or an automated analyzer integrated in Abella—can reformulate the main result of Section 11.2 as a detailed proof outline.

11.7.2 Example   The proofs in Figure 11.3 establish the commutativity of the addition relation between natural numbers as the theorem pluscom. A proof by simple induction relies on two auxiliary lemmas, one for each case of the induction on natural numbers (zero and successor), themselves proved separately by simple inductions.

We shall look at the certificate for the main theorem, which involves the application of lemmas (the proofs of those two lemmas are simpler inductions in the manner of Example 11.7.1). The complete certificate is given by the term:

```
(induction?
  (case? 0
    (apply? 1 0 (idx "plus0com") search)
    (apply? 2 0 (idx "local") (apply? 0 0 (idx "plusscom")
      (apply? 0 0 (idx "local") search)))
  )
)
```

Compare this with the proof script in Figure 11.3. In the certificate, decides on lemmas refer to those by their given names earlier in the Abella session. Again, the structure is simple: after starting with the obvious induction on the first

argument of the addition, a `case?` branches out the proof certificate into the two cases for zero and successor. Both follow the same pattern of applying a lemma by focusing on it, from which the proof of the branch follows; the application of the corresponding lemma is preceded by a small amount of preprocessing.

The focused discipline of the outline FPCs imposes additional structure to the proofs, which in turn results in some amount of automated processing. In Abella, the proof of the successor branch of the induction opens with the following sequent:

```
Variables: M S N1
IH : forall N, nat N * ->
        (forall M, nat M ->
          (forall S, plus N M S -> plus M N S))
H2 : nat M
H3 : plus (s N1) M S
H4 : nat N1 *
============================
 plus M (s N1) S
```

Note that the first two steps in the proof script—i.e., the case analysis on "H3" and the application of the inductive hypothesis—are directly handled by the semantics of the FPC and not reflected in the outline: in a sense, there is nothing interesting in these two obvious steps. This is related to the concept of *progressing unfolding* as described by Baelde et al. (2010). After these, it is easy to identify which proof steps correspond to unfoldings and record these in the allowances made in the certificate terms.

The translation from proof scripts to outline certificates is mechanic but cumbersome for the user of a proof assistant. In practice, it is simpler to fall back to the more implicit but slower outlines: compare the original proof scripts in Figure 11.3 with the much more readable outlines given in Figure 11.12—the latter compact the outlines studied in Example 11.7.2. Clearly, more generous bounds will also arrive at the same proofs—provided that the necessary lemmas are available—, but it would be desirable to refine said bounds to be as tight as possible, and to add additional details to make proofs more efficient…all this without imposing the burden on the user. Certificate pairing (for which see Chapter 5) can be easily applied to enable these kinds of manipulations, in particular in combination with the simple outlines of Section 11.5 for purposes of certificate elaboration.

The essential component (alongside unfolding bounds) that separates the more implicit and more explicit forms of outlines is the specification of a skeleton for the proof tree decorated with the required decisions, in particular the necessary lemmas as they occur. An alternative representation that can be written succinctly (like implicit outlines) and paired easily may encapsulate the tree of decisions in a separate data structure, parallel to the unfolding bounds. The modus operandi will be to enrich the self-contained certificate constructors of simple outlines with such an abstract tree of decisions. This is achieved by extending the FPC definition of simple outlines with the following doubles of the self-contained constructors:

1. (`induction# B AU SU AC SC D`): decorated version of the self-contained certificate (`induction B AU SU AC SC`) with an extra parameter, `D`, representing the tree of decisions made throughout the proof.

2. (`apply# B AU SU AC SC D`): decorated version of the self-contained certificate (`apply B AU SU AC SC`), extended with a decision tree `D` as for the above case.

The third kind of self-contained certificate, `search`, involves no such decisions and therefore remains unaltered. For brevity, the `inductionS` variant is not considered here; it experiments identical changes. Clerk and expert analogs for the new copies are declared with identical behavior except for the new checks that decision trees must match the structure of the proof, as illustrated in Figure 11.13. The reason for declaring a new family of constructors instead of performing elaboration, say, from self-contained to continuation-style simple outlines is that non-disjoint constructors make certificate elaboration ambiguous—as discussed in Section 5.6—noting that continuation-style certificates can at any point finish a branch by a self-contained certificate. The type of decision trees is defined by the following constructors:

1. A *branching constructor*, `btbranch`, used to represent a branching point in the proof with the decision trees for each branch.

2. Two *decision constructors*, `btlocal` and `btlemma` for local and lemma decides, respectively, each taking an index describing the decision and a continuation certificate.

3. A *terminal constructor*, `btinit`, representing a subtree where no decisions take place.

```
orClerk (apply# N AU SU AC SC (btbranch IdxL IdxR))
        (apply# N AU SU AC SC IdxL)
        (apply# N AU SU AC SC IdxR)

decideLClerk (apply# N AU SU AC SC (btlocal idxlocal Idx))
             (apply# N AU SU AC SC Idx)
             idxlocal

decideLClerk' (apply# N AU SU AC SC (btlemma Idx Idx'))
              (apply# N AU SU AC SC Idx')
              Idx

decideRClerk (apply# N AU SU AC SC (btlocal idxlocal Idx))
             (apply# N AU SU AC SC Idx)
```

11.13 Figure   Decision trees are implemented in the FPC family for simple outlines by a small set of additions implemented as clerk and expert clauses, whose interesting cases are summarized here. In the closed world of Abella, these new clauses are added to the existing definitions and cannot be accumulated separately.

If the usual pairing combinator used to check two certificates in tandem is represented by (`pair# C1 C2`): the usual pairing combinator, a simple but useful elaboration is achieved by combining a self-contained outline (in the style of Section 11.5) with an extended outline (as given in this section), identical in every detail except that the decision tree is left unspecified as a logic variable. The standard outline drives proof reconstruction, while the extended outline—containing the same amount of information—is necessarily in agreement, but also records the branching points and decisions in its vacant structure. Upon a successful check, the decision tree will contain information about the number of decisions made, what lemmas were used, and the general structure of the proof. This data can be exploited to speed up subsequent checking, or even to further elaborate the new information in the form of a nested certificate in continuation-passing style.

11.7.3 Example   Suppose we find a valid compact certificate for `pluscom`—in the context of a session, where previously proved theorems are available as theorems—which takes the following form:

```
(induction 3 2 0 2 0)
```

(We know from our analysis and Figure 11.12 that these bounds are indeed sufficient in the presence of the auxiliary lemmas for the zero and successor cases.)

While this is easy to use, we may want to extract some more information from
the proof, be it to learn something from it (i.e., the list of used lemmas) or to
elaborate it into a derived certificate that is easier to check. To retrieve the tree of
decisions, it suffices to pair it with its double as described above:

```
(pair# (induction 3 2 0 2 0) (induction# 3 2 0 2 0 D))
```

The joint checking of these two certificates in lockstep leads to the recording
of the decision points in D, as desired.

Proof outlines can be applied with identical ease and flexibility to many
problem domains and to degrees of complexity beyond the simple case study
used in this chapter: arithmetic, data structures such as lists, metaprogramming
concepts like program contexts and calculi like CCS and the π-calculus, alongside
mathematical concepts like simulation and bisimulation, etc.

11.8 Notes

The certificates for outline families presented in this chapter were first published
in Blanco and Miller (2015).

The final technical step is to enable certificates such as these to be developed
and executed directly within Abella. An end-to-end solution is described in
Section 13.3. Our first system made use of the Bedwyr model checking system,
itself closely related to Abella. The changes to Abella in this delegated approach
were minimum, as the proof assistant sat at the top of an architecture designed to
produce and check proof outlines by a simple linear workflow in three steps:

1. A *theorem prover* (Abella in this case) is the source of the concrete syntax
   of definitions, theorems and proofs. Its direct involvement in the workflow
   can be quite limited. We extended the tactics language with `ship` tactic
   which marked proof obligations that were to be discharged by an external
   checker by means of a provided proof outline.

2. A *translator* specific to each theorem prover converts the concrete syntax of
   the theorem prover into that of the proof checker. It may simply use proof
   certificates declared by a `ship` tactic or derive certificates automatically
   starting from proof scripts and other similar evidence. Such a translator may
   be built into the theorem prover, thus encapsulating this and the previous
   step. The translator generates input usable directly by a general-purpose
   checker.

3. A *proof checker* implements a proof system according to the precepts of the FPC framework. We used Bedwyr to implement and execute a μLJF^a kernel—instanciated with the FPC definitions of outlines and the translations generated from previous steps.

A similar process of certification by outlines can be adopted by other theorem provers. It suffices to provide a connection between those provers and the proof checker by means of a custom translator. These components are free to implement sophisticated refinement mechanisms to produce more efficient certificates from the information contained in proof scripts—or provide limited guidance and rely on the checker to reconstruct the missing details.

The proof certificates described in this chapter differ from their use as an independent representation of proof evidence—rather, they represent decision procedures which reconstruct a proof from a collection of high-level descriptions of proofs. The abstract inference steps defined by those certificate constructors have been in generally decidable; commonly, discussions about Frege proofs involve polynomially checkable inference rules. This restriction is superfluous, and proof reconstruction may involve large numbers of choices and undecidable procedures. None of these generalizations are a concern of the FPC framework, which only forbids—by construction—menaces to the soundness of whatever proofs result from reconstruction. How long this reconstruction takes, or whether it terminates, has no bearing on the soundness property. The minimalistic proof outlines explored in the chapter are, indeed, fairly expensive to check (and are executed in a programming environment which is not itself optimized for performance), but their elaboration to more detailed forms of proof evidence mitigates these costs in subsequent runs.

As we have seen, obvious induction rules constitute an important exception to the dictum that it is easy to take the step of trusting the direct encoding of a sequent calculus in a proof checking kernel—provided that we are willing to trust the logic engine used to that end. Although the code involve in the computation of the obvious invariants may be more involved, even external to the kernel, said code does not need to be trusted, since we can always arrange to check the generated invariants.

12 Property-based testing

12.1 A model theory vision of proof theory

Property-based testing, or PBT (Fink and Bishop, 1997), is a methodology developed to assist in the testing of computer programs by automatically generating and executing test data. In its basic form, it PBT pairs a function to be tested with two additional categories of information: (a) a collection of *properties* constituting an *executable specification* of the function that relates inputs and outputs; and (b) a set of *data generators* for the types of the inputs of the function. Given all these elements, it is possible to automate the use of generators to create a certain number of inputs, pass them to the function, and verify whether the outputs satisfy the required properties. While generation of test data comes in manifold methods (Utting et al., 2012), the two principal strategies are random and (subject to certain constraints) exhaustive testing.

Used well, the technique enables quick exploration of the state space of a problem and rapid discovery of errors—either in the specification or in the implementation—in the form of concrete and actionable inputs that falsify the posited properties of the function. A third component is commonly added to the framework: (c) a set of *data shrinkers* which, given an input that contradicts the specification, attempt to find smaller, derived inputs that continue to exhibit the error. Hence, we obtain small counterexamples that are easier to understand and quicker to lead to the root causes of the error.

The concept of PBT was originally applied to programming languages and was notably pioneered by Haskell's QuickCheck (Claessen and Hughes, 2011), from where it spread to many other languages. Soon it made the jump from the world of programming languages to most major proof assistants, where simple ports of QuickCheck gave way to more specialized counterexample generators, such as Nitpick (Blanchette and Nipkow, 2010) for Isabelle. In this latter milieu,

it is used to complement theorem proving activities with a preliminary phase of conjecture testing—whose goal is to zero in on incorrect theorem statements before substantial effort is put into them until a dead end in the proof is found. (Although, as the Curry-Howard correspondence states, the difference between proofs and programs is not profound.)

In this chapter, we will adapt the FPC framework to develop a proof theoretical reconstruction of this style of testing for *relational specifications*—such as those commonly used to describe the semantics of programming languages—and explore its benefits in this environment. We do this by revisiting the concept of *proof outlines* introduced in Chapter 11, noting that the operation PBT strongly resembles the *complement* of a very succinct outline which, instead of attempting to prove a theorem, seeks to *disprove* it. Under the relational lens, (co)inductive definitions correspond to functions, theorem statements correspond to properties, and FPCs implement property-based testing proper. The certificates we write will be used to describe the shape and size constraints of possible counterexamples to a property. If such an outline can be completed in the *μLJF* calculus, we will have obtained a *certified counterexample* to the property in question.

The rest of the chapter is organized as follows. Section 12.2 presents and illustrates the techniques on standard, first-order (algebraic) data structures. Section 12.3 lifts those techniques to more interesting structures containing bindings, represented using λ-tree syntax. Section 12.4 defines disproof outlines as FPCs. Section 12.5 applies those outlines to various problems and benchmarks about the metatheory of programming languages—extracted from related tools such as PLT Redex (Felleisen et al., 2009; Klein and Findler, 2009)—encoding those problems in λProlog, i.e., the specification level of Abella. Section 12.6 looks at the same problem within pure Abella, i.e., at the meta level of the prover. Section 12.7 concludes the chapter.

12.2 Standard property-based testing

To commence, we introduce and advocate a relational view of property-based testing starting from a general scenario. Suppose we wish to prove instances of a class of formulas which obeys the following general pattern:

$$\forall\,(x : \tau)\,.P(x) \supset Q(x)$$

Here, both P and Q are given relational specifications (for clarity, application is denoted in functional notation). In this setting, it can be important to sometimes—as discussed in Section 11.2—move the type judgment for x into the logic of a formula by turning the type τ into a predicate $\tau(\cdot)$ and adding it to the premise of the theorem statement thus:

$$\forall x.(\tau(x) \wedge P(x)) \supset Q(x)$$

In general, proving such theorems can involve significant work since their proofs may be complex and involve the clever invention of prior lemmas and induction invariants. Indeed, it will often be the case in practice that a conjecture of this form will be stated, yet its formal statement will not, in fact, be a theorem of the logic because of *specification errors* in the relational definition of either P or Q, regardless of whether the *intuition* of the conjecture is correct. Therefore, it can be valuable to first attempt to find counterexamples to such formulas before any proof attempts, in the hope of finding many typical but shallow errors, and possibly some deeper ones. We would instead attempt to prove the negation of a conjecture, i.e., a formula of the form:

$$\exists x.(\tau(x) \wedge P(x)) \wedge \neg Q(x)$$

That is, if a term t of type τ can be discovered such that $P(t)$ holds while $Q(t)$ does not, then one can return to the specifications of P and Q and revise them using the concrete evidence in t to determine how the specifications are wrong. Let us call these formulas *counterexample lemmas*. The process of writing and revising relational specifications could be aided if proofs of such lemmas and their associated counterexamples were discovered quickly.

12.2.1 Example Suppose that we wish to write a relational specification for reversal of lists (say, of natural numbers). There are many ways to write such a specification, but in every case the statement of the idempotency of reverse should be a theorem. If the specification is named *rev*, the statement can be written as:

$$\forall (L : \textit{list nat}).\forall (R : \textit{list nat}). \forall (L' : \textit{list nat}). \textit{rev } L\ R \supset \textit{rev } R\ L' \supset L = L'$$

More compactly, L' could be dropped and the condition on *rev* written as *rev L R ⊃ rev R L*. Here, we assume the standard definition for a polymorphic type of lists instantiated with natural numbers as the type of elements. (While

λProlog supports such polymorphic types, this feature is not yet part of Abella's mainline; this limitation extends to the embedded λProlog interpreter of the specification logic.)

The literature describes two interpretations of relational specifications written in the style of Horn-like clauses in proof theoretical terms.

1. In specifications written directly in a relational style, say, in a language like Prolog or λProlog, we can interpret some of those specifications directly as, say, Horn clauses—provided that they are in that fragment of the logic. For example, each purely positive clause for the specification of addition of natural numbers in Figure 4.2 corresponds to one of the Horn clauses in Figure 4.1. In this encoding, proof search is structured as a series of alternations between a *goal reduction* phase and a *backchaining phase* (Miller et al., 1991). Focused proof systems generalize this style of proof reconstruction, where goal reduction and backchaining correspond to the asynchronous and synchronous phases, respectively, and proof search is thus organized in bipoles. *LKF* is a representative of such a system.

2. A complementary approach to the proof theory of Horn clauses involves encoding those clauses as fixed points, hence effecting the *closed-world assumption*. (For example, the same Prolog-style specification of Figure 4.1 can be transcribed as the fixed point definitions of Figure 9.1.) In a proof system enriched with fixed points—such as *μLJF*—proofs of counterexample lemmas span a single bipole: first, a synchronous *generation phase* followed on all its premises by a single, asynchronous *testing phase* that completes the proof. Consequently, implementing the testing phase is an easy job; the difficulties lie in efficiently steering the generation phase through potentially large amounts of nondeterminism. Complexity is thus concentrated in the design and combination of generators.

12.2.2   Example   The counterexample lemma for the specification in Example 12.2.1 is the following, where *list_nat* is a generator of lists of natural numbers:

$$\exists L.\exists R.\,(\textit{list\_nat } L \wedge \textit{list\_nat } R) \wedge (\textit{rev } L\, R \wedge (\textit{rev } R\, L \supset \bot))$$

Here, we associate conjunctions in two groups. On the left is the test data generation phase, charged with the selection of inputs. On the right is the purely computational checking phase, which includes validates the combination

of inputs (i.e., the reverse of L must be R) and attempts to prove the negation of the conclusion. Note a naive generation phase like the one shown here assigns independent values to variables which are functionally related, resulting in backtracking and wasted work. If we take the view that *rev* "computes" R from L, we can simply drop the superfluous generator *list_nat R*.

In the above example, the property theorem is correct, and in consequence, failure to prove it (i.e., success in proving a counterexample lemma) must arise from an error in the specification of *rev*. Properties (and generators) can also be defective: if we claim that a list is always equal to its reverse (*rev L L*), we can test it by looking for instances of its counterexample lemma:

$$\exists L. \exists R.\ list\_nat\ L \wedge (rev\ L\ R \wedge (L = R \supset \bot))$$

Here, any non-symmetrical list of two or more elements, such as $[0; 1]$, will falsify the pseudo-property. PBT should help us find both types of errors quickly.

12.3 Treating metatheoretical properties

Describing computational tasks using proof theory often allows us to lift descriptions based on first-order (i.e., algebraic) terms to descriptions based on higher-order abstract syntax, specifically the *λ-tree syntax* representation (Miller and Palamidessi, 1999; Miller, 2000), which gives clean, declarative readings of variable binders. These possibilities extend to the two logical frameworks used throughout this thesis and represented by the two interpretations of relational specifications studied in this chapter.

1. Once logic programming is described in terms of proof search, it is natural to extend the treatment of first-order terms and Horn clauses (in Prolog) to the general manipulation of λ-terms (in λProlog).

2. Similarly, a sequent calculus presentation of model checking and inductive theorem proving in a first-order logic with fixed points (Baelde, 2012; Heath and Miller, 2017) leads to generalizations based on λ-terms like Bedwyr and Abella, respectively.

The full treatment of λ-tree syntax in a logic with fixed points is usually accommodated by the addition of nominal quantification with the nabla quantifier, as treated in Section 9.4. An important result about nabla is the following:

Types	A, B	::=	$int \mid ilist \mid A \to B$
Terms	M	::=	$x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \mid c \mid err$
Constants	c	::=	$n \mid plus \mid nil \mid cons \mid hd \mid tl$
Values	V	::=	$c \mid \lambda x{:}A.\ M \mid plus\ V \mid cons\ V \mid cons\ V_1\ V_2$

12.1   Figure   Syntax of the *Stlc* language.

$$\frac{}{hd\ (cons\ M_1\ M_2) \longrightarrow M_1}\ \text{E-HD} \qquad \frac{}{tl\ (cons\ M_1\ M_2) \longrightarrow M_2}\ \text{E-TL}$$

$$\frac{}{(\lambda x : A.\ M)\ V \longrightarrow [x \mapsto V]M}\ \text{E-ABS}$$

$$\frac{M_1 \longrightarrow M_1'}{M_1\ M_2 \longrightarrow M_1'\ M_2}\ \text{E-APP1} \qquad \frac{M \longrightarrow M'}{V\ M \longrightarrow V\ M'}\ \text{E-APP2}$$

$$\frac{}{\vdash_\Sigma err : A}\ \text{T-ER} \quad \frac{\Sigma(c) = A}{\vdash_\Sigma c : A}\ \text{T-K} \quad \frac{x : A \in \Gamma}{\Gamma \vdash_\Sigma x : A}\ \text{T-VAR}$$

$$\frac{\Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x : A.\ M : A \to B}\ \text{T-ABS} \qquad \frac{\Gamma \vdash_\Sigma M_1 : A \to B \quad \Gamma \vdash_\Sigma M_2 : A}{\Gamma \vdash_\Sigma M_1\ M_2 : B}\ \text{T-APP}$$

12.2   Figure   Static semantics (evaluation and typing relations) of the *Stlc* language. Variable types are implicitly based on the typographic conventions for the various syntactic categories defined in Figure 12.1.

12.3.1   Theorem   If fixed point definitions do not contain implications and negations (i.e., they are essentially positive), then moving between the universal quantifier $\forall$ and the nabla quantifier ∇ does not affect the provability of atomic formulas.

Proof.   Follows from the properties of nabla in Miller and Tiu (2005, Section 7.2).

$\square$

Because the present study is limited to Horn style recursive definitions, there will be no observable differences between both quantifiers. In consequence, in the first setting, we will be able to use, say, the λProlog implementation of the universal quantifier, **pi**, to implement nabla.

Among other applications, nabla has been used to formalize the metatheory of systems like the λ-calculus and the π-calculus. We illustrate its applications to property-based testing on a variation of the first: the simply-typed λ-calculus extended with primitives for natural numbers and lists of natural numbers, following the PLT Redex benchmark. We call this language *Stlc*, and its syntax is given in Figure 12.1; its static semantics—which shall become the basis of

our implementation—is in Figure 12.2. The direct treatment of λ-terms within the PBT setting will allow us to apply the same generate-and-test proofs to find bugs in implementations of programming languages—such as failure of expected properties like type preservation—seamlessly lifted here to terms with binders.

12.3.2   **Example**   Let *step* a specification of the small-step evaluation relation and *wt* a specification of the typing relation—reflecting the language semantics in Figure 12.2, while the signature (with binders) obeys the syntax laid out in Figure 12.1. The type preservation property can be given form as a candidate theorem:

$$\forall E. \forall E'. \forall T . step\ E\ E' \supset wt\ E\ T \supset wt\ E'\ T$$

If *is_exp* is a generator of expressions, i.e., λ-terms, and *is_ty* is a generator of simple types, the following counterexample lemma can be used to attempt to uncover faults in the specification:

$$\exists E. \exists E'. \exists T . (is\_exp\ E \wedge is\_exp\ E' \wedge is\_ty\ T) \wedge (step\ E\ E' \wedge wt\ E\ T \wedge (wt\ E'\ T \supset \bot))$$

As in previous examples, we note that both E' and T should be functionally dependent on E and, if this fact has been established beforehand, the independent generation of dependent data can be substantially simplified.

Before we show how to implement this framework in each of the two logical settings under consideration (in λProlog and Abella, respectively), we need to encode the framework inside these two systems together with the FPC definitions which implement PBT around generate-and-test disproof outlines. This last element, common to both frameworks, is presented in the next section.

12.4   Disproof outlines

Chapter 11 presented FPC definitions to describe classes of problems via high-level *outlines*, in particular describing the general shape of proofs and the application of lemmas—as is customary in, say, proof assistants. Likewise, previous sections in this chapter have described and given proof theoretical justification to another broad class of problems of practical interest: that of counterexample search whose mechanization has been popularized by PBT frameworks. We will now proceed to the description of these *disproof outlines*, which will support both exhaustive and random generation of inputs. By convention, the FPCs we define will operate

on counterexample lemmas of the form:

$$\exists X_1. \cdots \exists X_n. Generate(X_1, \ldots, X_n) \wedge Test(X_1, \ldots, X_n)$$

Here, the "top-level" conjunction separates the two phases of the process. The generation phase will be a conjunction of generators of *ground terms* assigning values to all or some of the existential variables. The testing phase will perform conjunctive computations that assign values to the remaining variables, verify conditions, and ultimately attempt to prove the negation of the conclusion of the original candidate theorem. A simple and complete class of generators for inductive types are the *typing judgments* discussed in Section 11.2, although many other schemes are possible.

Let us first present the design for exhaustive testing, following the model of SmallCheck (Claessen and Hughes, 2009). The FPC definition provides a single top-level certificate constructor, `qstart`, which takes two certificates, one for the generation phase and one for the checking phase. As the proof starts, the formula immediately obtains focus on the right-hand side. The initial certificate and is tasked with the traversal of the formula, generating logic variables for existential quantifiers until it reaches the top-level conjunction, and there the two continuation certificates are distributed, each to its respective branch. Each phase then draws from its own set of certificate constructors:

1. The *generation phase* is controlled by a certificate constructor, `qgen`, which takes a descriptor that places concrete bounds on the size of the terms to generate. These bounds, together with the generator predicates, precisely define the search space for the run. All the action takes place under focus on the right, as positive conjunctions branch out. Generator predicates are expected to embody purely positive, terminating computations—carried out by the unfold expert, ever on the right-hand side. Two standard definitions are contemplated:

   (a) A *height bound*, or `qheight`, which constraints the depth of the derivation trees for the generated terms. This bound is decremented as the generators perform right unfoldings and propagated unaltered on branching points. Notably, this implies that the initial bound applies to every term in the generation phase.

   (b) A *size bound*, or `qsize`, which limits the size of terms in number of constructors. As in the previous case, allowances are decremented on

right unfoldings. However, the size bound is defined so as to span the sum of constructors of all generated terms. Therefore, sizes must be communicated across both sides of a branching point (i.e., a positive conjunction); an intermediate variable is used to this end.

2. The *testing phase* uses a simple constructor, `qsearch`. It follows the generation phase and is designed to be much simpler. As in the generation phase, a number of conjunctive subgoals are computed in separate branches; these subgoals, too, are purely positive, terminating computations. Commonly, the final goal is the negation of the original conclusion and must therefore also allow enough asynchrony to release the focus, move the original conclusion to the left-hand side and search until an application of equality on a unification problem without solutions finishes the proof.

Figure 12.3 shows the FPC definition for this exhaustive search procedure. Note how, in conformance with the single-bipole proof outline, indexing is unusually absent from the picture. Vis-à-vis FPC definitions for *LKF* in λProlog, where the open-world assumption reigns, all the "clauses" of a given clerk or expert must be grouped together in an inductive definition, which limits modularity. In addition, it demands that all clerks and experts be part of the definition—though unused clerks and experts are absent from the figure. This results in somewhat less readable definitions, but does not detract from the simplicity of the design, as seen here. Of course, other generation schemes beyond the two expounded in this section, many other strategies are possible by making available more guidance information through the certificates.

12.4.1   **Example**   A standard certificate with a height bound of 5 can be written:

```
(qstart (qheight 5) qsearch)
```

And a certificate with a size bound of 5 can be written:

```
(qstart (qsize 5 X) qsearch)
```

Note that here 5 is given as the upper bound, whereas the lower bound is left open as a logic variable: it is indifferent how many constructors we use, as long as the bound is not surpassed. Bound propagation is carried out by `andExpert` and `eqExpert` in Figure 12.3. (In both examples, natural bounds are written in algebraic notation for clarity.)

The second variant is random testing as implemented by QuickCheck-style tools (Claessen and Hughes, 2011). The structure of proofs is not affected by the

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

% Generation
Kind    qbound    type.
Type    qheight   numidx -> qbound.
Type    qsize     numidx -> numidx -> qbound.

Type    qgen    qbound -> cert.

% Checking
Type    qsearch    cert.

% Staging
Type    qstart    cert -> cert -> cert.

% Clerks and experts
Define releaseRExpert : cert -> cert -> prop by
releaseRExpert qsearch qsearch.

Define decideRClerk : cert -> cert -> prop by
decideRClerk (qgen Bound) (qgen Bound) ;
decideRClerk (qstart Gen Check) (qstart Gen Check).

Define storeRClerk : cert -> cert -> prop by
storeRClerk (qgen Bound) (qgen Bound) ;
storeRClerk (qstart Gen Check) (qstart Gen Check).

Define unfoldRExpert : cert -> cert -> prop by
unfoldRExpert (qgen (qheight (s Height)))
              (qgen (qheight Height)) ;
unfoldRExpert (qgen (qsize (s In) Out)  )
              (qgen (qsize In Out)  ) ;
unfoldRExpert qsearch qsearch.
```

12.3 Figure   The SimpleCheck-style FPC for (bounded) exhaustive property-based testing in Abella. For brevity, "undefined" clerks and experts (which must be given a false definition under the closed-world assumption) are not shown. Also for clarity, the more concise Bedwyr flavor of syntax is used here (see Section 10.2).

```
Define unfoldLClerk : cert -> cert -> prop by
unfoldLClerk qsearch qsearch.

Define someExpert : cert -> cert -> i -> prop by
someExpert (qgen Bound) (qgen Bound) _ ;
someExpert (qstart Gen Check) (qstart Gen Check) _ ;
someExpert qsearch qsearch _.

Define someClerk : cert -> ( i -> cert ) -> prop by
someClerk qsearch (_\ qsearch).

Define eqExpert : cert -> prop by
eqExpert (qgen (qheight _)  ) ;
eqExpert (qgen (qsize In In)) ;
eqExpert qsearch.

Define orExpert : cert -> cert -> choice -> prop by
orExpert (qgen Bound) (qgen Bound) _ ;
orExpert qsearch qsearch _.

Define andExpert : cert -> cert -> cert -> prop by
andExpert (qgen (qheight Height))
          (qgen (qheight Height) )
          (qgen (qheight Height)  ) ;
andExpert (qgen (qsize In Out)  )
          (qgen (qsize In Middle))
          (qgen (qsize Middle Out)) ;
andExpert qsearch qsearch qsearch ;
andExpert (qstart Gen Check) Gen Check.

Define eqClerk : cert -> cert -> prop by
eqClerk qsearch qsearch.

Define impClerk : cert -> cert -> prop by
impClerk qsearch qsearch.

Define orClerk : cert -> cert -> cert -> prop by
orClerk qsearch qsearch qsearch.

Define andClerk : cert -> cert -> prop by
andClerk qsearch qsearch.
```

12.4 Figure   The SimpleCheck-style FPC for (bounded) exhaustive property-based testing (continued). Presentation conventions are shared with Figure 12.3.

change in strategy, and indeed the checking phase remains unaltered: changes are limited to the dynamic behavior of the generation phase, that is, how test data are generated to attempt to progress through the checking phase and arrive at a proof. These modifications are reflected in the certificate terms and the clerks and experts which consume those certificates; the generators themselves are unaltered, but play an indirect and fundamental role in the design work that follows.

Critically, recall that relational specifications of generators of inductive data types can be encoded as fixed points. The bodies of those fixed point expressions are built out of a series of disjuncts, one for each constructor—for instance, recall the transcription of the typing judgment on natural numbers in Figure 9.1. For each branching point (i.e., a disjunction, one of a series of choices that leads to the encoding of a constructor clause), we will assign a *fixed weight* to each branch, and select one of the branches proportionally and at random. This regime induces the following changes in the FPC design:

1. First, we need to fix the number of times the generation phase will be executed. Generation needs to satisfy a further requirement: if a set of generated values fails to complete the proof—thus falsifying the original property—the complete set of values must be discarded *en bloc*. That is, backtracking must roll the proof back to the start of the generation phase and randomly assemble a completely new, independent set of inputs. In consequence, `qstart` must be augmented with such a *number of tries* parameter, which can be decremented each time we split the initial certificate at the top-level conjunction. Block backtracking will be the responsibility of the generation phase, next.

2. Implementing random branching for generators, we encounter another instance of a *lockstep requirement pattern* (as observed in Section 11.6 for administrative outlines) whereby information about formulas which cannot be attached to formulas without annotations must become part of the certificate and mimic some of the behaviors of the kernel. Consequently, the generation certificate, `qcert`, is structured around an abstract description of generators, `qform`, which mimics their branching structure (i.e., conjunctions, `qand`, and weighted disjunctions, `qor`) down to named recursive generator calls, `qname`, and don't-care branches without generators, `qnone`. The generation certificate thus contains:

(a) A map that assigns names of generators with their `qform` structure. All generators used in such descriptors must have corresponding entries in the map.

(b) Two fields that mirror the workbench zones of the sequent with their `qform` descriptors and progress in lockstep with the kernel.

The lockstep requirement is visible in particular in the synchronous rules for conjunction, disjunction, and unfolding. Moreover, the disjunctive expert must procure some source of randomness from which to determine which branch to take; it must also fail upon backtracking, so that the generation phase runs from beginning to end each time, without intermediate states.

The certificate definition for random PBT is given in Figure 12.5. The presentation relies not only on Bedwyr syntax for readability, like Figure 12.3, but also on some handy features: shallow, like the predefined string type; and deep, like I/O predicates. In fact, both λProlog and Bedwyr offer system interfaces which, though rudimentary, allow the passing of information to a *system call* or *coprocess* to, say, generate a random number between a certain range, or directly pick left or right based on the weight of each branch—and the semantics of I/O predicates correspond to the desired *block backtracking* behavior.

In the purely relational model of Abella, no such interactive facilities avail. For example, suppose a size bound carries over to the maximum number of random choices involved in data generation. In such a situation, a list of random numbers could be passed in the certificate and consumed by: (a) the various iterations of the generation phase; and (b) each instance of the generation phase. Treatment of these external sources of randomness would parallel the propagation of bounds seen in the `qsize` certificates of Figure 12.3, although the solution is not as satisfactory as extending Abella with some form of I/O predicate, following the example of Bedwyr—which was extended precisely for purposes such as this.

12.4.2 Example Consider the final formulation of the counterexample lemma for the obviously false property of list reversal in Example 12.2.2. In it, the generation phase consists of a single list of naturals, for which we may use the standard typing judgments. For natural numbers, this corresponds to the *nat* fixed point of Figure 9.1; *list_nat* is in turned defined in terms of *nat* and the usual constructors:

```
% Generation
Kind    qform    type.
Type    qor      nat -> nat -> qform -> qform -> qform.
Type    qand                      qform -> qform -> qform.
Type    qname    string -> qform.
Type    qnone    qform.

Kind    qmap     type.
Type    qmap     string -> qform -> qmap.

Type    qcert    list qmap -> list qform -> qform -> cert.

% Staging
Type    qstart   numidx -> cert -> cert -> cert.

% Number of attempts
Define iterate : numidx -> prop by
iterate (s _) ;
iterate (s N) := iterate N.

% Clerks and experts
Define releaseRExpert : cert -> cert -> prop by
releaseRExpert qsearch qsearch.

Define decideRClerk : cert -> cert -> prop by
decideRClerk (qcert Map Delta Goal) (qcert Map Delta Goal) ;
decideRClerk (qstart Tries Gen Chk) (qstart Tries Gen Chk).

Define storeRClerk : cert -> cert -> prop by
storeRClerk (qcert Map Delta Goal) (qcert Map Delta Goal) ;
storeRClerk (qstart Tries Gen Chk) (qstart Tries Gen Chk).

Define unfoldRExpert : cert -> cert -> prop by
unfoldRExpert (qcert Map Delta qnone)
              (qcert Map Delta qnone) ;
unfoldRExpert (qcert Map Delta (qname Name))
              (qcert Map Delta Form) :=
       member (qmap Name Form) Map ;
unfoldRExpert qsearch qsearch.

Define unfoldLClerk : cert -> cert -> prop by
unfoldLClerk qsearch qsearch.
```

12.5 Figure   The QuickCheck-style FPC for random property-based testing in
Bedwyr. Presentation conventions are shared with Figure 12.3; unchanged blocks
are omitted as well.

```
Define someExpert : cert -> cert -> i -> prop by
someExpert (qcert Map Delta Goal) (qcert Map Delta Goal) _ ;
someExpert (qstart Tries Gen Chk) (qstart Tries Gen Chk) _ ;
someExpert qsearch qsearch _.

Define someClerk : cert -> ( i -> cert ) -> prop by
someClerk qsearch (_\ qsearch).

Define eqExpert : cert -> prop by
eqExpert (qcert _ _ _) ;
eqExpert qsearch.

Define orExpert : cert -> cert -> choice -> prop by
orExpert (qcert Map Delta qnone)
         (qcert Map Delta qnone) _ ;
orExpert (qcert Map Delta (qor Pr1 Pr2 Form1 Form2))
         (qcert Map Delta Form)   Choice :=
         read Random /\
         (
                 Random <= Pr1 /\
                 Form = Form1 /\
                 Choice = left
         \/
                 Random > Pr1 /\
                 Form = Form2 /\
                 Choice = right
         ) ;
orExpert qsearch qsearch _.

Define andExpert : cert -> cert -> cert -> prop by
andExpert (qcert Map Delta qnone)
         (qcert Map Delta qnone) (qcert Map Delta qnone) ;
andExpert (qcert Map Delta (qand Form1 Form2))
         (qcert Map Delta Form1) (qcert Map Delta Form2) ;
andExpert qsearch qsearch qsearch ;
andExpert (qstart Tries Gen Chk) Gen Chk :=
         iterate Tries.

Define eqClerk : cert -> cert -> prop by
eqClerk qsearch qsearch.

Define impClerk : cert -> cert -> prop by
impClerk qsearch qsearch.

Define orClerk : cert -> cert -> cert -> prop by
orClerk qsearch qsearch qsearch.

Define andClerk : cert -> cert -> prop by
andClerk qsearch qsearch.
```

12.6 Figure   The QuickCheck-style FPC for random property-based testing in Bedwyr (continued). Presentation conventions are shared with Figure 12.5.

$$list\_nat \equiv \mu(\lambda ListNat.\lambda l. \hspace{4cm} l = nil \lor$$
$$\exists n.\exists l'.l = cons\ n\ l' \land^+ nat\ n \land^+ ListNat\ l')$$

The following certificate will guide the proof by generating 10 lists of naturals. An empty list is chosen with 50% probability, otherwise a natural and a continuation list are generated. In generating natural numbers, zero is chosen 90% of the time, otherwise we return the successor of the recursively generated number.

```
(qstart
  10
  (qcert
    ((qmap "is_natlist"
      (qand qnone
        (qor 50 50 qnone
          (qand qnone
            (qand
              (qname "is_nat") (qname "is_natlist"))))))
    :: (qmap "is_nat"
      (qand qnone
        (qor 90 10 qnone (qand qnone (qname "is_nat")))))
    :: nil)
    nil
    (qname "is_natlist")
  )
  qsearch
)
```

Note that, for example, the formula structure of *list_nat* is reproduced as a qmap entry labeled with the name "is_natlist", and the positions of fixed point generators are likewise marked with the names of the entries in the map. In this fashion, the certificate knows what it is generating and how to take random choices.

As with other appearances of this lockstep pattern, the burdensome part of the certificates is an easy target for automation, and extends gracefully to other, possibly more sophisticated random generation strategies. For the remainder of the chapter, illustrations will be based on the cleaner exhaustive FPC, which runs in current versions of Abella without changes.

12.5 Hosted PBT in λProlog

The first of the two extension methods discussed in Section 12.2 makes use of some higher-order features of λProlog to work on signatures with binders using λ-tree syntax. Following McDowell and Miller (2002), we introduce a simple specification logic, which in this case is basically a basic Prolog-like meta-interpreter whose sole peculiarity is its interpretation of nabla as λProlog's universal quantifier. The definition of the interpreter drives the derivation of our object logic, which represents Horn-style clauses through the two-place predicate `prog` which relates heads and bodies of the object clauses. These clauses are built out of object-level logical constants (`tt`, `or`, `and`, `nabla`) and user-defined predicate constructors.

Figure 12.7 presents the kernel; in this purely positive presentation, each logical constant, as well as the unfolding of `prog` clauses, is controlled by a client-defined expert predicate—nabla remains transparent to the kernel, as usual. Two predicates are in charge of the computation, `interp` and `check`, differing only on the use in the latter of the expert predicates to steer search. Both interpret object-level connectives as λProlog code, and look up and unfold client-side constructors in program database of `prog`.

Porting the FPC definition of Figure 12.3 to this embedded kernel is straightforward and much more succinct, only partly due to the slightly simplified model of the checker; Figure 12.8 presents this version of the FPC. Note that there is no need to handle the disjunctive expert: because `prog`-based specifications allow alternatives in the form of variant clauses, disjunctions can be dispensed with at this level. However, it is important to note that—at the specification level—there is no alternative to disjunction in recursive definitions, and therefore the proof theory must give the connective full consideration.

12.5.1 Example We now revisit, in full, the disproof of the property in Example 12.2.2. Again, we want to generate lists of natural numbers and compute their reverse, and find out something about the combination of specification and property. Again, we want to falsify the assertion that the reverse of a list is equal to itself. Figure 12.9 gives the encoding of the problem as `prog` clauses in the hosted kernel. (Note how the specification makes full use of λProlog's type system.) To prove the counterexample lemma, it suffices to pose a goal like the following:

```
cexrev XS YS :- check (qgen (qheight 3)) (is_natlist XS),
```

```
% Formulas and their terms
kind    prolog        type.
type    tt            prolog.
type    and, or       prolog -> prolog -> prolog.
type    some, nabla   (A -> prolog) -> prolog.

% Program and interpreter
type    prog      prolog -> prolog -> o.
type    interp    prolog -> o.

% Certificates
kind    choice        type.
type    left, right   choice.


kind    idx    type.


kind    cert            type.
type    tt_expert       cert -> o.
type    or_expert       cert -> cert -> choice -> o.
type    and_expert      cert -> cert -> cert -> o.
type    unfold_expert   cert -> cert -> o.

% Checker
type    check    cert -> prolog -> o.

% Interpreter implementation
interp tt.
interp (and G1 G2) :- interp G1, interp G2.
interp (or G1 G2) :- interp G1; interp G2.
interp (nabla G) :- pi x\ interp (G x).
interp A :- prog A G, interp G.

% Checker implementation
check Cert tt :- tt_expert Cert.
check Cert (and G1 G2) :- and_expert Cert Cert1 Cert2,
                          check Cert1 G1, check Cert2 G2.
check Cert (or G1 G2) :- or_expert Cert Cert' Choice,
                         ((Choice = left,  check Cert' G1);
                          (Choice = right, check Cert' G2)).
check Cert (nabla G) :- pi x\ check Cert (G x).
check Cert A :- unfold_expert Cert Cert',
                prog A G, check Cert' G.
```

12.7 Figure   The hosted kernel in λProlog. Signature and module names and accumulations of kernel signatures are omitted.

```
% Signature
kind   qbound    type.
type   qheight   int -> qbound.
type   qsize     int -> int -> qbound.
type   qgen      qbound -> cert.

% Implementation of clerks and experts
tt_expert (qgen (qheight _)).
tt_expert (qgen (qsize In In)).

and_expert (qgen (qheight H))
           (qgen (qheight H)) (qgen (qheight H)).
and_expert (qgen (qsize In Out))
           (qgen (qsize In Mid)) (qgen (qsize Mid Out)).

unfold_expert (qgen (qheight H)) (qgen (qheight H')) :-
        H > 0, H' is H - 1.
unfold_expert (qgen (qsize In Out)) (qgen (qsize In' Out))
        :- In > 0, In' is In - 1.
```

12.8   Figure   The SimpleCheck-style FPC for (bounded) exhaustive property-based testing in embedded λProlog. Signature and module names and accumulations of kernel signatures are omitted.

$$\text{interp (rev XS YS), not (XS = YS).}$$

Here, we determine that, since the generation phase alone require guidance, we generate candidate lists in accordance with the limits of a generation certificate, here up to a certain depth by means of `qheight`, pairing it to the goal via `check`. The testing phase performs deterministic computation, each of which components can be delegated to the meta-interpreter `interp`. Finally, the negated conclusion employs *negation-as-failure* (NAF). Given that this NAF involves ground terms exclusively, the call is logically sound.

We are now ready to lift the full implementation—programming environment, kernel, FPC definition—to λ-tree syntax. Figure 12.10 translates the static semantics of Figure 12.2 to `prog` clauses in the hosted λProlog kernel, making use of a signature with the obvious type declarations for constants.

The encoding in λProlog is standard: we declare constructors for terms, constants, and types, while carving out values via an appropriate predicate, `is_value`. Similarly to values, another predicate, `is_err`, characterizes the threading in the operational semantics of the special `error` expression—used to

```
% Signature
kind    nat       type.
type    zero      nat.
type    succ      nat -> nat.
type    is_nat    nat -> prolog.

kind    lst            type -> type.
type    null           lst A.
type    cons           A -> lst A -> lst A.
type    is_natlist     lst nat -> prolog.
type    append         lst A -> lst A -> lst A -> prolog.
type    rev            lst A -> lst A -> prolog.

% Module
prog (is_nat zero) (tt).
prog (is_nat (succ N)) (is_nat N).

prog (is_natlist null) (tt).
prog (is_natlist (cons Hd Tl))
     (and (is_nat Hd) (is_natlist Tl)).

prog (append null K K) (tt).
prog (append (cons X L) K (cons X M)) (append L K M).

prog (rev null null) tt.
prog (rev (cons X XS) RS)
     (and (rev XS SX) (append SX (cons X null) RS)).
```

12.9 Figure Reversal of lists of natural numbers in hosted λProlog. Presentation conventions are shared with Figure 12.7.

model runtime errors such as taking the head of an empty list. Third in line is the small-step evaluation relation, `step`. Progress is defined in terms of these three predicates in the homonymous predicate that embodies the dynamic semantics. The static semantics is contained by the typing predicate `wt` (for "with type"); this assigns an arbitrary type to `error` and types constants via a table encoded by `tcc`. Note that the encoding we have chosen uses *explicit contexts* as opposed to the hypothetical judgments of McDowell and Miller (2002) (see also Section 4.3). This choice avoids implications in the body of the typing predicate and, as a result, allows us to use λProlog universal quantification to implement nabla at the reasoning level.

This version of the λ-calculus enjoys some usual properties, of which we will focus on two. First, it satisfies subject reduction and progress—where progress

```
prog (is_value (c _)) (tt).
prog (is_value (lam _ _)) (tt).
prog (is_value (app (c cns) V)) (is_value V).
prog (is_value (app (app (c cns) V1) V2))
        (and (is_value V1) (is_value V2)).

prog (is_error error) (tt).
prog (is_error (app (c hd) (c nl))) (tt).
prog (is_error (app (c tl) (c nl))) (tt).
prog (is_error (app E1 _)) (is_error E1).
prog (is_error (app E1 E2))
        (and (is_value E1) (is_error E2)).

prog (step (app (c hd) (app (app (c cns) X) XS)) X)
        (and (is_value X) (is_value XS)).
prog (step (app (c tl) (app (app (c cns) X) XS)) XS)
        (and (is_value X) (is_value XS)).
prog (step (app (lam M T) V) (M V)) (is_value V).
prog (step (app M1 M2) (app M1' M2)) (step M1 M1').
prog (step (app V M2) (app V M2'))
        (and (is_value V) (step M2 M2')).

prog (progress V) (is_value V).
prog (progress E) (is_error E).
prog (progress M) (step M N).

prog (memb X (cons X _)) (tt).
prog (memb X (cons Y Gamma)) (memb X Gamma).

prog (tcc (toInt _) intTy) (tt).
prog (tcc nl listTy) (tt).
prog (tcc hd (funTy listTy intTy)) (tt).
prog (tcc tl (funTy listTy listTy)) (tt).
prog (tcc cns (funTy intTy (funTy listTy listTy))) (tt).

prog (wt Ga M A) (memb (bind M A) Ga).
prog (wt _ error _) (tt).
prog (wt _ (c M) T) (tcc M T).
prog (wt Ga (app X Y) T)
        (and (wt Ga X (funTy H T)) (wt Ga Y H)).
prog (wt Ga (lam F Tx) (funTy Tx T))
        (nabla x\ wt (cons (bind x Tx) Ga) (F x) T).
```

12.10 **Figure**   Dynamic semantics of the *Stlc* language, implemented in hosted λProlog, corresponding to the rules presented in Figure 12.2 under an appropriate type signature.

```
% Simple generation
prog (is_exp (c Cnt)) (is_cnt Cnt).
prog (is_exp (app Exp1 Exp2))
    (and (is_exp Exp1) (is_exp Exp2)).
prog (is_exp (lam ExpX Ty))
    (and (nabla x\ is_exp (ExpX x)) (is_ty Ty)).
prog (is_exp error) (tt).

% Maintaining a context of lambda variables
prog (is_exp' _ (c Cnt)) (is_cnt Cnt).
prog (is_exp' Ctx (app Exp1 Exp2))
    (and (is_exp' Ctx Exp1) (is_exp' Ctx Exp2)).
prog (is_exp' Ctx (lam ExpX Ty))
    (and (nabla x\ is_exp' (cons x Ctx) (ExpX x))
         (is_ty Ty)).
prog (is_exp' _ error) (tt).
prog (is_exp' Ctx X) (tt) :-
  memb_ctx Ctx X.
```

12.11 **Figure** Top-level (λ-term) expression generators written in hosted λProlog. For bound variables to appear in the generated expressions a dedicated context needs to be maintained and its contents drawn as valid expressions. This latter step is performed in raw λProlog outside the `prog` interpreter so that it does not wrongly correlate `prog` steps taken inside a generator with size bounds of generated terms.

means, from a direct reading of the `progress` predicate, "being either a value, an error, or able to take an evaluation step." In fact we can easily prove those results in a theorem prover like Abella. Furthermore, by defining λ-term generators in the usual manner PBT can be applied to this setting, as well. The only point of interest concerns the role of nabla and the use of variables generated by it in the resulting expressions, as illustrated in Figure 12.11.

12.5.2 **Example** Complementing a proof of subject reduction, we may wonder whether the calculus enjoys the *subject expansion* property, as well. The sagacious reader will promptly note this is highly unlikely, but will also observe that, rather than waste time in a fruitless proof attempt, we can define a counterexample lemma and let the machine disprove the property for us:

```
cexsexp M M' A :- check (qgen (qsize 8 _)) (step M M'),
                  interp (wt null M' A),
                  not (interp (wt null M A)).
A  = listTy
```

```
M' = c nl
M  = app (c hd) (app (app (c cns) (c nl)) (c _))
```

There are many other interesting queries we can explore in this fashion. Are there terms that do not have a type? Are there terms for which evaluation does not converge to a value? This, and many others, are useful applications of the PBT approach.

As a more comprehensive validation we addressed the nine mutations proposed by the PLT Redex benchmark, to be identified as violations the preservation or progress properties.

12.5.3 Example   The first mutation in the PLT Redex benchmark introduces a bug in the typing rule for application, matching the range of the function type to the type of the argument. The change occurs in the T-APP rule in Figure 12.2:

$$\frac{\Gamma \vdash_\Sigma M_1 : A \to B \quad \Gamma \vdash_\Sigma M_2 : \boxed{B}}{\Gamma \vdash_\Sigma M_1\, M_2 : B} \; \text{T-APP-B1}$$

In the specification, the mistake translates into replacing the program clause for typing applications with the following faulty code in Figure 12.10:

```
prog (wt Ga (app X Y) T)
     (and (wt Ga X (funTy H T)) (wt Ga Y T)).
```

And, as we can verify, the mutation causes both properties to fail:

```
cexprog M A :- check (qgen (qsize 6 _)) (wt null M A),
               not (interp (progress M)).
A = intTy
M = app (c hd) (c (toInt zero))

cexpres M M' A :- check (qgen (qsize 8 _)) (wt null M A),
                  interp (step M M'),
                  not (interp (wt null M' A)).
A  = funTy listTy intTy
M' = lam (x\ c hd) listTy
M  = app (lam (x\ lam (y\ x) listTy) intTy) (c hd)
```

The table in Figure 12.12 summarizes the results. In this comparison, αCheck is set to use negation-as-failure for fairness, although this is not always the optimal choice (Cheney et al., 2016). In any case, the λProlog implementation completes all the problems in the alloted time, consistently maintaining a time advantage

bug	check	αC	λP	cex	Description/Rating
1	preservation	0.3	0.05	(λx:int. λy:ilist. x) hd	range of function in app rule
	progress	0.1	0.02	hd 0	matched to the arg (S)
2	progress	0.27	0.06	(cons 0) nil	value (cons v)v omitted (M)
3	preservation	0.04	0.01	(λx:int. cons) cons	order of types swapped
	progress	0.1	0.04	hd 0	in function pos of app (S)
4	progress	t.o.	207.3	((plus 0) ((cons 0) nil))	the type of cons returns int (S)
5	preservation	t.o.	0.67	tl ((cons 0) nil)	tail reduction returns the head (S)
6	progress	24.8	0.4	hd ((cons 0) nil)	hd reduction on part. applied cons (M)
7	progress	1.04	0.1	hd ((λx:ilist. err) nil)	no eval for argument of app (M)
8	preservation	0.02	0.01	(λx:ilist. x) nil	lookup always returns int (U)
9	preservation	0.1	0.02	(λx:ilist. cons) nil	vars do not match in lookup (S)

12.12 **Figure** Results of PBT on the *Stlc* benchmark. In order, each column represents: a *bug* number, the *property* that is the subject of the test (preservation or progress), *checking times* in αCheck and λProlog (αC and λP, respectively, with a timeout threshold of 300 seconds), one of the smallest *counterexamples* found, and a *description* of each bug along with a rating of increasing difficulty (Shallow, Medium, Unnatural).

over αCheck which only grows larger as the challenges become more complex. Bug #6 requires exploring the state space up to a depth of 11, the largest of the problem suite. Under this discipline, it is trivial to adapt the pairing FPC of Section 5.2 to the hosted kernel and seamlessly combine both families of bounds. It is also interesting to note that, for this particular set of problems, Teyjus and ELPI perform indistinguishably from one another—in contrast to their showing in Section 8.5, where ELPI consistently beat Teyjus.

The developments in this section can be run directly in λProlog or inside Abella by loading the λProlog kernel, FPC definition and programs by means of the `Specification` command (see Section 10.2).

12.6 Native PBT in Abella

The second of the two extension methods discussed in Section 12.2 encodes relational specifications directly as fixed points. These fixed point encodings can be written directly in a logic like *μLJF*—as implemented in Section 10.3, directly embedded in Abella. However, the deep connection between Abella and logic (itself compatible to a large extent with *μLJF*; see also Section 10.2) does imply that relational specifications can be programmed directly as Abella definitions and transparently reflected into the logic (this aspect is studied in Section 13.4).

In order to accommodate rich specifications involving nominal quantification, the rich kernel described in Section 10.3, in particular Figure 10.8, must be used. From the point of view of the user, specifications are written in pure Abella and transparently reflected into the kernel; working directly with Abella has the added advantage of a richer type system for user terms than the kernel implements—and the disadvantage of working with closed inductive definitions, less modular than the λProlog code of the previous section (so an external element of composition would be needed). For example, compare Figure 12.13 with its isomorph in Figure 12.10, above. Other specification code obeys the same shallow syntactic translation rules and therefore requires no particular attention. To make the code in the figure embeddable, it suffices that contexts and bindings be implemented in terms of user-defined types instead of the predefined types `o` and `olist`.

12.6.1 Example   In the correct implementation of the semantics of *Stlc* of Figure 12.13, there will be a (relatively involved) proof of the progress theorem:

Theorem prog : forall E T, wt nil E T -> progress M.

```
Define is_value : exp -> prop by
  is_value (c N) ;
  is_value (lam M T) ;
  is_value (app (c cons) V) := is_value(V) ;
  is_value (app (app (c cons) V1) V2) :=
    is_value(V1) /\ is_value(V2).

Define is_err : exp -> prop by
  is_err error ;
  is_err (app(c hd) (c nl)) ;
  is_err (app(c tl) (c nl)) ;
  is_err (app E1 E2) := is_err(E1) ;
  is_err (app V1 E2) := is_value(V1) /\ is_err(E2).

Define step : exp -> exp -> prop by
  step (app (c hd) (app (app (c cons) X) XS)) X :=
    is_value X /\ is_value XS ;
  step (app (c tl) (app (app (c cons) X) XS)) XS :=
    is_value X /\ is_value XS ;
  step (app (lam M T) V) (M V) := is_value V ;
  step (app M1 M2) (app M1' M2) := step M1 M1' ;
  step (app V M2) (app V M2') := is_value V /\ step M2 M2'.

Define progress : exp -> prop by
  progress V := is_value V ;
  progress E := is_err E ;
  progress M := exists N, step M N.

Define memb : o -> olist -> prop by
  memb X (X :: Gamma) ; memb X (Y :: Gamma) := memb X Gamma.

Define tcc : cnt -> ty -> prop by
  tcc (toInt N) intTy;
  tcc nl listTy ;
  tcc hd (funTy listTy intTy) ;
  tcc tl (funTy listTy listTy) ;
  tcc cons (funTy intTy (funTy listTy listTy)).

Define wt : olist -> exp -> ty -> prop by
  wt Ga M A := memb (bind M A) Ga ;
  wt Ga error T ;
  wt Ga (c M) T := tcc M T ;
  wt Ga (app X Y) T :=
    exists H, wt Ga X (funTy H T) /\ wt Ga Y H ;
  wt Ga (lam F Tx) (funTy Tx T) :=
    nabla x, wt (bind x Tx :: Ga) (F x) T.
```

12.13 **Figure** Dynamic semantics of the *Stlc* language, implemented in Abella.

Suppose now the typing relation `wt` is replaced with the version that contains bug #1 of the PLT Redex benchmark, as shown in Figure 12.14. In this case, `prog` is not a theorem, i.e., it can be falsified. One way to show this is to write the corresponding counterexample lemma, `cexprog`, and prove it. This could be done in Abella by coming up with witnesses manually, using them to instantiate the existentials and completing the proof. More interestingly, we can use PBT with a counterexample outline to `certify` the counterexample lemma—simply by adding generators to the statement of the manually proved lemma. Even more, by adding the typing judgments to the original `prog` non-theorem, we can `falsify` it by proving the associated counterexample lemma with the same certificate, i.e., a disproof outline.

As opposed to Example 12.2.1, which pairs a correct implementation with an incorrect specification, this example combines a correct specification (i.e., property) with a buggy implementation. Noting that given a ground term `E` its type `T` is determined by the typing relation `wt`, it is possible to balance the generation phase by producing only an independent set of inputs from which others (in this case the type) are derived by computation.

The resulting system is functionally equivalent to the hosted λProlog kernel of the previous section. However, the architecture in its current form it is not competitive with λProlog in terms of performance. This is principally due to the runtime behavior of the pure embedded Abella kernel. Executing the kernel as part of an Abella proof involves large numbers of redundant clause lookups. What is worse, pattern matching the arguments of a call to the main relations (`async`, etc.) with the various clauses representing inference rules creates multiple, also redundant, unification problems; even though each of these effectively matches formal parameters and actual arguments at the top, smaller, trivial subproblems are generated and vacuously solved after this step. If we use the kernel with nabla, the unification problems are slightly more complex, but even a simpler implementation of the unification algorithm—such as is described in Section 13.2—has a negligible effect (a small penalty factor) compared to the grind imposed by Abella's search—for a discussion on these issues, see also Section 13.1. Nonetheless, as we note, this organization of the system yields equivalent results, even as significant redesigns are needed to make it usable in practice.

```
Define wt : olist -> exp -> ty -> prop by
  wt Ga M A := memb (bind M A) Ga ;
  wt Ga error T ;
  wt Ga (c M) T := tcc M T ;
% wt Ga (app X Y) T :=
%   exists H, wt Ga X (funTy H T) /\ wt Ga Y H ;
  wt E (app M N) U :=
    exists T, wt E M (funTy T U) /\ wt E N U ;
  wt Ga (lam F Tx) (funTy Tx T) :=
    nabla x, wt (bind x Tx :: Ga) (F x) T.

Theorem cexprog : exists E T,
  wt nil E T /\ (progress E -> false).
skip.

Theorem cexprog : exists E T,
  gen_exp E /\ gen_ty T /\ wt nil E T /\
  (progress E -> false).
certify (qstart (qgen (qheight 5)) qsearch).

Theorem prog : forall E T,
  gen_exp E -> gen_ty T -> wt nil E T -> progress M.
falsify (qstart (qgen (qheight 5)) qsearch).
```

12.14 **Figure** Buggy implementation of the wt typing relation, replacing that given in Figure 12.13 according to bug #1 of the PLT Redex benchmark. In this version of the semantics, counterexample lemmas are provable and their associated non-theorems are falsifiable by the application of PBT techniques.

12.7 Notes

The present proof theoretical treatment of property-based testing was first presented in Blanco et al. (2017b).

Property-based testing validates code against an executable specification by the automatic generation test data, typically following random or exhaustive regimes, or a combination of both. It was originally conceived for its use in testing applied to programming languages (Claessen and Hughes, 2000) and has since to most major proof assistants (Blanchette et al., 2011; Paraskevopoulou et al., 2015) to complement theorem proving with a preliminary phase of conjecture testing. For a comprehensive review, refer, e.g., to Cheney and Momigliano (2017). In the more specific arena of model checking applied to metatheoretical pursuits, one of the most representative initiatives is PLT Redex (Felleisen et al., 2009), an executable domain-specific language for mechanizing semantic models built on top of the Scheme dialect DrRacket. It supports QuickCheck-style random testing and its usefulness has been demonstrated in several impressive case studies (Klein et al., 2012).

However, PLT Redex offers limited support for relational specifications and none whatsoever for binding signatures. αCheck's role is to address those deficiencies (Cheney and Momigliano, 2017). On top of the logic programming language αProlog, the tool adds a checker for relational in the same vein as has been done in this chapter. One of several possible implementation techniques is based as well on NAF—as far as testing of the conclusion is concerned. The generation phase is instead "wired in" via iterative-deepening search based on the height of derivations; in this regard αCheck is less flexible than our FPC-based architecture (and can be interpreted as offering a fixed choice of clerks and experts). Finally, more distant cousins in the logic programming world are *declarative debugging* (Naish, 1997) and the Logic-Based Model Checking project at Stony Brook (LMP).

As we have noted, the implementation of random PBT is not directly supported by Abella due to a lack of I/O functionality while in proof mode. Clerks and experts can be programmed and run in Bedwyr, as shown above, or equivalently in λProlog—note, however, that the I/O capabilities of λProlog are not available in the integrated interpreter that Abella implements, but ELPI integration is an appealing possibility. Neither λProlog nor Bedwyr offer a stable system interface nor a random number generator, so that a source of randomness must be obtained by a relatively rudimentary interface with the runtime system,

e.g., via file descriptors or coprocesses. A possible "pure" workaround involves parameterizing the random disproof outlines by a list of random choices which would be obtained by some external means and passed as an argument. This alternative can be executed in pure Abella and in its embedded λProlog, but must be carefully threaded through clerks and experts so that different parts of the sequence are passed to different branches of a proof.

Interesting examples of the metatheory of programming languages require the addition of nabla to the logic, presented in Section 9.4, but our problem domain also involves simplifications to the logic from features that are not required. In particular, the initial rules are absent and only equality is used to close proof branches; all atoms are defined as fixed points. If our model is limited to PBT, only a subset of the connectives is used for this purpose, and structural rules are severely restricted (i.e., no stores or decides). Furthermore, polarization is fixed. All these facts can be combined used to provide a simplified kernel.

As observed in Example 12.6.1, the dependencies between the various generators and computational predicates paves the way to a certain amount of shifting obligations between the generation and the testing phase. This is particularly important for performance, because generating independent data and filtering only those sets that satisfy certain judgments (e.g., generating terms and types independently and accepting only pairs such that the term has the selected type) is potentially very inefficient.

In the chapter, we have considered generators that are complete for given inductive types, but this is not a strict requirement. It is possible and often interesting to define generators for specific subtypes such as, say, "small" integers. Independent notions such as exhaustive or random generation, combined with custom generators, can be flexibly combined into composite certificate definitions simply by the usual mechanism of module accumulation in logic programming.

More in general, we may want to use a certificate not simply to witness a counterexample, but to point to the specific point where a proof fails, therefore avoiding the inspection of all the paths that (unhelpfully for the counterexample) succeed. The proof process can be seen as a dialogue between the doomed derivation of the impostor theorem and the proof of its negation—the counterexample lemma. Potentially, this could assist in repairing the specification or the property, and therefore the proof. The notion of productive use of failure Ireland and Bundy (1996) may serve as inspiration. For example, in Example 12.6.1, we want to skip directly to the case of a list with two or more distinct elements, avoiding

empty and singleton lists, which offer no useful information. This would suggest modifying the generator of lists to account for these minimum size requirements.

Another natural area of interest is in applications of model checking, for instance applied to graphs and properties like reachability in degenerate cases. This could be useful to find bad behaviors in program analysis. A proof theoretical presentation of this area has been initiated by Heath and Miller (2015, 2017).

13 Certificate integration in a proof assistant

13.1 Abella architecture

The Abella proof assistant is an OCaml program implemented as a collection of modules. Some of those modules define a public interface and encapsulate critical data structures, whereas others have a less obtuse structure. Nevertheless, the overall architecture is clearly laid out and welcomes modular extensions of the sort required to bring the developments of this part to executable form. The issues and technical considerations involved in this process—on which the systems described throughout Part III depend—are outlined in this chapter.

In a preliminary overview, we introduce the primary components where development is concentrated. At the heart of Abella—and central also for our purposes—is the `Term` module that represents higher-order λ-terms and models the *suspension calculus* on which their manipulation is based (Nadathur, 2002). The interface type of terms is reproduced in Figure 13.1. Interestingly, the variables of Abella terms are simply memory cells, and their comparison is given by pointer equality: their attributes do not factor into such operations as substitution, and seemingly identical but separate variables can be created—that is, a variable is created only once. A problematic point is that this representation decision is neither hidden by the semantics of the module interface nor even by its publicly declared definition. Over the course of our developments, we have encountered and corrected several bugs and misbehaviors.

Also related to term manipulation, but less critical for our purposes, are the modules `Metaterm` of meta-level terms, i.e., Abella's propositions, and `Typing`, where the still-untyped terms read in interactive use are defined. Other basic modules that we will manipulate include `Prover`, which acts as the main interface

257

```
(* Terms. The use of references allow in-place normalization
 * but is completely hidden by the interface. *)

type ptr
type tyctx = (id * ty) list

type term = private
  | Var of var
  | DB of int
  | Lam of tyctx * term
  | App of term * term list
  | Susp of term * int * int * env
  | Ptr of ptr
    (* Sorry about this one, hiding it is costly.. *)
and envitem = Dum of int | Binding of term * int
and env = envitem list
```

13.1 **Figure**   The Abella term structure interface exposes the standard constructors
for constants, bound variables, abstractions and applications, as well as suspended
terms and pointers used for side effects. The representation relies on helper
functions to inspect and manipulate terms, but these invariants are not strictly
enforced by the `Term` module.

to the core of the prover, and `Tactics`, where the language of tactics is defined.
Those two modules will be our gateway to building the FPC framework inside
the very nucleus of Abella. The last module of special interest is `Unify`, where a
heavily customized implementation of unification on Abella terms—with substi-
tutions as side effects on the imperative-style terms—is provided (Nadathur and
Linnell, 2005). This unification module, critical Abella's execution of proofs, will
be less amenable to the kinds of direct manipulation we have in store. (However,
this impediment will not be as pervasive as the leaky representation of terms.)

   We must note that Abella's computational engine is not particularly efficient.
When used to execute an embedded checker, it generates large numbers of redun-
dant unification problems. At its core, a checker performs proof reconstruction
by means of an implementation of an augmented sequent calculus. From the entry
point, an inductively defined proof object is constructed by mutually recursive
calls from the bottom up. A successful recursive call represents the application
of an inference rule of the proof system on the conclusion and the generation of
the premises. The relations and their clauses correspond to the inference rules
organized by the kind of sequent of their conclusion: in μLJF^a, these are `async`,

`syncL` and `syncR`. Calling the checker, say, on an unfocused sequent collects all clauses operating on unfocused sequents and pattern matches the arguments via unification on all of them. Since the kernel has few functions with "many" clauses, this leads to much repeated and wasted work. Furthermore, because most pattern matching consists in copying inputs around or decomposing at the top level of formulas only, the fairly costly unification algorithms spend their substantial runtime mostly on trivial identities.

The rest of the chapter is organized as follows. Section 13.2 presents an extended, hierarchical unification model for Abella which allows more flexible use of higher-order unification and the effective embedding of kernels with nabla for use with the FPC framework. Section 13.3 begins to develop the integration of the FPC framework in Abella proper, introducing the FPC-based tactics used in previous chapters as a complement, or replacement, of the standard tactics of Abella—the sole limitation being that our tactics are limited to accepting certificate terms representing complete proofs in current versions of Abella instead of being used to build a proof interactively. Section 13.4 completes the proof theoretical integration of the framework by connecting the logic of Abella with the logic implemented by the embedded kernel. Section 13.5 considers the uniform organization of FPC definitions in λProlog and their modular composition extended, as well, to Abella. Section 13.6 concludes the chapter.

13.2 Extended unification

Abella, like Bedwyr (with some limitations) and all modern implementations of λProlog, implements *(higher-order) pattern unification*, a decidable fragment of full higher-order unification (Huet, 1975). In general, the extension of first-order unification (Robinson, 1965; Martelli and Montanari, 1982) to higher-order terms preserves none of the desirable qualities of the former: in particular, higher-order unification is *undecidable* (more precisely, semidecidable, so there are no guarantees of termination: we cannot know if a solution exists); it is *non-determinate* (so, if there may be arbitrarily many incomparable solutions which cannot be expressed as a single, *most general unifier*, or MGU); and it is *typed* (in the sense that term-level typing information plays an important role in the search for solutions). In contrast, unification problems on higher-order terms satisfying the *pattern restriction* preserve the good traits of first-order unification (decidability, determinacy and type-freeness), being the weakest extension from

first-order to the higher-order setting in which the usual rules of $\alpha\beta\eta$-conversion hold (Miller, 1991a).

Typically, pattern unification is applied *dynamically*. That is, in solving a higher-order unification problem, we assume that the problem is in the pattern fragment; if at some point we find an equation which fails to satisfy the pattern restriction, it is set aside until—by the effect of substitutions dictated by other equations in the problem—it becomes a pattern equation. Although pattern unification is considered "almost complete in practice" with problems outside the fragment occurring very rarely, there are at least two reasons to look beyond the status quo. First, a controlled, interactive application of full higher-order unification inside a proof assistant can avoid its pitfalls while allowing a user to guide the solution of complex problems. Second, there are significant use cases which have simple, well-behaved solutions in spite of falling outside the pattern fragment in non-fundamental ways—the enriched kernel needed to represent nabla at the object level in Section 12.6 is one such case.

The second case will be addressed by implementing a recent technique proposed by Libal and Miller (2016) called *functions-as-constructors* (FC) unification, or FCU. In pattern unification, applications with existential variables at the head require that all arguments be distinct variables, universally quantified within the scope of the head. FCU extends patterns (as well as some prior proposed generalizations) by observing that the requirement that bound variables be used as arguments can be extended to *term constructors*, which very often are acting as *functions* and do not alter the desirable properties of unification patterns—the traditional head and cons operations on lists are examples of this. The conditions of operation of the extended algorithm must be given first.

13.2.1 Definition   In a formula, an occurrence of a bound variable is *essentially universal* if it is bound by a positive occurrence of the universal quantifier, a negative occurrence of the existential quantifier or a term-level abstraction. All other bound variables, bound by a positive existential quantifier or a negative universal quantifier, are said to be *essentially existential*. Essentially universally quantified variables can be instantiated only with eigenvariables, and essentially existentially quantified variables can be instantiated with general terms (in logic programming, this involves logic variables and unification).

In the context of a unification problem, an occurrence of term is *rigid* if its head is a (term-level) bound variable or an eigenvariable; it is *flexible* if its head is

a logic variable. A unification equation can be classified according to the rigidity of its terms as rigid-rigid, rigid-flexible, flexible-rigid, or flexible-flexible.

Here we shallow follow Libal and Miller (2016) for general notation and naming conventions. Given a signature of non-logical constants C and a signature of essentially universally quantified variables Σ, a *restricted term* in an equation e is defined as follows, where $BV(e)$ is the set of bound variables of e:

1. A variable in Σ or $BV(e)$ is a restricted term.

2. A (non-vacuous) application $(f\,t_1 \cdots t_n)$ is a restricted term if f is either in C, Σ, or $BV(e)$, and $t_1, \ldots, t_n$ are all restricted terms.

A system of equations satisfies the *FCU property* iff it satisfies the following three restrictions:

1. *Argument restriction:* for all (non-vacuous) applications $(X\,t_1 \cdots t_n)$ where X is an essentially existentially quantified variable, $t_1, \ldots, t_n$ are all restricted terms.

2. *Local restriction:* for all (non-vacuous) applications $(X\,t_1 \cdots t_n)$ where X is an essentially existentially quantified variable, no argument t_i is a subterm of a different argument t_j.

3. *Global restriction:* for all pairs of (non-vacuous) applications $(X\,t_1 \cdots t_n)$ and $(Y\,s_1 \ldots s_m)$ where both X and Y are essentially existentially quantified variables, no argument t_i of the first application is a strict subterm of an argument s_j of the second application.

13.2.2 Example   By inclusion, all unification problems in the pattern fragment are also in FC. Given $C = \{cons, nil, fst, rst\}$ and $\Sigma = \{l, z\}$ at appropriate types:

1. *cons* $(X\,(fst\,l))\,(rst\,l) = rst\,(Y\,(fst\,l)\,(fst\,(rst\,l)))$ is in FC.

2. $X\,(cons\,z\,nil) = rst\,l$ breaks the argument restriction.

3. $X\,(fst\,l)\,l = cons\,z\,l$ breaks the local restriction.

4. $X\,(fst\,l) = rst\,(Y\,(cons\,(fst\,l)\,(rst\,l)))$ breaks the global restriction.

These examples were originally given by Libal and Miller (2016). The explicit encoding of sequents explored by McDowell and Miller (2002, Section 4) and applied to the construction of proof checking kernels with nominal quantification

in Section 10.3 and Figure 10.8 naturally result in unification problems involving the *fst* and *rst* constructors: these problems are conceptually simple, but strictly fall outside the pattern fragment, but are covered by FC unification.

The restrictions of the FCU property are sufficient conditions to maintain determinacy. A series of results in Libal and Miller (2016), together with the algorithm given there, show that unification problems satisfying the FCU property are decidable, determinate and (essentially) type-free—since, although the algorithm works in a typed setting, it is the presence of a constructor or a bound variable that guides application of the algorithm (as part of the restricted terms of Definition 13.2.1), not the types of those constructors and variables. The original presentation of the FCU algorithm follows closely that of pattern unification in Miller (1991a), with minor changes in some of the rules. Like pattern unification, it uses a *pruning substitution* (also slightly modified from its original form) that assists in optimizing the term-traversal computations involved in the performance of the occurs-check—an operation called *variable elimination*. The pruning operation is applied to exhaustion before the applicable steps of the main algorithm; the paper makes no attempt at organizing the sequence of applications of pruning to avoid inefficiencies.

In order to open up unification—both by allowing controlled use of full higher-order unification and by treating additional classes of solvable problems without user intervention—we integrate a *hierarchy* of unification algorithms in the Abella proof assistant. The first addition is a purely functional implementation of higher-order unification as presented by Huet (1975), assuming the η-rule—as implied by the axiom of functional extensionality. The algorithm operates by building a *matching tree* where nodes correspond to unification problems—with the original problem at the root—and edges correspond to substitutions; leaves can be *failure nodes* indicating a failed solution attempt or *success nodes*, each indicating a solution to the original problem as the sequence of substitutions from root to success node. Solutions are *preunifiers* since outstanding flexible-flexible equations are underconstrained and left untreated by the algorithm.

To ensure tractability, some technical changes are needed. First, termination is imposed by computing matching trees down to a certain, finite depth. Information is preserved by introducing a third kind of pseudo-leaf, the *suspend node*, from which tree construction can resume incrementally. Second, the pointer-based, shared term representation (see Figure 13.1) is ill-suited to the construction of matching trees, where several potential substitutions for the same variable need

to be considered simultaneously. Instead of relying on Abella's side-effecting implementation (which cannot support multiple competing substitutions for a given variable) a purely functional implementation of term operations like normalization and substitution—in which new copy-terms are returned and variable equality is based on attributes and not on pointer equality—is provided.

The development is encapsulated in a new module that follows the general structure of the original `Unify` module—Abella's implementation of pattern unification and its interface with the rest of the system. Like that module, it parameterizes essentially existential and universal variables by two sub-modules: universal (resp. existential) quantification represents either: (a) essential universal (resp. existential) quantification on the right; or (b) essential existential (resp. universal) quantification on the left. Nominal abstraction is easily added and treated in both cases as constant-like. The module signature is shown in Figure 13.2.

13.2.3 Example In addition to serving as the basis for specialized unification algorithms, the `Huet` module can be more directly utilized to expose unification to the user of Abella through specialized tactics—as opposed to the common view of it as a black box that either succeeds or fails. This exposure allows one to judiciously apply the power of full higher-order unification when a problem requires it, all the while maintaining automation of a well-behaved fragment like patterns or FC.

As a particularly useful illustration, we have implemented a tactic, `match`, which in its simplest mode of operation seeks to solve the unification problem defined by the current goal. For example, given the signature $C = \{a, g\}$, a goal of the form $F\,a = g\,a\,a$ represents unification a problem with four distinct unifiers: $\{\langle F, \lambda x.g\,x\,x\rangle\}$, $\{\langle F, \lambda x.g\,a\,x\rangle\}$, $\{\langle F, \lambda x.g\,x\,a\rangle\}$, and $\{\langle F, \lambda x.g\,a\,a\rangle\}$. If we apply `match` to such a goal, we obtain a disjunction of goals generated by applying each of the unifiers to the original goal, from which we need to be able to finish the proof at hand. While the result in this example is simple, more complex behaviors arise in richer unification problems.

To manage the possible nontermination of higher-order unification, external depth bounds must be added to the tactic. At a given point, construction of the matching tree may halt and the results associated to any success nodes returned. The only risk is that proofs that rely on a solution that has not yet been found cannot be finished without further, possibly incremental exploration—until the right disjunct of the goal is found. Applying the `match` tactic to the premises is trickier, as shown by the fact that for a unification problem with one solution

```
(* Pairs of terms, esp. disagreement pairs *)
type pair = Term.term * Term.term
val pair_to_string : pair -> string

(* Information about failures *)
type huet_restriction =
  | Type_restriction of Term.term * Term.term

exception Not_huet of huet_restriction

(* Bounded matching trees *)
type mtree =
  | Success
  | Failure
  | Suspend
  | Node of pair list * (pair * mtree) list
val mtree_to_string : mtree -> string

(* Disagreement set nodes for simplification *)
type node =
  | NSuccess
  | NFailure
  | NPairs of pair list

(* Basic signature of unification modules *)
module type Unification = sig
  val is_flexible : Term.term -> bool
  val is_rigid : Term.term -> bool
  val simpl : node -> node
  val umatch : pair -> pair list
  val from_pairs : int -> pair list -> mtree
  val from_mtree : int -> mtree -> mtree
  val unifiers : mtree -> (pair list * pair list) list
end

(* Higher-order unification on the left and on the right *)
module Left : Unification
module Right : Unification
```

13.2 Figure   Huet unification module in Abella. The two sub-modules implement
a common unification interface for use on both sides of a two-sided sequent.

at a certain depth, failing to go deep enough to find that solution and simply reporting an empty (if partial) list of solutions may be confused with a success by negation-as-failure. During case analysis, Abella performs a single step of Huet's algorithm covering the simple scenario of application of such a tactic.

To complement `match`, a `simpl` tactic that simplifies and updates sequents operates as a complement to the former tactic and can be used to perform a manually controlled application of higher-order unification.

The second addition to the unification framework is a functional implementation of the FCU algorithm. A line of attack that circumvents its still relatively undeveloped algorithmic presentation harnesses the general higher-order unification algorithm (just described) by the following observation: Huet's algorithm applied to a problem which satisfies the FCU condition must terminates by the properties of the FCU fragment; moreover, if the problem has a solution, the matching tree has a single success node which represents the preunifier of the most general unifier that is the solution of the problem. The hosted version of the FCU algorithm follows this high-level outline:

1. Check the FCU property for the problem. If it fails, notify that the problem falls outside the supported fragment.

2. If the check is successful, iteratively grow the matching tree by the bounded version of Huet's algorithm until the success node is reached, and extract the associated preunifier.

3. Turn the preunifier into the unifier by generating pruning substitutions followed by the FCU substitutions for flexible-flexible pairs, repeating the substitution to exhaustion.

4. Apply the substitutions of the most general unifier thus obtained to the original problem.

Implementation is straightforward with two important technical notes. First, whereas Huet's algorithm uses η-expanded terms throughout its treatment, subsequent treatment (including by the FCU algorithm) relies on their η-contracted form, so the interface code must adjust term representation accordingly. Second, any extensions of unification based on a functional representation of terms (such as in Huet's matching trees) must translate the resulting unifier (i.e., a list of substitutions) into the corresponding side-effecting substitutions implemented

by Abella's term library, taking care not to alter the original terms and their pointers—on which the standard substitution operation of Abella relies. In the case of FC unification, which strictly extends the functionality of pattern unification, the most flexible solution is to encapsulate the single unification operation in a module, `Fcunify`, which exactly reproduces Abella's native `Unify` interface (shown in Figure 13.3) with minimal changes.

An appealing possibility is to make FCU the default unification algorithm in Abella—begin stronger than pattern unification while retaining its high-level computational properties. This poses several technical challenges. First, the figure makes clear that the interface to the unification *algorithm* depends on a particular unification algorithm, namely pattern unification. In particular, unification failures and errors are not generic enough. The negative impact of this coupling is that a full replacement of pattern unification by the more general FC unification cannot properly succeed without (fairly minor) changes in the interface. Secondly, and independently from the interface's reliance on a particular algorithm, Abella depends (implicitly) on a particular *implementation* of unification. Aspects like the generation of names of new existential variables or the order in which possible solutions to a case analysis step are generated depend implicitly on undocumented and unplanned behavior. Therefore, changes that preserve the semantics of unification can break existing proof scripts and modify Abella's interaction with the user. However, only a complete redesign at the core of the program can remove this limitation.

In spite of these considerations, both versions of the tactics—using either pattern and FC unification—can coexist gracefully. Additionally, tactics which use unification in the course of attempting to complete a full subgoal—notably, `search`—can be completely replaced by the more powerful FCU algorithm, as their succeed-or-fail behavior leaves no partial proof state that may differ from Abella's standard. This extended, FCU-based search is exactly what is needed to run the enriched kernel natively in Abella, as required in Section 12.6. There is a performance penalty arising from the added complexity and unoptimized implementation, but the replacement does not significantly affect the trends observed in that part. To our knowledge, this is the first implementation and practical use of the FCU algorithm.

```
type unify_failure =
  | OccursCheck
  | ConstClash of (term * term)
  | Generic
  | FailTrail of int * unify_failure

val explain_failure : unify_failure -> string

exception UnifyFailure of unify_failure

type unify_error =
  | NotLLambda

val explain_error : unify_error -> string

exception UnifyError of unify_error

val right_unify :
  ?used:(id * term) list -> term -> term -> unit
val left_unify :
  ?used:(id * term) list -> term -> term -> unit

val try_with_state : fail:'a -> (unit -> 'a) -> 'a

val try_right_unify :
  ?used:(id * term) list -> term -> term -> bool
val try_left_unify :
  ?used:(id * term) list -> term -> term -> bool

val try_left_unify_cpairs :
  used:(id * term) list ->
  term -> term -> (term * term) list option
val try_right_unify_cpairs :
  term -> term -> (term * term) list option

val left_flexible_heads :
  used:(id * term) list ->
  sr:Subordination.sr ->
  ((id*ty) list * term * term list) ->
  ((id*ty) list * term * term list) ->
    term list
```

13.3 Figure   Unify unification module in Abella. It implements left and right uni-
fication in several variations (standard, returning success or a list of conflict pairs).
This existing interface is adopted as a wrapper to expose our new developments
to Abella, noting that error reporting generalizes the contents of the standard
interface, which are specific to pattern unification.

13.3 Certifying tactics in Abella

In Section 4.4, we saw how a proof system like LKF^a admits a natural, direct encoding as a logic program in a logic programming language like λProlog. As fixed points and equality are added to the logic, resulting in proof systems like μLJF^a, the same techniques are applicable by selecting a logic programming language with fixed points whose semantics corresponds closely to those of the logic, such as the common core of both Bedwyr and Abella. In this latter context, a kernel is an Abella program—more precisely, a definition—operating on (polarized) formulas built on the kernel's object logic, itself defined as regular Abella data types; this approach was introduced in Section 10.3. In Chapter 11 and Chapter 12, we glossed over the applied aspects of applying certification in this environment. In fact, Chapter 11 is formulated in pure Abella and assumes that we can call a full checker by means of special tactics. Those tricks will be revealed now.

Up to this point, we have a kernel and FPC definitions written as Abella programs. Thus, the checker can exist as part of an Abella development. At this level, we can define fixed points and natively—through the checker—prove properties about formulas based on those fixed points. Nonetheless, and insofar as Abella's $\mathcal{G}$ logic (or a fragment thereof) is compatible with the intuitionistic calculus μLJF, an intriguing possibility is to use Abella to certify properties defined at the reasoning level of Abella itself. In order to have Abella's hosted μLJF^a kernel, also at the reasoning level, discharge proof obligations inside **Theorem** environments, we need to make Abella aware of the kernel and expose its functionality through new tactics, e.g., the `certify` of Chapter 11. In addition, Abella's theorems need to be reified into formulas of the kernel's object logic in order to serve as valid inputs.

First, let us account for the additions to the tactics language. The `certify` tactic is easily defined in terms of a call to `search`. We start considering the scenario that maps directly to the kernel's interface, in which a single formula wrapped in the initial sequent is checked against a certificate. That is, a certificate will be given to discharge the entire proof of a theorem, with the language of FPCs as a full replacement of Abella's language of proof scripts (for full proofs). This is achieved by the following pseudo-algorithm:

1. Reify the statement of the full theorem in Abella into a (polarized) formula of the kernel's object logic. (This process is covered after the present discussion on tactics.)

2. Given a certificate term (of type `cert`), execute the kernel's entry point `check` on this certificate and the translated formula. This execution is carried out by a call to the `search` tactic without depth search restrictions.

Under this scheme, we seek to use the sequent calculus embodied by the kernel to find a full proof of (the projection of) the original theorem. Indeed, given a certificate term and suitable FPC definitions, it suffices to allow the full checker to run and either succeed, thus proclaiming the theoremhood of the reified theorem statement—and, provided that the translation is adequate, of the original. The computational character of proof checking means that all we need to do is behind the scenes is `search` on the embedded kernel. However, to obtain efficient proof checking we need to make the tactic aware that its execution is controlled not by an explicit bound, but implicitly by the nature of the kernel. By default, the tactic employs a stateless version of iterative deepening which attempts to find proofs of growing depth $1, 2, \ldots$, while repeating all the work at lower bounds. We enrich the tactic with the ability to perform depth-first search—the standard choice for deep proofs whose backtracking points are completely determined by the kernel. (This behavior is not always what interactive users want, and can have a negative effect on the performance of Abella proof scripts where finding short proofs with less control is more important. However, DFS can be toggled in the call to the extended `search` inside the `certify` tactic wrapper.)

13.3.1 Example With this apparatus in place, sessions like the one in Figure 11.12 (prefixed by its corresponding declarations and definitions) can be executed natively in Abella. Likewise, property-based testing as described in Section 12.6 becomes directly executable (for this, see Example 13.3.4).

A first generalization of `certify` consists in employing it as a first-class tactic. Instead of being a full replacement of the legacy tactical language of Abella and using it to prove full theorems, a certificate term can be provided at any point in the proof in an attempt to find a subproof, thus discharging the current goal. The reifier must now inspect an arbitrary Abella sequent—essentially, a collection of a goal, a list of hypotheses and a list of eigenvariables—and fold back the new components in the translation. Namely, given eigenvariables $x_1, \ldots, x_m$,

hypotheses $H_1, \ldots, H_n$ and goal G, it must provide a translation of the formula:

$$\forall x_1. \cdots \forall x_m. H_1 \supset \cdots \supset H_n \supset G$$

The translation of this representation of the sequent can then be passed to the checker. Suppose `certify` takes a certificate Ξ. It is a simple matter to define a certificate constructor which reconstructs the original sequent (or rather, its reified form) in the kernel by asynchronously introducing both eigenvariables and hypotheses and then applying the supplied certificate to the exact sequent for which it was written. Such a preprocessing wrapper can be written in exact form as (`preprocess` m n Ξ). Its definition is completely straightforward.

13.3.2   Example   Recall the proof by certificate of the commutativity of addition of natural numbers developed in Example 11.7.2. A hybrid proof may perform the induction in the standard Abella vernacular, reusing the first part of the script in Figure 11.3:

```
induction on 1. intros. case H1.
```

This brings us to the first subgoal:

```
Variables: M S
IH : forall N, nat N * ->
       (forall M, nat M ->
         (forall S, plus N M S -> plus M N S))
H2 : nat M
H3 : plus z M S
============================
 plus M z S

Subgoal 2 is:
 plus M (s N1) S
```

Now, instead of resorting to standard Abella tactics, we may choose to prove the goal with a certificate. For example, from the original example, we may use:

```
certify (apply? 1 0 (idx "plus0com") search).
```

Which will allow us to progress to the second subgoal. On the other hand, a weak tactic like `certify search.` will fail to prove the goal, much like a regular `search` would. In this way we can use certificates at any point in a proof.

A second orthogonal generalization of the basic `certify` tactic is the addition of *interactivity*. Instead of presuming a certificate—a description of an

entire proof—is known in advance, it is possible to define certificates with holes in them. If those holes are undecorated logic variables, they are not in themselves useful, but adequate support from the FPC definition can be obtained. Suppose a certificate constructor, `ask`, is provided. The role of such a certificate is simply to as the client for a certificate, read it and use it to try to finish the current goal. Embedded as a continuation inside a more complex certificate, this simple extension implements the strategy to "use a certificate to continue the proof until a certain point, and when this point is reached ask for more information to proceed." In effect, in this way we reproduce in the FPC framework the interactive proving loop characteristic of proof assistants like Abella, and enables a flexible and complete revamp of its tactical language. This proving style can be implemented in Bedwyr—where I/O predicates were added precisely for this purpose—, but not yet in Abella, which currently lacks the ability for tactics to interact with the user—though, paralleling Bedwyr, they are a feasible design extension which could be used in Abella.

13.3.3 Example Still illustrating these concepts from refinements of Example 11.7.2 and its associated Figure 11.3, suppose now that we wish to perform a proof through the FPC framework that resembles the information flow of the Abella proof script—in regular practice, we commonly lack a formal proof at the outset and build one through the use of the proof assistant. The first step is clear: perform an induction and case analysis to obtain the zero and successor cases. What to do in each subgoal is not yet clear. Therefore, we may leave these unspecified by writing a full certificate which is, nonetheless, missing crucial information, though these point are explicitly annotated as follows:

```
(induction?
  (case? 0
    ask
    ask
  )
)
```

In fact, this is a more concrete representation of the incremental construction of a certificate in Example 11.7.1. Applied to `pluscom`, this certificate in fact succeeds and stops at the same sequent shown in Example 13.3.2. At this point, we can continue writing suitable certificates for each branch, such as, as noted for the zero case in the example:

```
(apply? 1 0 (idx "plus0com") ask)
```

Here, note that as a continuation certificate we decide to simply apply the lemma and then return for further instructions: this successive refinement closely parallels the common language of standard tactics. However, note that once we step into the world of FPCs we are committed to it: it is not considered at this point that one could drop back into the world of uncertified tactics.

The complement of the `certify` family of tactics is the `falsify` tactic that is necessary to directly implement the treatment of property-based testing developed in Chapter 12—certainly, counterexample lemmas can themselves be explicitly written as theorems and proved (as they were given in the examples in that chapter), but this tedium can be averted by integrating the process in the proof environment, i.e., as a dedicated tactic. Said tactic never succeeds in proving the current goal; rather, it is of interest for its "side effects," i.e., its informative output: if a counterexample is found, instantiations for the variables are shown—which is to prompt an interruption of the user's proof efforts until the problem is repaired. The most modular alternative is to define `falsify` as a wrapper around `certify` in the following manner:

1. From the current goal (or sequent), attempt to derive a counterexample lemma according to the patterns described in Section 12.2.

2. If a counterexample lemma avails, wrap it in a fresh sequent and apply to it the `certify` tactic (reification and checking). The `falsify` tactic is parameterized by a certificate term—a priori, typically a disproof outline, but this is not a strict requirement—, which is propagated to `certify`.

3. Upon success, inspect Abella's proof witness and extract the values of the prefix of existential variables, and output them to the console. Note that success in the counterexample lemma does not extend to the original goal.

13.3.4 Example   The disproof outlines in both positive and negative form (i.e., through the tactics `certify` and `falsify`) thus enabled permit the full range of experiments presented in Example 12.6.1 and Figure 12.14.

Note that, in our discussion, tactics like `certify` take a full certificate term as argument and use it to guide search for a complete proof by cleverly driving a native `search`-like tactic. These certifying tactics can be embedded in standard proof scripts. The addition of interactivity at the certificate level will eventually make it possible to dispense with legacy tactics and build proofs in Abella fully within the FPC framework.

13.4 Connection between Abella and the kernel

Second, after describing the new tactics added to Abella, we turn to the *reification* process by which Abella terms are reflected into terms of μLJF^a as implemented by the embedded kernel. The core of the logic $\mathcal{G}$ on which Abella is based is an intuitionistic and predicative subset of Church's Simple Theory of Types extended with generic judgments via the nabla quantifier (Gacek et al., 2011). To those core rules are also added rules for definitions, induction and coinduction with the standard fixed point interpretation. The resulting logic can be given a sequent calculus presentation which largely coincides with μLJ—described in Section 9.1, and from which the proof systems that are the central subject of Part III derive through focusing (μLJF) and augmentation (μLJF^a). Owing to the relatedness of both systems, moving between them is, in this instance, quite simple, as established by the next definition.

13.4.1 Definition The reification function from formulas of $\mathcal{G}$ to formulas of μLJ is defined as a function $[\![\cdot]\!]$. Connectives of each logic are distinguished by subscripts. For formulas in the core (first-order) fragment of $\mathcal{G}$, map the top-level connective to its corresponding version in μLJ, and recurse on its subformulas. For example:

$$[\![A \wedge_{\mathcal{G}} B]\!] = [\![A]\!] \wedge_{\mu LJ} [\![B]\!]$$
$$[\![\forall_{\mathcal{G}} A]\!] = \Pi x. \forall_{\mu LJ} [\![Ax]\!]$$

Here, Π denotes the meta-level quantifier; other connectives receive analogous treatment. Definitions in $\mathcal{G}$ are given by a finite set of clauses $\Pi \bar{x}. p\, \bar{x} \triangleq B\, p\, \bar{x}$. Here, p is a predicate constant that takes a number of arguments given by the length of $\bar{x}$. A predicate is defined by exactly one clause with body B—with standard restrictions to guarantee the existence of fixed points—, whose interpretation is given by the unfolding rules (analogous to those in Section 9.1). Instead of the generic $\triangleq$, we write $\stackrel{\mu}{=}$ and $\stackrel{\nu}{=}$ for least and greatest fixed point definition clauses, respectively. Thus, we have:

$$[\![\Pi \bar{x}. p\, \bar{x} \stackrel{\mu}{=} B\, p\, \bar{x}]\!] = \Pi p. \Pi \bar{x}. \mu [\![B\, p\, \bar{x}]\!]$$
$$[\![\Pi \bar{x}. p\, \bar{x} \stackrel{\nu}{=} B\, p\, \bar{x}]\!] = \Pi p. \Pi \bar{x}. \nu [\![B\, p\, \bar{x}]\!]$$

It is immediate that the operation of reification thus defined is an isomorphism across the fragments of both logics—being essentially identical—and therefore preserves provability between $\mathcal{G}$ and μLJ, as supported by Baelde and Miller (2007,

Proposition 3). In fact, the kernel developed in Section 9.3 implements the focused and augmented version of μLJ directly in terms of Abella's implementation of $\mathcal{G}$. The connection between the unpolarized logic μLJ considered now and its extension μLJF^a is given in two steps: from μLJ to μLJF by Baelde et al. (2010, Theorem 1), and from μLJF to μLJF^a by Theorem 9.3.1.

In a similar vein, the translation operation as implemented in tactics like `certify` proceeds in two steps, each requiring its own considerations. First, an Abella (i.e., $\mathcal{G}$) formula is reified into μLJ, following Definition 13.4.1. Ostensibly, no attempts are made to reify Abella sequents in mid-(co)induction: inductive reasoning in Abella is modeled by *size restriction annotations* which sustain the cyclic reasoning used by induction and coinduction tactics (Baelde et al., 2014, Section 5). Second, the reified μLJ formula moves into the world of polarized formulas common to μLJF and μLJF^a. To this end, a *polarization function* is needed. In our experience, a purely positive polarization (for least fixed points) is the most useful in practice and can be made the default, but we need to be able to specify other polarization strategies. For this, the `certify` tactic—and therefore `falsify`, which must supply adequate parameters to it—must accept a second (optional) parameter to convert from an unpolarized μLJ formula to a formula suitable for use by the kernel.

13.4.2   Example   In the *LKF* setting, a simple scheme that has been used in previous work, e.g., by Chihani et al. (2016b), parameterizes the relation that converts unpolarized client-side formulas into polarized kernel-side formulas (in negation normal form) by two logical connectives: a conjunction and a disjunction. In so doing, it allows the user to select the polarities of both connectives.

```
type   nnf   (form -> form -> form -> o) ->
             (form -> form -> form -> o) ->
             bool -> form -> o.
```

Here, `bool` is the type of unpolarized formulas and `form` is the type of polarized formulas. This simple approach works in most cases, but more sophisticated conversions are possible—of course, in an intuitionistic logic like μLJF, the polarity of disjunction is fixed. Such relations can be loaded as part of the specification, written in λProlog, and referenced by name in the syntax of certifying tactics.

After covering both reification and additions to the tactics language (in the previous section), all that remains is to connect the structure and evolution of an Abella development session to our logical framework—hence establishing the

basis for trust in the embedded certification framework. At its core, a session is a sequence of three types of commands, each with a corresponding image in the certified world of μLJF^a:

1. **Kind** and **Type** declarations of types and constructors build up the signature. All constructors are modeled on top of the *term monotype* used by the kernel, i, assisted as needed by *typing judgments*—as introduced in Section 11.2, and by means of which ambiguities are resolved.

2. **Define** and **CoDefine** definitions give names to (respectively, least and greatest) fixed point expressions. References to existing, stratified definitions inline their corresponding fixed points at the point where they occur in subsequent expressions. For the sake of efficiency, the certifying version if Abella maintains a *shadow table* that stores the μLJ fixed point alongside the native Abella definition, indexed by name. This mechanism is also used to define formulas and refer to them by name inside certificates—especially useful for complicated and possibly reoccurring formulas like induction invariants and cut formulas.

3. **Theorem** statements accompanied by full, succeeding proof scripts. These are trusted to be elaborated into full proofs by Abella—and now, alternatively by a combination of Abella and the embedded kernel if they rely on `certify` tactics. Previously defined theorems are admissible as lemmas under the assumption that a full, formal proof for them is available, i.e., they are available for use in a "lemma context" and need not be re-proved each time they are used. (The `skip` tactic admits an obligation without proof, unsoundly assuming the existence of an arbitrary proof; it must be disallowed in any serious development and will not be considered here.)

By composing this sequence of signature extensions, definitions, and proofs, we can obtain a pure, proof theoretical view of an Abella development as a single proof, say, expressed in μLJF, displayed in Figure 13.4. This is yet another instance of the *cut backbone* proof pattern first observed in the resolution proofs of Section 3.6. In this reading, the session can always be finished by inspecting a trivial goal, or the proof can be continued by formulating a new result and prolonging the session by a cut. On one side, the proof of the theorem is given; on the other, the theorem is added to the lemma context and can be used in subsequent "proof branches." If every theorem is proved by tactics in the

`certify` family, such a "session proof tree" is actually being built; otherwise, Abella's soundness needs to be trusted as well. A tactics language based on the FPC framework solves this problem organically; the feasibility of this approach has been explored in Section 11.7 and is hereby given technical substance and logical legitimacy.

The final question concerns the threading of seemingly isolated calls to the checker by `certify` within the larger context of the session proof tree: namely, all previously proved lemmas must be made available as *lemma decides*—motivated back in Section 11.4. The most faithful encoding extends the kernel interface with a `check_with_lemmas` call which populates the lemma map, Λ, with the available theorems. To offer finer control over lemma decides, the `certify` tactics are extended with a third optional parameter: in its basic form, a list of theorem names—a subset of the current context of proved theorems—whose use as lemmas is allowed within the proof.

13.4.3   Example   Once again in the context of Example 11.7.2, `pluscom` is preceded by lemmas `plus0com` and `plusscom`. Therefore, calls to the `certify` tactic correspond to calls to the kernel interface `check_with_lemmas` where the context of lemmas is populated by the two previous results. In this way, the backbone of theorems that compose an Abella session are connected to the formal proof represented in Figure 13.4.

It should be noted that the proof of `plusscom` has `plus0com` (as well as any previous results) available as lemma decides. In large sessions, a wealth of uninformative decision points has potentially deleterious effects on performance, which motivates the introduction of filters on the list of previous (restricted to *usable*) lemmas.

13.5   Clerks and experts as specifications

Thus far, there has been a clear separation between kernels written in pure λProlog (in Part II) and kernels written in Abella (in Part III). Noticeably, the specification of an FPC definition in λProlog is not only more compact and readable—partly, because the open-world assumptions enables us to declare only what we need and elide the rest—, but also more modular—for the same reason, we can seamlessly compose definitions, whereas in Abella all clerks and experts need to be defined, and do so at once. (Compare, for example, the (abridged) presentation of the SimpleCheck FPC in Abella in Figure 12.3 with the code

$$
\cfrac{
  \cfrac{
    \Pi_1 \atop \Gamma \Uparrow \vdash T_1 \Downarrow
  }{
    \cfrac{
      \cfrac{
        \Pi_2 \atop \Gamma, T_1 \Uparrow \vdash T_2 \Downarrow
      }{
        \cfrac{
          \cfrac{
            \Pi_n \atop \Gamma, T_1, \ldots, T_{n-1} \Uparrow \vdash T_n \Downarrow
          }{
            \cfrac{
              \cfrac{
                \cfrac{\Gamma, T_1, \ldots, T_n \Uparrow \vdash \vdash \Uparrow \mathcal{R}}{\Gamma, T_1, \ldots, T_{n-1} \Uparrow T_n \vdash \Uparrow \mathcal{R}}\,{}^{\text{decide}}_{\text{freeze}}
              }{\Gamma, T_1, \ldots, T_{n-1} \Uparrow \vdash \Uparrow \mathcal{R}}\,{}^{\text{cut}}
            }{\Gamma, T_1, \ldots, T_n \Uparrow \vdash \Uparrow \mathcal{R}}
          }{
            \cfrac{
              \cfrac{\Gamma, T_1, T_2 \Uparrow \vdash \Uparrow \mathcal{R}}{\Gamma, T_1 \Uparrow T_2 \vdash \Uparrow \mathcal{R}}\,{}^{\text{freeze}}
            }{\Gamma, T_1 \Uparrow \vdash \Uparrow \mathcal{R}}\,{}^{\text{cut}}
          }{\Gamma, T_1 \Uparrow \vdash \Uparrow \mathcal{R}}
        }{
          \cfrac{\Gamma, T_1 \Uparrow \vdash \Uparrow \mathcal{R}}{\Gamma \Uparrow T_1 \vdash \Uparrow \mathcal{R}}\,{}^{\text{freeze}}
        }{\Gamma \Uparrow \vdash \Uparrow \mathcal{R}}\,{}^{\text{cut}}
      }{\Gamma \Uparrow \vdash \Uparrow \mathcal{R}}
    }{\Gamma \Uparrow \vdash \Uparrow \mathcal{R}}
  }{\Gamma \Uparrow \vdash \Uparrow \mathcal{R}}
}{}
$$

13.4 Figure Representation of an Abella session as a focused proof. The session starts with the default signature and a goal $\mathcal{R}$ that is trivially provable, e.g., by finishing the session. In terms of proofs, a session is a sequence of theorem statements represented by cuts: on one premise, a proof Π_i for the theorem T_i must be given; on the other premise, the proved lemma is stored (by freezing) and ready to be used in subsequent proofs. This picture elides definitions which augment the effective signature and does not contemplate the introduction of the unsoundness by skipped proofs.

written for λProlog in Figure 12.8; the simplified execution model in the latter case hardly accounts for the diminished complexity.) However, for all practical purposes, the code of an FPC definition—i.e., its clerks and experts—are simple logic programs that inhabit the common fragment of λProlog and Abella, and consequently, a unified treatment can be foreseen.

In Abella, the existence of a *specification logic* in the form of a slightly trimmed down version of λProlog (for details, see Section 10.2) can be exploited to take advantage of the terseness of λProlog specifications. This change involves minor changes—or, alternatively, additions—to the kernel, where calls to clerks and experts are delegated to the specification level by the curly bracket notation. Specification-level predicates of type `o` thus replace reasoning-level predicates of type `Prop`, and a specification including those predicates must be loaded before the kernel, as usual. In particular:

1. Clerks and experts are defined as λProlog predicates in the style of Part II.

2. Clerks and experts are called from the kernel in curly brackets, thus representing their provenance at the specification level.

Otherwise, the implementation remains unchanged and as obviously correct—and amenable to formalization—as its counterpart in pure Abella. (This extension is not compatible with Bedwyr, however.)

By using specifications, λProlog-defined FPC definitions can be combined in a flexible manner, but once imported into Abella by the `Specification` command, the open world is closed and cannot be modified. Moreover, the FPC declarations become part of whatever client-defined specifications are require by the user. A further improvement from modularity comes from allowing multiple names specifications to be imported and handled separately—this is achieved by a simple technical modification. The default, nameless namespace in Abella can be preserved for backwards compatibility. With this increased modularity, the user can load several FPC specifications separately, whether they cooperate or they conflict. All that is missing is to allow the `certify` family of tactics to (optionally) specify a subset of FPC specification namespaces to select clauses only from the designated names. This provides increased control to seamlessly change the style of certification within a single development session and without risk of conflicts. The modification is made at the level of the basic `search` tactic, though it can be extended to others, as well.

To summarize, we have explored various levels of modularity and indirection and various connections between λProlog, Bedwyr and Abella and their connections to the Abella world, along with requisite background and extensions:

1. A kernel and FPC definitions can be defined in λProlog, loaded at the specification level and executed purely at that level, as was done in Section 12.5.

2. The kernel can be defined as Abella code, and FPC definitions can be given indistinctly as Abella of λProlog clauses—as discussed in this section. To prove properties written directly as Abella code, reification to the kernel logic is employed. If no λProlog code is utilized, proof obligations can be "shipped" to Bedwyr for execution as an alternative.

3. Given the contact surface between $\mathcal{G}$ and μLJF^a, a third possibility involves executing a kernel which manipulates Abella formulas almost directly, with the sole addition of polarization. The lack of embedding then implies that a simple `search` no longer succeeds, and the language of tactics must be upgraded to feature a slightly more complete, Bedwyr-like search.

13.6   Notes

The FCU algorithm is one of several computationally simple restrictions of higher-order unification (and extensions of higher-order pattern unification) than can compute MGUs for the unification problems generated during the execution of a kernel with support for nominal abstraction. Previously, Tiu (2002) proposed a limited extension to the pattern unification algorithm of Miller (1991a,b) modified to account exclusively for the constructors of the lists that compose the local contexts in their explicit representation. FCU recently received another, independent implementation by Hamana (2017).

In the event of success of the `falsify` tactic, it is desirable to obtain a printout of the witness terms that effectively falsify the target property (by proving the counterexample lemma, which sees the universal quantifiers as existentials which are instantiated by the generators). This information is easily extracted by the trace of the proof contained in Abella's witness terms, and is then trivially added to `falsify`'s wrapper around `certify`, which produces said witness from the counterexample lemma.

Afterword

"*A thesis is not finished: it is abandoned.*" On the conduct and culmination of doctoral studies a professor once offered me this curious maxim. By that jocular koan he exorcized the powerful symbolism the artifact often holds in the eyes of the student: that one should not attempt in vain to square the circle in a quest for perfect completeness. There is—there should always be—more.

Indeed, a thesis is not a living document as much as it is a travel photograph, laboriously developed at a milestone, just as we set for the next one. Ultimately, it is not an end in and of itself; rather, it is the first leg in a journey of scientific research—or is it really the first? Either way, it should not be the last.

To conclude, I reproduce a poem by Antonio Machado, Spanish exile in France. It is very well known, likely because of its simple poignancy.

> *Caminante, son tus huellas*
> *el camino, y nada más;*
> *caminante, no hay camino:*
> *se hace camino al andar.*
> *Al andar se hace camino,*
> *y al volver la vista atrás*
> *se ve la senda que nunca*
> *se ha de volver a pisar.*
> *Caminante, no hay camino,*
> *sino estelas en la mar.*

The journey continues.

Bibliography

Alan R. Anderson and Nuel D. Belnap. *Entailment: The Logic of Relevance and Necessity*. Princeton University Press, Princeton, NJ, 1975.

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992. doi: 10.1093/logcom/2.3.297.

Peter B. Andrews. Refutations by matings. *IEEE Trans. Computers*, 25(8):801–807, 1976. doi: 10.1109/TC.1976.1674698.

Peter B. Andrews. Theorem proving via general matings. *J. ACM*, 28(2):193–214, 1981. doi: 10.1145/322248.322249.

ARCADE. Arcade—automated reasoning: Challenges, applications, directions, exemplary achievements. `http://www.cs.man.ac.uk/~regerg/arcade/`, 2017.

Andrea Asperti. Proof, message and certificate. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics - Proceedings of AISC, DML, and MKM 2012*, volume 7362 of *LNCS*, pages 17–31. Springer, 2012. doi: 10.1007/978-3-642-31374-5.

Ali Assaf and Guillaume Burel. Translating HOL to Dedukti. In Cezary Kaliszyk and Andrei Paskevich, editors, *Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015*, volume 186 of *EPTCS*, pages 74–88, 2015. URL `http://dx.doi.org/10.4204/EPTCS.186`.

Ali Assaf and Raphaël Cauderlier. Mixing HOL and Coq in Dedukti (extended abstract). In Cezary Kaliszyk and Andrei Paskevich, editors, *PxTP 20215: Fourth Workshop on Proof eXchange for Theorem Proving*, volume 186 of *EPTCS*,

pages 89–96, Berlin, Germany, August 2015. URL `http://dx.doi.org/10.4204/EPTCS.186`.

Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory. Unpublished, 2016. URL `http://www.lsv.ens-cachan.fr/~dowek/Publi/expressing.pdf`.

Arnon Avron. Hypersequents, logical consequence and intermediate logics for concurrency. *Ann. Math. Artif. Intell.*, 4:225–248, 1991.

Arnon Avron. The method of hypersequents in the proof theory of propositional non-classical logics. In *Logic: from foundations to applications: European logic colloquium*, pages 1–32. Clarendon Press, 1996.

David Baelde. On the expressivity of minimal generic quantification. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, number 228 in ENTCS, pages 3–19, 2008a. doi: 10.1016/j.entcs.2008.12.113.

David Baelde. *A linear approach to the proof-theory of least and greatest fixed points.* PhD thesis, Ecole Polytechnique, December 2008b. URL `http://www.lix.polytechnique.fr/~dbaelde/thesis/`.

David Baelde. On the proof theory of regular fixed points. In Martin Giese and Arild Waller, editors, *TABLEAUX 09: Automated Reasoning with Analytic Tableaux and Related Methods*, number 5607 in LNAI, pages 93–107, 2009. URL `http://www.lix.polytechnique.fr/~dbaelde/productions/pool/baelde09tableaux.pdf`.

David Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1), April 2012. doi: 10.1145/2071368.2071370. URL `http://tocl.acm.org/accepted/427baelde.pdf`.

David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, volume 4790 of *LNCS*, pages 92–106, 2007. doi: 10.1007/978-3-540-75560-9_9. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/lpar07final.pdf`.

David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *21th Conf. on Automated Deduction (CADE)*, number 4603 in LNAI, pages 391–397, New York, 2007. Springer. doi: 10.1007/978-3-540-73595-3_28. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/cade2007.pdf`.

David Baelde, Dale Miller, and Zachary Snow. Focused inductive theorem proving. In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, number 6173 in LNCS, pages 278–292, 2010. doi: 10.1007/978-3-642-14203-1_24. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/ijcar10.pdf`.

David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2), 2014. doi: 10.6092/issn.1972-5787/4650. URL `http://jfr.unibo.it/article/download/4650/4137`.

Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *J. of Automated Reasoning*, 28(3):321–336, 2002.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004. URL `http://www.labri.fr/publications/l3a/2004/BC04`.

Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

Jasmin C. Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9, 2016.

Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: a counterexample generator for higher-order logic based on a relational model finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP*, volume 6172 of *LNCS*, pages 131–146, Edinburgh, July 2010. Springer.

Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic proof and disproof in Isabelle/HOL. In *FroCoS*, pages 12–27, 2011.

Roberto Blanco and Zakaria Chihani. An interactive assistant for the definition of proof certificates. In David Baelde, Amy Felty, Gopalan Nadathur, Vivek Nigam, and Alexis Saurin, editors, *Dale Fest: Seminar in Honor of the 60th Birthday of Dale Miller*, Paris, France, December 2016. URL `http://www.lsv.fr/~baelde/dale-fest/`.

Roberto Blanco and Zakaria Chihani. A methodology, architecture and assistant for foundational proof certificates. Submitted for publication, 2017.

Roberto Blanco and Dale Miller. Proof outlines as proof certificates: a system description. In Iliano Cervesato and Carsten Schürmann, editors, *Proceedings First International Workshop on Focusing*, volume 197 of *Electronic Proceedings in Theoretical Computer Science*, pages 7–14, Suva, Fiji, November 2015. Open Publishing Association. doi: 10.4204/EPTCS.197.2. URL `http://www.cs.cmu.edu/~wof15/`.

Roberto Blanco, Tomer Libal, and Dale Miller. Defining the meaning of TPTP formatted proofs. In Boris Konev, Stephan Schulz, and Laurent Simon, editors, *IWIL-2015. 11th International Workshop on the Implementation of Logics*, volume 40 of *EPiC Series in Computing*, pages 78–90, Suva, Fiji, 2016. EasyChair. URL `http://www.eprover.org/EVENTS/IWIL-2015.html`.

Roberto Blanco, Zakaria Chihani, and Dale Miller. Translating between implicit and explicit versions of proof. In Leonardo de Moura, editor, *CADE-26: 26th International Conference on Automated Deduction*, volume 10395 of *LNCS*, pages 255–273, Gothenburg, Sweden, August 2017a. Springer. doi: 10.1007/978-3-319-63046-5_16. URL `http://cade-26.info/`.

Roberto Blanco, Alberto Momigliano, and Dale Miller. Property-based testing via proof reconstruction: work-in-progress. In Marino Miculan and Florian Rabe, editors, *LFMTP 2017: Workshop on Logical Frameworks and Meta-Languages*, pages 17–22, Oxford, UK, September 2017b. URL `http://lfmtp.org/workshops/2017/home.shtml`.

Mathieu Boespflug. *Conception d'un noyau de vérification de preuves pour le λΠ-calcul modulo*. PhD thesis, École polytechnique, 2011.

Mathieu Boespflug and Guillaume Burel. CoqinE : Translating the calculus of inductive constructions into the λΠ-calculus modulo. In *Second International Workshop on Proof Exchange for Theorem Proving (PXTP 2012)*, 2012. URL `https://hal.archives-ouvertes.fr/hal-01126128`.

Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, *Proceedings of PxTP2012: Proof Exchange for Theorem Proving*, pages 28–43, 2012.

Sascha Böhme and Tobias Nipkow. Sledgehammer: judgement day. In *Automated Reasoning*, pages 107–121. Springer, 2010.

P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Third Annual Symposium on Software Development Environments (SDE3)*, pages 14–24, Boston, 1988.

Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *International Symposium on Theoretical Aspects of Computer Software*, pages 515–529. Springer, 1997.

Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.

Kai Brünnler. *Nested Sequents*. Habilitationsschrift, Universität Bern, 2010. URL `http://arxiv.org/abs/1004.1845v1`.

Samuel R. Buss. An introduction to proof theory. In Samuel R. Buss, editor, *Handbook of Proof Theory*, pages 1–78. Elsevier Science Publishers, Amsterdam, 1998.

CASC. The CADE ATP System Competition. `http://www.cs.miami.edu/~tptp/CASC/`, 1996.

Raphaël Cauderlier and Catherine Dubois. FoCaLiZe and Dedukti to the rescue for proof interoperability. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *ITP 2017: 8th International Conference on Interactive Theorem Proving*, volume 10499 of *LNCS*, pages 131–147, Brasília, Brazil, September 2017. Springer. URL `http://itp2017.cic.unb.br/`.

Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In G. Ausiello, J. Karhumäki, G. Mauri, and L. Ong, editors, *Fifth International Conference on Theoretical Computer Science*, volume 273 of *IFIP*, pages 383–396. Springer, September 2008a.

Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. *J. of Automated Reasoning*, 40(2-3):133–177, March 2008b. doi: 10.1007/s10817-007-9091-0.

Kaustuv Chaudhuri, Stefan Hetzl, and Dale Miller. A multi-focused proof system isomorphic to expansion proofs. *J. of Logic and Computation*, 26(2):577–603, 2016. doi: 10.1093/logcom/exu030. URL `http://hal.inria.fr/hal-00937056`.

James Cheney and Alberto Momigliano. αCheck: A mechanized metatheory model checker. *Theory and Practice of Logic Programming*, 17(3):311–352, 2017.

James Cheney, Alberto Momigliano, and Matteo Pessina. Advances in property-based testing for αProlog. In Bernhard K. Aichernig and Carlo A. Furia, editors, *Tests and Proofs - 10th International Conference, TAP 2016, Vienna, Austria, July 5-7, 2016, Proceedings*, volume 9762 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2016.

Zakaria Chihani. *Certification of First-order proofs in classical and intuitionistic logics*. PhD thesis, École polytechnique, August 2015.

Zakaria Chihani and Dale Miller. Proof certificates for equality reasoning. In Mario Benevides and René Thiemann, editors, *Post-proceedings of LSFA 2015: 10th Workshop on Logical and Semantic Frameworks, with Applications. Natal, Brazil.*, number 323 in ENTCS, 2016. doi: 10.1016/j.entcs.2016.06.007.

Zakaria Chihani, Dale Miller, and Fabien Renaud. Foundational proof certificates in first-order logic. In Maria Paola Bonacina, editor, *CADE 24: Conference on Automated Deduction 2013*, number 7898 in LNAI, pages 162–177, 2013. doi: 10.1007/978-3-642-38574-2_11.

Zakaria Chihani, Tomer Libal, and Giselle Reis. The proof certifier Checkers. In Hans De Nivelle, editor, *Proceedings of the 24th Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, number 9323 in LNCS, pages 201–210. Springer, 2015.

Zakaria Chihani, Danko Ilik, and Dale Miller. Classical polarizations yield double-negation translations. Technical report, Inria Saclay, August 2016a. URL https://hal.inria.fr/hal-01354298.

Zakaria Chihani, Dale Miller, and Fabien Renaud. A semantic framework for proof evidence. *J. of Automated Reasoning*, 2016b. doi: 10.1007/s10817-016-9380-6. URL http://dx.doi.org/10.1007/s10817-016-9380-6. Published electronically doi:10.1007/s10817-016-9380-6.

Alonzo Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 268–279. ACM, 2000.

Koen Claessen and John Hughes. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. *ACM SIGPLAN Notices*, 44(2):37–48, February 2009. URL http://doi.acm.org/10.1145/1543134.1411292.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *ACM SIGPLAN Notices*, pages 53–64, April 2011. URL http://doi.acm.org/10.1145/1988042.1988046.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999. ISBN 0-262-03270-8.

Stephen A. Cook. The complexity of theorem-proving procedures. In *Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, Ohio, 1971. ACM Press.

Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified rat verification. In Leonardo

de Moura, editor, *CADE-26: 26th International Conference on Automated Deduction*, volume 10395 of *LNCS*, pages 220–236, Gothenburg, Sweden, August 2017a. Springer.

Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In Axel Legay and Tiziana Margaria, editors, *TACAS 2017: International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 10205 in LNCS, pages 118–135, Uppsala, Sweden, April 2017b. Springer.

Marcello D'Agostino and Marco Mondadori. The taming of the cut. Classical refutations with analytic cut. *J. of Logic and Computation*, 4(3):285–319, 1994.

Martin Davis. Obvious logical inferences. In Ann Drinan, editor, *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, pages 530–531, Los Altos, CA, August 24–28 1991. William Kaufmann. ISBN 1-86576-059-4.

Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, New York, 1980.

Dedukti. The Dedukti system. `https://www.rocq.inria.fr/deducteam/Dedukti/index.html`, 2013.

DeepSpec. The science of deep specification. `https://deepspec.org/`, 2016.

Joëlle Despeyroux, Amy Felty, and Andre Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138, April 1995.

Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λProlog interpreter. In Martin Davis, Ansgar Fehnker,

Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *LNCS*, pages 460–468, 2015.

Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. of Symbolic Logic*, 57(3):795–807, September 1992.

Gabriel Ebner, Stefan Hetzl, Giselle Reis, Martin Riener, Simon Wolfsteiner, and Sebastian Zivota. System description: GAPT 2.0. In Nicola Olivetti and Ashish Tiwari, editors, *Proceedings of the 8th International Joint Conference on Automated Reasoning, IJCAR 2016*, volume 9706 of *LNCS*, pages 293–301. Springer, 2016. ISBN 978-3-319-40228-4. doi: 10.1007/978-3-319-40229-1.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

Gilda Ferreira and Paulo Oliva. On the relation between various negative translations. *Logic, Construction, Computation, Ontos-Verlag Mathematical Logic Series*, 3(23):227–258, 2012.

George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, pages 74–80, July 1997. URL http://doi.acm.org/10.1145/263244.263267.

Andrew Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, volume 5195 of *LNCS*, pages 154–161. Springer, 2008. URL http://arxiv.org/abs/0803.2305.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. Reasoning in Abella about structural operational semantics specifications. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, number 228 in ENTCS, pages 85–100, 2008a. doi: 10.1016/j.entcs.2008.12.118.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23th Symp. on*

Logic in Computer Science, pages 33–44. IEEE Computer Society Press, 2008b. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/lics08a.pdf`.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011. doi: 10.1016/j.ic.2010.09. 004.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012. doi: 10.1007/s10817-011-9218-1. URL `http://arxiv.org/abs/0911.2993`.

Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *ISAIM*, 2008. URL `http://isaim2008.unl.edu/index.php?page=proceedings`.

Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935. doi: 10.1007/BF01201353.

Ulysse Gérard and Dale Miller. Separating functional computation from relations. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*, volume 82 of *LIPIcs*, pages 23:1–23:17, 2017. doi: 10.4230/LIPIcs.CSL.2017.23.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. doi: 10.1016/0304-3975(87)90045-4.

Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE '03: 2003 Conference and Exhibition on Design, Automation and Test in Europe*, pages 886–891, Munich, Germany, March 2003. IEEE.

Georges Gonthier. A computer checked proof of the four colour theorem. Technical report, Microsoft Research Cambridge, 2005.

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

Orna Grumberg and Helmut Veith, editors. *25 years of model checking: history, achievements, perspectives*. Number 5000 in LNCS. Springer, 2008.

Alessio Guglielmi. A system of interaction and structure. *ACM Trans. on Computational Logic*, 8(1):1–64, January 2007.

Alessio Guglielmi. Deep inference. In David Delahaye and Bruno Woltzenlogel Paleo, editors, *All about Proofs, Proofs for All*, volume 55 of *Mathematical Logic and Foundations*, pages 164–172. College Publications, London, UK, January 2015. ISBN 978-1-84890-166-7.

Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the kepler conjecture, 2015. Available at `http://arxiv.org/abs/1501.02155`.

Makoto Hamana. A functional implementation of function-as-constructor higher-order unification. In Adriá Gascòn and Christopher Lynch, editors, *UNIF 2017: 31st International Workshop on Unification*, Oxford, UK, September 2017.

John Hannan. Extended natural semantics. *J. of Functional Programming*, 3(2): 123–152, April 1993.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical report, Citeseer, 1995.

John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In J. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 135–214. North Holland, 2014. doi: 10.1016/B978-0-444-51624-4.50004-6.

Quentin Heath and Dale Miller. A framework for proof certificates in finite state exploration. In Cezary Kaliszyk and Andrei Paskevich, editors, *Proceedings of the Fourth Workshop on Proof eXchange for Theorem Proving*, number 186 in Electronic Proceedings in Theoretical Computer Science, pages 11–26. Open Publishing Association, August 2015. doi: 10.4204/EPTCS.186.4. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/pxtp2015.pdf`.

Quentin Heath and Dale Miller. A proof theory for model checking: An extended abstract. In Iliano Cervesato and Maribel Fernández, editors, *Proceedings Fourth International Workshop on Linearity (LINEARITY 2016)*, volume 238 of *EPTCS*, January 2017. doi: 10.4204/EPTCS.238.1.

Marijn J. H. Heule and Armin Biere. Proofs for satisfiability problems. In David Delahaye and Bruno Woltzenlogel Paleo, editors, *All about Proofs, Proofs for All*, volume 55 of *Mathematical Logic and Foundations*, pages 1–22. College Publications, London, UK, January 2015. ISBN 978-1-84890-166-7.

Marijn J. H. Heule, Jr. Warren A. Hunt, and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *CADE 24: Conference on Automated Deduction 2013*, number 7898 in LNAI, pages 345–359, Lake Placid, NY, June 2013a. Springer.

Marijn J. H. Heule, Jr. Warren A. Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In Barbara Jobstmann and Sandip Ray, editors, *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD) 2013*, pages 181–188, Portland, OR, October 2013b. IEEE.

Marijn J. H. Heule, Martina Seidl, and Armin Biere. A unified proof system for QBF preprocessing. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *IJCAR 2014: 7th International Joint Conference on Automated Reasoning*, number 8562 in LNCS, pages 91–106, Vienna, Austria, July 2014. Springer.

Marijn J. H. Heule, Jr. Warren A. Hunt, and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In Amy P. Felty and Aart Middeldorp, editors, *CADE-25: International Conference on Automated Deduction 2015*, number 9195 in LNCS, pages 591–606, Berlin, Germany, August 2015. Springer.

Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *SAT 2016: International Conference on Theory and Applications of Satisfiability Testing*, number 9710 in LNCS, pages 228–245, Bordeaux, France, July 2016. Springer.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2006.

Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

Joe Hurd. First-order proof tactics in higher-order logic theorem provers. *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003.

Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:79–111, 1996.

Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR 2012: 6th International Joint Conference on Automated Reasoning*, number 7364 in LNCS, pages 355–370, Manchester, UK, June 2012. Springer.

Cezary Kaliszyk and Josef Urban. PRocH: Proof reconstruction for HOL light. In *24th Conf. on Automated Deduction (CADE)*, pages 267–274. Springer, 2013.

Cezary Kaliszyk and Josef Urban. Learning-assisted theorem proving with millions of lemmas. *Journal of Symbolic Computation*, 69:109–128, 2015.

Casey Klein and Robert Bruce Findler. Randomized testing in plt redex. In John Clements, editor, *Proceedings of the 10th Workshop on Scheme and Functional Programming*, pages 26–36, 2009.

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM.

Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, 2015.

Imre Lakatos. *Proofs and Refutations*. Cambridge University Press, 1976.

Peter Lammich. Efficient verified (UN)SAT certificate checking. In Leonardo de Moura, editor, *CADE-26: 26th International Conference on Automated Deduction*, volume 10395 of *LNCS*, pages 237–254, Gothenburg, Sweden, August 2017. Springer.

Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):
    107–115, 2009. doi: 10.1145/1538788.1538814.

Pierre Letouzey. Extraction in coq: an overview. In Arnold Beckmann, Costas
    Dimitracopoulos, and Benedikt Löwe, editors, *CiE 2008: 4th Conference on
    Computability in Europe*, volume 5028 of *LNCS*, pages 359–369. Springer, June
    2008.

Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic,
    and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009. doi:
    10.1016/j.tcs.2009.07.041.

Tomer Libal and Dale Miller. Functions-as-constructors higher-order unification.
    In D. Kesner and B. Pientka, editors, *1st International Conference on Formal
    Structures for Computation and Deduction (FSCD 2016)*, page 26:1–26:17, 2016.
    ISBN 978-3-9597701-0-1. doi: 10.4230/LIPIcs.FSCD.2016.0.

Tomer Libal and Marco Volpe. A general proof certification framework for
    modal logic. In David Baelde, Amy Felty, Gopalan Nadathur, Vivek Nigam,
    and Alexis Saurin, editors, *Dale Fest: Seminar in Honor of the 60th Birthday of
    Dale Miller*, Paris, France, December 2016a. URL `http://www.lsv.fr/
    ~baelde/dale-fest/`.

Tomer Libal and Marco Volpe. Certification of Prefixed Tableau Proofs for
    Modal Logic. In *the Seventh International Symposium on Games, Automata,
    Logics and Formal Verification (GandALF 2016)*, number 226 in EPTCS, pages
    257–271, Catania, Italy, September 2016b. doi: 10.4204/EPTCS.226.18. URL
    `https://hal.archives-ouvertes.fr/hal-01379625`.

LMP. The Logic-based Model Checking website. `http://www.cs.sunysb.
    edu/~lmc`, 1998.

Donald W Loveland. Mechanical theorem-proving by model elimination. *Journal
    of the ACM (JACM)*, 15(2):236–251, 1968.

Donald MacKenzie. *Mechanizing Proof*. MIT Press, 2001.

Sonia Marin, Dale Miller, and Marco Volpe. A focused framework for emulating
    modal proof systems. In Lev Beklemishev, Stéphane Demri, and András
    Máté, editors, *11th conference on "Advances in Modal Logic"*, number 11 in
    Advances in Modal Logic, pages 469–488, Budapest, Hungary, August 2016.

College Publications. URL `https://hal.archives-ouvertes.fr/hal-01379624`.

João P. Marques-Silva and Karem A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Lanuages and Systems*, 4(2):258–282, April 1982.

W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2010.

William McCune and Olga Shumsky. IVY: A preprocessor and proof checker for first-order logic. In *Computer-Aided reasoning*, pages 265–281. Springer, 2000.

Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In Glynn Winskel, editor, *12th Symp. on Logic in Computer Science*, pages 434–445, Warsaw, Poland, July 1997. IEEE Computer Society Press.

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

Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.

Dale Miller. A compact representation of proofs. *Studia Logica*, 46(4):347–370, 1987.

Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991a. MIT Press.

Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991b.

Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.

Dale Miller. Observations about using logic as a specification language. In M. Sessa, editor, *Proceedings of GULP-PRODE'95: Joint Conference on Declarative Programming*, Marina di Vietri (Salerno-Italy), September 1995.

Dale Miller. Abstract syntax for variable binders: An overview. In John Lloyd and *et al.*, editors, *CL 2000: Computational Logic*, number 1861 in LNAI, pages 239–253. Springer, 2000. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/cl2000.pdf`.

Dale Miller. Formalizing operational semantic specifications in logic. In *Proceedings of the 17th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2008)*, volume 246, pages 147–165, August 2009. doi: 10.1016/j.entcs.2009.07.020.

Dale Miller. A proposal for broad spectrum proof certificates. In J.-P. Jouannaud and Z. Shao, editors, *CPP: First International Conference on Certified Programs and Proofs*, volume 7086 of *LNCS*, pages 54–69, 2011. doi: 10.1007/978-3-642-25379-9_6. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/cpp11.pdf`.

Dale Miller. Communicating and trusting proofs: The case for broad spectrum proof certificates. In P. Schroeder-Heister, W. Hodges, G. Heinzmann, and P. E. Bour, editors, *Logic, Methodology, and Philosophy of Science. Proceedings of the Fourteenth International Congress*, pages 323–342. College Publications, 2014.

Dale Miller. Proof checking and logic programming. *Formal Aspects of Computing*, 29(3):383–399, 2017. doi: 10.1007/s00165-016-0393-z. URL `http://dx.doi.org/10.1007/s00165-016-0393-z`.

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

Dale Miller and Vivek Nigam. Incorporating tables into proofs. In J. Duparc and T. A. Henzinger, editors, *CSL 2007: Computer Science Logic*, volume 4646 of *LNCS*, pages 466–480. Springer, 2007. doi: 10.1007/978-3-540-74915-8_35.

Dale Miller and Catuscia Palamidessi. Foundational aspects of syntax. *ACM Computing Surveys*, 31, September 1999.

Dale Miller and Alwen Tiu. Encoding generic judgments. In *Proceedings of FSTTCS*, number 2556 in LNCS, pages 18–32. Springer, December 2002.

Dale Miller and Alwen Tiu. A proof theory for generic judgments: An extended abstract. In Phokion Kolaitis, editor, *18th Symp. on Logic in Computer Science*, pages 118–127. IEEE, June 2003.

Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005. doi: 10.1145/1094622.1094628. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/tocl-nabla.pdf`.

Dale Miller and Alwen Tiu. Extracting proofs from tabled proof search. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, number 8307 in LNCS, pages 194–210, Melburne, Australia, December 2013. Springer. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/cpp2013.pdf`.

Dale Miller and Marco Volpe. Focused labeled proof systems for modal logic. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, number 9450 in LNCS, pages 266–280, November 2015. doi: 10.1007/978-3-662-48899-7_19.

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

Dale A. Miller. Expansion tree proofs and their conversion to natural deduction proofs. In R. E. Shostak, editor, *Seventh Conference on Automated Deduction*, volume 170 of *LNCS*, pages 375–393, Napa CA, May 1984. Springer.

Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

Alberto Momigliano and Alwen Tiu. Induction and co-induction in sequent calculus. In Mario Coppo, Stefano Berardi, and Ferruccio Damiani, editors, *Post-proceedings of TYPES 2003*, number 3085 in LNCS, pages 293–308, January 2003.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *DAC '01: 38th Annual Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001. ACM.

Martin Mundhenk and Felix Weiß. The complexity of model checking for intuitionistic logics and their modal companions. In *4th International Workshop on Reachability Problems*, volume 6227 of *LNCS*, pages 146–160. Springer, 2010. doi: 10.1007/978-3-642-15349-5_10.

Gopalan Nadathur. The suspension notation for lambda terms and its use in metalanguage implementations. *Electr. Notes Theor. Comput. Sci.*, 67:35–48, 2002.

Gopalan Nadathur and Natalie Linnell. Practical higher-order pattern unification with on-the-fly raising. In *ICLP 2005: 21st International Logic Programming Conference*, volume 3668 of *LNCS*, pages 371–386, Sitges, Spain, October 2005. Springer.

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

Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 287–291, Trento, 1999. Springer.

Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

George C. Necula. Proof-carrying code. In *Conference Record of the 24th Symposium on Principles of Programming Languages 97*, pages 106–119, Paris, France, 1997. ACM Press.

Sara Negri and Jan von Plato. *Structural Proof Theory*. Cambridge University Press, 2001.

Duckki Oe, Andrew Reynolds, and Aaron Stump. Fast and flexible proof checking for smt. In *SMT '09: Seventh International Workshop on Satisfiability Modulo Theories*, pages 6–13, Montreal, Canada, August 2009. ACM.

Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015.

Lawrence C Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In *Theorem Proving in Higher Order Logics*, pages 232–245. Springer, 2007.

Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *4th Symp. on Logic in Computer Science*, pages 313–321, Monterey, CA, June 1989.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

Frank Pfenning and Carsten Schürmann. System description: Twelf — A metalogical framework for deductive systems. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206, Trento, 1999. Springer. doi: 10.1007/3-540-48660-7_14.

Gordon D. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

Gordon D. Plotkin. A structural approach to operational semantics. *J. of Logic and Algebraic Programming*, 60-61:17–139, 2004.

Robert Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 1998.

Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, Uppsala, 1965.

Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David Scott Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV97)*, number 1254 in LNCS, pages 143–154, 1997.

Stephen Read. *Relevant Logic*. Blackwell, Oxford, 1988.

Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002.

Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In *TPHOLs*, volume 3603, pages 294–309. Springer, 2005.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12:23–41, January 1965.

SatComp. The international SAT Competitions. `http://www.satcompetition.org/`, 2002.

Stephan Schulz. System description: E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 735–743. Springer, 2013.

Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In *Theorem Proving in Higher Order Logics: 16th International Conference, TPHOLs 2003*, volume 2758 of *LNCS*, pages 120–135. Springer, 2003. doi: 10.1007/10930755_8.

Kurt Schütte. Schlussweisen-kalküle der prädikatenlogik. *Mathematische Annalen*, 122:369–389, 1950.

Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2): 3–36, 1995.

Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge MA, 1986.

Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.

Geoff Sutcliffe. Semantic derivation verification: Techniques and implementation. *International Journal on Artificial Intelligence Tools*, 15(06):1053–1070, 2006.

Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

Geoff Sutcliffe, Jürgen Zimmer, and Stephan Schulz. TSTP data-exchange formats for automated theorem proving tools. In Weixiong Zhang and Volker Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, volume 112 of *Frontiers in Artificial Intelligence and Applications*, pages 201–216. IOS Press, 2004.

William W. Tait. Normal derivability in classical logic. In K. J. Barwise, editor, *The syntax and semantics of infinitary languages*, volume 72 of *Lecture Notes in Mathematics*, pages 204–236. Springer, 1968.

Gaisi Takeuti. *Proof Theory*. North Holland, 2nd edition, 1987.

Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004. URL `http://etda.libraries.psu.edu/theses/approved/WorldWideIndex/ETD-479/`.

Alwen Tiu. Model checking for π-calculus using proof search. In Martín Abadi and Luca de Alfaro, editors, *Proceedings of CONCUR'05*, volume 3653 of *LNCS*, pages 36–50. Springer, 2005.

Alwen Tiu. A logic for reasoning about generic judgments. In A. Momigliano and B. Pientka, editors, *Int. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06)*, volume 173 of *ENTCS*, pages 3–18, 2006.

Alwen Tiu and Alberto Momigliano. Induction and co-induction in sequent calculus. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES 2003: International Workshop on Types for Proofs and Programs 2003*, number 3085 in LNCS, pages 293–308, Torino, Italy, April 2003. Springer.

Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and co-induction. *Journal of Applied Logic*, 10(4):330–367, 2012. doi: 10.1016/j.jal.2012.07.007.

Alwen Tiu, Gopalan Nadathur, and Dale Miller. Mixing finite success and finite failure in an automated prover. In *Empirically Successful Automated Reasoning in Higher-Order Logics (ESHOL'05)*, pages 79–98, December 2005.

Alwen F. Tiu. An extension of L-lambda unification. `http://www.ntu.edu.sg/home/atiu/llambdaext.pdf`, September 2002. URL `http://www.ntu.edu.sg/home/atiu/llambdaext.pdf`.

A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 2 edition, 2000.

Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5): 297–312, 2012. doi: 10.1002/stvr.456.

Alexandre Viel and Dale Miller. Proof search when equality is a logical connective. Presented to the International Workshop on Proof-Search in Type Theories, July 2010. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/unif-equality.pdf`.

Jan von Plato. Gentzen's proof of normalization for natural deduction. *Bulletin of Symbolic Logic*, 14(2):240–257, June 2008.

Jan von Plato. *Elements of Logical Reasoning*. Cambridge University Press, 2013.

Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek, and Gopalan Nadathur. Reasoning about higher-order relational specifications. In Tom Schrijvers, editor, *Proceedings of the 15th International Symposium on Princples and Practice of Declarative Programming (PPDP)*, pages 157–168, Madrid, Spain, September 2013. doi: 10.1145/2505879.2505889. URL `http://chaudhuri.info/papers/draft13hhw.pdf`.

Nathan Wetzler, Marijn J. H. Heule, and Jr. Warren A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014. doi: 10.1007/978-3-319-09284-3_31.

Sean Wilson, Jacques Fleuriot, and Alan Smaill. Inductive proof automation for Coq. In *Second Coq Workshop*, 2010. URL `http://hal.archives-ouvertes.fr/inria-00489496/en/`.

Ping Yang, C. R. Ramakrishnan, and Scott A. Smolka. A logical encoding of the pi-calculus: model checking mobile processes using tabled resolution. *International Journal on Software Tools for Technology Transfer*, 6(1):38–66, 2004.

Title: Applications of Foundational Proof Certificates in theorem proving

Abstract: Formal trust in an abstract property, be it a mathematical result or the behavior of a computer program or a piece of hardware, is founded on the existence of a proof of its correctness. Many different kinds of proofs are written by mathematicians or generated by theorem provers, with the common problem of ascertaining whether those claimed proofs are themselves correct. The recently proposed Foundational Proof Certificate (FPC) framework harnesses advances in proof theory to define the semantics of proof formats, which can be verified by an independent and trusted proof checking kernel written in a logic programming language. This thesis extends initial results in certification of first-order proofs in several directions. It covers various essential logical axes grouped in meaningful combinations as they occur in practice: first, classical logic without fixed points and proofs generated by automated theorem provers; later, intuitionistic logic with fixed points and equality as logical connectives and proofs generated by proof assistants. The role of proof certificates is no longer limited to representing complete proofs to enable independent checking, but is extended to model proof transformations where details can be added to or subtracted from a certificate. These transformations yield operationally simpler certificates, around which increasingly trustworthy and performant proof checkers are constructed. Another new role of proof certificates is writing high-level proof outlines, which can be used to represent standard proof patterns as written by mathematicians, as well as automated techniques like property-based testing. We apply these developments to fully certify results produced by two families of standard automated theorem provers. Another application is the design of programmable proof description languages for a proof assistant.