HAL
archives-ouvertes.fr

A Semantic Framework for Proof Evidence

Zakaria Chihani, Dale Miller, Fabien Renaud

To cite this version:

Zakaria Chihani, Dale Miller, Fabien Renaud. A Semantic Framework for Proof Evidence. Journal of Automated Reasoning, Springer Verlag, 2017, 59 (3), pp.287-330. 10.1007/s10817-016-9380-6 . hal-01390912

HAL Id: hal-01390912

https://hal.inria.fr/hal-01390912

Submitted on 2 Nov 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A semantic framework for proof evidence

Zakaria Chihani · Dale Miller ·
Fabien Renaud

Draft: July 2, 2016

Abstract Theorem provers produce evidence of proof in many different formats, such as proof scripts, natural deductions, resolution refutations, Herbrand expansions, and equational rewritings. In implemented provers, numerous variants of such formats are actually used: consider, for example, such variants of or restrictions to resolution refutations as binary resolution, hyper-resolution, ordered-resolution, paramodulation, etc. We propose the *foundational proof certificates* (FPC) framework for defining the semantics of a broad range of proof evidence. This framework allows both producers of proof certificates and the checkers of those certificates to have a clear formal definition of the semantics of a wide variety of proof evidence. Employing the FPC framework will allow one to separate a proof from its provenance and to allow anyone to construct their own proof checker for a given style of proof evidence. The foundation on which FPC relies is that of proof theory, particularly recent work into *focused proof systems*: such proof systems provide *protocols* by which a checker extracts information from the certificate (mediated by the so called *clerks and experts*) as well as performs various deterministic and non-deterministic computations. While we shall limit ourselves to first-order logic in this paper, we shall not limit ourselves in many other ways. The FPC framework is described for both classical and intuitionistic logics and for proof structures as diverse as resolution refutations, natural deduction, Frege proofs, and equality proofs.

1 Introduction

Of all topics in mathematics and computer science, one might expect that logic has provided us with clearly defined and delimited standards. This expectation is particularly high since the study of logic and proof has existed since early work by Boole, Frege, Hilbert, Russell, Whitehead, Gödel, and Gentzen and since the resulting logical systems are actively being studied and applied to a wide range of applications in mathematics and computer science. But in practice, sadly, formal proofs are often technology-specific: such "proof objects" are usually meant for a particular prover and even a particular version of that prover. There should be, however, considerable value

Inria-Saclay and LIX, Ecole Polytechnique, Palaiseau, France

in being able to have the multitude of computational logic systems—theorem provers, model checkers, type checkers, static analyzers, etc.—share and check each other's proofs. Such advantages have been explicitly recognized in the SMT community by Van Gelder when he wrote: "It is important to get our propositional house in order to provide an adequate foundation for the more sophisticated challenge of producing independently checkable proofs for SMT." [38].

One way to realize the world-wide sharing and checking of proofs is to ask computational logic systems to export their proof evidence as documents with clear semantics that can be validated by trusted checkers. If checkers can be small and formally specified, then one might be able to trust their correctness by formally proving them correct or by allowing any number of people to reimplement them. (In contrast, theorem provers are, generally speaking, complex and evolving systems that are more difficult to trust.) By the term *proof certificate* we mean an exported document that contains the proof evidence discovered by a computational logic system. We propose here a framework for defining the semantics of a wide range of proof certificates: since we use proof-theoretic concepts to define this framework, we refer to this framework as "foundational" in contrast to, say, technological. Our framework will allow the sequent calculus of Gentzen (with a number of improvements) to actually serve as the semantic framework for defining a wide range of the proof evidence that is (or could easily be) output from theorem provers.

Two bright spots in standardization. One bright spot in the use of logic to provide a standard in computational logic system is the role of simply typed λ-terms as a framework for defining term structures and logical formulas. As Church has shown in [22], his Simple Theory of Types (STT) provides an elegant and well understood framework for defining terms, formulas, binding, and substitution. By varying the signature of type constants, that framework can provide a specification of propositional formulas as well as multi-sorted first-order logic formulas and various modal logics. Such a notion of formula is popular and supported by a wide range of computational logic systems today, such as Isabelle [73], λProlog [66], and Twelf [76]. A second bright spot is that classical and intuitionistic logics have been identified as being important for most of computational logic and that the set of theorems of these logics is well defined: see, for example, textbooks such as [37, 43, 90]. We shall assume for the rest of this paper that terms, formulas, and theorems for first-order classical and intuitionistic logics are well established. Having fixed these, we turn our attention to the specification of proofs.

1.1 Dealing with many proof languages

If the designer of a prover wishes it to be part of a larger world where theorems and proofs are shared, checked, and stored, that prover must output some document that can be checked by trusted proof checkers. The problem is, of course, that there can be many different kinds of proof evidence and, hence, many different kinds of "proof languages" with which to deal. A resolution theorem prover might output a presentation of a resolution refutation; a constructive logic prover might output a typed λ-term; a bisimulation checker might output a set of pairs denoting a bisimulation; etc. Is it necessary to have a separate checker for each of these many proof evidence formats? Even then, how can one be completely clear and formal in providing the details of such proof formats?

A computer scientist will recognize that similar problems have existed before. For example, in the early days of computer science, the structure of a programming language was often defined by a particular piece of technology - a parser. Eventually, the framework of grammars (specifically certain classes of context-free grammars) was used to describe the structure of programming languages. In that way, any number of people could implement parsers for a language described by a given grammar. Furthermore, cross compilers could be written that automatically transformed grammar specifications into actual parsers. In a similar fashion, the meaning of early programming languages was often defined by a particular compiler or interpreter. Today we generally insist that the semantics of a programming language is given by a formal framework such as *denotational semantics* [87], *structural operational semantic* (SOS) [77,78], or *natural semantics* [57]. The semantics of Standard ML, for example, has been defined using natural semantics [70].

Just as frameworks were proposed for programming language structures (via grammars) and semantics (via denotational, structured operational, and natural semantics), we will provide in this paper a framework for defining the semantics of proof languages. The framework that we present here is not the only possible universal framework possible or proposed: for example, the $\lambda\Pi$-modulo system [24] has been proposed, implemented, and applied to the checking of proofs from a number of constructive logic theorem prover (see Section 13).

1.2 What can be learned about proof structure from proof theory

Several different and concrete approaches to structuring proofs have been developed within the proof theory literature. We overview a few of these here.

The earliest formalized notions of proof are now often called *Frege-Hilbert proofs* (Frege proofs, for short): such proofs are sequences of formulas such that any formula is either an axiom or is the result of applying an inference rule to formulas that precede it. Such proofs provide a high degree of trust in what they prove. The LCF approach to theorem proving [47] can be directly linked to such proof systems. In particular, the axioms are given the type `thm`, the inference rules are then (partial) functions of type, say, `thm -> thm` and `thm -> thm -> thm`. Finally, the type `thm` is declared to be an abstract datatype (thus, no additional primitives for building terms of type `thm` can be added). Any object that can be build of type `thm` in such a well-typed programming language as ML can be trusted to encode a theorem. As a notion of proof itself, such proof objects have little useful structure. In the LCF setting, *proof scripts*—specific instructions that lead the prover to a proof—are often used as proofs although modern LCF provers are capable of generating more abstract proof objects as well.

Natural deduction proofs for intuitionistic logic [39,79] provide a much richer setting for studying proof. For example, proofs can be normal or non-normal and functional computations can be performed on non-normal proofs in order to produce normal proofs. In addition, computational information can often be extracted from normal proofs. Furthermore, this paradigm can be augmented via the *deduction modulo* framework to allow even more opportunities for functional-style computations within proofs [30]. Dependently typed λ-calculi have been used to encode natural deduction proofs in a number of computer systems, for example, Automath, Coq, Twelf, and Agda, to mention a few.

While *sequent calculus proof* systems [39] are a flexible way to compare proofs in classical, intuitionistic, and linear logics and to prove cut-elimination theorems, such proofs are surprisingly unstructured and chaotic. In one of the earliest applications of sequent calculus to computational logic [67], logic programming was described as the search for certain normal forms of sequent calculus proofs (the so-called *uniform proofs*) that involved two alternating phases of inference rules—one phase captured *goal reduction* and one phase captured *backchaining*. Andreoli [1] lifted this notion of two-phase alternating proof to all of linear logic and, thereby, introduced the notion of *focused proof system*. Various focused proof systems for intuitionistic logic [14, 32, 50, 53] and classical logic [25, 42, 59] appeared shortly afterwards.

Focused proof systems will play a central role in the FPC framework. In particular, we shall use the *LJF* and *LKF* focused proof systems of [60] (see Figures 5 and 13) since they offer a framework in which most other focused proof systems can be seen as subsystems.

1.3 The atoms, molecules, and chemistry of inference

The sequent calculus of Gentzen (especially as it is refined by Girard in linear logic [41]) provides us with very small elements of inference: in particular, the introduction rules for connectives, the structural rules of weakening and contraction (we shall avoid Gentzen's exchange rule), and the identity rules of initial and cut. We shall (informally) refer to these inference rules as the *atoms of inference*. Sequent calculus proofs can be rather chaotic in structure since the appearance of one occurrence of an inference rule does not generally predict the occurrence of any adjacent inference rule.

Since focused proof systems are composed of alternating phases, we will be able to identify entire *phases* with larger units of inference. Thus, we can say that focused proof systems provide the *rules of chemistry* that identify which atoms stick together (to form phases) and which separate to form boundaries (between phases). In classical and intuitionistic logics, there is a great deal of flexibility in how such phases can be constructed. We shall equate the *molecules of inference* (also called *synthetic rules* or *macro-level rules*) with these phases.

This chemistry of inference will be used to build a framework for defining proof semantics that satisfies the four desiderata laid out in [64] for a general notion of proof certificates.

D1: *A simple checker can, in principle, check if a proof certificate denotes a proof.* Simplicity is helped by the fact that the checker need only implement the atoms of inference and the rules of chemistry. Since both of these are small and closed sets, the checker size and complexity can, in principle, be limited.

D2: *The format for proof certificates must support a wide range of proof systems.* Since the rules of chemistry allow for flexibility in how the atoms of inference can be organized into phases, these phases can be made to match the reasoning steps taken within different styles of proof.

D3: *A proof certificate is intended to denote a proof in the sense of structural proof theory.* The ultimate elaboration of a proof certification involves the elaboration of the molecules of inference into their constituent sequent calculus inference rules. Thus, a sequent calculus proof is implicitly built during checking, satisfying this desideratum.

D4: *A proof certificate can simply leave out details of the intended proof.* Thus a proof checker must be able to do some *proof reconstruction.* As we shall illustrate later, the search space for focused proofs will only use allowed molecules and will not produce unintended molecules of inference by randomly assembling atomic inference rules.

1.4 Machine-machine transmission and checking of formal proofs

In what follows, we shall be concerned with formal and not with informal proofs. Our goal is to describe how one machine might generate a proof that another machine checks. There is no intention here that humans should be able to read or learn from such formal proofs. It might well be the case that certain proof evidence is both formal and human readable: such a happy accident is not to be expected in general. Of course, one might want to browse and interact with formal proofs in order to understand them. To that end, tools that allow such interactions might be built since, by desiderata D3 above, formal proof semantics are not ad hoc structures because they ultimately elaborate into sequent calculus proofs which have a rich meta-theory (including, for example, cut-elimination).

2 Proof checking as computation and interaction

The *Poincaré's Principle* [4] states that proofs do not need to contain computations since computations are routine and can be performed by a proof checker without instructions coming from the proof. For example, a proof checker can verify that $2+2=4$ without having computation steps occupy space in the proof. Most illustrations of this principle generally view computation as a functional program: in particular, computation is *deterministic*. However, any general approach to proof representation should also allow *non-deterministic* computation in the sense of relational programming. It is well known that non-determinism is a powerful computing resource and if one is concerned about the size of proofs certificates, non-determinism can help reduce the size of certificates: witness, for example, the difference is sizes between deterministic and non-deterministic finite state machines accepting the same language, or the difference between the complexity classes P and NP. Also, a focus on non-deterministic computations does not remove our ability to use only deterministic computations within proof checking if one wishes.

Communication and interaction can also be seen as intimately related to proof and proof checking. There are, of course, well-known connections between proof and winning strategies in game theory [27, 51, 62]. We shall view proof checking as involving interactions between the *proof checker* and the *proof certificate*.

2.1 Navigating a robot through corridors and mazes

In order to better illustrate the kind of interaction and computation that we have in mind for proof checking, consider the problem of having a simple robot navigate through the maze of offices and corridors in an office complex. The proof that there is a path from office A to office B can be witnessed by navigating a robot through the

corridors and offices without going through or over walls. Navigation instructions can be structured as follows.

When the robot is at the start of a corridor, we tell the robot (via a radio link) to traverse that corridor. At that point, the robot can turn off its radio (to save power) and compute its path through the corridor. Once the robot reaches the end of the corridor, it turns on its radio again and waits for instructions to navigate the maze of offices it has encountered. We would then interact repeatedly with the robot in order to offer instructions (such as "turn left then forward 5m then turn right") until it reaches its final destination or another corridor. Thus, in this way, we can move through an office complex with communications active only during actual mazes of office and not during corridors.

Moving through corridors is a deterministic computation. Navigating a maze does not necessarily require complete and precise instructions. If the robot has enough sensors, the navigator can also give instructions such as "find the second left turn and take that." Non-deterministic computations can also be valuable here in this setting. For example, every night, when the robot is guided back to the warehouse for recharging, we can navigate the robot to the entrance of the warehouse and then ask it to search the warehouse for an open electric plug. In principle, this computation might be non-deterministic since there might be several such plugs and any one might be picked. We might well expect, however, such a search to always terminate given our knowledge of how warehouses are designed and how many other robots might be parked there.

2.2 Sorting out this analogy

When we present our first *focused proof system* in Section 4, its most striking feature is that they build proofs in two different phases, called the *asynchronous* phase and the *synchronous* phase. A robot moving down a long corridor can be modeled by an asynchronous phase: that phase is characterized by determinate computations and no communications. The robot moving through a maze of offices corresponds to the synchronous phase where communications is useful in order to make a backtrack-free path.

If one gives the checker some additional computational capabilities, we can, like a robot with sensors and the ability to do search, give less explicit and less precise instructions, such as, "the final destination is immediately behind one of the doors on the right". An implementation of proof checking within a logic programming setting allows such high-level instructions to be effectively executed by the checker. In the implementation of such a checker, a non-deterministic computation can be captured via backtracking search.

3 Two proof systems for propositional classical logic

To illustrate how a sequent calculus proof system can be used to describe interactive protocols, we consider in this section two different proof systems, *LKneg* and *LKpos*, for classical propositional logic. In both cases, we shall assume that formulas are in negation normal form (that is, negations have only atomic scope), that the logical connectives are f, $\vee$, t, and $\wedge$, and that sequents are one-sided. The soundness of both

$$\frac{\vdash \cdot ; B}{\vdash B}\ \mathit{start} \qquad \frac{\vdash \Delta, L; \Gamma}{\vdash \Delta; L, \Gamma}\ \mathit{store} \qquad \frac{}{\vdash \Delta, A, \neg A; \cdot}\ \mathit{init}$$

$$\frac{\vdash \Delta; \Gamma}{\vdash \Delta; f, \Gamma} \qquad \frac{\vdash \Delta; B, C, \Gamma}{\vdash \Delta; B \vee C, \Gamma} \qquad \frac{}{\vdash \Delta; t, \Gamma} \qquad \frac{\vdash \Delta; B, \Gamma \quad \vdash \Delta; C, \Gamma}{\vdash \Delta; B \wedge C, \Gamma}$$

Fig. 1 The *LKneg* proof system

$$\frac{\vdash B; \cdot ; B}{\vdash B}\ \mathit{start} \qquad \frac{\vdash B; \mathcal{N}, \neg A; B}{\vdash B; \mathcal{N}; \neg A}\ \mathit{restart} \qquad \frac{}{\vdash B; \mathcal{N}, \neg A; A}\ \mathit{init}$$

$$\frac{\vdash B; \mathcal{N}; B_i}{\vdash B; \mathcal{N}; B_1 \vee B_2}\ i \in \{1, 2\} \qquad \frac{}{\vdash B; \mathcal{N}; t} \qquad \frac{\vdash B; \mathcal{N}; B_1 \quad \vdash B; \mathcal{N}; B_2}{\vdash B; \mathcal{N}; B_1 \wedge B_2}$$

Fig. 2 The *LKpos* proof system

LKneg and *LKpos* is trivial to establish: their completeness follows directly from the completeness of the *LKF* system that we present in Section 4.

3.1 *LKneg* and the invertible inference rules

The *LKneg* proof system (see Figure 1) will use only the invertible inference rules for introducing both conjunction and disjunction. In this proof system, a sequent contains two zones and is written as $\vdash \Delta; \Gamma$, where Δ is a multiset of literals, Γ is a list of formulas, and L (in the store rule) is a literal. Notice that the four introduction rules and the store rule work only on the first member of the second zone of a sequent. A dot is used to denote both an empty list and an empty multiset.

Clearly, the *LKneg* proof system describes a decision procedure for propositional classical logic in the following sense.

1. Given a formula B, there is a unique derivation Π_B of $\vdash B$ with premises of the form $\vdash \Delta; \cdot$. That is, Π_B is functionally determined from B.
2. The formula B is a tautology if and only if every premise of Π_B is such that its first zone contains an atomic formula and its complement.

The implied decision procedure for proving B is first to build Π_B and then check that every premise in Π_B contains complementary pairs of literals. Clearly, this decision procedure is essentially the same as rewriting B into conjunctive normal form and then checking that every disjunction in that normal form has complementary literals. It is also clear that this decision procedure has exponential time complexity.

There is some non-determinism possible in this proof system in the sense that there might be more than one way to decompose a multiset of literals into $\Delta, A, \neg A$ in the *init* rule. For example, the formula $(p \vee p) \vee \neg p$ can be seen as having two different proofs.

3.2 *LKpos* and non-invertible rules

The *LKpos* proof system (see Figure 2) uses the non-invertible inference rule for introducing disjunction. In this proof system, a sequent contains three zones and is written as $\vdash B; \Delta; C$, where B is a (fixed) formula, Δ is a multiset of negated atoms, and C is a formula. Notice that introduction rules only introduce the formula in the third zone

$$\frac{C \vdash B; \cdot; B}{C \vdash B}\ start \qquad \frac{C \vdash B; \mathcal{N}, \neg A; B}{C \vdash B; \mathcal{N}; \neg A}\ restart \qquad \frac{}{emp \vdash B; \mathcal{N}, \neg A; A}\ init$$

$$\frac{C \vdash B; \mathcal{N}; B_1}{l(C) \vdash B; \mathcal{N}; B_1 \vee B_2} \qquad \frac{C \vdash B; \mathcal{N}; B_2}{r(C) \vdash B; \mathcal{N}; B_1 \vee B_2} \qquad \frac{}{emp \vdash B; \mathcal{N}; t} \qquad \frac{L \vdash B; \mathcal{N}; B_1 \quad R \vdash B; \mathcal{N}; B_2}{c(L, R) \vdash B; \mathcal{N}; B_1 \wedge B_2}$$

Fig. 3 The *LKpos* proof system augmented with a certificate

$$\frac{\langle C, emp\rangle \vdash B; \cdot; B}{C \vdash B}\ start \qquad \frac{\langle I, O\rangle \vdash B; \mathcal{N}, \neg A; B}{\langle I, O\rangle \vdash B; \mathcal{N}; \neg A}\ restart \qquad \frac{}{\langle I, I\rangle \vdash B; \mathcal{N}, \neg A; A}\ init$$

$$\frac{\langle I, O\rangle \vdash B; \mathcal{N}; B_1}{\langle l(I), O\rangle \vdash B; \mathcal{N}; B_1 \vee B_2} \qquad \frac{\langle I, O\rangle \vdash B; \mathcal{N}; B_2}{\langle r(I), O\rangle \vdash B; \mathcal{N}; B_1 \vee B_2}$$

$$\frac{}{\langle I, I\rangle \vdash B; \mathcal{N}; t} \qquad \frac{\langle I, M\rangle \vdash B; \mathcal{N}; B_1 \qquad \langle M, O\rangle \vdash B; \mathcal{N}; B_2}{\langle I, O\rangle \vdash B; \mathcal{N}; B_1 \wedge B_2}$$

Fig. 4 The *LKpos* proof system augmented with a certificate

of a sequent. The restart rule is responsible for both "storing" a negated literal in the third zone as well as restarting the proof process by copying the formula B in the first zone and placing it into the third zone.

The *LKpos* proof system does not yield a decision procedure in the same, direct sense that *LKneg* does. In particular, the restart rule can lead to unbounded proof search: consider, for example, proving $\vdash \neg p$, for some propositional constant p.

The bottom-up development of an *LKpos* proof for a given sequent is determinate except for the introduction rule for the disjunction. In that case, the proof system needs to choose between the left and right disjunction. Thus, one could view the essence of a proof in *LKpos* as the information needed to answer that one question every time a disjunction is encountered (reading proofs from conclusion to premises). It is, thus, an easy matter to develop a certificate for *LKpos* proofs that explicitly provides those choices. Such a certificate can be formalized using terms built from the following constructors: l and r are constructors of one argument, c is a constructor of 2 arguments, and emp is a constructor of no arguments. Notice that the formula $(\neg p \vee q) \vee p$ (for propositional constants p and q) has a proof in *LKpos* that is given by the term $l(l(r(emp)))$: that is, $l(l(r(emp))) \vdash (\neg p \vee q) \vee p$ is provable using the rules in Figure 3.

This notion of certificate is similar to the use of "oracles as streams of bits" as representations of proofs in [72]. More precisely, the certificate structure used in Figure 3 is better named an *oracle tree* since it uses a binary constructor c to pair together two other oracle trees. A simple variation of the certificate format described above can reorganize the choices of left/right into, instead, *oracle strings*. Consider removing the c constructor and use, instead, a pair of two *oracle strings*. Figure 4 contains a suitably instrumented version of the *LKpos* inference rule: there, a certificate is such a list of left/right choices but internally to the proof system, a pair of such lists is used. In general, when an inference rule is labeled with the certificate term $\langle I, O\rangle$, then O is a tail of the list I and the formal difference between these two lists is the sequence of left/right choices that were consumed in building this proof.

3.3 A proof system can yield a protocol

The description of a specific proof of a given formula in *LKneg* and in *LKpos* are radically different. For example, consider interacting with someone (called the *checker*) attempting to build a proof bottom-up in *LKneg*: except for the *init* rule, there is no choice in how the proof building process is conducted. At every step, the checker simply needs to apply the only rule that is possible in the only way that is possible. With the *init* rule, a choice is possible since we need to select a complementary pair of literals from a multiset of literals and there can be multiple such pairs. Our interaction with the checker building such a proof might either involve telling her the pair or letting her find such a pair on her own, given that such a search is bounded and simple.

On the other hand, assume that the checker is attempting to build a proof in *LKpos*. Here, the checker will need to ask from us what to do with every disjunction: should she select the left or right disjunct? There is also a choice in the *init* rule again: this time, the choice is between possibly different negated atoms within the $\mathcal{N}$ zone to pick and we can let the checker search for her own answer to the *init* question.

Consider how these two protocols work on the formula $(\neg p \vee C) \vee p$, where C is some, possibly large propositional formula. This formula is clearly a tautology. A checker building an *LKneg* proof will initially fall silent and build a possibly huge derivation, stopping only when she wants to apply the *init* rule (then either asking for a pair of complementary literals or searching for them). Of course, a possibly exponential (in the size of C) number of such premises might be built and p and $\neg p$ will occur in every one of them. Thus, this checker might run in exponential time but consume no information from an external source. Therefore, checking time can be exponential but interaction can be constant sized. On the other hand, the *LKpos* proof can consume some "clever" information from a proof certificate and, as a result, terminate quickly. In particular, the certificate $l(l(r(emp)))$ can steer the (augmented) *LKpos* proof system to a proof independent of the formula C. Thus, if we are willing to communicate some information to the checker, we can have drastic improvements in checking time.

We shall now present a more ambitious proof system for classical logic, one that, in a sense, mixes the inference rules in *LKneg* and *LKpos* into one, hybrid proof system. It will also include first-order quantification.

4 *LKF* as a framework for classical focused proofs

Figure 5 presents the *LKF* focused proof system for first-order classical logic [60]. The sequents of this proof system contain *polarized* formulas instead of ordinary (unpolarized) formulas. A polarized formula results from taking an ordinary first-order classical formula and replacing every occurrence of propositional connectives and constants with the corresponding occurrence of a positive or negative version of that connective or constant. The polarized versions are formed by affixing a superscripted $+$ or $-$ symbol. Thus, an occurrence of $\vee$ can be replaced by an occurrence of either $\vee^+$ or $\vee^-$. If B is a formula with n occurrences of propositional connectives and constants then there are 2^n different polarized versions of B. The quantifiers have unambiguous polarities: the existential quantifier is positive and the universal quantifier is negative.

A polarized formula is *positive* if its top-level connective is a positive connective ($\vee^+$, $\wedge^+$, t^+, f^+, $\exists$) or an atomic formula. Similarly, a polarized formula is *negative* if its top-level connective is a negative connective ($\vee^-$, $\wedge^-$, t^-, f^-, $\forall$) or a negated

Asynchronous introduction rules

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

Synchronous introduction rules

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

Identity rules

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

Structural rules

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

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. The proviso marked as † is the usual eigenvariable restriction: y is not free in Θ, in Γ, nor in $\forall x.B$.

Fig. 5 *LKF*: a focused proof systems for classical logic

atomic formula. The notion of de Morgan duals is extended to polarized connectives following the pairing t^{-}/f^{+}, t^{+}/f^{-}, $\vee^{+}/\wedge^{-}$, $\vee^{-}/\wedge^{+}$, and $\forall/\exists$. Negation is not a proper logical connective: we assume that all classical formulas are in *negation normal form*, meaning that the $\neg$ symbol has only atomic scope. When we write $\neg B$, for a non-atomic polarized formula B, we mean the polarized formula that results from computing the de Morgan dual of the connectives and literals within B. We shall sometimes refer to polarized formulas as simply formulas when it is clear from context which we mean.

The *LKF* proof system uses two kinds of sequents. The up-arrow sequents $\vdash \Gamma \Uparrow \Theta$ and the down-arrow sequents $\vdash \Gamma \Downarrow B$. In both cases, Γ is a multiset of formulas, Θ is a list of formulas, and B is a formula. We say that the up or down arrows divide sequents into two *zones*, namely the collection of formulas to the left or to the right of the arrow. The left zone is called *storage*. It is most accurate to consider such sequents as one-sided sequents since these two sequents can be approximated as the sequents $\vdash \Gamma, \Theta$ and $\vdash \Gamma, B$, respectively. Introduction rules are applied to the first element of the second zone of both of these sequents.

Proofs in *LKF* are built using two kinds of alternating *phases*. The *asynchronous* phase is composed of invertible inference rules and only involves $\Uparrow$-sequents in the conclusion and premise. The other phase is the *synchronous* phase: here, applications of such inference rules often require choices. In particular, the introduction rule for the disjunction requires selecting either the left or right disjunct and the introduction rule for the existential quantifier requires selecting a term for instantiating the quantifier. The initial rule can terminate a synchronous phase and the cut rule can restart an asynchronous phase. Finally, there are three structural rules in *LKF*. The *store* rule recognizes that the first formula to the right of the $\Uparrow$ is either a negative atom or a positive formula: such a formula does not have an invertible introduction inference rule and, hence, its treatment is delayed by storing it on the left. The *release* rule is used when the formula under focus (i.e., the formula to the right of the $\Downarrow$) is no longer positive: at such a moment, the phase changes to the asynchronous phase. Finally, the

decide rule is used at the end of the asynchronous phase to start a synchronous phase by selecting a previously stored positive formula as the *focus*.

Notice that negative non-literal formulas are treated linearly in the sense that they are never weakened nor contracted. Only positives formulas are contracted (in the decide rule) and only negative literals and positive formulas are weakened (in the init rule). The following theorem about *LKF* is proved in [60].

Theorem 1 *Let B be a classical, first-order formula.*

1. *If B is a theorem then for every polarization $\hat{B}$ of B, the sequent $\vdash \cdot \Uparrow \hat{B}$ has an LKF proof.*
2. *If $\hat{B}$ is a polarization of B and if $\vdash \cdot \Uparrow \hat{B}$ has an LKF proof then B is a theorem.*
3. *If a sequent has an LKF proof, it has a cut-free LKF proof.*

The right zone in the up-arrow sequent is a list. It is possible to make that zone into a multiset instead but we refrain from doing this for two reasons. First, the completeness of *LKF* (Theorem 1) is a stronger result when the Θ context is a list, and second, the behavior of proof construction in *LKF* is more deterministic and that lends itself to a more predictable protocol between a proof checker and proof certificate.

It is straightforward to prove the completeness of the *LKneg* and *LKpos* proof systems given Theorem 1. In particular, the *LKneg* system corresponds to using *LKF* when all propositional connectives have been polarized negatively and the *LKpos* system corresponds to using *LKF* when all propositional connectives have been polarized positively. In the late case, the rule called *restart* in *LKpos* is a combination of the *LKF* rules *store* and *decide*.

Example 1 In order to present a simple macro-level inference rule, assume that Γ contains the formula $a \wedge^+ b \wedge^+ \neg c$. An *LKF* proof in which the root inference rule is a *decide* rule applied to this formula must have the following form.

$$\dfrac{\dfrac{\dfrac{}{\vdash \Gamma \Downarrow a}\,\mathit{init} \quad \dfrac{}{\vdash \Gamma \Downarrow b}\,\mathit{init} \quad \dfrac{\dfrac{\vdash \Gamma, \neg c \Uparrow \cdot}{\vdash \Gamma \Uparrow \neg c}\,\mathit{store}}{\vdash \Gamma \Downarrow \neg c}\,\mathit{release}}{\vdash \Gamma \Downarrow a \wedge^+ b \wedge^+ \neg c}}{\vdash \Gamma \Uparrow \cdot}\,\mathit{decide}$$

Given the design of the focused initial rule, this derivation is possible if and only if Γ is of the form $\neg a, \neg b, \Gamma'$. Thus, deciding on this formula encodes the following synthetic inference rule.

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

While checking the applicability of individual sequent calculus inference rules can be done effectively, checking synthetic inference rules may, in fact, require exponential time and or space: consider, for example, the synthetic inference rules that correspond to the *LKneg* proof system. Thus, synthetic rules are not always inference rules in the sense of Cook [23] where inference rules must be polynomially checkable.

Asynchronous Rules

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

$$\frac{\Xi_1 \vdash \Gamma \Uparrow A, B, \Theta \quad \vee_c(\Xi_0, \Xi_1)}{\Xi_0 \vdash \Gamma \Uparrow A \vee^- B, \Theta} \qquad \frac{}{\Xi_0 \vdash \Gamma \Uparrow t^-, \Theta} \qquad \frac{(\Xi_1\, y) \vdash \Gamma \Uparrow (B\, y), \Theta \quad \forall_c(\Xi_0, \Xi_1)}{\Xi_0 \vdash \Gamma \Uparrow \forall x.B, \Theta}\ \dagger$$

Synchronous Rules

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

$$\frac{\Xi_1 \vdash \Gamma \Downarrow B_i \quad \vee_e(\Xi_0, \Xi_1, i)}{\Xi_0 \vdash \Gamma \Downarrow B_1 \vee^+ B_2} \qquad \frac{\Xi_1 \vdash \Gamma \Downarrow (B\, t) \quad \exists_e(\Xi_0, \Xi_1, t)}{\Xi_0 \vdash \Gamma \Downarrow \exists B}$$

Identity rules

$$\frac{\Xi_1 \vdash \Gamma \Uparrow B \quad \Xi_2 \vdash \Gamma \Uparrow \neg B \quad cut_e(\Xi_0, \Xi_1, \Xi_2, B)}{\Xi_0 \vdash \Gamma \Uparrow \cdot}\ cut \qquad \frac{l : \neg P_a \in \Gamma \quad init_e(\Xi_0, l)}{\Xi_0 \vdash \Gamma \Downarrow P_a}\ init$$

Structural rules

$$\frac{\Xi_1 \vdash \Gamma \Uparrow N \quad release_e(\Xi_0, \Xi_1)}{\Xi_0 \vdash \Gamma \Downarrow N}\ release \qquad \frac{\Xi_1 \vdash \Gamma \Downarrow P \quad l : P \in \Gamma \quad decide_e(\Xi_0, \Xi_1, l)}{\Xi_0 \vdash \Gamma \Uparrow \cdot}\ decide$$

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

The proviso † requires that y is not free in Ξ, Θ, Γ, B. Notice also that in that same rule Ξ_1 is a term-level abstraction over certificates and B is a term-level abstraction over formulas (as it is also in the $\exists$-introduction rules).

Fig. 6 The augmented *LKF* proof system LKF^a.

5 Augmented *LKF* and checking certificates

We are now in a position to describe in detail how the *LKF* proof system can be used to provide a protocol for mediating the communications between a proof checker, whose architecture is described in this section, and foundational proof certificates, introduced in the following section.

We present an analogy to help convey the spirit of the intended protocol. Imagine an accounting office which is charged with checking if a certain mound of financial documents represents a legal transaction as judged by some particular tax code. The tax office staff is divided into two groups. The workers called *experts* are given the responsibility of looking into the mound and extracting information: they must *decide* into which series of transactions to dig and they need to know when to *release* their findings for storage and later reconsideration. On the other hand, the workers called *clerks* are responsible for taking information released by the experts and performing various computations on them, including their *indexing* and *storing*. The justification of this division of effort between clerks and experts comes from the structure of focused sequent proof systems: experts operate during the *synchronous* phase of proof construction while clerks operate during the *asynchronous* phase. Furthermore, the vocabulary of *decide*, *release*, and *store* comes directly from the structural rules of *LKF*.

A formal definition of the augmentation of *LKF*, seen in Figure 6, is presented along three directions.

1. Every sequent in every inference rule is decorated with an additional argument, labeled as Ξ, Ξ_1, or Ξ_2. These additional arguments are *certificate terms* or just *certificates*, for short.
2. The first zone in all sequents has been changed from being a multiset of formulas to being a multiset of pairs of the form $l : B$ where B is a formula and l is an *index*. Along with this change, the *store* and *decide* rules refer to indexes as well as formulas.
3. Every inference rule in *LKF* (except the introduction rule for t^-) is given an extra premise which is an atomic formula with a top-level predicate that is either a "clerk predicate" (subscripted with the letter "c") or an "expert predicate" (subscripted with the letter "e"). The definition of these "agents" provide the formal meaning of certificates terms.

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 $l : B$ with just B. For this reason, any proof in LKF^a is a proof in *LKF*, which guarantees the soundness of the LKF^a system.

An implementation of the augment focused proof system LKF^a will be called a *kernel* (for *LKF*). We describe in the next section the items that must be added to the kernel—formal definitions of clerks and experts as well as the data structures for indexes and certificate terms—in order to construct a complete proof checker. In Section 8 we describe how to build a kernel along similar lines for intuitionistic first-order logic and in Section 11 we describe an approach to implementing kernels using the λProlog programming language.

6 Foundational proof certificates

A foundational proof certificate consists of a proof object written in a suitable language along with the semantic definition of that language. A proof language is given meaning by specifying how a proof object written in it can unfold into a sequent calculus proof, in the same way a programming language is given meaning by specifying how its instructions can be interpreted using some other low-level programming language (e.g., assembly language). The sequent calculus is arguably standard enough to serve as the "assembly language of proof languages". While a kernel based on it may not be able to deal with certain aspects of proofs—such as "is this proof minimal"—it can provide a means to ascertain that a given logical formula is a theorem.

The semantic definition of a class of proof objects is given through the specification of five parameters. A proper proof certificate is then a pair, containing a specific proof object and a reference to the formal semantic definition that is then used by an implementation of LKF^a to provide a proof checker for that proof object. The five parameters—polarization, certificate terms, indexes, clerks, and experts—are described below and are, collectively, called an *FPC*.

Polarization. Since provers work with *unpolarized* formulas but LKF^a requires polarized formulas, the first step in specifying a proof semantics involves choosing how to polarize formulas. Exactly how to choose polarity depends on many considerations. For example, an inspection of Figure 5 shows that the *decide* rule incorporates the

only occurrence of the contraction rule. Therefore a positive polarity must be given to formulas that one wishes to make "contractible". Similarly, if one wishes to perform a deterministic computation within a proof—such as reducing a clause to a multiset of its literals—then all the connectives in such a clause need to be given a negative polarity. There is a great flexibility in polarizing connectives since it is possible to polarize some occurrences of disjunctions and conjunctions as positive and some as negative, even in the same polarized formula. Finally, the polarity choice of atoms can also greatly influence the proofs one gets. Each atom can receive either a positive polarity (turning the negation of that atom into a negatively polarized atom) or a negative polarity (turning the negation of that atom into a positively polarized atom). In *LKF* we have chosen to give all atomic formulas positive polarity: another treatment of the polarity of atoms is described in Section 8.1.

Certificate terms. The signature for the certificate terms augmenting the *LKF* sequents (via the schema variable Ξ) must be declared. When we explicitly list such signatures below, we will identify constructors of certificate terms as any constant whose type has target type `cert`. The different constructors are used to denote different "regions" of the proof checking/reconstruction process: generally speaking, the clerk and expert relations operate differently in different regions of the proof checking process. Terms of type `cert` are threaded throughout the LKF^a inference rules in a systematic way. Because they are, in essence, the containers of a proof's evidence, these terms will often be referred to simply as "certificates".

Indexes. The *store* and *decide* inference rules are responsible for moving formulas in and out of storage. When a formula is put into storage using the *store* rule, it is assigned an index and when we need to take a formula from storage in the *decide* rule, that formula is addressed via an index. The exact structure of indexes is open to the author of an FPC: in what follows, an index is any term of type `index`. Examples of indexes can be numbers (such as de Bruijn indexes or clause numbers in resolution refutations). They can also be terms representing occurrences of subformulas with a formula. There is no assumption that indexes uniquely determine formulas: in some of the examples we present starting in Section 7, many stored formulas have the same label: thus, when the *decide* rule selects a formula by supplying an index, the dereferencing of that index might be non-deterministic. Formulas can themselves be used as indexes in which case such dereferencing is, of course, deterministic.

Experts. The synchronous inference rules in the augmented proof system LKF^a are given an extra premise that invokes an expert predicate that is responsible for examining a certificate term and extracting information from it, yielding continuation certificates in the process. For example, the existential introduction rule invokes the existential expert predicate in order to extract a term for witnessing the existential instance. Keeping with the theme of making the FPC framework as flexible and general as possible, experts may not exhibit actual expert behavior. For example, when the disjunction expert is asked which disjunct to select, it might suggest both left and right. Similarly, the exists expert is allowed to guess at any (every) term. This kind of flexibility is desirable since deterministic behavior is certainly allowed but non-determinism is more general. The proof context itself might be a useful way to determine (via search) the value of an existential witness. In other words, sometimes the best thing an expert

can do is to let the checker search for the right instantiation term. We illustrate this below in, for example, the resolution refutation checker.

Clerks. The asynchronous inference rules in the augmented proof system LKF^a are given an extra premise that invokes a clerk predicate that specifies the computations that take place in that phase. For example, clerks are used to compute indexes for use in the *store* rule and, if necessary, on which branch of a two-premise inference rule the proof checking phase continues.

When we present examples of FPCs in the following sections, we use the concrete syntax of λProlog [66]. We make this choice for several reasons [18]. First, it is useful to have a typed specification language so that we can explicitly specify the constructors of different datatypes denoting, for example, indexes, certificate terms, and formulas. Such declarations can be written as λProlog signatures. Second, clerks and experts are relations that are defined by inductive specifications: the Horn clauses available within λProlog immediately and naturally allow such definitions to be written. Third, the λ-*tree syntax* approach to treating bindings available in λProlog yields an immediate, declarative, and effectively implemented approach to the computational treatment of formula-level bindings (quantifiers) and proof-level bindings (eigenvariable). In any case, the use of λProlog syntax is only one of convenience: when we use that syntax we are describing nothing more than formulas in, say, Church's Simple Theory of Types (STT) [22]. No non-logical features of λProlog are used in this paper. All the λProlog code described in this paper is available [20].

For readers familiar with, say, Prolog or ML, the syntax of λProlog should be easy to read. We illustrate here two specific aspects of λProlog's design: the syntax for signatures and the syntax for term and formula binders.

To introduce a new type constructor, a `kind` declaration is used. Following the example in Section 3.2, the line

```
kind  oracle    type.
```

can be used to declare a new type called `oracle`. Introducing the constructors for building terms is done with `type` declarations. For example, the type declarations

```
type emp              oracle.
type l, r             oracle -> oracle.
type c                oracle -> oracle -> oracle.
```

introduce four constants that can be used to build oracles.

The λ-abstraction is written as an infix backslash: that is, the term $\lambda x.t$ is written in λProlog syntax as `x\ t`. The scope of this infix operator extends to the right as far as possible consistent with parentheses: thus, `(x\ y\ z\ t)` is the λProlog encoding of $\lambda x(\lambda y(\lambda z.t))$. Following Church [22], quantifiers in formulas are decomposed into a λ-binder and a constant denoting that quantifier. Thus, the existential and universal quantifiers $\exists x.B$ and $\forall x.B$ that are part of λProlog specifications are written as `(sigma x\ B)` and `(pi x\ B)`, respectively. Later in Section 9.2 we will need to explicitly encode first-order universal formulas as terms and in that encoding, we shall similarly decompose that object-level quantifier, using the constant `forall` (instead of `pi` which is used to denote the λProlog universal quantifier).

While we use λProlog code to specify most aspects of particular FPCs, it is only meant to convey the formal semantic definition of a proof format. Whether or not one actually uses λProlog to implement a proof checker for the defined proof format

is a separate concern. We have used the Teyjus [71, 80] implementation of λProlog to build prototype proof checkers and to debug our FPC specifications. A number of other means for implementing a formal semantic definition can, of course, be considered. The logic programming perspective on proof checking is, however, rather powerful since it incorporates in a familiar and systematic fashion unification and backtracking search. Both of these tools can play useful roles in proof reconstruction during proof checking [65].

7 Examples of FPCs in classical, first-order logic

We shall now present examples of FPCs in classical logic. For each example, the semantic definition follows the same steps:

1. Provide a mapping of unpolarized formulas into polarized versions. Since the focused proof system *LKF* has twice the number of propositional connectives as standard first-order logic, one must decide how to treat, for example, the disjunction: is it treated as invertible or not. The actual polarization for propositional connectives and literals is chosen in concert with choosing the definition of the clerks and experts (item 4 below).
2. Provide constructors of the type `cert`: this type is used to build the certificate terms that are represented by the schema variable Ξ in Figure 6.
3. Provide constructors of the type `index`: this type is used to represent legal indexes that are represented by the schema variable l in Figure 6.
4. Define the clerk and expert predicates.

Note that if an n-ary predicate (clerk, expert, or otherwise) is not specified by any clauses, that predicate will never be provable and, hence, denotes the empty relation on n arguments.

The names for clerks and experts will be written as tokens with three components: the name for the connective introduced or the name for the structural rules, the string `_k`, and either the letter `e` (for expert) or `c` (for clerk). When we later present clerks and experts for intuitionistic logic, we replace the `_k` with `_j`. For example, `orNeg_kc` is the $\vee_c^-$ clerk predicate from Figure 6.

7.1 CNF decision procedure

In Section 3.1, we introduced a proof system (and decision procedure) for propositional classical logic that used only invertible introduction rules. We can describe an FPC that can steer the *LKF* proof system to emulate the same proof system as follows. First, we translate unpolarized, propositional classical logic formulas into polarized formulas by translating $\wedge$, t, $\vee$, f as $\wedge^-$, t^-, $\vee^-$, and f^-, respectively. The rest of this FPC is given in Figure 7. In particular, we declare each of the types `cert` and `index` as having exactly one member each, namely, `cnf` and `lit`, respectively. Thus, the certificate essentially contains just the name of a decision procedure and the only index used will have the name `lit`. Finally, meaning is given to both of these constructors as well as to the four clerk and the three expert predicates

$$\wedge_c(\cdot,\cdot,\cdot),\ f_c(\cdot,\cdot),\ \mathit{store}_c(\cdot,\cdot,\cdot),\ \vee_c(\cdot,\cdot),\ \mathit{release}_e(\cdot,\cdot),\ \mathit{init}_e(\cdot,\cdot),\ \text{and}\ \mathit{decide}_e(\cdot,\cdot,\cdot).$$

```
type lit                index.
type cnf                 cert.

andNeg_kc       cnf cnf cnf.
orNeg_kc            cnf cnf.
false_kc            cnf cnf.
release_ke          cnf cnf.
initial_ke              cnf lit.
decide_ke           cnf cnf lit.
store_kc            cnf cnf lit.
```

Fig. 7 An FPC providing a decision procedure based on conjunctive normal forms

by the theories for these predicates displayed in Figure 7. The clerk and expert predicates not in this list are given an "empty theory" and thus do not hold for any arguments.

As we have commented before (in Section 5), soundness is guaranteed: that is, it is immediate to show that if $\hat{B}$ is the polarization of B using the negative versions of disjunction and conjunction, then a proof of $\texttt{cnf} \vdash \cdot \Uparrow \hat{B}$ implies $\vdash B$ in classical logic. We can also make the following additional observations about this FPC.

- If $\Xi \vdash \Gamma \Uparrow \Theta$ is provable then Ξ is the constant `cnf` and Θ is a multiset of pairs $\texttt{lit} : L$ where L is a literal.
- If $\Xi \vdash \Gamma \Downarrow B$ is provable then Ξ and Γ are as above and B is an atom such that $\texttt{lit} : B$ and $\texttt{lit} : \neg B$ are members of Θ. An occurrence of such a sequent must be both the premise of a *decide* rule and the conclusion of an *initial* rule.

Using these observations, the completeness of this encoding is simple to establish.

7.2 *LKpos* example

To provide a slightly more interesting notion of proof certificate, we can show how the proof evidence (oracle strings) described in Section 3.2 can be described as an FPC. First, we translate unpolarized, propositional classical logic formulas into polarized formulas by translating $\wedge$ and $\vee$ as $\wedge^+$ and $\vee^+$, respectively (similarly for their units). The rest of this FPC is given in Figure 8. In particular, the type `index` has two inhabitants: `root` is used to label the restart formula and `lit` is used to label (negative) literals. The proof certificate contains oracles, these have the same structure as the proof evidence described in Section 3.2 and Figure 3. Since more than one inference rule of LKF^a is needed to encode the *start* and *restart* rules of *LKpos*, additional constructors are used in certificates to connect these two inference rules in LKF^a.

7.3 Resolution refutations

Binary resolution is a popular form of proof and we demonstrate an FPC that is able to check an alleged (binary) resolution refutation. We review the key ideas behind resolution refutations.

A (resolution) *clause* is a closed formula that is composed of universal quantifiers around a disjunction of literals (the empty disjunction is identified with false). We use

```
kind oracle          type.
type emp             oracle.                      % empty
type l, r            oracle -> oracle.            % left, right
type c               oracle -> oracle -> oracle.  % conjunction
kind cert            type.
type start, restart  oracle -> cert.
type consume         oracle -> cert.
type root, lit       index.

decide_ke  (start Oracle) (consume Oracle) root.
store_kc   (start Oracle) (start Oracle) root.
decide_ke  (restart Oracle) (consume Oracle) root.
store_kc   (restart Oracle) (restart Oracle) lit.
true_ke    (consume emp).
initial_ke (consume emp) lit.
orPos_ke   (consume (l Oracle)) (consume Oracle) left.
orPos_ke   (consume (r Oracle)) (consume Oracle) right.
andPos_ke  (consume (c Left Right)) (consume Left) (consume Right).
release_ke (consume Oracle) (restart Oracle).
```

Fig. 8 An FPC based on oracle strings

C as a schema variable ranging over clauses. We assume that a certificate for resolution contains the following items:

1. The proposed theorem $\neg C_1 \vee \cdots \vee \neg C_n$ which is the disjunction of a number of negated clauses.
2. A list of clauses $C_{n+1}, \ldots, C_p$.
3. A list of triples $\langle i, j, k\rangle$ where each such triple claims that C_k is a binary resolution (with factoring) of C_i and C_j.

If the implementer of a traditional, binary resolution prover wished to output a resolution refutation, a document containing these items should be easy to provide. Of course, for such a structure to denote a proper refutation, one of the clauses $C_{n+1}, \ldots, C_p$ (usually the last one) must be the empty clause (denoted here as false).

As before, the first step in defining an FPC is describing how to polarize formulas. For this FPC, we polarize disjunctions and false in clauses as negative and the conjunction and true in negated clauses as positive. We shall always assume that clauses have negative polarity and that negated clauses always have positive polarity. If, for example, a clause is just an atomic formula A (which has positive polarity), then we must write that clause as, for example, $A \vee^{-} f^{-}$. In what follows, the expression $\neg C_i$ denotes a polarized negated clause, that is, an existentially quantified positive-conjunction of literals.

The first phase in building an LKF^a proof of the proposed theorem $\neg C_1 \vee \cdots \vee \neg C_n$ is the asynchronous phase using the clerks described in Figure 9 that are responsible for interpreting the `cert` constructor `start`. These clerks steer the LKF^a kernel to build the following augmented, synthetic inference rule.

$$\frac{\texttt{(start}\ (n+1)\ \texttt{R)} \vdash \texttt{(idx 1)} : \neg C_1, \ldots, \texttt{(idx}\ n\texttt{)} : \neg C_n \Uparrow}{\texttt{(start 1 R)} \vdash \Uparrow \neg C_1 \vee \cdots \vee \neg C_n}$$

That is, this initial phase of proof building simply stores the negated clause $\neg C_i$ with the index `(idx` i`)`.

```
type idx      int -> index.
type start    int -> list method -> cert.

orNeg_kc (start C Resol) (start C Resol).
false_kc (start C Resol) (start C Resol).
store_kc (start C Resol) (start D Resol) (idx C) :- D is C + 1.
```

Fig. 9 Region one: Store the negated clauses that comprise the alleged theorem

```
kind method   type.
type resol    int -> int -> int -> method.
type rlist            list method -> cert.
type rlisti   int ->  list method -> cert.
type rdone                          cert.
type lemma    int -> form -> o.

cut_ke (start C Resol) C1 C2 Cut :- cut_ke (rlist Resol) C1 C2 Cut.
cut_ke (rlist (resol I J K::Rs)) (dlist [I,J]) (rlisti K Rs) Cut :-
  lemma K Cut.
store_kc  (rlisti K Rs) (rlist Rs) (idx K).
decide_ke (rlist []) rdone (idx I).
true_ke   rdone.
```

Fig. 10 Region two: a sequence of cuts leading to the selection of the empty clause

The second phase in building this proof involves translating the resolution steps into uses of the cut rule. An inspection of the inference rules in Figure 6 shows that the only inference rules that are available when the context on the right of the $\Uparrow$ is empty are the rules for cut and for decide. The specification in Figure 10 shows that when the certificate term has `start` as its top-most constructor, that certificate can only be used to build a cut-inference. In particular, if the list of resolvents `R` has $\langle i, j, k\rangle$ as its first triple, then the cut-formula will be taken from the list of lemmas at position k. Note that the predicate `lemma` is not part of the kernel but is code supplied by the resolution refutation certificate: we use this predicate as a convenient storage for the additional resolvents $C_{n+1}, \ldots, C_p$.

If the first resolution step claims that clauses C_i and C_j $(i, j \in \{1, \ldots, n\})$ resolve together to yield C_k then this claim is translated into the following proof fragment. Here, Γ denotes the context $\texttt{(idx 1)} : \neg C_1, \ldots, \texttt{(idx } n\texttt{)} : \neg C_n$.

$$\dfrac{\texttt{(dlist [}i,j\texttt{])} \vdash \Gamma \Uparrow C_k \qquad \dfrac{\texttt{(rlist R)} \vdash \Gamma, \texttt{(idx k)} : \neg C_k \Uparrow}{\texttt{(rlisti R } k\texttt{)} \vdash \Gamma \Uparrow \neg C_k}\ \mathit{store}}{\texttt{(start } (n+1)\texttt{ (resol } i\ j\ k\texttt{::R)} \vdash \Gamma \Uparrow}\ \mathit{cut}$$

Note that the constructor `rlisti` is used to link the store inference rule (which needs an index, here, k) and the cut rule (whose certificate terms contains the needed index). This proof fragment reduces proof checking to two subproblems. The left premise contains the subproblem of showing that the clause C_k is derivable using the context Γ (in particular, using only clauses C_i and C_j). The right premise contains the subproblem of showing that the enlarged sequent $\vdash \texttt{(idx 1)} : \neg C_1, \ldots, \texttt{(idx } (n+1)\texttt{)} : \neg C_{n+1} \Uparrow$ can be checked using the remaining resolution triples. The search for a proof of the right premise will now be guided by the constructor `rlist` which will cause all the resolution triples in its argument to generate, in a similar fashion, additional cut inferences. Once no more triples are stored in the certificate term, that is, the proof

```
type lit              index.
type dlist            list int    -> cert.

all_kc     (dlist L) (x\ dlist L) & orNeg_kc (dlist L) (dlist L).
false_kc   (dlist L) (dlist L) & store_kc (dlist L) (dlist L) lit.
decide_ke  (dlist L) (dlist [J]) (idx I) :- L = [I,J] ; L = [J,I].
decide_ke  (dlist [I]) (dlist []) (idx I).
decide_ke  (dlist L) (dlist []) lit :- L = [I]; L = [].
initial_ke (dlist L) lit.
true_ke    (dlist L).
andPos_ke  (dlist L) (dlist L) (dlist L).
some_ke    (dlist L) (dlist L) T.
release_ke (dlist L) (dlist L).
```

Fig. 11 Region three: a short proof checks an individual resolution step

reconstruction process reduces the proof certificate on the right branch of the proof to the term `(rlist [])`, the final step of that branch is the following proof fragment.

$$\dfrac{\dfrac{}{\texttt{rdone} \vdash \texttt{(idx 1)} : \neg C_1, \ldots, \texttt{(idx } p\texttt{)} : \neg C_p \Downarrow \neg C_i}\ t^+}{\texttt{(rlist [])} \vdash \texttt{(idx 1)} : \neg C_1, \ldots, \texttt{(idx } p\texttt{)} : \neg C_p \Uparrow}\ \mathit{decide}$$

The decide rule expert for the certificate term `(rlist [])` provides no special information as to which index to select to start the focused phase: instead, it is allowed to succeed with *any* index. At the same time, the certificate term is changed to `rdone` and proof checking with this certificate term can succeed if and only if the formula under focus, namely $\neg C_i$ is t^+ and this is only possible if C_i is f^- (the empty clause). Thus, the forced pairing of the decide rule and the t^+ rule guarantees that this branch terminates only if an empty clause has been reached. While we can restrict that empty clause to be C_p, that is not necessary.

We are left with checking a series of subproofs of augmented sequents of the form $\texttt{(dlist [}i,j\texttt{])} \vdash \Gamma \Uparrow C_k$, where the resolution refutation information contained in the triple $\langle i, j, k\rangle$. It is a simple matter to prove the following: if clauses C_i and C_j yield resolvent C_k as a binary resolvent (allowing also factoring), then there exists a shallow, focused proof of the sequent $\vdash \neg C_i, \neg C_j \Uparrow C_k$. In particular, this proof can be characterized as being built by focusing first on $\neg C_i$ then on $\neg C_j$ or by focusing first on $\neg C_j$ then on $\neg C_i$. In either case, that proof may need to be terminated with one additional decide rule but this time on a literal. Thus, such a proof is bounded by at most three decide rules: such proofs will, in principle, be easy to reconstruct. The clerk and expert clauses in Figure 11 actually describe the full details of how such a proof can be built. In particular, left premises of the cut rules will all start with proof fragments that have the following shape.

$$\dfrac{\texttt{(dlist [}i,j\texttt{])} \vdash \Gamma', \mathit{lit} : \rho L_1, \ldots, \mathit{lit} : \rho L_q \Uparrow}{\texttt{(dlist [}i,j\texttt{])} \vdash \Gamma' \Uparrow C_k}$$

Here, we assume the following.

- The clause C_k is of the form $\forall \bar{x}[L_1 \vee^- \ldots \vee^- L_q]$ for a list of variables $\bar{x}$ and a list of literals $L_1, \ldots, L_q$.
- The context Γ' is of the form $\texttt{(idx 1)} : \neg C_1, \ldots, \texttt{(idx } m\texttt{)} : \neg C_m$ for some m such that $n \leq m \leq q$

– The substitution ρ denotes some renaming of the variables in the list $\bar{x}$ and new eigenvariables that have been inserted by the $\forall$-introduction rule.

Thus, this initial phase of proof construction essentially dissolves the clause C_k into eigenvariables and a collection of stored literals. At this point, the decide expert predicate non-deterministically selects either $\neg C_i$ or $\neg C_j$ to start the synchronous phase. Which ever clause is selected, the certificate term will have the corresponding index withdrawn so that the next time the decide rule must select an index, the other index will be selected. Finally, the proof is completed by either attempting to prove $\wedge^+$ or by selecting a literal which has a complement in the same context.

Example 2 Consider the following theorem, written as the disjunction of three negated clauses.

$$\neg(r\ z) \vee^- [\exists x.(r\ x) \wedge^+ \neg(r\ (k\ x))] \vee^- (r\ (k\ (k\ (k\ (k\ z)))))$$

A resolution refutation for this formula can be given by making the following additional assumptions used to name the clauses generated by resolutions.

$$\texttt{lemma}\ 4\ (\forall x.\neg(r\ x) \vee^- (r\ (k\ (k\ x)))).$$
$$\texttt{lemma}\ 5\ (\forall x.\neg(r\ x) \vee^- (r\ (k\ (k\ (k\ (k\ x)))))).$$
$$\texttt{lemma}\ 6\ (r\ (k\ (k\ (k\ (k\ z))))).$$
$$\texttt{lemma}\ 7\ f^-.$$

Finally, the following certificate term denotes a particular resolution refutation that can be used to prove the proposed theorem above.

```
start 1 (resol 2 2 4 :: resol 4 4 5 :: resol 1 5 6 :: resol 6 3 7 :: nil)
```

The indexes 1, 2, and 3 refer to the formulas that are stored in the first steps of the proof construction.

One can specify various generalizations and specializations of this technique, including, for example, hyper-resolution. We present here a simple variation to account for *factoring*. If clause C_j results from factoring in clause C_i, then there is a simple proof of the sequent $\vdash \neg C_i \Uparrow C_j$ and this proof can be guided by the clerk and expert clauses contained in Figure 12. In order to accommodate factoring as a refutation step, the type declarations in Figure 12 include a new construct of type `method`: the certificate term `(factor` i j`)` that claims that clause C_j results from factoring in clause C_i. Similar to the treatment of the `dlist` constructor above, this certificate term instructs the proof checking engine to first decide on $\neg C_i$ and attempt to finish the proof using only decides on literals.

Note that the proof certificate for resolution does not contain any explicit information regarding how quantifiers should be instantiated: while this kind of information can be included in the proof certificate, it seems far more natural and compact to leave it out. This is particularly true in the setting of first-order logic where unification is a simple and decidable operation. An implementation of the proof checking machinery directly in a logic programming language, such as λProlog in these examples, makes the reconstruction of substitution instances a simple feature of the above definition of the resolution refutation format.

One final observation to make about this checker for resolution refutations: while we have described a *sound* checker, one might wish to have a converse guarantee,

```
type factor          int -> int -> method.
type factr                 int    -> cert.
type fdone                           cert.

cut_ke (rlist (factor I K ::Rs)) (factr I) (rlisti K Rs) Cut :-
  lemma K Cut.
all_kc     (factr I) (x\ factr I).
orNeg_kc   (factr I) (factr I) & store_kc (factr I) (factr I) lit.
false_kc   (factr I) (factr I) & decide_ke (factr I) fdone (idx I).
true_ke    fdone               & andPos_ke fdone fdone fdone.
some_ke    fdone fdone T       & decide_ke fdone fdone lit.
release_ke fdone fdone         & store_kc  fdone fdone lit.
initial_ke fdone lit.
```

Fig. 12 Describing how to check factoring between two clauses

namely, that if the checker succeeds with a given certificate term, then that term denotes a proper resolution refutation. That property is not, however, the case for the certificate format that we have described above. For example, while the resolution of $\forall x[p(x) \vee r(f(x))]$ and $\forall x[\neg p(f(x)) \vee q(x)]$ is $\forall x[r(f(f(x))) \vee q(x)]$, the following clause is also provable.

$$\vdash \exists x[\neg p(x) \wedge \neg r(f(x))], \exists x[p(f(x)) \wedge \neg q(x)] \Uparrow \forall x[r(f(f(f(x)))) \vee q(f(x)) \vee s(f(x))].$$

This formula is similar to a resolvent except it uses a unifier that is not most general and it has an additional literal. The proof certificate mechanism above will actually validate this entailment which is not a problem from the point-of-view of soundness.

8 Intuitionistic first-order logic

The structure of sequent calculus proof for intuitionistic logic is usually presented with a two-sided sequent calculus and with the restriction that there is at most one formula on the right-hand side of the sequent [39]. The sequent calculi of classical and intuitionistic logics are close enough to make it possible to give a focusing system for intuitionistic first-order logic that has many similarities with focusing in classical logic (see Section 10 for a detailed relationship between focusing in these two logics). Below we describe the *LJF* proof system for focusing first-order intuitionistic logic [60].

8.1 The *LJF* focused proof system

Focused proof systems involve *polarized* formulas. As we observed in the classical case, both the conjunction and disjunction (and their units) can be polarized either positively or negatively. When moving to the intuitionistic setting, we find that there are, indeed, two polarities for conjunction and for truth which we write as $\wedge^+$, $\wedge^-$, t^+, and t^-. On the other hand, there are not two polarities for $\vee$ but rather the two connectives $\vee^+$ and $\supset$. As a result, we shall simply write $\vee$ instead of $\vee^+$ and write its unit as f instead of f^+. The polarity of $\supset$ is negative and it has no unit. The polarities for $\forall$ and $\exists$ are the same in the intuitionistic setting as in the classical setting.

In *LKF*, the polarity of atomic formulas was set globally to be positive. One could instead arbitrarily assign polarity to atomic formulas although such flexibility is not

Asynchronous Rules

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

$$\frac{\Gamma \Uparrow \vdash [y/x]B \Uparrow}{\Gamma \Uparrow \vdash \forall x.B \Uparrow} \qquad \frac{\Gamma \Uparrow [y/x]B, \Theta \vdash \mathcal{R}}{\Gamma \Uparrow \exists x.B, \Theta \vdash \mathcal{R}} \qquad \frac{}{\Gamma \Uparrow f, \Theta \vdash \mathcal{R}}$$

$$\frac{\Gamma \Uparrow A, B, \Theta \vdash \mathcal{R}}{\Gamma \Uparrow A \wedge^+ B, \Theta \vdash \mathcal{R}} \qquad \frac{\Gamma \Uparrow \Theta \vdash \mathcal{R}}{\Gamma \Uparrow t^+, \Theta \vdash \mathcal{R}} \qquad \frac{\Gamma \Uparrow A, \Theta \vdash \mathcal{R} \quad \Gamma \Uparrow B, \Theta \vdash \mathcal{R}}{\Gamma \Uparrow A \vee B, \Theta \vdash \mathcal{R}}$$

Synchronous Rules

$$\frac{\Gamma \vdash A \Downarrow \quad \Gamma \Downarrow B \vdash R}{\Gamma \Downarrow A \supset B \vdash R} \qquad \frac{\Gamma \vdash A_i \Downarrow}{\Gamma \vdash A_1 \vee A_2 \Downarrow} \qquad \frac{\Gamma \Downarrow A_i \vdash R}{\Gamma \Downarrow A_1 \wedge^- A_2 \vdash R}$$

$$\frac{\Gamma \vdash A \Downarrow \quad \Gamma \vdash B \Downarrow}{\Gamma \vdash A \wedge^+ B \Downarrow} \qquad \frac{}{\Gamma \vdash t^+ \Downarrow} \qquad \frac{\Gamma \Downarrow [t/x]B \vdash R}{\Gamma \Downarrow \forall x.B \vdash R} \qquad \frac{\Gamma \vdash [t/x]B \Downarrow}{\Gamma \vdash \exists x.B \Downarrow}$$

Identity rules

$$\frac{}{\Gamma \Downarrow N_a \vdash N_a}\ init_l \qquad \frac{}{\Gamma, P_a \vdash P_a \Downarrow}\ init_r \qquad \frac{\Gamma \Uparrow \vdash F \Uparrow \qquad \Gamma \Uparrow F \vdash \Uparrow R}{\Gamma \Uparrow \vdash \Uparrow R}\ cut$$

Structural rules

$$\frac{\Gamma, N \Downarrow N \vdash R}{\Gamma, N \Uparrow \vdash \Uparrow R}\ decide_l \qquad \frac{\Gamma \vdash P \Downarrow}{\Gamma \Uparrow \vdash \Uparrow P}\ decide_r \qquad \frac{\Gamma \Uparrow P \vdash \Uparrow R}{\Gamma \Downarrow P \vdash R}\ release_l \qquad \frac{\Gamma \Uparrow \vdash N \Uparrow}{\Gamma \vdash N \Downarrow}\ release_r$$

$$\frac{C, \Gamma \Uparrow \Theta \vdash \mathcal{R}}{\Gamma \Uparrow C, \Theta \vdash \mathcal{R}}\ store_l \qquad \frac{\Gamma \Uparrow \vdash \Uparrow D}{\Gamma \Uparrow \vdash D \Uparrow}\ store_r$$

Here, P is positive, N is negative, C is a negative formula or positive atom, D a positive formula or negative atom, N_a is a negative atom, and P_a is a positive atom. Other formulas are arbitrary.

Fig. 13 The intuitionistic sequent calculus *LJF*.

necessary in a classical setting: for example, if one wishes to have the atomic formula p positive but the atomic formula q negative, one could simply change all occurrences of q to $\neg q$ and once again assume that q is positively polarized. Note that the intuitionistic negation $B \supset f$ is always negative no matter the polarity of the formula B. Thus in intuitionistic logic, the option of assigning polarities to atomic formulas arbitrarily is an important choice.

The synchronous sequent in *LJF* comes in two kinds, namely, the left focused sequent $\Gamma \Downarrow B \vdash R$ and the right focused sequent $\Gamma \vdash B \Downarrow$. In both of these cases, the formula B is under focus. Asynchronous sequents also come in two kinds, namely, $\Gamma \Uparrow \Theta \vdash R \Uparrow$ or $\Gamma \Uparrow \Theta \vdash \Uparrow R$. In both the synchronous and asynchronous sequents, B and R are formulas and the zone marked with Γ is a multiset of formulas while the zones in asynchronous sequents marked with Θ are lists of formulas. We shall sometimes write an asynchronous sequent as $\Gamma \Uparrow \Theta \vdash \mathcal{R}$ where $\mathcal{R}$ can be of the form $\Uparrow R$ or $R \Uparrow$.

The inference rules for *LJF* are given in Figure 13. The basic structure of *LJF* is rather similar to that of *LKF* in that they both classify inference rules as either identity rules (initial and cut), structural rules (*decide*, *release*, and *store*), or introduction rules (divided among asynchronous and synchronous rules). Let B be a (polarized) intuitionistic formula and let Π be an *LJF* proof of the (end)sequent $\Uparrow \vdash B \Uparrow$. We list some invariants that hold for the various sequents that appear in Π.

1. Every occurrence of the sequent $\Gamma \Uparrow \Theta \vdash \mathcal{R}$ or the sequent $\Gamma \Downarrow B \vdash R$ in Π is such that Γ is a multiset of negative formulas and positive atoms.
2. Similarly, every occurrence of the sequent $\Gamma \Uparrow \Theta \vdash \Uparrow R$ or the sequent $\Gamma \Downarrow B \vdash R$ in Π is such that R is a positive formula or a negative literal.
3. If $\Gamma \Uparrow \Theta \vdash R \Uparrow$ is the conclusion of a right-introduction rule, then Θ is empty.

The asynchronous introduction rules can actually be applied in any order without changing the shape of focused proofs when one ignores the internal structure of phases. As a result, we fix the order of processing asynchronous inferences in the following natural order. First, elements of the list in the zone labeled Θ are introduced in the order that they appear in that list. Second, an asynchronous introduction on the right is applied only when the Θ zone is empty.

Besides the fact that sequents in *LKF* and *LJF* different in that the latter has two sides and four zones, there are a couple of other key differences between these two proof systems.

- While *LKF* provides two polarities of disjunction, *LJF* contains only the positive disjunction while also allowing for the (negative polarity) implication.
- The *store* inference rule of *LKF* associates a formula with an index and that formula remains stored with that index in all sequents appearing above that occurrence of *store* rule. In *LJF*, the $store_l$ rule associates formulas to indexes is a similar fashion. However, the $store_r$ rule stores the right-hand formula in a linear fashion: that is, there can be at most one formula stored on the right in any sequent and the $decide_r$ rule removes that formula from storage. Thus, there is no need to provide an index to the formula stored on the right since there is only *the* right-stored formula.

Despite these differences, it is possible to formally relate these two proof systems. More about that relationship will be described in Section 10.

8.2 The LJF^a focused proof system

Following the design of the LKF^a proof system, we present the result of augmenting *LJF* with suitable notions of clerks, experts, and indexes in order to get the LJF^a proof system of Figure 14. Note that, as in Section 5, we have added calls to clerk predicates to all premises of all asynchronous introduction rules as well as the *storeL* and *storeR* rules. Similarly, we have added calls to expert predicates to all other inference rules, in particular, the synchronous introduction rules as well as the identity rules and the release structural rules. We have also added an indexing mechanism that labels formulas that are placed on the left of the sequents: that is, in LJF^a, the Γ zone (on the left) is a multiset of pairs $\langle l, B\rangle$ where l is an index and B is a formula. Such indexes are used by clerks and experts to refer to formulas. In particular, the $storeL_c(\cdot,\cdot,\cdot)$ predicate is responsible for computing an index for a stored formula while the $decideL_e(\cdot,\cdot,\cdot)$ predicate is responsible for identifying a formula on which to focus by providing an index for such a formula. We do not require that a context Γ containing an association of indexes to formulas is functional: it is certainly possible and useful for there to be a one-to-many relationship between indexes and formulas. Finally, note that the store and decide structural rules also function on the right-hand-side of a sequent: in these cases, there is exactly one formula on the right so no indexing mechanism is needed to refer to it.

Asynchronous Rules

$$\frac{\Xi_1 : \Gamma \Uparrow A \vdash B \Uparrow \quad \supset_c (\Xi_0, \Xi_1)}{\Xi_0 : \Gamma \Uparrow \vdash A \supset B \Uparrow} \qquad \frac{(\Xi_1 y) : \Gamma \Uparrow \vdash [y/x]B \Uparrow \quad \forall_c(\Xi_0, \Xi_1)}{\Xi_0 : \Gamma \Uparrow \vdash \forall x.B \Uparrow}$$

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

$$\frac{\Xi_1 : \Gamma \Uparrow A, B, \Theta \vdash \mathcal{R} \quad \wedge_c^+(\Xi_0, \Xi_1)}{\Xi_0 : \Gamma \Uparrow A \wedge^+ B, \Theta \vdash \mathcal{R}} \qquad \frac{\Xi_1 : \Gamma \Uparrow \Theta \vdash \mathcal{R} \quad t_c^+(\Xi_0, \Xi_1)}{\Xi_0 : \Gamma \Uparrow t^+, \Theta \vdash \mathcal{R}}$$

$$\frac{\Xi_1 : \Gamma \Uparrow A, \Theta \vdash \mathcal{R} \quad \Xi_2 : \Gamma \Uparrow B, \Theta \vdash \mathcal{R} \quad \vee_c(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0 : \Gamma \Uparrow A \vee B, \Theta \vdash \mathcal{R}} \qquad \frac{f_c^+(\Xi_0)}{\Xi_0 : \Gamma \Uparrow f, \Theta \vdash \mathcal{R}}$$

$$\frac{(\Xi_1 y) : \Gamma \Uparrow [y/x]B, \Theta \vdash \mathcal{R} \quad \exists_c(\Xi_0, \Xi_1)}{\Xi_0 : \Gamma \Uparrow \exists x.B, \Theta \vdash \mathcal{R}}$$

Synchronous Rules

$$\frac{\Xi_1 : \Gamma \vdash A \Downarrow \quad \Xi_2 : \Gamma \Downarrow B \vdash R \quad \supset_e (\Xi_0, \Xi_1, \Xi_2)}{\Xi_0 : \Gamma \Downarrow A \supset B \vdash R}$$

$$\frac{\Xi_1 : \Gamma \vdash A_i \Downarrow \quad \vee_e(\Xi_0, \Xi_1, i)}{\Xi_0 : \Gamma \vdash A_1 \vee A_2 \Downarrow} \qquad \frac{\Xi_1 : \Gamma \Downarrow A_i \vdash R \quad \wedge_e^-(\Xi_0, \Xi_1, i)}{\Xi_0 : \Gamma \Downarrow A_1 \wedge^- A_2 \vdash R}$$

$$\frac{\Xi_1 : \Gamma \vdash A \Downarrow \quad \Xi_2 : \Gamma \vdash B \Downarrow \quad \wedge_e^+(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0 : \Gamma \vdash A \wedge^+ B \Downarrow} \qquad \frac{t_e^+(\Xi_0)}{\Xi_0 : \Gamma \vdash t^+ \Downarrow}$$

$$\frac{\Xi_1 : \Gamma \vdash [t/x]B \Downarrow \quad \exists_e(\Xi_0, \Xi_1, t)}{\Xi_0 : \Gamma \vdash \exists x.B \Downarrow} \qquad \frac{\Xi_1 : \Gamma \Downarrow [t/x]B \vdash R \quad \forall_e(\Xi_0, \Xi_1, t)}{\Xi_0 : \Gamma \Downarrow \forall x.B \vdash R}$$

Identity rules

$$\frac{\mathit{initL}_e(\Xi_0)}{\Xi_0 : \Gamma \Downarrow N_a \vdash N_a} \qquad \frac{(l, P_a) \in \Gamma \quad \mathit{initR}_e(\Xi_0, l)}{\Xi_0 : \Gamma \vdash P_a \Downarrow}$$

$$\frac{\Xi_1 : \Gamma \Uparrow \vdash F \Uparrow \quad \Xi_2 : \Gamma \Uparrow F \vdash \Uparrow R \quad \mathit{cut}_e(\Xi_0, \Xi_1, \Xi_2, F)}{\Xi_0 : \Gamma \Uparrow \vdash \Uparrow R}$$

Structural rules

$$\frac{l : N \in \Gamma \quad \Xi_1 : \Gamma \Downarrow N \vdash R \quad \mathit{decideL}_e(\Xi_0, \Xi_1, l)}{\Xi_0 : \Gamma \Uparrow \vdash \Uparrow R} \qquad \frac{\Xi_1 : \Gamma \vdash P \Downarrow \quad \mathit{decideR}_e(\Xi_0, \Xi_1)}{\Xi_0 : \Gamma \Uparrow \vdash \Uparrow P}$$

$$\frac{\Xi_1 : \Gamma \Uparrow P \vdash \Uparrow R \quad \mathit{releaseL}_e(\Xi_0, \Xi_1)}{\Xi_0 : \Gamma \Downarrow P \vdash R} \qquad \frac{\Xi_1 : \Gamma \Uparrow \vdash N \Uparrow \quad \mathit{releaseR}_e(\Xi_0, \Xi_1)}{\Xi_0 : \Gamma \vdash N \Downarrow}$$

$$\frac{\Xi_1 : l : C, \Gamma \Uparrow \Theta \vdash \mathcal{R} \quad \mathit{storeL}_c(\Xi_0, \Xi_1, l)}{\Xi_0 : \Gamma \Uparrow C, \Theta \vdash \mathcal{R}} \qquad \frac{\Xi_1 : \Gamma \Uparrow \vdash \Uparrow D \quad \mathit{storeR}_c(\Xi_0, \Xi_1)}{\Xi_0 : \Gamma \Uparrow \vdash D \Uparrow}$$

Fig. 14 The augmented intuitionistic sequent calculus LJF^a. Here, P is positive, N is negative, C is a negative formula or positive atom, D a positive formula or negative atom, N_a is a negative atom, and P_a is a positive atom. Other formulas are arbitrary.

8.3 Simply typed λ-terms as proof certificates

It is a well-known observation that simply typed λ-terms can be seen as proofs (usually, as natural deduction proofs in minimal logic) of the formulas that are encoded as types. Thus, we should expect that a typed λ-term can be used as a proof certificate for its type. We illustrate this by showing how simply typed λ-terms in η-long β-normal form can be used as certificates.

A simple type is an expression built from *primitive types*, say, i, j, and k, and the *function type constructor*, $\rightarrow$. If we identify primitive types with propositional variables and $\rightarrow$ with intuitionistic implication, then simple type expressions can be identified with propositional intuitionistic formulas. For example, the type expressions $i \rightarrow j$ and $(i \rightarrow i) \rightarrow i \rightarrow i$ correspond to the propositional formulas $i \supset j$ and $(i \supset i) \supset i \supset i$. Note that the first of these types is not a theorem while the second is.

A simply typed λ-term is in η-long β-normal form if it can be written in the form $\lambda x_1 \ldots \lambda x_n.(h\ t_1 \ldots t_m)$ where $n \geq 0$ and $m \geq 0$, $h, x_1, \ldots, x_n$ are variables, and $t_1, \ldots, t_m$ is a list of terms that are in η-long β-normal form. Furthermore, for a term to be in "η-long form", the type of the term $(h\ t_1 \ldots t_m)$ must be a primitive type.

A representation of simply typed λ-terms that is suitable as a proof certificate is the "nameless dummy" presentation of the λ-terms of de Bruijn [26]. In such a representation of (untyped) λ-terms, a variable occurrence is replaced by a number which indicates the number of intervening bindings between that occurrence and the actual binder for that variable. Instead of formally defining this familiar concept, we present in Figure 15 a few examples of λ-terms and their representation using de Bruijn indexes along with their simple type. A quick reflection on the structure of terms using

$\lambda x.x$	$\lambda 0$	$i \rightarrow i$
$\lambda x \lambda y.y$	$\lambda\lambda 0$	$j \rightarrow i \rightarrow i$
$\lambda x \lambda y.(x\ y)$	$\lambda\lambda(1\ 0)$	$(i \rightarrow j) \rightarrow i \rightarrow j$
$\lambda x \lambda y \lambda z.(z\ (z\ x))$	$\lambda\lambda\lambda(0\ (0\ 2))$	$i \rightarrow j \rightarrow (i \rightarrow i) \rightarrow i$
$\lambda x \lambda y.y\ (\lambda z.(z\ x))$	$\lambda\lambda(0\ (\lambda(0\ 2)))$	$i \rightarrow (((i \rightarrow j) \rightarrow j) \rightarrow k) \rightarrow k$

Fig. 15 A few λ-terms with their representation using de Bruijn indexes and a simple type.

de Bruijn indexes reveals that if we are using such a term to check an intuitionistic formula, the λ symbol corresponds to the invertible implication-right introduction. As a result, its presence in the proof certificate is not necessary. The key information, however, is the name (i.e., the offset) of the head variable of the λ-terms: the type of that head variable provides the formula on which the *decideL* inference rule focuses.

This observation suggests that a rather simple design for proof certificates can be based on a nested structure of pairs, lists, and integers. In particular, let t be a λ-term in η-long β-normal form, let θ be a function from the free variables of t to the non-negative integers, and let d be a non-negative integer. We now define by recursion the function $[\![\theta \mid t]\!]_d$ via the equation

$$[\![\theta \mid \lambda x_1 \ldots \lambda x_n(h\ t_1 \ldots t_m)]\!]_d = \langle \theta' h, [[\![\theta' \mid t_1]\!]_{d'}, \ldots, [\![\theta' \mid t_m]\!]_{d'}] \rangle.$$

Here, $d' = d + n$ and θ' is the function θ extended so that for $i \in \{1, \ldots, n\}$, θ' maps x_i to $d + i$. The structure that results from such a computation is a recursive structure based on the pairing of an integer with a list (possibly empty) of such structures. The result of computing $[\![\theta_0 \mid t]\!]_0$ on the five λ-terms in Figure 15 yields

$$\langle 0, [] \rangle, \quad \langle 0, [] \rangle, \quad \langle 1, [\langle 0, [] \rangle] \rangle, \quad \langle 0, [\langle 0, [\langle 2, [] \rangle] \rangle] \rangle, \quad \text{and} \quad \langle 0, [\langle 0, [\langle 2, [] \rangle] \rangle] \rangle.$$

Note that the first two and last two structures are identical. Since the input λ-terms are closed, the function θ_0 can be taken to be any function from variables to natural numbers.

```
kind deb             type.
type apply           int -> list deb -> deb.
type lc              int ->      deb -> cert.
type args            int -> list deb -> cert.
type idx                            int -> index.

arr_jc       (lc C D) (lc C D).
storeR_jc    (lc C D) (lc C D).
releaseR_je  (lc C D) (lc C D).
storeL_jc    (lc C D) (lc C' D) (idx C) :- C' is C + 1.
decideL_je   (lc C (apply H A)) (args C A) (idx V) :- V is C - H - 1.
initialL_je  (args C []).
arr_je       (args C (A::As)) (lc C A) (args C As).
```

Fig. 16 The FPC definition of the proof evidence of simply typed η-long β-normal λ-terms.

Figure 16 provides an FPC definition of η-long β-normal terms as proof evidence for this propositional intuitionistic logic. Note that the type `deb` captures the nested structure we have just described (the constructor `apply` corresponds to pairing in the definition of $[\![\cdot \mid \cdot]\!]$ above). Certificates—inhabitants of the `cert` type—are built using the two constructors `lc` and `args`. In this FPC, all atomic formulas are given negative polarity and the indexing structure is used to associate level counts (needed to compute offsets) with assumed (stored) formulas. When providing names for clerks and experts in this figure, we have used `arr` to denote implication, and we have used `_j` in all names to signify that they belong to an intuitionistic logic proof system.

Formally speaking, let t be a λ-term in η-long β-normal form with simple type τ. Let D be the term $[\![\theta_0 \mid t]\!]_0$ and let B be the intuitionistic formula that results from replacing $\rightarrow$ with $\supset$ and the primitive types with atomic formulas. Then it must be the case that the sequent

$$(\texttt{lc}\ 0\ D)\colon\ \cdot \Uparrow \cdot \vdash B \Uparrow \cdot$$

is provable in the LJF^a proof system. It is also easy to see that this check is, in fact, a decision procedure since the certificate associated to sequents gets smaller as we move from one phase to the next.

It is important to realize, however, that the converse of the statement above does not hold. That is, it is possible for the sequent $(\texttt{lc}\ 0\ D)\colon\ \cdot \Uparrow \cdot \vdash B \Uparrow \cdot$ to be provable even when D does not encode a simply typed λ-term of a type that corresponds to B. For example, if B is t (the true constant), then this sequent is always provable even if D is, say, $\langle 0, [\langle 0, [] \rangle] \rangle$, a term that encodes the self application $\lambda x(x\ x)$. The focus of this FPC is the theorem-hood of a formula and not the structure of λ-terms. While that structure might be used to guide the proof of a formula, this FPC may not actually analyze the full structure of that term. Using techniques described in Section 9, it is certainly possible to design an FPC that can validate a statement such as "t is a typed term of type τ" instead of the statement "B is an intuitionistic formula".

8.4 Variation on certificates based on de Bruijn indexes

A common representation of λ-terms based on de Bruijn indexes contains more information than the certificate structures presented above: in particular, the exact placements of λ's within a term are usually part of the syntax (as in Figure 15). What if one wants

to require more of the structure of the λ-term to be encoded within a certificate and checked? Since the occurrences of λ in terms based on de Bruijn indexes corresponds to the right-introduction of implication and since the certificate does not need to be consulted during the asynchronous phase, we will assume that a properly polarized formula forces a polarity shift by placing a *positive delay* ($\varrho(\cdot)$) around all positive occurrences of implication. This positive delay can be seen as either a new constructor of polarized formulas or it can be defined by, say, $\varrho(B) = B \wedge^+ t^+$. The formula $\varrho(B)$ is positive no matter what the polarity of B is. We also introduce an additional constructor

```
type lambda          deb -> deb.
```

for encoding such extended term structures. We also need to add one new clause to handle the new structure of certificate terms.

```
storeR_jc   (lc C (lambda D)) (lc C (lambda D)).
decideR_je  (lc C (lambda D)) (lc C (lambda D)).
andPos_je   (lc C (lambda D)) (lc C D) (lc C (lambda D)).
true_je     (lc C D).
```

It is also possible to accommodate simply typed λ-terms that are not in η-long β-normal form. In order to handle the lack of β-normal forms, the cut inference rule is needed and to handle the lack of η-long forms, it is necessary to check that $B \vdash B$ holds for non-atomic formulas. We now present the *mimic* FPC which can be used for this purpose.

8.5 The mimic FPC

One way to prove the (unfocused) sequent $B \vdash B'$ is to check if B and B' are, in fact, equal as formulas. In that case, this sequent is provable using Gentzen's initial rule. In our focused proof systems, however, the initial rule can be used only for atomic formulas. We present an FPC that will prove this sequent whenever B and B' are equal. To see how this FPC can be organized, consider proving the fact that restricting initial rules to only be atomic initial rules (i.e., instances of the initial rule in which the formula on the right is atomic and is present on the left) does not lose completeness. The proof of this theorem proceeds by induction on the structure of formulas and with proof figures such as

$$\dfrac{\dfrac{B \vdash B \qquad C \vdash C}{B \supset C, B \vdash C}}{B \supset C \vdash B \supset C} \qquad \text{and} \qquad \dfrac{\dfrac{B \vdash B \qquad C \vdash C}{B, C \vdash B \wedge C}}{B \wedge C \vdash B \wedge C}\ .$$

Showing the completeness of atomic initials is significantly simpler using Gentzen's original LJ proof system than it is using the focused *LJF* system directly. As these two displayed derivations show, we simply need to pair a left and right introduction rule together in order to make the formulas involved in an initial rule smaller. In a focused proof system, however, a number of introduction rules of one phase (say, $\supset$-right, $\vee^+$-left then $\wedge^-$-right) are done together before the corresponding rules ($\supset$-left, $\vee^+$-right then $\wedge^-$-left) can be done. The clerks must record the steps taken in the asynchronous phase so that the experts can follow those steps.

More formally, the mimic FPC deals with sequents of the following form $\Gamma \Uparrow F \vdash F \Uparrow$. Depending on the polarity of F, one of the two occurrences will be immediately

```
type root     index.
type mL, mR   index -> index.
type mimic                                          index -> cert.
type aphaseL  index -> list index -> list index -> index -> cert.
type aphaseR  index -> list index               -> index -> cert.
type sphaseL           list index ->      index -> index -> cert.
type sphaseR           list index               -> index -> cert.

arr_jc (mimic I) (aphaseL I [] [I] I).
decideL_je (aphaseR I Cs X) (sphaseL Cs I X) I.
decideR_je (aphaseR I Cs I) (sphaseR Cs   I).
storeL_jc (aphaseL Rt Cs [I,J|R] X) (aphaseL Rt Cs [J|R] X) I.
storeL_jc (aphaseL Rt Cs [I] X)     (aphaseR Rt Cs X) I.
storeR_jc (aphaseR Rt Cs X)         (aphaseR Rt Cs X).
initialL_je (sphaseL _ I I).
initialR_je (sphaseR _   I) I.
releaseL_je (sphaseL _ I X) (aphaseL I [] [I] X).
releaseR_je (sphaseR _   I) (aphaseR I []     I).
arr_jc    (aphaseR Rt Cs I) (aphaseL Rt Cs [mL I] (mR I)).
andNeg_jc (aphaseR Rt Cs I) (aphaseR Rt [mL I|Cs] (mL I))
                            (aphaseR Rt [mR I|Cs] (mR I)).
andPos_jc (aphaseL Rt Cs [I|R] X) (aphaseL Rt Cs [mL I, mR I|R] X).
true_jc   (aphaseL Rt Cs [I,J|R] X) (aphaseL Rt Cs [J|R] X).
true_jc   (aphaseL Rt Cs [I] X)     (aphaseR Rt Cs X).
or_jc     (aphaseL Rt Cs [I|R] X) (aphaseL Rt [mL I|Cs] [mL I|R] X)
                                  (aphaseL Rt [mR I|Cs] [mR I|R] X).
arr_je    (sphaseL Cs I X) (sphaseR Cs (mL I)) (sphaseL Cs (mR I) X).
andPos_je (sphaseR Cs   I) (sphaseR Cs (mL I)) (sphaseR Cs (mR I)).
true_je   (sphaseR Cs I).
or_je     (sphaseR Cs I)   (sphaseR Cs' (mL I))   left  &
andNeg_je (sphaseL Cs I X) (sphaseL Cs' (mL I) X) left  :-
          memb_and_rest (mL I) Cs Cs'.
or_je     (sphaseR Cs I)   (sphaseR Cs' (mR I))   right &
andNeg_je (sphaseL Cs I X) (sphaseL Cs' (mR I) X) right :-
          memb_and_rest (mR I) Cs Cs'.
```

Fig. 17 The mimic FPC for intuitionistic logic

stored (call it the *mirror* F). As said before, the output (in terms of sequents) of an asynchronous phase is uniquely determined by its input sequent. These sequents will be of the form $\Gamma, F_1, .., F_n \Uparrow \vdash \Uparrow F'$ where F_i are negative or atomic subformulas of F, F' is a positive or atomic subformula of F, and where exactly one of these subformulas is the *mirror* F which was stored at the beginning of this asynchronous phase. The derivation then continues with a *decide* rule on this *mirror* F, and then introduces exactly the same connectives as the previous phase and ends at the same subformulas, either with an *init* rule (if the subformulas are atomic) or a *release* rule (if the subformulas are not atomic). Indeed, one notices that if a formula can be subject to a *store* rule on the left (resp. the right) then that same formula is subject to a *release* or *init* rule if it appears on the right (resp. the left). Furthermore, a formula that can be subject to a *release* on the left (resp. the right) can be subject to a *decide* rule on the right (resp. the left). This appears clearly when one examines the behavior of the *release* experts of Figure 17, recording the current index `I` as the first parameter of the `aphaseL` and `aphaseR cert` constructors, and the behavior of the *decide* experts using precisely that parameter to start the focused phase.

9 Checking proofs instead of provability

In the previous sections, we placed primary concern on checking whether or not a certain formula is a theorem (of intuitionistic or classical logic). Proof evidence was used to guide the proof checking and proof reconstruction steps. In the end, when the checker has successfully executed a given FPC over a given proof certificate, the only guarantee our kernel provides is that the formula is, in fact, a theorem. The kernel, in and of itself, does not guarantee any other properties about certificates.

In this section, we illustrate that it is possible to write FPCs that succeed only if the proof evidence it contains satisfies certain strong structural restrictions.

9.1 Justified Horn clause proofs

By Horn clause, we shall mean a closed formula of the form

$$\forall x_1 \ldots \forall x_m (A_1 \supset \ldots \supset A_n \supset A_0) \qquad (m \geq 0,\ n \geq 0)$$

where $A_0, \ldots, A_n$ are atomic formulas. If m is 0, we do not write any universal quantifiers and if n is 0, we write no implications. Thus, if $m = n = 0$ then the displayed Horn clause is a (closed) atomic formula. By a *Horn clause entailment* we shall mean a formula of the form $H_1 \supset \cdots \supset H_n \supset A$ where A is an atomic formula and $H_1, \ldots, H_n$ ($n \geq 1$) are Horn clauses.

We now wish to introduce a simple proof structure, which we called here a *justified Horn clause proof*. We first illustrate this kind of proof with an example.

Example 3 Considered the following sequence of indexed Horn clauses that describes the adjacency graph of a simple graph and defines the relationship of path within that graph.

(1)	adj a b.
(2)	adj b a.
(3)	adj a c.
(4)	adj c d.
(5)	$\forall x \forall y (\text{adj } x\ y \supset \text{path } x\ y)$.
(6)	$\forall x \forall y \forall z (\text{path } x\ y \supset \text{path } y\ z \supset \text{path } x\ z)$.

A consequence of these Horn clauses is the atom (path a d); that is, the Horn clause entailment (1) $\supset \ldots \supset$ (6) $\supset$ path a d is a theorem of classical (and intuitionistic) logic. Given that the underlying graph has a cycle and that the construction of paths is associative, there are a number of proofs of this entailment. The following sequence of triples, containing a label, an atomic formula and a *justification* is an example of a justified Horn clause proof.

(7)	path a b	$\langle 5, [1] \rangle$	(11)	path a c	$\langle 6, [9, 10] \rangle$
(8)	path b a	$\langle 5, [2] \rangle$	(12)	path c d	$\langle 5, [4] \rangle$
(9)	path a a	$\langle 6, [7, 8] \rangle$	(13)	path a d	$\langle 6, [11, 12] \rangle$
(10)	path a c	$\langle 5, [3] \rangle$			

In order to formally define what we mean by justified Horn clause proofs, we will use the machinery of FPCs (polarization, clerks, and experts). We must first define the syntax of intuitionistic formulas used to form Horn clauses. For this, we shall use the kind and type declarations

```
kind just                                                    type.
type tup        index -> iform -> index -> list index -> just.

type i                                              int -> index.
type terminal                                              index.

type load                     list index -> list just -> cert.
type jlist                                  list just -> cert.
type apply                        index -> list index -> cert.
type args                                 list index -> cert.
type finish                                    index -> cert.
type initL, done                                         cert.
```

Fig. 18 The constants used by the FPC for checking justified Horn clause proofs

```
kind iform, i        type.
type imp             iform -> iform -> iform.
type forall          (i -> iform) -> iform.
```

Here, formulas will have the type `iform` and first-order quantifiers range over the type `i`. The predicates denoting adequacy and path are declared the following type.

```
type adj, path    i -> i -> iform.
```

Next consider the constructors declared in Figure 18. Clearly, indexes are either based on numbers (as in Example 3) or are equal to the internally used index `terminal`. The type `just` represent the individual steps of a justified proof: thus, the proof evidence in Example 3 can be written as the following term of type `(list just)`:

```
[  tup (i  7) (path a b) (i 5) [i 1],
   tup (i  8) (path b a) (i 5) [i 2],
   tup (i  9) (path a a) (i 6) [i 7, i 8],
   tup (i 10) (path a c) (i 5) [i 3],
   tup (i 11) (path a c) (i 6) [i 9, i 10],
   tup (i 12) (path c d) (i 5) [i 4],
   tup (i 13) (path a d) (i 6) [i 11, i 12]  ].
```

There are six constructors for building certificates and these will correspond to six different parts of the proof reconstruction process that happens during checking of this proof. We describe each of these different constructors and their roles separately.

The `load` *constructor.* Since the theorems that we are attempting to prove have the form $H_1 \supset \cdots \supset H_n \supset A$, the asynchronous phase is a series of n alternations of $\supset_r$ and S_l rules, such as those displayed here:

$$\dfrac{\dfrac{\Gamma, H \Uparrow \vdash G \Uparrow}{\Gamma \Uparrow H \vdash G \Uparrow}\, S_l}{\Gamma \Uparrow \vdash H \supset G \Uparrow}\, \supset_r$$

The term guiding this part of the proof construction has the form `(load Is Js)` where `Is` is a list of the indexes to be assigned (by the `store_jc` clerk) to the Horn clause assumptions. In our example above, this list is

```
[i 1, i 2, i 3, i 4, i 5, i 6]
```

Notice that the only clerk and expert predicates defined for the constructor `load` are `arr_jc`, `storeL_jc`, and (when the list of indexes is exhausted) `storeR_jc`.

```
arr_jc       (load (I::Tlabs) Justs) (load (I::Tlabs) Justs).
storeL_jc    (load (I::Tlabs) Justs) (load Tlabs Justs) I.
storeR_jc    (load nil Justs) (jlist Justs).
cut_je       (jlist (tup I Atom Rule Premises::Justs))
             (apply Rule Premises)
             (jlist (tup I Atom Rule Premises::Justs))
             (p Atom).
storeL_jc    (jlist (tup I Atom Rule Premises::Justs))
             (jlist Justs) I.
decideR_je   (jlist nil) done.
initialR_je  done I.
storeR_jc    (apply Rule Premises) (apply Rule Premises).
decideL_je   (apply Rule Premises) (args Premises) Rule.
all_je       (args Premises) (args Premises) T.
arr_je       (args (P::Premises)) (finish P) (args Premises).
releaseL_je  (args nil) initL.
storeL_jc    initL initL terminal.
decideR_je   initL initL.
initialR_je  initL terminal.
initialR_je  (finish P) P.
```

Fig. 19 The clerks and experts used for defining justified Horn clause proofs

The `jlist` *constructor.* This constructor is used to build a sequent of cut rules, one for each of the justifications in the list that is the argument of the `jlist` constructor. In particular, when this constructor is paired with a non-empty set of justifications, then the `cut_je` expert and the `storeL_jc` clerk are defined and are responsible for building proofs of the following form.

$$\dfrac{\Pi_1 \quad \dfrac{\Pi_2}{\Gamma, B \Uparrow \vdash \Uparrow A}}{\Gamma \Uparrow \vdash B \Uparrow \quad \Gamma \Uparrow B \vdash \Uparrow A}\ S_l \qquad \Gamma \Uparrow \vdash \Uparrow A\ Cut$$

The proof Π_1 is constructed (as described below) by applying a Horn clause to some atomic formulas. The proof Π_2 accumulates that formula B as a new assumption and then continues to insert cuts as long as there are justifications (arguments to the `jlist` constructor). Notice that the conclusion of this derivation is the same as the conclusion of the subproof Π_2 except that the atomic formula B is added to the left-hand context. When there are no more justifications remaining, the `decideR_je` expert shifts the certificate term to `done` and the only clerk or expert that can be used with that constructor is the `initialR_je` expert. If that instance of the initial rule succeeds, the formula stored on the right (A in the clauses illustrated above) must also be on the left: that is, A was either one of the original assumptions or it was inserted by a previous cut rule.

The `apply`*,* `args`*, and* `finish` *constructors.* These three constructors are used to build the proofs that arise on the left premise of the cut rule above (the proof labeled Π_1 above). By way of example, consider the proof in Figure 20 that the formula labeled by (9) (i.e., path a a) follows from transitivity of the path predicate (the formula labeled (6)) and formulas labeled (7) and (8).

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \text{path } a\ b \Downarrow}\ I_r \quad \dfrac{\dfrac{}{\Gamma \vdash \text{path } b\ a \Downarrow}\ I_r \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \text{path } a\ a \vdash \text{path } a\ a \Downarrow}\ I_r}{\Gamma, \text{path } a\ a \Uparrow \vdash \Uparrow \text{path } a\ a}\ D_r}{\Gamma \Uparrow \text{path } a\ a \vdash \Uparrow \text{path } a\ a}\ S_l}{\Gamma \Downarrow \text{path } a\ a \vdash \text{path } a\ a}\ R_l}{\Gamma \Downarrow \text{path } b\ a \supset \text{path } a\ a \vdash \text{path } a\ a}\ \supset_l}{\Gamma \Downarrow \text{path } a\ b \supset \text{path } b\ a \supset \text{path } a\ a \vdash \text{path } a\ a}\ \supset_l}{\Gamma \Downarrow \forall x \forall y \forall z . \text{path } x\ y \supset \text{path } y\ z \supset \text{path } x\ z \vdash \text{path } a\ a}\ \forall_r \times 3}{\Gamma \Uparrow \vdash \Uparrow \text{path } a\ a}\ D_l}{\Gamma \Uparrow \vdash \text{path } a\ a \Uparrow}\ S_r
$$

Fig. 20 A focused proof about paths

Note that the two initial rules on the left of this proof are guided by the terms `(finish (i 7))` and `(finish (i 8))`, respectively. Since these terms explicitly mention certain indexes, the premises to the transitivity axiom cannot be just any two path-atoms that precede the current one in the proof but must be the specific ones labeled by the indexes. Thus, this FPC is not just checking that a particular formula is provable, it is also controlling many of the fine points in how it is proved.

9.2 Frege proofs

The example of checking justified Horn clause proofs was introduced in the previous section in order to make the checking of Frege proofs immediate. A Frege proof in propositional logic is a list of formulas such that every element of that list is either the instance of an axiom schema or is the result of an inference rule applied to one or more formulas that appear before it in that list.

By way of example, consider the following three axiom schemas for propositional classical logic.

(1)	$X \supset Y \supset X$
(2)	$((X \supset (Y \supset Z)) \supset ((X \supset Y) \supset (X \supset Z))))$
(3)	$(((X \supset \bot) \supset \bot) \supset X))$

Together with the rule of modus ponens, we have a Frege proof of the formula $(\bot \supset w)$ (taken from [54]).

(4)	$((w \supset \bot) \supset \bot) \supset w$	by (3)
(5)	$(((w \supset \bot) \supset \bot) \supset w) \supset (\bot \supset (((w \supset \bot) \supset \bot) \supset w))$	by (1)
(6)	$\bot \supset (((w \supset \bot) \supset \bot) \supset w)$	by *mp* (4), (5)
(7)	$(\bot \supset (((w \supset \bot) \supset \bot) \supset w)) \supset ((\bot \supset ((w \supset \bot) \supset \bot)) \supset (\bot \supset w))$	by (2)
(8)	$(\bot \supset ((w \supset \bot) \supset \bot)) \supset (\bot \supset w)$	by *mp* (6), (7)
(9)	$\bot \supset ((w \supset \bot) \supset \bot)$	by (1)
(10)	$\bot \supset w$	by *mp* (9), (8)

We can easily check the claim that these displayed formulas are a Frege proof by first encoding this problem into one of checking justified Horn clauses. To do this, we use the following declaration.

```
type  v, w  i.            % Two propositional variables
type  bot   i.            % classical false
type  ar    i -> i -> i.  % classical implication
infix ar    4.
type  pv    i -> iform.   % Provability predicate
```

Given these types and typed constructors, we can then form the following Horn clause entailment.

```
((forall X\ forall Y\ (pv X) imp (pv (X ar Y)) imp (pv Y)) imp
((forall X\ forall Y\ pv (X ar (Y ar X))) imp
((forall X\ forall Y\ forall Z\ pv ((X ar (Y ar Z)) ar
                                    ((X ar Y) ar (X ar Z)))) imp
((forall X\ pv (((X ar bot) ar bot) ar X)) imp
     pv (bot ar w)))))
```

The predicate pv denotes the property that its argument (an encoding of a propositional classical logic formula) is provable. The first assumption of this entailment encodes modus ponens and the other three assumptions encode the three axiom schemas. Given this rather natural encoding of this problem, the Frege proof claimed above can be directly encoded as the following list of justifications.

```
[ tup (i 4) (pv (((w ar bot) ar bot) ar w)) (i 3) [],
  tup (i 5) (pv ((((w ar bot) ar bot) ar w) ar
                 (bot ar (((w ar bot) ar bot) ar w)))) (i 1) [],
  tup (i 6) (pv (bot ar (((w ar bot) ar bot) ar w))) (i 0) [i 4,i 5],
  tup (i 7) (pv ((bot ar (((w ar bot) ar bot) ar w)) ar
                 ((bot ar ((w ar bot) ar bot)) ar (bot ar w))))
                 (i 2) [],
  tup (i 8) (pv ((bot ar ((w ar bot) ar bot)) ar (bot ar w)))
            (i 0) [i 6,i 7],
  tup (i 9) (pv (bot ar ((w ar bot) ar bot))) (i 1) [],
  tup (i 10) (pv (bot ar w)) (i 0) [i 9,i 8] ].
```

As is required by the FPC for checking justified Horn clause proofs, the polarity bias for the pv predicate must be positive.

Using such an encoding, we have illustrated how checking Frege proofs can be reduced to the checking of justified Horn clause entailments. It would be easy to change the object-level logic from classical propositional logic to, say, a modal logic and its associated Frege proof rules.

10 Hosting LKF^a on an LJF^a kernel

The semantic definition of proof evidence starts with first identifying which logic—intuitionistic or classical—to use to interpret that evidence. Depending on that choice, a different set of polarization choices and a different choice of clerks and experts are involved. The syntactic details behind these two logics are, however, remarkably similar. For example, Gentzen's original unfocused sequent calculus for classical (LK) and intuitionistic (LJ) logics used the same technical devices (contexts, two-sided sequents, structural rules, eigenvariables, etc) and separated LJ proofs from LK proofs by one simple restriction ("at most one formula on the right"). Similarly, Liang and Miller [61] have shown how focused classical and intuitionistic (and linear logic) proofs can be seen as restrictions of a common focusing framework. Since classical and intuitionistic proofs share a great deal in common, it is natural to ask whether we could use the

kernel for one of these logics to check proof evidence for the other logic: in this way, one would only need to build (and trust) one kernel instead of two.

Various relationships between classical and intuitionistic provability and proofs are well known. The double negation translations of, say, Gödel [44], Gentzen [40], and Kolmogorov [58] can map classical logic provability directly into intuitionistic logic. On the other hand, translations of intuitionistic logic into classical logic all seem to use additional devices, not in propositional or first-order classical logics. For example, intuitionistic provability has been encoded by Gödel using classical, modal logic [45] and by Girard [41] using linear logic and the ! exponential. In fact, Gentzen's single-conclusion restriction for intuitionistic sequent can easily be seen as adding a kind of exponential in the sense of linear logic. In particular, since the right-hand side of LJ sequents cannot have commas, the right-hand side is a *linear* context while the left-hand side is *classical* (i.e., allows contractions). At the same time, the single-conclusion restriction enforces another distinction between these two contexts that must be encoded into the implication. Consider, for example, the LK inference rule

$$\frac{\Gamma_1 \longrightarrow A, \Delta_1 \qquad \Gamma_2, B \longrightarrow \Delta_2}{\Gamma_1, \Gamma_2, A \supset B \longrightarrow \Delta_1, \Delta_2} \supset L$$

If we impose the LJ restriction here, then it must be the case that Δ_1 is empty. This same "modal" distinction between formulas on the left and right of the sequent arrow is captured by the linear logic exponential. In particular, if the left premise of the inference figure

$$\frac{!\Gamma_1 \vdash !A, \Delta_1 \qquad !\Gamma_2 \vdash B, \Delta_2}{!\Gamma_1, !\Gamma_2 \vdash !A \otimes B, \Delta_1, \Delta_2}$$

is the introduction rule for ! then, again, Δ_1 must be empty.

Since it appears that the classical logic proofs (without modal-like or exponential-like operators) are less expressive than intuitionistic logic proofs, it is then natural to try to use a kernel for intuitionistic logic as the heart of a kernel for classical logic. Given our approach to proof checking—which includes proof reconstruction—our kernels need to follow proof evidence to successful conclusions but we must also follow unsuccessful attempts at proof. We need to require that no matter how we polarize classical logic formulas, we can translate the resulting polarized formulas into polarized intuitionistic formulas so that phases in *LKF* are in one-to-one correspondence with phases in *LJF* of the translated formulas. For example, if we know that there is a small proof of a given sequent in *LKF* (say, involving a couple of decide rules), then we need to know that there must also be a similarly small proof of the translated sequent in *LJF*.

We shall not directly use double negation translations since these are usually presented for unpolarized classical logic formulas. Instead, we shall use a translation introduced by Chaudhuri [11] for encoding *LKF* formulas and proofs into *LJF* formulas and proofs. The two mappings, $[\![\cdot]\!]^+$ and $[\![\cdot]\!]^-$, defined in Figure 21 describe two mappings of *LKF* formulas to *LJF* formulas. Here, q is a fixed, negative atom that is not allowed in the input *LKF* formulas. All other atoms in the target *LJF* formula are assigned positive polarity even if it is the result of encoding a negative *LKF* atom. The output of both $[\![\cdot]\!]^+$ and $[\![\cdot]\!]^-$ are either a positive formula or a formula of the form $B \supset q$ for some positive formula B. We will subsequently call these functions collectively the $[\![\cdot]\!]^\pm$ translation.

It is proved in [11, Theorem 12] that phases in *LKF* are in one-to-one correspondence with phases in *LJF* when the $[\![\cdot]\!]^\pm$ is used. More precisely, [11] shows that an

$$
\begin{array}{rcl@{\qquad}rcl}
[\![t^+]\!]^+ & = & t & [\![t^-]\!]^- & = & f \\
[\![f^+]\!]^+ & = & f & [\![f^-]\!]^- & = & t \\
[\![B \vee^+ C]\!]^+ & = & [\![B]\!]^+ \vee [\![C]\!]^+ & [\![B \vee^- C]\!]^- & = & [\![B]\!]^- \wedge^+ [\![C]\!]^- \\
[\![B \wedge^+ C]\!]^+ & = & [\![B]\!]^+ \wedge^+ [\![C]\!]^+ & [\![B \wedge^- C]\!]^- & = & [\![B]\!]^- \vee [\![C]\!]^- \\
[\![\exists x.A]\!]^+ & = & \exists x.[\![A]\!]^+ & [\![\forall x.A]\!]^- & = & \exists x.[\![A]\!]^- \\
[\![A^+]\!]^+ & = & A^+ & [\![A^-]\!]^- & = & A^+ \\
[\![N]\!]^+ & = & [\![N]\!]^- \supset q & [\![P]\!]^- & = & [\![P]\!]^+ \supset q
\end{array}
$$

Fig. 21 The functions $[\![\cdot]\!]^+$ and $[\![\cdot]\!]^-$ which map *LKF* formulas onto *LJF* formulas.

```
type fcert   cert.

andPos_jc   C C'         :- orNeg_kc   C C'.
andPos_je   C C' C''     :- andPos_ke  C C' C''.
decideL_je  C C' I       :- decide_ke  C C' I.
initialR_je C I          :- initial_ke C I.
or_jc       C C' C''     :- andNeg_kc  C C' C''.
or_je       C C' Choice  :- orPos_ke   C C' Choice.
releaseR_je C C'         :- release_ke C C'.
some_jc     C C'         :- all_kc     C C' .
some_je     C C' T       :- some_ke    C C' T.
storeL_jc   C C' I       :- store_kc   C C' I.
true_jc     C C'         :- false_kc   C C'.
true_je     C            :- true_ke    C.
arr_jc      C C.
storeR_jc   C C.
arr_je      C C  fcert.
initialL_je fcert.
```

Fig. 22 The definition of *LJF* clerks and experts based on those given for *LKF*

LKF-phase ending with the sequent $\vdash \Theta \Uparrow \Gamma$ corresponds to an *LJF*-phase ending with the sequent $[\![\Theta]\!]^- \Uparrow [\![\Gamma]\!]^- \vdash q \Uparrow$ and that an *LKF*-phase ending with the sequent $\vdash \Theta \Downarrow B$ corresponds to an *LJF*-phase ending with the sequent $[\![\Theta]\!]^- \Downarrow [\![B]\!]^- \vdash q$.

We can make the following additional statements about the translation of *LKF* proofs into *LJF*.

1. Formally, the zones in the sequents of the focused calculi in [11] are multisets. In our setting, the asynchronous zones Γ in both the classical sequent $\vdash \Theta \Uparrow \Gamma$ and the intuitionistic sequent $\Theta \Uparrow \Gamma \vdash q \Uparrow$ are lists. The correspondence of phases is still maintained given this change.
2. The $store_r$ rule is only applied to the atom q.

Given these observations, it is possible to refine Chaudhuri's theorem to fit our purposes here: we can actually show a precise translation of each *LKF* rule into a small group of *LJF* rules. This makes it possible to define the clerks and experts needed for guiding the *LJF* kernel directly from clerks and experts destined for the *LKF* kernel. In fact, a precise way to show this correspondence on proofs is to show the implementation of this definition, given in Figure 22. Note also that indexes used for storing and deciding on formulas in the *LKF* setting can be used, unchanged, for the same purposes in the *LJF* setting.

We left the cut-rule and its associated expert from the presentation above since it is the only case where something more subtle needs to be done. First the mapping of formulas and proofs from Chaudhuri needs to be extended slightly to account for

$$\frac{\Xi_1 \vdash \Theta \Uparrow N \quad \Xi_2 \vdash \Theta \Uparrow \neg N \quad cut_e(\Xi, \Xi_1, \Xi_2, N)}{\Xi \vdash \Theta \Uparrow \cdot}$$

$$\frac{\Xi_1 \vdash \Theta \Uparrow P \quad \Xi_2 \vdash \Theta \Uparrow \neg P \quad cut_e(\Xi, \Xi_1, \Xi_2, P)}{\Xi \vdash \Theta \Uparrow \cdot}$$

$$\frac{\Xi_1 : \Gamma \Uparrow \vdash [\![\neg N]\!]^{-} \Uparrow \quad \Xi_2 : \Gamma \Uparrow [\![\neg N]\!]^{-} \vdash \Uparrow R \quad cut_e(\Xi, \Xi_1, \Xi_2, [\![\neg N]\!]^{-})}{\Xi : \Gamma \Uparrow \vdash \Uparrow R}$$

$$\frac{\Xi_2 : \Gamma \Uparrow \vdash [\![P]\!]^{-} \Uparrow \quad \Xi_1 : \Gamma \Uparrow [\![P]\!]^{-} \vdash \Uparrow R \quad cut_e(\Xi, \Xi_1, \Xi_2, [\![P]\!]^{-})}{\Xi : \Gamma \Uparrow \vdash \Uparrow R}$$

Fig. 23 Various instances of the augmented LJF^a cut rule

proofs with the cut-inference rule. Second, the exact way that the *LJF* cut expert calls the *LKF* cut expert depends on the polarity of the formula returned by the *LKF* cut expert. However, given the two different instances of the cut-rule in LKF^a in Figure 23, it is an easy matter to see how the cut-experts for LJF^a must be defined. That formal definition is given by the clause

```
cut_je C C1 C2 F :- cut_ke C C1 C2 D,
                    (isNeg D, negate D P, trans- P F;
                     isPos D,             trans- D F).
```

Here, we assume that the predicate `trans-` and `negate` are defined so that `trans- F F'` holds if and only if $F' = [\![F]\!]^{-}$ and `negate F F'` holds if and only if F' is the negation normal form of the negation of F.

11 A reference proof certificate checker

In this section, we outlined how the logic programming paradigm can be used to implement FPCs. Such implementations are best referred to as *reference checkers* and not as an implementation of a *universal proof checker* since the effective implementation of such a broad-spectrum checker for all possible FPCs is unlikely. The relation between FPC specification and proof checker implementation is probably similar to the relationship between the specification of a context-free grammar (CFG) and the implementation of a parser (returning to an analogy offered in Section 1). Following the introduction of CFGs as a framework for describing the structure of a concrete language [21], a great deal of additional study was needed before CFGs could be used to provide practical tools for routine use by computer scientists. For example, such grammars needed to be analyzed in order to discover (1) that determining whether or not a grammar is ambiguous is undecidable; (2) that deterministic and non-deterministic CFGs have different expressive strengths; and (3) that practical parsing required serious restriction (such as those related to LALR parsing). Similar developments also need to be considered for the FPC framework.

Another analogy between CFG and FPC that is worth pointing out here is the role that the logic programming paradigm can play. Logic programs provide natural means of getting parsers from grammars [75,84] (even though serious parsers implement greatly restricted subsets of CFG with rather specialized tools, such a YACC [56]). It is also the case, as we discuss below, that moving from an FPC specification to a naive logic programming implementation of a checker is straightforward.

Techniques for turning a proof system into a logic program that captures provability has been known for a long time [35,66]. Prolog-style, depth-first search is too naive to turn such specifications into useful theorem provers. In the setting of proof checking, however, certificates usually contain information that can provide meaningful bounds to restrict depth-first search. One might thus expect that bounded depth-first search implementations of the augmented proof systems LKF^a and LJF^a might provide useful implementations of proof checkers. We describe below a reference proof checker we have written in λProlog that has been used to validate all the example FPCs in this paper.

11.1 Kernels as logic programs

As we mentioned in Section 5, a kernel is an implementation of an augmented focused systems, such as LKF^a or LJF^a. While we have specified these kernels as collections of inference rules (in Figures 6 and 14), these could have been equally well written as clauses within the λProlog programming language. To be more concrete, we illustrate just such an encoding for LKF^a.

The two styles of sequents used in LKF are of the form $\vdash \Theta \Uparrow \Gamma$ and $\vdash \Theta \Downarrow B$. When we move to LKF^a, the members of Θ are pairs of formula and index. Our logic programming specification of the association Θ will make use of the λProlog's ability to perform hypothetical reasoning: that is, the association between indexes and formulas will be made by sets of assumptions of the form `storage I C`, where `I` is an index and `C` is a formula. On the other hand, the Γ context will be represented by a list of formulas. By introducing the declarations

```
kind seq                          type.
type unf              list form -> seq.
type foc                   form -> seq.
type storage        index -> form -> o.
```

we can then encode the LKF^a sequent $\vdash \Theta \Downarrow B$ as the term `(foc B)` and $\vdash \Theta \Uparrow \Gamma$ as the term `(unf Gamma)`: here, `B` encodes the formula B and `Gamma` encodes the list of formulas in Γ. Of course, one must look to the set of assumptions to determine the value of (the now implicit) Θ context.

Given this encoding of an LKF^a sequent, the checking process can be encoded directly as a recursive λProlog program that axiomatizes the predicate

```
type check            cert -> seq -> o.
```

that is meant to determine whether or not a given certificate term leads to a proof of the given formula. For example, the implementation of the two inference rules from Figure 6 for introducing the negative polarized disjunction and conjunction can be written as follows.

```
check Cert (unf [A !-! B|Rest]) :- orNeg_kc Cert Cert',
  check Cert' (unf [A, B|Rest]).
check Cert (unf [A &-& B|Rest]) :- andNeg_kc Cert CertA CertB,
  check CertA (unf [A|Rest]), check CertB (unf [B|Rest]).
```

Here, the symbols `!-!` and `&-&` denote the internal names for the negatively polarized disjunction and conjunction (these are local to just the kernel code). Similarly, two inference rules that use the storage of formulas—store and decide—can be implemented simply as follows.

```
check Cert (unf [C|Rest]) :- (isPos C ; isNegAtm C),
   store_kc Cert Cert' I, (storage I C => check Cert' (unf Rest)).
check Cert (unf nil) :- decide_ke Cert Cert' I, storage I P,
   isPos P, check Cert' (foc P).
```

Note the use of the hypothetical implication in the body of the first clause.

In this manner, a λProlog specification of an LKF^a kernel can be a short program, consisting of one clause for every inference rule in LKF^a plus code to define predicates such as `isPos`, `isNeg`, `isNegAtom`, etc.

All along, we have been presenting the specification of clerks and experts as logic programs. In general, the logical structure of such specifications are rather simple: for example, the clerks and experts for deciding conjunctive normal forms (Figure 7), for checking oracle strings (Figure 8), and for checking justified Horn clauses proofs (Figure 19) are all given as (universally quantified) atomic Horn clauses. Even the other specifications of clerks and experts—for binary resolution (Section 7.3), $\beta\eta$-long type checking (Figure 16), the mimic decision procedure (Figure 17)—are almost all atomic specifications with an occasional clause having a simple body.

11.2 Implementations of checkers

Although proof checking has a long tradition, that tradition has most frequently been based on functional programming language implementations and denotational semantic specification techniques. There are, however, a number of reasons for preferring the logic programming paradigm over the functional programming paradigm [65]. In addition to the benefits of using λProlog given above (availability of hypothetical reasoning) and in Section 6 (availability of typing, relational specifications, and λ-tree syntax) to present FPCs, we also state the following additional benefits of using λProlog for the construction of proof checking kernels.

1. As with all logic programming languages, λProlog implements unification and backtracking search, both of which are useful for proof reconstruction.
2. One of the benefits of using the λ-*tree syntax* [66] approach to syntax with bindings, is that it incorporates logically supported mechanisms for eigenvariables, even in the presence of unification. As a result, full support for alternating quantification within formulas being checked is allowed: prenex normal forms and skolemization are not needed.
3. The logic underlying λProlog also provides for not only modular construction of programs but also the ability to hide constructors and clauses (in the sense of abstract datatypes). Thus, a kernel can be largely isolated from other programs that may invoke it and this makes it much easier to reason about the correctness of the resulting kernel implementations.

While these features are useful and provide a flexible framework in which to design, test, and execute proof checkers based on FPCs, not all these features are needed for all proof checking efforts. While modularity and typing are useful for the usual reasons, they are not strictly necessary for building checkers. Similarly, if one is interested only in checking proofs in propositional logic, then the support for λ-tree syntax is not needed. Finally, the association between indexes and formulas within sequent contexts can also be managed by explicit structures, such as association lists, instead of implicitly using hypothetical reasoning. Hence, it is easy to modify everything presented here so that Prolog (not λProlog) could be used to interpret FPCs over propositional logic.

The one feature of logic programming that would seem the most difficult to eliminate completely is the availability of unification and backtracking search: these two features made it possible for several of our certificates to not contain explicit term structures. If one is willing, however, to work with certificate formats that provide a lot of explicit information, doing without these two features is possible. For example, it is possible to extend the FPC for justified Horn clauses (Figure 19) so that justification contain a list of terms to be used to instantiate the universal quantifiers surrounding a Horn clause under focus. By using such a justification, unification is no longer needed for checking such certificates. Removing unification in this way means, of course, that the certificates for justified Horn clauses (and, similarly, Frege proofs) are significantly larger.

Similarly, an implementation of full hypothetical reasoning (as it is implemented in, for example, Teyjus [71, 80]) is not necessary since it is only used in the kernel to store atomic formulas of the form `(storage I L)` where `L` is a literal and `I` is an index. Since indexes can be very specific structures in some applications (e.g., integers, tokens, and formula occurrences), a more effective implementation of storing and recalling atomic formulas can be done by replacing the usual "first-argument-indexing" technique employed by many logic programming implementations with specialized indexing techniques.

Since kernels must be trusted, the fact that we can write them as small, declarative programs has given us a great deal of trust in the kernels we have implemented. We do, of course, have to trust as well the Teyjus implementation to provide a sound implementation of logic. Fortunately, since the elements of λProlog that we use here are all based on intuitionistic logic with quantification over simply typed λ-terms, other implementations of that logic exist and still others can be built as desired. For example, there is the ELPI implementation of λProlog [31] and both the Minlog prover [83] and Isabelle/Pure [74] implement much of that logic.

12 Future work

12.1 Developing more examples of FPCs

Many more proofs systems than those illustrated here can be defined as FPCs: see, for example, the specification of FPCs for equational rewriting and paramodulation [16, 17]. The proof theoretic results about focused proof systems for modal logics in [69] were designed in part to allow a natural approach to defining FPCs for labeled proofs of modal logic formulas. A formal FPC definition of the dependently typed λ-calculus—such as $\lambda\Pi$ [6] and LF [48]—should be possible by first encoding such a typed λ-calculus into intuitionistic logic [33, 86] and then by generalizing the justified Horn clause certificate format in Section 9 to a "justified hereditary Harrop formula" certificate format. This typed λ-calculus has also recently been extended to LFSC [88] and to $\mathrm{LF}_{\mathcal{P}}$ [52] so that various kinds of computations can be treated by the type checker instead of being explicitly detailed within the typed λ-term itself. Formal FPC definitions of such proofs should similarly be captured in our setting.

Developing a kernel for linear logic should be a rather straightforward exercise given that focusing for linear logic is well understood [1]. In fact, our first attempt to build kernels for classical and intuitionistic logics centered on first building a kernel for the LKU system [61] that allows mixing linear logic, classical, and intuitionistic logic.

Given the fact that a kernel for LJF^a could also be used to power a kernel for LKF^a (Section 10), we did not need the additional flexibility offered by an LKU-based kernel.

12.2 Moving beyond first-order logic

In this paper, we have limited ourselves to first-order logic. An interesting and important setting for developing FPCs are model checkers and inductive theorem provers. The paper by Baelde [3] on adding least and greatest fixed point operators to linear logic provides some of the proof theoretic foundations on which to start that effort. Initial results on certifying model checking problems related to reachability and bisimulation appear in [49] and initial designs for certifying inductive theorem proving appear in [8].

While the sequent calculus proof systems are known and well studied for higher-order versions of both classical and intuitionistic logics, there has not been much work on general focusing systems for such logics. While focusing can be extended to higher-order logic when a monolithic assignment of polarity is made upfront (all connectives and atomic formulas are all negative or all positive) [25, 34, 67], more flexible polarity assignments have not yet been studied explicitly.

12.3 Theories

In many situations, theorems are proved with respect to some *theory*. Set theory is one such popular theory and provides the basis of the Mizar theorem prover. It is also possible to view type theories and higher-order logic as theories in first-order logic [28, 29, 33]. Proving theorems from theories can generally be encoded in logic by viewing theories as additional assumptions. Thus, checking proofs in logic is an important first step in dealing with consequences within theories. Many questions related to reasoning with theories—such as how to related conclusions derived from different theories—are not immediately treated by reference to an underlying logic and such questions will require their own treatment at some point.

12.4 Extensions to the kernel design

Recognizing and treating parallelism in proof structures has been a recurrent theme in proof theory: for examples, expansion trees [63] and proof nets [41] provide proof systems that make a minimal commitment to the order in which inference rules are applied. The notion of multifocusing—that is, the focusing on several positive formulas at once instead of just one—has been introduced into the setting of focused proofs [68] in order to capture such parallelism [12, 13]. For a (single-focus) treatment of expansion trees as an FPC, see [15, Section 5.4].

The following inference rule, called the *multicut* rule, is often used as a technical generalization to the cut rule to help prove cut-elimination theorems (see, for example, [39, 85]).

$$\frac{\Delta_1 \longrightarrow B_1 \quad \cdots \quad \Delta_n \longrightarrow B_n \qquad B_1, \ldots, B_n, \Gamma \longrightarrow C}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc \quad (n \geq 1)$$

While this rule can be seen as encoding n separate applications of the cut-rule, this rule states that the n-lemmas that are proved (the n left-most premises) were actually proved separately from each other. Thus, if a proof relies on several lemmas that have been proved independently of each other, their independence can be encoded using a multicut. If such a multicut rule is not checked directly, the author of the proof certificate might need to serialize the various cuts and thereby introduce spurious dependences between the various lemmas and their assumptions.

Our kernels should also be extended so that previous proved theorems can be used in establishing new theorems. It is easy to imagine a situation where *libraries of theorems* are responsible for checking proof certificates and for making their theorems available to other provers. Similarly, there might be situations where we simply wish to trust some (special purpose) prover and that we check all other aspects of a proof: current uses of SMT provers often follow this pattern. In any case, the kinds of kernels we have described would need (simple) modifications so that they can relegate trust to other computational systems.

13 Related work

The notion of having trusted kernels for checking proofs is a well established design element of many theorem provers. For example, the LCF [47] family of provers, including Coq [7], Isabelle [73], and the HOL provers [46], all separate kernels that need to be trusted from more general proof search mechanisms that do not need to be trusted. While theorem provers are often evolving programs in which complex implementation of search algorithms and heuristics are tested and deployed, kernels are intended to be small and largely static. As a result, one should be able to analyze a kernel and have high-confidence that they will not be unsound. A theorem prover that contains a trusted subsystem that checks alleged proofs is said to satisfy the *de Bruijn criterion* [5]. In such a system, one only has to trust the correctness of the simpler checking subsystem and not the large and complex theorem prover.

Most of the theorem provers that contain kernel subsystems do not generally provide their proofs in a format that can be exported, checked, and used by other systems. In recent years, there has been an increasing interest and need to facilitate the sharing of proofs. For example, the OpenTheory project [55] aims at having various HOL theorem provers share proofs. Still other projects attempt to connect SAT/SMT systems with more general theorem provers, e.g., [2, 10, 36]. In order for prover A to not blindly trust proofs from prover B, prover B may be required to generate a certificate that demonstrates that it has formally found a proof. Prover A will then need to check the correctness of that certificate. In this way, prover A only needs to check individual certificates and not rely on trusting the whole of prover B. For example, in [36], an SMT prover was modified to output proof evidence as Isabelle proof scripts that can be checked by Isabelle; similarly, in [2], a SAT/SMT prover is modified to output proof evidence that can be checked within Coq. To the extent that such approaches are *ad hoc* and technology-based, this kind of proof sharing can break every time a new version of either prover A or B is released.

Several projects have arose to improve this situation and to provide more principled and universal formats for sharing proof structures. Some specialized theorem provers related to SAT solving have adopted standards for outputting their proof evidence (see, for example, [38, 91]). The dependently typed λ-calculus LF has been extended

to LFSC (the "Logical Framework with Side Conditions") [88] for which a checker is available. This checker has been used to check proofs emitted by two different SMT provers [89]. The MMT proof tool [81] abstracts away from specific logics and allows proof systems to be defined, analyzed, implemented, and checked, hence, allowing the construction of proof checkers that are independent of specific theorem provers. The Dedukti proof checking system [9, 82] is based on a still richer logical framework, namely $\lambda\Pi$ modulo [24], that incorporates constructive and dependently typed λ-calculus plus induction (arithmetic). Several implemented systems—in particular, Coq, HOL, FoCaLize, Matita, iProver, and Zenon—have been augmented in such a way that they output proofs that Dedukti can check.

Our goal here has been to provide a technology-independent means of defining the semantics of a range of proof formats in classical and intuitionistic logics. While our specifications are, in principle, implementable as relational specifications (logic programs), this paper has not been concerned with exploring the many ways that such specifications can be converted into effective proof checkers.

14 Conclusions

The promise of focused proof systems has always been that they can be used to describe synthetic inference rules. We have shown how the focused proof systems, *LJF* and *LKF*, for first-order intuitionistic and classical logics can be augmented in such a way that the simple relational specifications—the clerk and expert predicates—can be used to describe such synthetic rules. We have then gone on to illustrate how a range of proof systems can be encoded as just such synthetic rules. In this way, we view our augmented focused proof system as both a formal definition of those proof systems as well as an executable specification that can be used to check (and partially reconstruct) such proof systems. We have also shown how the more expressive intuitionistic logic proof system can easily and directly be used to define and check classical logic proof systems.

Acknowledgments This paper is an extension of the conference paper [19] by the authors. This work has been funded by the ERC Advanced Grant ProofCert. We thank Roberto Blanco, Danko Ilik, Matthias Puech, and anonymous reviewers for their comments on an earlier draft of this paper.

References

1. J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.
2. M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs (CPP 2011)*, LNCS 7086, pages 135–150, 2011.
3. D. Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1), Apr. 2012.
4. H. Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Symbolic Logic*, 3(2):181–215, 1997.
5. H. Barendregt and E. Barendsen. Autarkic computations in formal proofs. *J. of Automated Reasoning*, 28(3):321–336, 2002.
6. H. P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, Apr. 1991.

7. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
8. R. Blanco and D. Miller. Proof outlines as proof certificates: a system description. In I. Cervesato and C. Schürmann, editors, Proceedings First International Workshop on *Focusing,* Suva, Fiji, 23rd November 2015, volume 197 of *Electronic Proceedings in Theoretical Computer Science*, pages 7–14. Open Publishing Association, Nov. 2015.
9. M. Boespflug, Q. Carbonneaux, and O. Hermant. The $\lambda\Pi$-calculus modulo as a universal proof language. In D. Pichardie and T. Weber, editors, *Proceedings of PxTP2012: Proof Exchange for Theorem Proving*, pages 28–43, 2012.
10. S. Böhme and T. Weber. Designing proof formats: A user's perspective. In P. Fontaine and A. Stump, editors, *PxTP 2011: First International Workshop on Proof eXchange for Theorem Proving*, pages 27–32, Aug. 2011.
11. K. Chaudhuri. Classical and intuitionistic subexponential logics are equally expressive. In A. Dawar and H. Veith, editors, *CSL 2010: Computer Science Logic*, LNCS 6247, pages 185–199, Brno, Czech Republic, Aug. 2010. Springer.
12. K. Chaudhuri, S. Hetzl, and D. Miller. A multi-focused proof system isomorphic to expansion proofs. *J. of Logic and Computation*, 26(2):577–603, 2016.
13. K. Chaudhuri, D. Miller, and A. 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*, IFIP 273, pages 383–396. Springer, Sept. 2008.
14. 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, 2008.
15. Z. Chihani. *Certification of First-order proofs in classical and intuitionistic logics*. PhD thesis, Ecole Polytechnique, Aug. 2015.
16. Z. Chihani, T. Libal, and G. Reis. The proof certifier checkers. In H. D. Nivelle, editor, *Proceedings of the 24th Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, LNCS 9323, pages 201–210. Springer, 2015.
17. Z. Chihani and D. Miller. Proof certificates for equality reasoning. In M. Benevides and R. Thiemann, editors, *Post-proceedings of LSFA 2015: 10th Workshop on Logical and Semantic Frameworks, with Applications. Natal, Brazil*, ENTCS 18612, 2016.
18. Z. Chihani, D. Miller, and F. Renaud. Checking foundational proof certificates for first-order logic (extended abstract). In J. C. Blanchette and J. Urban, editors, *Third International Workshop on Proof Exchange for Theorem Proving (PxTP 2013)*, volume 14 of *EPiC Series*, pages 58–66. EasyChair, 2013.
19. 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*, LNAI 7898, pages 162–177, 2013.
20. Z. Chihani, D. Miller, and F. Renaud. Supporting λProlog code, Apr. 2016. `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/fpc-support.tar`.
21. N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, Sept. 1956.
22. A. Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.
23. S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. *J. of Symbolic Logic*, 44(1):36–50, 1979.
24. 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*, LNCS 4583, pages 102–117. Springer, 2007.
25. 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.
26. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with an application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.
27. O. Delande, D. Miller, and A. Saurin. Proof and refutation in MALL as a game. *Annals of Pure and Applied Logic*, 161(5):654–672, Feb. 2010.

28. G. Dowek. Skolemization in simple type theory: the logical and the theoretical points of view. In *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday*, number 17 in Studies in Logic, pages 244–255. College Publications, 2008.
29. G. Dowek, T. Hardin, and C. Kirchner. HOL-$\lambda\sigma$ an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science*, 11(1):1–25, 2001.
30. G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *J. of Automated Reasoning*, 31(1):31–72, 2003.
31. C. Dunchev, F. Guidi, C. S. Coen, and E. Tassi. ELPI: fast, embeddable, λProlog interpreter. In M. Davis, A. Fehnker, A. McIver, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 460–468, 2015.
32. R. Dyckhoff and S. Lengrand. Call-by-value λ-calculus and LJQ. *J. of Logic and Computation*, 17(6):1109–1134, 2007.
33. A. Felty. Transforming specifications in a dependent-type lambda calculus to specifications in an intuitionistic logic. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.
34. A. Felty. Encoding the calculus of constructions in a higher-order logic. In M. Vardi, editor, *8th Symp. on Logic in Computer Science*, pages 233–244. IEEE, June 1993.
35. A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, Aug. 1993.
36. P. Fontaine, J.-Y. Marion, S. Merz, L. P. Nieto, and A. F. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In H. Hermanns and J. Palsberg, editors, *TACAS: Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference*, LNCS 3920, pages 167–181. Springer, 2006.
37. J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row, 1986.
38. A. V. Gelder. Producing and verifying extremely large propositional refutations: Have your cake and eat it too. *Annals of Mathematics and Artificial Intelligence*, 65(4):329–372, 2012.
39. G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935.
40. G. Gentzen. Die widerspruchfreiheit der reinen zahlentheorie. *Mathematische Annalen*, 112:493–565, 1936. Reprinted in English translation as "The consistency of Elementary Number Theory" in The collected papers of Gerhard Gentzen, M. E. Szabo, ed.
41. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
42. J.-Y. Girard. A new constructive logic: classical logic. *Math. Structures in Comp. Science*, 1:255–296, 1991.
43. J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.
44. K. Gödel. Zur intuitionistischen arithmetik und zahlentheorie. *Ergebnisse eines Mathematischen Kolloquiums*, pages 34–38, 1932. English translation in *The Undecidable* (M. Davis, ed.) 1965, 75-81.
45. K. Gödel. Eine interpretation des intuitionistischen aussagenkalkuls. *Ergebnisse eines Mathematischen Kolloquiums.*, 4:39–40, 1933. Available in "Kurt Gödel: Collected Works. Volume 1" edited by S. Feferman and et al.
46. M. Gordon. From LCF to HOL: a short history. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 169–186. MIT Press, 2000.
47. M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, LNCS 78. Springer, 1979.
48. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
49. Q. Heath and D. Miller. A framework for proof certificates in finite state exploration. In C. Kaliszyk and A. 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, Aug. 2015.
50. 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.

51. W. Hodges. Logic and games. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, spring 2013 edition, 2013.
52. F. Honsell, M. Lenisa, L. Liquori, P. Maksimovic, and I. Scagnetto. LFP: a logical framework with external predicates. In A. Chlipala and C. Schürmann, editors, *LFMTP 2012: Proceedings of the seventh international workshop on Logical frameworks and meta-languages, theory and practice*, pages 13–22. ACM New York, 2012.
53. 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.
54. D. J. D. Hughes. Proofs without syntax. *Annals of Mathematics*, 143(3):1065–1076, 2006.
55. J. Hurd. The OpenTheory standard theory library. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *The Third International Symposium on NASA Formal Methods*, LNCS 6617, pages 177–191, 2011.
56. S. C. Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.
57. G. Kahn. Natural semantics. In F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, LNCS 247, pages 22–39. Springer, Mar. 1987.
58. A. N. Kolmogorov. On the principle of the excluded middle. *Matematicheskii sbornik*, 32:646–667, 1925. English translation by Jean van Heijenoort in *From Frege to Gödel*.
59. O. Laurent. *Etude de la polarisation en logique*. PhD thesis, Université Aix-Marseille II, Mar. 2002.
60. C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.
61. C. Liang and D. Miller. A focused approach to combining logics. *Annals of Pure and Applied Logic*, 162(9):679–697, 2011.
62. P. Lorenzen. Ein dialogisches konstruktivitätskriterium. In *Infinitistic Methods: Proceed. Symp. Foundations of Math.*, pages 193–200. PWN, 1961.
63. D. Miller. A compact representation of proofs. *Studia Logica*, 46(4):347–370, 1987.
64. D. 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.
65. D. Miller. Proof checking and logic programming. In M. Falaschi, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR)*, LNCS 9527, pages 3–17. Springer, 2015.
66. D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.
67. 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.
68. D. Miller and A. Saurin. From proofs to focused proofs: a modular proof of focalization in linear logic. In J. Duparc and T. A. Henzinger, editors, *CSL 2007: Computer Science Logic*, LNCS 4646, pages 405–419. Springer, 2007.
69. D. Miller and M. Volpe. Focused labeled proof systems for modal logic. In M. Davis, A. Fehnker, A. McIver, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, LNCS 9450, Nov. 2015.
70. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
71. 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)*, LNAI 1632, pages 287–291, Trento, 1999. Springer.
72. G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In C. Hankin and D. Schmidt, editors, *28th ACM Symp. on Principles of Programming Languages*, pages 142–154, 2001.
73. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.
74. L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, Sept. 1989.
75. F. C. N. Pereira and S. M. Shieber. *Prolog and Natural-Language Analysis*, volume 10. CLSI, Stanford, CA, 1987.
76. 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)*, LNAI 1632, pages 202–206, Trento, 1999. Springer.

77. G. D. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, Aarhus, Denmark, Sept. 1981.
78. G. D. Plotkin. The origins of structural operational semantics. *J. of Logic and Algebraic Programming*, 60:3–15, 2004.
79. D. Prawitz. *Natural Deduction.* Almqvist & Wiksell, Uppsala, 1965.
80. X. Qi, A. Gacek, S. Holte, G. Nadathur, and Z. Snow. The Teyjus system – version 2, 2015. `http://teyjus.cs.umn.edu/`.
81. F. Rabe. The future of logic: Foundation-independence. *Logica Universalis*, pages 1–20, 2016.
82. R. Saillard. Towards explicit rewrite rules in the $\lambda\Pi$-calculus modulo. In S. Schulz, G. Sutcliffe, and B. Konev, editors, *IWIL-10th International Workshop on the Implementation of Logics*, 2013.
83. H. Schwichtenberg. Minlog. In F. Wiedijk, editor, *The Seventeen Provers of the World*, LNCS 3600, pages 151–157. Springer, 2006.
84. S. M. Shieber, Y. Schabes, and F. C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, 1995.
85. J. Slaney. Solution to a problem of Ono and Komori. *Journal of Philosophic Logic*, 18:103–111, 1989.
86. Z. Snow, D. Baelde, and G. Nadathur. A meta-programming approach to realizing dependently typed logic programming. In T. Kutsia, W. Schreiner, and M. Fernández, editors, *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 187–198, 2010.
87. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.
88. A. Stump. Proof checking technology for satisfiability modulo theories. In A. Abel and C. Urban, editors, *Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.
89. A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.
90. A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge University Press, 2 edition, 2000.
91. N. Wetzler, M. J. H. Heule, and J. W. 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*, LNCS 8561, pages 422–429. Springer, 2014.