Proof Checking and Logic Programming

Dale Miller

Inria and LIX/École Polytechnique
dale.miller@inria.fr

Abstract. In a world where trusting software systems is increasingly important, formal methods and formal proof can help provide trustable foundations. Proof checking can help to reduce the size of the *trusted base* since we do not need to trust an entire theorem prover if we can check the proofs they produce by a trusted (and smaller) checker. Many approaches to building proof checkers require embedding within them a full programming language. In most many modern proof checkers and theorem provers, that programming language is a functional programming language, often a variant of ML. In fact, parts of ML (e.g., strong typing, abstract datatypes, and higher-order programming) were designed to make ML into a trustworthy "meta-language" for checking proofs. While there is considerable overlap in the foundations of logic programming and proof checking (both benefit from unification, backtracking search, efficient term structures, etc), the discipline of logic programming has, in fact, played a minor role in the history of proof checking. I will argue that logic programming can have a major role in the future of this important topic.

1 Introduction

There are a number of theorem provers used by academics and industry and the kinds of formalisms that they take on are becoming increasingly complex and important. For example, computer systems such as Coq, HOL/Lite, and Isabelle have been used to help formally prove the four color theorem [18], the Kepler conjecture [20], and the correctness of a compiler [26] and a micro kernel [24]. For theorem provers to be part of our approach to trusting formalized mathematics and software systems, such provers must be trusted as well. However, trusting modern theorem provers is proving difficult since over time they become increasingly more complex: they evolve to allow for stronger inference rules and for the integration of specialized proving technology. Furthermore, one might not wish to require that theorem provers are formally trusted since anything that is formally validated has stopped evolving and is not generally subject to innovation and improvements.

1.1 Validate proofs, not provers

One method for addressing the correctness of theorem provers is to move from formally validating an entire theorem prover to simply validating individual

proofs that are emitted by provers. With such a move, we need to trust the proof checker instead of the entire theorem prover: presumably a checker is much simpler and does not need to evolve frequently. Some provers, like Coq, separate the activity of theorem proving from proof checking by providing its own *kernel* which checks all proposed proofs. There are, however, several reasons why it is desirable to move the proof checking operation to be *outside* a theorem prover.

- One of the philosophically motivated aspects of proofs must surely be their ability to communicate across time and space the reason to trust that a formula is indeed true [30]. Moving proof checkers outside a prover emphasizes that proofs are meant to be *communicated*, at least from prover to checker.
- When the kernel is part of a prover, there is a great tendency for the kernel to provide exactly what the prover needs: such a communication would evolve to just involve two entities (the kernel and prover) instead of the many possible other actors which might also want to check, trust, and use a proof.
- Finally, when the checker is formally separated from the prover, the structure of emitted proofs and the semantics of the kernel would become independent of the technology of the prover. Anyone could, therefore, reimplement the kernel and check the emitted proofs. Having several kernels by several different teams of implementers provides a well recognized path to having increased trust in software.

An example of an architecture for moving proof checking outside of theorem provers is currently being explored and implemented within the Dedukti system [11]. Dedukti is based the $\lambda\Pi$-modulo formal framework of Cousineau and Dowek [9] which mixes two well-known and powerful frameworks, one for hypothetical reasoning (via the dependently typed λ-calculus known as LF [21] and $\lambda\Pi$) and one for functional programming style computations (via confluent rewriting). The Dedukti project has recently developed software that allows several existing theorem provers—e.g., Coq, HOL, Matita—to output proofs into a format that Dedukti can check independently from those provers [2].

1.2 Proof checking vs proof reconstruction

What is typically called proof checking generally contains elements of *proof reconstruction*: that is, the process of checking whether or not a given document is a proof might require computing some details that are not present explicitly in the document. For example, even for rather low-level and detailed notions of proof, it is seldom the case that one would expect to have every detail present within a formal proof object. For example, in order to check that the assumptions $(p \supset p \supset q) \supset (p \supset q)$ and $(p \supset p \supset q)$ and the rule of *modus ponens*, written schematically as "from A and $A \supset B$, conclude B", infers $p \supset q$, it is not likely that one needs to provide *in the proof itself* the explicit ordering of assumption and the binding of schematic variables $[A \mapsto (p \supset p \supset q),\ B \mapsto (p \supset q)]$. Such an ordering and binding can easily be computed, leaving less to store in the proof document.

Existing theorem provers often contain much more significant notions of proof reconstruction. For example, the Boyer-Moore theorem prover attempts to fill in significant gaps between lemmas using various proof procedures parametrized by the collection of previously proved lemmas [5]. If the proof procedure is not able to fill in a gap in a sequence of lemmas, the user must design some additional lemmas to split up that gap into more manageable parts. Other theorem provers, particularly those based on the LCF framework [19], allow functional programs (usually in a variant of ML) to be executed in order to compute ways to complete the gaps between lemmas (or between hypotheses and goal). In other systems, an interactive theorem prover might call a completely automated prover in order to complete a step of inference: for example, Isabelle can call the Vampire theorem prover to close a gap in a user's proof attempt [28].

1.3 The community of logic programming

Oddly enough, the community that has invested a great deal of energy into providing effective implementations of logic—namely, the logic programming community—has not traditionally been involved with proof checking. There are at least three likely reasons for this mismatch between that community and those interested in theorem proving and proof checking.

Efficiency versus soundness In proof checking, logical soundness is *everything*: there is no reason to be doing proof checking if one is not confident that the underlying logic engine is logically sound. The logic programming community has often emphasized efficiency instead of logical soundness: for example, many Prolog systems have not supported the occurs-check in unification since that was seen as a feature only needed for toy examples [34, Section 3.3]. Experience with automated theorem proving shows, however, that soundness of inference critically depends on the presence of occurs-check in unification. Of course, a programming language like Prolog can still be used to implement proof checkers even when unification is unsound since *any programming language* can be used to build, in principle, any programming task. One would suspect, however, that logic programming languages should have a much more immediate and transparent ways to support the effective implementation of logic.

Lack of logical expressiveness Another aspect of most logic programming languages is that they do not have direct support for quantified formulas and the concomitant operations of substitution into and unification of expressions containing bindings. While individual Prolog clauses are interpreted as universally quantified, such quantification is implicit. Furthermore, no direct and logical support for bindings is available within Prolog even though the interplay between formula-level bindings (quantifiers) and term-level bindings (λ-abstractions) has been well understood since Church's Simple Theory of Types [8]. Thus, while Prolog provides many logical principles that could be used to implement proof checkers, that support does not extend to much of logic itself.

Lack of abstractions There is still at least one other reason that logic programming can be a poor match with proof checking: most Prolog systems do not support rich forms of *abstractions*, such as abstract datatypes and procedural (higher-order) abstraction. Each of these features were explicitly introduced into the first design and implementation of the functional programming language ML since they were seen as important for building the LCF proof system [19]. Abstract datatypes help provide guarantees that, for example, there are only certain ways to build objects of type `thm` (the type for theorems): once `thm` is established, it is made into an abstract datatype and the design of ML's type system enforces that theorems can only arise from the originally provided constructors and functions. Similarly, higher-order programming in ML was introduced to allow for certain "kernel" operations (the tacticals) to have their trusted code separated from the "clients" code (the tactics).

The lack of expressiveness and abstraction can be addressed within logic programming if one is willing to move beyond first-order Horn clauses for fragments of higher-order, intuitionistic logic [29]. In fact, λProlog [12, 31, 33] and Twelf [35] are two logic programming languages that treat bindings in expressions and proofs directly as part of their logical foundations. Furthermore, λProlog also exploits features of its underlying logic to provide logically sound notions of modules and abstract datatypes as well as higher-order programming (such as is found in most functional programming languages).

Where should we begin in looking for connections between the logic programming paradigm and proof checking? Remarkably, one has to look no further than the recent literature of proof theory (see references in Section 3) to find a framework where relations and not functions dominate, where bounded backtracking search has obvious and immediate applications, and where formula-level and term-level abstractions occur naturally together. In addition, the topic of proof theory presents a mathematically and not a technologically notion of proof.

2 Proof theory as a framework

If a modern theorem prover outputs a proof as a (persistent) document, that proof document is usually based on specific technology built into the prover. On some other occasions, provers output documents meant for tracing and debugging. It is the exceptional theorem prover that outputs a document that is intended to outlast the (version of the) prover itself.[1] Given the existence of the mathematical literature on *proof theory* initiated by Frege and Gentzen, proofs-as-documents can be as *eternal* as Peano numerals: 0, $s(0)$, $s(s(0))$, $s(s(s(0)))$, etc.

Once proofs are liberated from the technology that produces them, then they can be checked by independently constructed checking programs which can be written by anyone keen to develop their own trusted base of code instead of adopting someone else's code. Once checked, such proofs can be placed in

[1] Some specialized theorem provers related to SAT solving have adopted standards of outputting their proof evidence (see, for example, [36]).

libraries that survive changes in theorem proving and proof checking technologies. Thus, proofs can be used to communicate trust between different provers and across changes in technology. In order to reach such a status in the sharing and trusting of proofs, we probably need to design a framework based on a mathematically well defined and sophisticated notion of proof. Examining the literature on proof theory, however, reveals a number of formally defined proof structures. In the earliest days, Frege and Hilbert proposed rather simple, linear proof structures; Gentzen introduced both the sequent calculus as well as natural deduction; later, resolution refutations and tableaux proof systems were also introduce, in part, to support automation of theorem proving. Still more structures can be found that can be accepted as proofs, such as proof nets, matings, deep inference, and winning strategies.

In this paper, I will outline a multi-year effort that proposes to use the sequent calculus as the *assembly language* of proof and to describe how to compile many other higher-level notions of proof into that assembly language. The formal devices for making such definition of proof languages will be based on the notion of *focused proofs* for first-order classical and intuitionistic logics. We illustrate such a proof system in the next section.

3 Focused versions of sequent calculi

The sequent calculus of Gentzen provides an appealing form of formal proof structure since they can be used to describe proofs in both classical, intuitionistic, and linear logics. They also support propositional, first-order, and higher-order logics and do so in a modular and clear fashion. The cut-elimination theorem [15] also reveals that this notion of proof supports sophisticated manipulations (such as substitution and composition of proofs). On the other hand, the proofs in the sequent calculus can be chaotic: if a proof of a given sequent exists, many trivial and not-so-trivial variations of that proof also exists. All these variants work to hide structure. Relying on the small inference steps that are part of Gentzen's presentation of the sequent calculus not only makes finding sequent calculus proofs difficult, it also makes communicating them challenging. Consider the following example (taken from [4]). Attempting to prove the sequent $\Gamma \vdash \exists x \exists y[(p\ x\ y) \vee ((q\ x\ y) \vee (r\ x\ y))]$, where Γ contains, say, a hundred formulas. The search for a (cut-free) proof of this sequent can confront the need to choose from among a hundred-and-one introduction rules. If we choose the right-side introduction rule, we will then be left with, again, a hundred-and-one 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 choices.

One of the first attempts to use sequent calculus in computer science needed to develop a normal form of sequent proof that discarded a great deal of those variants. The *uniform proofs* of [29, 32]—with its notion of alternating phases of goal-reduction and backchaining—was used to provide a general framework for defining proof search in logic programming. With the advent of linear logic [16], that two phase structure was extended to all of linear logic using Andreoli's *fo-*

cused proof system [1] and the notion of *polarity* [1, 17]. Soon afterwards, various focused proof systems for intuitionistic logic [6, 13, 22, 23] and classical logic [10, 17, 25] appeared. The *LJF* and *LKF* frameworks of [27] can be seen as offering a general framework and generalization to these various classical and intuitionistic focused proof systems.

We limit our attention here to first-order classical logic. A similar development holds for intuitionistic logic as well.

Polarizing connectives The emphasis on focused proofs is an emphasis on proof structure and not provability. For example, consider the following different ways to write the introduction rules for disjunction and conjunction in a one-sided sequent system.

$$\frac{\vdash \Gamma, B_1 \quad \vdash \Gamma, B_2}{\vdash \Gamma, B_1 \wedge B_2} \quad \frac{\vdash \Gamma_1, B_1 \quad \vdash \Gamma_2, B_2}{\vdash \Gamma_1, \Gamma_2, B_1 \wedge B_2} \quad \frac{\vdash \Gamma, B_1, B_2}{\vdash \Gamma, B_1 \vee B_2} \quad \frac{\vdash \Gamma, B_i}{\vdash \Gamma, B_1 \vee B_2}\, i \in \{1, 2\}$$

Given that the structural rules of weakening and contractions are available in classical logic, the first pair of rules and the second pair of rules are inter-admissible inference rules: any sequent provable with one element of the pair is provable also with the second member of the pair. Notice also that the first member of each pair is *invertible* while the second member is *not invertible*. People presenting proof systems for classical logic or who are implementing such systems generally pick one member from each pair and that choice is usually the invertible rule. Given our interest here in proof structures (an not just provability), our eventual focused proof system will contain all four of these introduction rules. They will be distinguished from each other by having them introduce different *polarized* versions of disjunction and conjunction.

$$\frac{\vdash \Gamma, B_1 \quad \vdash \Gamma, B_2}{\vdash \Gamma, B_1 \wedge^- B_2} \quad \frac{\vdash \Gamma_1, B_1 \quad \vdash \Gamma_2, B_2}{\vdash \Gamma_1, \Gamma_2, B_1 \wedge^+ B_2} \quad \frac{\vdash \Gamma, B_1, B_2}{\vdash \Gamma, B_1 \vee^- B_2} \quad \frac{\vdash \Gamma, B_i}{\vdash \Gamma, B_1 \vee^+ B_2}\, i \in \{1, 2\}$$

The introduction rules for the negative polarized connectives ($\wedge^-$ and $\vee^-$) are invertible while the introduction rules for the positive polarized connectives ($\wedge^+$ and $\vee^+$) are not invertible. The units for these connectives are also polarized similarly: t^-, t^+, f^-, and f^+, and these have the following introduction rules.

$$\frac{}{\vdash \Gamma, t^-} \qquad \frac{}{\vdash t^+} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, f^-}$$

(There is no introduction rule for the positive false f^+.)

Some connectives have fixed polarity: universal quantification is negative and its de Morgan dual, existential quantification, is positive. Atoms can be either positive or negative: this choice can be made in an arbitrary but fixed fashion. The negated atom $\neg A$ has the opposite polarity to A. A formula has positive or negative polarity depending only on its top-level logical connective (if it has one) or on the polarity as a literal.

Grouping don't-care and don't-know non-determinism If several invertible rules can be applied to yield a given sequent then those rules can be applied in any order and, in fact, in all possible orderings to yield a proof. In order to factor away such don't-care non-determinism, we introduce the notion of the $\Uparrow$ phase in focused proof construction display the invertible rules as follows:

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

$$\frac{\vdash \Theta \Uparrow [y/x]B, \Gamma}{\vdash \Theta \Uparrow \forall x.B, \Gamma}\ \dagger \qquad \frac{\vdash \Theta, C \Uparrow \Gamma}{\vdash \Theta \Uparrow C, \Gamma}\ \mathit{store}$$

Here, sequents are of the form $\vdash \Theta \Uparrow \Gamma$, where Θ is a schematic variable ranging over *multisets* of formulas and Γ is a schematic variable ranging over *lists* of formulas. A list is used here instead of a multiset as a way to reduce the don't-care non-determinism: we only need to consider introduction rules on the first formulas of that list. In the $\forall$-introduction rule, the $\dagger$ proviso is the usual one: the variable y is not free in the lower sequent. In the store rule, C is a positive formula or negative literal: this rule is responsible for recognizing that the first formula in the right-hand context cannot be introduced by an invertible inference. In general, the context Θ contains only positive formulas and negative literals.

Another phase contains sequents of the form $\vdash \Theta \Downarrow B$ where Θ is as before and B is a formula. The introduction rules associated to this phase are written as follows.

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

Structural and Identity rules The following two "structural" rules are needed to move between these two phases.

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

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

Here, P is a positive formula; N a negative formula; P_a a positive literal; C a positive formula or negative literal; and $\neg B$ is the negation normal form of the negation of B.

Synthetic inference rules One of the purposes of introducing a focused proof system is to make the following identification: the phases introduce new, synthetic inference rules. Gentzen's introduction and structural rules form the assembly instructions of proof and the synthetic inference rules form the higher-level notions of proof. For example, assume that Θ contains the formula $a \wedge^+$

$b \wedge^+ \neg c$ and that we have a derivation that Decides on this formula.

$$\dfrac{\dfrac{}{\vdash \Theta \Downarrow a}\ \mathit{Init} \quad \dfrac{}{\vdash \Theta \Downarrow b}\ \mathit{Init} \quad \dfrac{\dfrac{\vdash \Theta, \neg c \Uparrow \cdot}{\vdash \Theta \Uparrow \neg c}\ \mathit{Store}}{\vdash \Theta \Downarrow \neg c}\ \mathit{Release}}{\dfrac{\vdash \Theta \Downarrow a \wedge^+ b \wedge^+ \neg c}{\vdash \Theta \Uparrow \cdot}\ \mathit{Decide}}\ \wedge^+$$

This derivation is possible if and only if Θ is of the form $\neg a, \neg b, \Theta'$. Thus, the "macro-rule" is

$$\dfrac{\vdash \neg a, \neg b, \neg c, \Theta' \Uparrow \cdot}{\vdash \neg a, \neg b, \Theta' \Uparrow \cdot}$$

Soundness and completeness of focusing The formulas used in *LK* are unpolarized while those in *LKF* are polarized. In order to state soundness and completeness of focusing, we must introduce the notion of polarizing a formula. Given the unpolarized formula B, let $\hat{B}$ be one of the exponentially many formulas that result by placing $+$ or $-$ on the occurrences of $\vee$ and $\wedge$ as well as attributing polarization to the atoms in B. (The quantifiers have fixed polarities: $\forall$ is negative and $\exists$ is positive.) The soundness theorem for *LKF* is immediate: Assume that $\vdash \cdot \Uparrow \hat{B}$ has an *LKF* proof. We can recover an *LK* proof by simply replacing the $\Uparrow$ and $\Downarrow$ with commas, deleting some repetitions of sequents, and dropping the $+$ and $-$ annotations on the propositional connectives. Conversely, completeness (which is proved in [27]) states that if B is a first-order theorem and $\hat{B}$ is any polarization of B then $\vdash \cdot \Uparrow \hat{B}$ is provable in *LKF*. A consequence of soundness and completeness implies that if any polarization of B is provable in *LKF* then every polarization is provable in *LKF*. Clearly, polarization is not relevant to *provability* but is relevant to the structure of proofs.

4 Foundational proof certificates

The two phases in *LKF* are strikingly different. The invertible $\Uparrow$ phase can be built from the bottom up as a purely deterministic computation: one just applies the inference rules in a straightforward fashion. On the other hand, the $\Downarrow$ phase is not straightforward since some of the inference rules need information that is lacking from the conclusion: in particular, the $\exists$ introduction rule requires a substitution term and the $\vee^+$ introduction rule requires an indicator of whether to select the left or right disjunct. If this information is lacking, then the construction of the proof can be seen as a non-deterministic computation, where choices and substitution terms are guessed.

It is now easy to see that the focused proof system in Section 3 provides a *communication protocol* between an entity that possesses some evidence that a formula is a theorem and a low-level tool attempting to build a sequent calculus proof from the bottom-up of that proposed theorem.

$$\frac{}{\Xi \vdash \Theta \Uparrow t^{-}, \Gamma} \qquad \frac{\Xi_1 \vdash \Theta \Uparrow A, \Gamma \quad \Xi_2 \vdash \Theta \Uparrow B, \Gamma \quad \wedge_c(\Xi, \Xi_1, \Xi_2)}{\Xi \vdash \Theta \Uparrow A \wedge^{-} B, \Gamma} \qquad \frac{\Xi' \vdash \Theta \Uparrow \Gamma \quad f_c(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow f^{-}, \Gamma}$$

$$\frac{\Xi' \vdash \Theta \Uparrow A, B, \Gamma \quad \vee_c(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow A \vee^{-} B, \Gamma} \qquad \frac{\Xi' \vdash \Theta \Uparrow [y/x]B, \Gamma \quad \forall_c(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow \forall x.B, \Gamma}\ \dagger$$

$$\frac{true_e(\Xi)}{\Xi \vdash \Theta \Downarrow t^{+}} \qquad \frac{\Xi_1 \vdash \Theta \Downarrow B_1 \qquad \Xi_2 \vdash \Theta \Downarrow B_2 \qquad \wedge_e(\Xi, \Xi_1, \Xi_2)}{\Xi \vdash \Theta \Downarrow B_1 \wedge^{+} B_2}$$

$$\frac{\Xi' \vdash \Theta \Downarrow B_i \qquad i \in \{1, 2\} \qquad \vee_e(\Xi, \Xi', i)}{\Xi \vdash \Theta \Downarrow B_1 \vee^{+} B_2} \qquad \frac{\Xi' \vdash \Theta \Downarrow [t/x]B \qquad \exists_e(\Xi, \Xi', t)}{\Xi \vdash \Theta \Downarrow \exists x.B}$$

$$\frac{\Xi_1 \vdash \Theta \Uparrow B \qquad \Xi_2 \vdash \Theta \Uparrow \neg B \qquad cut_e(\Xi, \Xi_1, \Xi_2, B)}{\Xi \vdash \Theta \Uparrow \cdot}\ cut$$

$$\frac{\Xi' \vdash \Theta \Uparrow N \quad release_e(\Xi, \Xi')}{\Xi \vdash \Theta \Downarrow N}\ release \qquad \frac{init_e(\Xi, l) \quad \langle l, \neg P_a \rangle \in \Theta}{\Xi \vdash \Theta \Downarrow P_a}\ init$$

$$\frac{\Xi' \vdash \Theta \Downarrow P \qquad decide_e(\Xi, \Xi', l) \qquad \langle l, P \rangle \in \Theta \qquad \text{positive}(P)}{\Xi \vdash \Theta \Uparrow \cdot}\ decide$$

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

Fig. 1. The augmented *LKF* proof system LKF^a.

One way to implement such a protocol would be to instrument the focused proof system *LKF* with certain augmentations as shown in Figure 1. The augmentation is accomplished in three simple steps: (*i*) a proof certificate term, denoted by the syntactic variable Ξ is added to every sequent; (*ii*) every inference rule of *LKF* is given an additional premise using either an *expert predicate* or a *clerk predicate*; and (*iii*) the multiset of formulas to the left of the arrows $\Uparrow$ and $\Downarrow$ is replaced with a multiset of pairs of an *index* and a formula. (Viewing this figure in color shows the augmentations in blue.) Clearly, the *LKF* proof system can be recovered from LKF^a by removing all occurrences of the syntactic variable Ξ and by removing all premises with a subscripted e or c as well as replacing all occurrences of tuples such as $\langle l, B \rangle$ with just B.

Notice that the extra premise added to invertible rules are called *clerks*: in such inference rules, only simple computations are done and, in general, no information in a certificate term needs to be consumed. On the other hand, the extra premise added to non-invertible rules are called *experts*: these predicates are responsible for examining certificates and, possibly, extracting information from them. We also allow for experts to invoke non-determinism: that is, they can guess additional information (such as substitution terms).

Depending on exactly how one defines and uses certificate terms, indexes, and the clerk and expert predicates (e.g., $\wedge_c(\cdot, \cdot, \cdot)$, $\exists_e(\cdot, \cdot, \cdot)$, etc), the LKF^a inference rules can be directed to build widely varying proof structures. Collecting

together such definitions yields what we call an *FPC*, that is, a *foundational proof certificate* definition. We shall present an example shortly.

5 Proof checking as logic programming

It is possible to see the inference rules in Figure 1 as describing a logic program: in particular, Figure 2 contains part of a λProlog specification for several of those rules. The first five lines of Figure 2 declare the types, constants, and predicates used to encode first-order polarized (*LKF*) formulas; the next seven lines declare the types and predicates of clerks and expert predicates; the next three lines provide the type declaration of the predicates that form the core of the proof checking kernel; and the remaining lines contain six clauses that are a direct specification of six inference rules from Figure 1: specifically of the inference rules for introducing $\wedge^-$, $\forall$, and $\exists B$, and the rules for store, decide, and initial. All the remaining augmented inference rules can be specified in a similar fashion.

We have used λProlog here instead of Prolog for the following two major reasons.

Bindings in formulas and proofs. λProlog encodes bindings in formulas (quantifiers) and in proofs (eigenvariables) directly and implements them into its unification and substitution mechanisms. Achieving a Prolog implementation is possible but would require implementing such binding structures and associated logical operations, all rather difficult things to get right.

Context management and its dynamics during proof search. Focused and augmented sequents contain a context denoted by the Θ: this is intended to be a multiset of pairs of indexes and formulas. This multiset only needs to support the following operations: add a pair to the multiset (the store rule) and select a formula (nondeterministically) from the multiset by providing an index (the decide and initial rules). Notice that we do not need to know how many members there are in Θ nor anything about possible orderings between pairs. Note also that no functional dependency is assumed to hold between indexes and formula: that is, many formulas may be associated to the same index within a given Θ context. For all these reasons, the hypothetical context on λProlog serves as an interesting and direct implementation of this aspect of these inference rules. (See, for example, the fourth clause in Figure 2.)

By forging such a direct link between a proof checking kernel and a logic program, that kernel has access to backtracking search and unification, which means that it can be used to support *proof reconstruction*: if a proof certificate does not contain all the details necessary to complete a (sequent calculus) proof, then it should be possible to allow backtracking and unification to discover some of them.

The full process of defining and checking a certain format of proof evidence can now be done as follows.

```
kind form, i          type.
type nand, por        form -> form -> form.
type all, some        (i -> form) -> form.
type complem          form -> form -> o.
type pos_or_lit       form -> o.

kind cert, index      type.
type andC             cert -> cert -> cert -> o.
type allC             cert -> (i -> cert)  -> o.
type storeC           cert -> cert -> index -> o.
type initE            cert ->         index -> o.
type someE            cert -> cert -> i     -> o.
type decideE          cert -> cert -> index -> o.

type uparrow          cert -> list form -> o.
type downarrow        cert  ->     form -> o.
type store            index ->     form -> o.

uparrow Cert ((nand A B)::Gamma) :- andC Cert Cert1 Cert2,
    uparrow Cert1 (A::Gamma), uparrow Cert2 (B::Gamma).
uparrow Cert ((all B)::Gamma) :- allC Cert Cert',
    pi x\ uparrow (Cert' x) ((B x)::Gamma).
downarrow Cert (some B) :- someE Cert Cert' T,
    downarrow Cert' (B T).
uparrow Cert (C::Gamma) :- pos_or_lit C,
    storeC Cert Cert' Idx, store Idx C => uparrow Cert Gamma.
uparrow Cert nil :- decideE Cert Cert' Index, store Index B,
    downarrow Cert' B.
downarrow Cert B :- initE Cert Idx, store Idx C, complem B C.
```

Fig. 2. A λProlog implementation of part of Figure 1

1. Pick some discipline to polarize a given classical logic formula into a polarized (*LKF*) formula.
2. Provide the signature (term constructors) for certificate (terms of type `cert`) and indexes (terms of type `index`). Any term structure possible in λProlog are allowed for these structures.
3. Provide logic program specifications of the clerk and expert relations.
4. Prove the goal `uparrow Cert (B::nil)` where `B` is the polarized form of the proposed theorem and `Cert` is the supplied certificate term (proof evidence).

6 Non-determinism in proof checking

In general, clerk predicates are intended to be functional: in the case of the conjunctive-clerk, this means that for every Ξ there exists at most one Ξ_1 and at most one Ξ_2 such that $\wedge_c(\Xi, \Xi_1, \Xi_2)$ is provable.

One could insist that this is also the case with experts. For example, if the $\exists$ expert predicate is functional then for every Ξ for which $\exists_e(\Xi, \Xi', t)$ is provable, the continuation certificate Ξ' and the substitution term t are uniquely determined. This can be achieved if, for example, the certificate simply stores this substitution term inside itself. While such proof certificates are clearly possible to design and check, one might want some flexibility in the design of certificates: for example, storing all substitution instances might require certificates to be huge. If such substitution terms could be inferred from context using, say, the unification mechanism of logic programming, then the size of a proof certificate might be much smaller. Thus, allowing the proof checker to also reconstruct details of a proof allows proof certificates to be possibly much smaller in size.

For a specific example of the trade-off between proof size and proof checking, consider the following example (taken from [7]). It is possible to convert some decision procedures into proof certificates. For example, consider the procedure for determining whether or not a given propositional formula is a tautology: first compute the conjunctive normal form of the formula and then check that all the resulting clauses contain complementary literals. It is an easy matter to define a proof certificate encapsulating this procedure. Following the four steps mentioned above, we choose to polarize the connectives in a propositional classical formula using the negative (invertible) connectives. We then arrive at the following code (which specifies the result of the second and third steps).

```
type lit       index.
type cnf        cert.

andC      cnf cnf cnf  &  initE          cnf lit.
orC           cnf cnf  &  decideE    cnf cnf lit.
falseC        cnf cnf  &  storeC     cnf cnf lit.
releaseE      cnf cnf.
```

Here, there is exactly one proof certificate—just the token `cnf`. Similarly, there is just one index—the token `lit`—which is used to index all stored formulas. Note that the only formulas stored in this way are (both positive and negative) literals. Thus, the association between indexes and formulas is not functional and, as a result, the decide rule will be asked to chose some formula with index `lit` for which a complement is found. Such a step works perfectly in a logic programming setting where the decide rule (on index `lit`) is immediately followed by the initial rule (on index `lit`): thus the decide rule will generate positive literals and the initial rule will test those against available negative literals. Notice that if we required the indexing mechanism as well as the decide and initial experts to be functional, we would need to insert into the certificate a great deal of indexing information: since there can be exponentially many clauses in the conjunctive normal form of a propositional formula, such certificates would be huge, in contrast to the description of the `cnf` certificate that is described here.

7 Conclusion

Proof checking has been part of the history and development of high-level languages, starting, in particular, with the LCF system and the ML meta-language for it [19]. I have argued here that proof checking can be closely linked to logic programming. Such a close linkage should benefit both communities. There are, however, some challenges ahead for such a close relationship to actually occur.

One such challenge is that implementers and designers of logic programming languages have often favored efficiency and expressiveness over logical soundness: witness the presence of unification without the occurs-check, negation-as-failure, assert/retract, etc. Proof checking is a setting where logical soundness is paramount. While soundness can be delivered using unsound, quasi-logical processing, the logic programming community should certainly be able to deliver much more interesting and ambitious approaches to the implementation of logical deduction.

A second such challenge would be to have powerful techniques for reasoning about logic programming specifications. It is possible to view provability from Horn clauses as a simple inductive definition (a feature that a number of theorem provers support), but more direct support seems desirable since provability has more properties than just any inductive definition. In the case of logic programming with hypothetical reasoning and with bindings, as with λProlog, the simple inductive style approach is problematic. Fortunately, initial steps to address this challenge have been made in the design and implementation of the Abella theorem prover [3, 14] which is capable of reasoning directly with specifications like those found in λProlog. Thus, it should be possible to develop formal proofs of correctness for λProlog based proof checkers using Abella and related tools.

Acknowledgments. Zak Chihani provided comments on an earlier version of this paper. The work presented here has been funded by the ERC Advanced Grant ProofCert.

References

1. J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.
2. A. Assaf. *A framework for defining computational higher-order logics.* PhD thesis, École Polytechnique, Sept. 2015.
3. D. Baelde, K. Chaudhuri, A. Gacek, D. Miller, G. Nadathur, A. Tiu, and Y. Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2), 2014.
4. R. Blanco and D. Miller. Proof outlines as proof certificates: a system description. Draft available online., Sept. 2015.
5. R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, 1979.
6. K. Chaudhuri, F. Pfenning, and G. Price. A logical characterization of forward and backward chaining in the inverse method. *J. of Automated Reasoning*, 40(2-3):133–177, Mar. 2008.

7. Z. Chihani, D. Miller, and F. Renaud. Foundational proof certificates in first-order logic. In M. P. Bonacina, editor, *CADE 24: Conference on Automated Deduction 2013*, number 7898 in LNAI, pages 162–177, 2013.
8. A. Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.
9. D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In S. R. D. 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.
10. V. Danos, J.-B. Joinet, and H. Schellinx. LKT and LKQ: sequent calculi for second order logic based upon dual linear decompositions of classical implication. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, number 222 in London Mathematical Society Lecture Note Series, pages 211–224. Cambridge University Press, 1995.
11. The Dedukti system. https://www.rocq.inria.fr/deducteam/Dedukti/index.html, 2013.
12. C. Dunchev, F. Guidi, C. Sacerdoti Coen, and E. Tassi. A fast interpreter for λProlog. In *LPAR-20: Logic Programming and Automated Reasoning, International Conference*, 2015. To appear.
13. R. Dyckhoff and S. Lengrand. Call-by-value λ-calculus and LJQ. *J. of Logic and Computation*, 17(6):1109–1134, 2007.
14. A. Gacek, D. Miller, and G. Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012.
15. G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935.
16. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
17. J.-Y. Girard. A new constructive logic: classical logic. *Math. Structures in Comp. Science*, 1:255–296, 1991.
18. G. Gonthier. The four colour theorem: Engineering of a formal proof. In D. Kapur, editor, *8th Asian Symposium on Computer Mathematics*, volume 5081 of *LNCS*, page 333. Springer, 2007.
19. M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.
20. T. C. Hales. A proof of the Kepler conjecture. *Annals of Mathematics*, 162(3):1065–1185, 2005.
21. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
22. H. Herbelin. *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes*. PhD thesis, Université Paris 7, 1995.
23. J. M. Howe. *Proof Search Issues in Some Non-Classical Logics*. PhD thesis, University of St Andrews, Dec. 1998. Available as University of St Andrews Research Report CS/99/1.
24. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles (22nd SOSP'09), Operating Systems Review (OSR)*, pages 207–220, Big Sky, MT, Oct. 2009. ACM SIGOPS.
25. O. Laurent. *Etude de la polarisation en logique*. PhD thesis, Université Aix-Marseille II, Mar. 2002.

26. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
27. C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.
28. J. Meng. *The integration of higher order interactive proof with first order automatic theorem proving*. PhD thesis, University of Cambridge, Computer Laboratory, 2015.
29. D. Miller. Abstractions in logic programming. In P. Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.
30. D. Miller. Communicating and trusting proofs: The case for broad spectrum proof certificates. In *Logic, Methodology, and Philosophy of Science. Proceedings of the Fourteenth International Congress*, pages 323–342. College Publications, 2014.
31. D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.
32. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
33. G. Nadathur and D. 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.
34. F. Pereira. *C-Prolog User's Manual, Version 1.5*, June 1988.
35. F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206, Trento, 1999. Springer.
36. N. Wetzler, M. J. H. Heule, and J. Warren A. Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing SAT 2014*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.