Translating between implicit and explicit versions of proof

Roberto Blanco[1], Zakaria Chihani[2], and Dale Miller[1]

[1] Inria & LIX/École polytechnique
[2] CEA-List

Abstract. The Foundational Proof Certificate (FPC) framework can be used to define the semantics of a wide range of proof evidence. For example, such definitions exist for a number of textbook proof systems as well as for the proof evidence output from some existing theorem proving systems. An important decision in designing a proof certificate format is the choice of how many details are to be placed within certificates. Formats with fewer details are smaller and easier for theorem provers to output but they require more sophistication from checkers since checking will involve some proof reconstruction. Conversely, certificate formats containing many details are larger but are checkable by less sophisticated checkers. Since the FPC framework is based on well-established proof theory principles, proof certificates can be manipulated in meaningful ways. In this paper, we illustrate how it is possible to automate moving from implicit to explicit (*elaboration*) and from explicit to implicit (*distillation*) proof evidence via the proof checking of a *pair of proof certificates*. Performing elaboration makes it possible to transform a proof certificate with details missing into a certificate packed with enough details so that a simple kernel (without support for proof reconstruction) can check the elaborated certificate. We illustrate how trust in only a single, simple checker of explicitly described proofs can be used to provide trust in a range of theorem provers employing a range of proof structures.

1 Introduction

The study and development of programming languages have been aided by the use of (at least) two frameworks: context-free grammars (CFG) are used to define the structure of programs and structural operational semantics (SOS) [44] are used to define the evaluation and behavior of programming languages. Both of these frameworks make it possible to define the structure and meaning of a programming language in a way that is independent of a particular parser and particular compiler. Specifications in these frameworks are both mathematically rigorous and easily given prototype implementations using the logic programming paradigm [10, 24, 34, 47].

The study and development of automated and interactive reasoning systems can similarly benefit from the introduction of frameworks that are capable of defining the meaning of proof descriptions that are output by provers. Such formal semantics of proof languages make it possible to separate the production of

proofs (via possibly untrusted and complex theorem provers) from the checking of proofs (via smaller and trusted checkers). In such a setting, the provenance of a proof should not be critical for checking a proof.

Separating theorem provers from proof checkers using a simple, declarative specification of proof certificates is not new: see [27] for a historical account. For example, the LF dependently typed λ-calculus [25] was originally proposed as a framework for specifying (natural deduction) proofs and the Elf system [41] provided both type checking and inference for LF: the proof-carrying code project of [40] used LF as a target proof language. The LFSC system is an extension of the dependently typed λ-calculus with side-conditions and an implementation of it has successfully been used to check proofs coming from the SMT solvers CLSAT and CVC4 [48]. Deduction modulo [18] is another extension to dependently typed λ-terms in which rewriting is available: the Dedukti checker, based on that extension, has been successfully used to check proofs from such systems as Coq [9] and HOL [4]. In the domain of higher-order classical logic, the GAPT system [22] can check proofs given by sequent calculus, resolution, and expansion trees and allows for checking and transforming among proofs in those formats.

Foundational Proof Certificates (FPC) is a recently proposed framework for defining the semantics of a wide range of proof languages for first-order classical and intuitionistic logic [13, 16, 17]. Instead of starting with dependently typed λ-calculus, the FPC framework is based on Gentzen's more low-level notion of sequent calculus proof. FPC definitions have been formulated for resolution refutations [46], expansion trees [38] (a generalization of Herbrand disjunctions), Frege proof systems, matings [2], simply typed and dependently typed λ-terms, equality reasoning [15], tableau proofs for some modal logics [30, 31, 37], and decision procedures based on conjunctive normal forms, truth table evaluation, and the G4ip calculus [21, 50]. Additionally, FPCs have been used to formalize proof outlines [8] and have been applied to model checking [28]. As with other declarative and high-level frameworks, proof checkers for FPC specifications can be implemented using the logic programming model of computation [14, 17, 35].

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

One approach to this trade-off is to invoke the Poincaré principle [7] which states that computation traces (such as that for $2 + 2 = 4$) should be left out of a proof and reconstructed by the checker. This principle requires a checker to be complex enough to contain a (possibly small) programming language interpreter. In LFSC and the Dedukti checker, such computations are performed using deterministic functional programs. The FPC framework goes a step beyond

that by allowing nondeterministic computation as well. As is familiar from the study of finite state machines, nondeterministic specifications can be exponentially smaller than deterministic specifications: such a possibility for shortening specifications is an interesting option to exploit in specifying proof certificates. Of course, deterministic computations are instances of nondeterministic computations: similarly, FPCs can be restricted to deterministic computation when desired.

The following example illustrates a difference between requiring all details to be present in a certificate and allowing a certificate to drop some details. A proof checker for first-order classical logic could be asked to establish that a given disjunctive collection of literals, say, $L_1 \vee \ldots \vee L_n$ is provable. An explicit certificate of such a proof could be a (unordered) pair $\{i, j\} \subseteq \{1, \ldots, n\}$ such that L_i and L_j are complementary. If we allow nondeterminism, then the indexes i, j do not need to be provided: instead, we could simply confirm that there exist guesses for i and j such that literal L_i is the complement of L_j. (Of course, there may be more than one such pair of guesses.) The use of nondeterminism here is completely sensible since a systematic and naive procedure for attempting a proof of such a disjunction can reconstruct the missing details.

Since the sequent calculus can be used as the foundation for both logic programming and theorem proving, the nature and structure of nondeterministic choices in the search for sequent calculus proofs have received a lot of attention. For example, Gentzen's original LK and LJ sequent calculus proof systems [23] contained so many choices that it is hard to imagine performing meaningful proof search directly in those proof systems. Instead, those original proof systems can be replaced by *focused sequent calculus proof systems* in order to help structure nondeterminism. In particular, the common dichotomy between *don't-care* and *don't-know* nondeterminism gives rise to two different phases of focused proof construction. Don't-know nondeterminism is employed in the *positive* phase where significant choices (choices determined by, say, an oracle or a proof certificate) are chained together. Don't-care nondeterminism is employed in the *negative* phase and it is responsible for performing determinate (functional) computation. As we shall see, this second phase provides support for the Poincaré principle.

The next two sections describe and illustrate the main ideas behind focused proof systems and the FPC framework. Following that, we introduce the *pairing* FPC and illustrate how we can use it to elaborate proof certificates (introduce more details) and to distil proof certificates (remove some details). We then illustrate how such transformations of proof certificates can be used to provide trust in proof checking.

2 The Foundational Proof Certificates framework

While we restrict our attention in this paper to first-order classical logic, much of what we develop here can also be applied to first-order intuitionistic logic and to logics with higher-order quantification and fixed points. We assume that the

$$\frac{\mathit{true}_c(\Xi)}{\Xi \vdash \Theta \Uparrow t^-, \Gamma} \qquad \frac{\Xi_1 \vdash \Theta \Uparrow A, \Gamma \quad \Xi_2 \vdash \Theta \Uparrow B, \Gamma \quad \wedge_c(\Xi, \Xi_1, \Xi_2)}{\Xi \vdash \Theta \Uparrow A \wedge^- B, \Gamma}$$

$$\frac{\Xi' \vdash \Theta \Uparrow \Gamma \quad f_c(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow f^-, \Gamma} \qquad \frac{\Xi' \vdash \Theta \Uparrow A, B, \Gamma \quad \vee_c(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow A \vee^- B, \Gamma}$$

$$\frac{\Xi' y \vdash \Theta \Uparrow [y/x]B, \Gamma \quad \forall_c(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow \forall x.B, \Gamma} \qquad \frac{\Xi_1 \vdash \Theta \Downarrow A \quad \Xi_2 \vdash \Theta \Downarrow B \quad \wedge_e(\Xi, \Xi_1, \Xi_2)}{\Xi \vdash \Theta \Downarrow A \wedge^+ B}$$

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

$$\frac{\Xi_1 \vdash \Theta \Uparrow B \quad \Xi_2 \vdash \Theta \Uparrow \neg B \quad \mathit{cut}_e(\Xi, \Xi_1, \Xi_2, B)}{\Xi \vdash \Theta \Uparrow \cdot}\mathit{cut} \qquad \frac{\mathit{init}_e(\Xi, l) \quad \langle l, \neg P_a \rangle \in \Theta}{\Xi \vdash \Theta \Downarrow P_a}\mathit{init}$$

$$\frac{\Xi' \vdash \Theta, \langle l, C \rangle \Uparrow \Gamma \quad \mathit{store}_c(\Xi, \Xi', l)}{\Xi \vdash \Theta \Uparrow C, \Gamma}\mathit{store} \qquad \frac{\Xi' \vdash \Theta \Uparrow N \quad \mathit{release}_e(\Xi, \Xi')}{\Xi \vdash \Theta \Downarrow N}\mathit{release}$$

$$\frac{\Xi' \vdash \Theta \Downarrow P \quad \mathit{decide}_e(\Xi, \Xi', l) \quad \langle l, P \rangle \in \Theta \quad \mathrm{positive}(P)}{\Xi \vdash \Theta \Uparrow \cdot}\mathit{decide}$$

Here, N is a negative formula, P is a positive formula, P_a is a positive literal, and C is a positive formula or a negative literal. The $\forall$-introduction rule has the proviso that the eigenvariable y is not free in the conclusion to that occurrence of the rule.

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

reader is familiar with the one-sided version of Gentzen's LK calculus [23]. The FPC framework is layered on that sequent calculus by taking the following steps.

First, we employ the *LKF focused* sequent calculus of [29] in which proofs are divided into two alternating phases of inference rule applications. The *negative* phase uses sequents with an $\Uparrow$ and organizes the don't-care nondeterminism of rule application. Dually, the *positive* phase uses sequents with a $\Downarrow$ and is organized around don't-know nondeterminism. This proof system operates on *polarized* formulas, which differ from ordinary formulas in that there are positive and negative variants of the propositional constants t^-, $\wedge^-$, f^-, $\vee^-$, t^+, $\wedge^+$, f^+, $\vee^+$. A non-atomic formula is *positive* if its top-level connective is t^+, $\wedge^+$, f^+, $\vee^+$, or $\exists$, while a formula is *negative* if its top-level connective is t^-, $\wedge^-$, f^-, $\vee^-$, or $\forall$. (While the two variants of the propositional connectives have the same truth conditions, they behave differently in *LKF* proofs.) Literals can be given polarity arbitrarily: here we choose to fix the polarity of atoms to be positive and the polarity of negated atoms to be negative. The two kinds of sequents used in *LKF* are of the form $\vdash \Theta \Uparrow \Gamma$ and $\vdash \Theta \Downarrow B$ where Γ is a list of formulas, B is a formula, and Θ is a multiset of positive formulas or negative literals. We shall refer to the formulas in the Θ zone as *stored* formulas.

Second, the *LKF* proof system is *augmented* to get the LKF^a proof system displayed in Figure 1. This augmentation consists of three kinds of items: *certificate terms* (the schematic variable Ξ), *indexes* (the schematic variable l), and *clerk and expert predicates*. Certificate terms are threaded through all inference rules by adding such a term to all LKF^a sequents. Every *LKF* inference rule is given an additional premise involving either a *clerk* predicate (identified by

a subscripted c) or an *expert* predicate (identified by a subscripted e). These predicates are parameters to LKF^a: different ways to define these predicates will describe different styles of proof certificates that LKF^a can check. (In later sections, we shall present several different sets of definitions for these predicates.) Indexes are used to help manage the "storage and retrieval" of formulas. In particular, when in the part of the proof system used for performing all invertible rules (i.e., the don't-care nondeterminism phase), any formula whose introduction rule might not be invertible must be delayed: this is achieved by *storing* that formula. In LKF^a, when the store rule performs this duty, the formula is stored along with an *index*: subsequent references to stored formulas (in the decide and initial rules) make use of such indexes for accessing formulas. Thus, in the two kinds of sequents used by LKF^a, namely, $\Xi \vdash \Theta \Uparrow \Gamma$ and $\Xi \vdash \Theta \Downarrow B$, Ξ is a certificate term and Θ is a multiset of pairs $\langle l, C\rangle$ where l is an index and C is a positive formula or a negative literal. The clerk and expert premises are responsible for processing certificate terms and providing the continuation certificates (for any sequent premises) along with additional information that can be used to further instantiate the inference rule.

The soundness of LKF^a is immediate since an LKF^a proof contains a (one-sided) LK proof (which are known to be sound). More precisely: let B be an unpolarized formula and let $\hat{B}$ be some polarization of B (that is, the result of placing plus and minus signs on the propositional constants). If there is a proof of $\Xi \vdash \cdot \Uparrow \hat{B}$ then B is a first-order theorem since any proof of $\Xi \vdash \cdot \Uparrow \hat{B}$ can be made into an LK proof of B simply by deleting the clerk and expert premises and changing the up and down arrows into commas (as well as replacing pairs such as $\langle l, C\rangle$ in the storage context with simply C). Thus, soundness holds for this proof system *no matter how the clerks and experts are defined.*

The expert predicates used in the $\vee^+$ and $\exists$ introduction rules can examine the certificate Ξ and extract information (the value of i or the term t) and the continuation certificate Ξ'. There is no assumption that such an extraction is functional: indeed, the expert for the $\exists$ introduction might simply (nondeterministically) guess at some term. Similarly, we do not assume that there is a functional dependency between index and formulas: many formulas may be associated with the same index.

When examining proof construction in LKF^a, note that the negative ($\Uparrow$) phase is essentially *determinate*: in other words, clerks do routine computation and storage operations. On the other hand, experts can be nondeterministic: in particular, the certificate may lack specific information and the experts may simply guess at possible details. In this sense, the experts *consume* resources by either extracting information contained in a certificate term or by invoking nondeterminism.

The definition of a particular FPC is given by providing the constructors of proof certificate terms (Ξ) and of indexes (l) as well as the definition of the clerk and expert predicates. Figure 3 contains an example of a particular FPC. While the FPC framework embraces a nondeterministic model of computation behind

proof checking (and, hence, proof reconstruction), that framework obviously admits deterministic computation as well.

Proof checking a given certificate term will lead to the construction of a sequent calculus proof (in this case, in *LKF*): while the construction of such a proof helps us to trust the checking process, such a proof is *performed* and neither stored nor output. The FPC framework, however, is intended to make it possible to check many other forms of proof evidence: our clients will not need to understand sequent calculus in order to use our checker. By analogy, programmers in a high-level language such as, say, OCaml do not need to know about the many issues involved with compiling their code to bytecode or native code even though the execution of OCaml programs does generate a sequence of very low-level instructions.

3 Proof checking kernels as logic programs

Given that almost everything about the proof theory we described in the previous section is based on *relations*, the logic programming paradigm is, in principle, well suited to providing implementations of trusted proof checkers, also called *kernels*. (For an extended argument supporting this conclusion, see [35].) Many people may not wish to trust the implementations of such complex operations as unification and backtracking search, which would be inherent to a logic programming-based kernel. Also, implementations of logic programming, such as Prolog, have often supported unsound logical operations (for example, unification without the occurs-check). Fortunately, there have been many who have implemented logic programming languages that not only focus on sound deduction but also include a great deal more logic than, say, Prolog. Such systems include the Teyjus [39] and ELPI [20] implementations of λProlog [36] and the Twelf [42] and Beluga [43] implementations of LF [25].

The proof system in Figure 1 can easily be seen as a program in λProlog, a language that supports hypothetical reasoning, variable bindings, (capture-avoiding) substitution, and unification that treats logic variables and eigenvariables soundly. To illustrate how inference rules can be specified in λProlog, the following four clauses specify the ∃-introduction rule, the ∀-introduction rule, the decide rule, and the store rule.

```
sync  Xi (some B)     :- someE Xi Xi' T, sync Xi' (B T).
async Xi [all B|Rest] :- allC Xi Xi', pi w\ async (Xi' w) [B w|Rest].
async Xi nil          :- decideE Xi Xi' I, storage I P, isPos P, sync Xi' P.
async Xi [C|Rest]     :- (isPos C ; isNegAtm C),
                          storeC Xi Xi' I, storage I C => async Xi' Rest.
```

Here, provability of ⇑ and ⇓ sequents is encoded using the `async` and `sync` predicates, respectively. (Historically, the negative and positive phases have also been called *asynchronous* and *synchronous*, respectively [1].) The syntax `pi w\` denotes the universal quantification of the variable `w`: operationally, a λProlog interpreter instantiates the bound variable `w` with an eigenvariable (a new, scoped constant) when interpreting such a goal. Here, the hypothetical reasoning mechanism of λProlog (the symbol `=>` denotes implication in a goal) is used to associate

indexes with stored (positive or atomic) formulas (using the `storage` predicate): as a result, the Θ zone in Figure 1 does not need to be an explicit argument in the specification of the kernel since it is encoded as hypothetical assumptions within λProlog.

The λProlog specification of LKF^a can be viewed as a trustworthy kernel. (Section 6 describes another trustworthy but more limited kernel written in OCaml.) Someone interested in having their proofs checked by this kernel must provide (in λProlog) the definition of certificate and index terms (of type `cert` and `index` respectively) and the definition of the clerk and expert predicates. The next section provides a few examples of such specifications.

4 Example FPCs

In this section, we provide the FPC definitions of three different proof formats.

4.1 Controlling the decide rule

The only place where Gentzen's structural rule of contraction is used within LKF^a is the decide rule. If contractions can be sufficiently controlled, naive search algorithms can often become decision procedures. To that end, it is easy to design a proof certificate that describes any LKF^a proof with an upper bound on its *decide depth* (that is, the maximum number of decide inference rules along any path in the proof). To convert this observation into an FPC, we need only one index, say, `indx` and we use just one form of certificate, namely, the term `(dd D)` where `D` is a natural number. Below is the specification of the clerk and expert predicates (here, `s` is the non-zero natural number constructor).

```
andNegC      (dd D) (dd D) (dd D).      orPosE       (dd D) (dd D) Choice.
andPosE      (dd D) (dd D) (dd D).      someE        (dd D) (dd D) T.
falseC       (dd D) (dd D).             storeC       (dd D) (dd D) indx.
releaseE     (dd D) (dd D).             initialE     (dd D) indx.
orNegC       (dd D) (dd D).             trueE        (dd D).
allC         (dd D) (x\ dd D).          decideE      (dd (s D)) (dd D) indx.
```

These clerks and experts leave the bound `D` untouched except for the `decideE` (the $decide_e$ predicate in Figure 1) which decrements that bound. The experts for the positive disjunction and the existential quantifier are nondeterministic since, for example, every term `T` is a possible instantiation allowed by the `someE` expert specification. The two predicates that deal with indexes—`storeC` and `decideE`—always make use of the same index. Since the cut expert `cutE` is not defined, this FPC will only allow checking cut-free proofs. This FPC provides a high-level means of describing proofs in the sense that the goal formula `(async (dd N) [B])` is provable from the kernel clauses and the clerk and expert clauses above if and only if `B` has an *LKF* proof of decide depth `N` or less.

Many other descriptions of proofs via FPCs are possible. For example, it is easy to design a certificate that is just a tree of nodes labeled with formulas that are used as cut formulas: all other details of the proof are unspecified. Another certificate design could be a tree of nodes labeled with indexes that record

when an index is used during the decide inference rule. For now, we consider such certificates as descriptive and we make no assumption that checking that a given certificate holds for a given formula is decidable: with many high-level descriptions of proofs, such checking might indeed be undecidable.

4.2 Conjunctive normal form: a decision procedure as an FPC

Converting a propositional formula to conjunctive normal form provides an (expensive) decision procedure for determining whether or not a propositional formula is a tautology. The following FPC encodes this decision procedure. First, we choose to polarize all propositional connectives negatively. An *LKF* proof with only such a polarized formula in its conclusion consists of exactly one large negative phase that has, as premises, sequents containing only stored literals. Such a sequent is provable if and only if there is an index, say i, that labels a positive literal and the complement of that literal exists with the index j. We need only one certificate constructor, say `cnf`, and one index, say, `lit`. The clerk and expert predicates for this FPC can be defined as follows.

```
andNegC    cnf cnf cnf.          initialE          cnf lit.
orNegC         cnf cnf.          decideE       cnf cnf lit.
falseC         cnf cnf.          storeC        cnf cnf lit.
releaseE       cnf cnf.
```

In this case, the proof certificate size is constant (just the token `cnf`) while checking time can be exponential.

A simple variation of this FPC would be a certificate that stores every literal with different indexes and then accumulates all pairs $\langle i, j\rangle$ such that i and j are complementary literals within the same premise. Such an FPC essentially contains a *mating* [3]. Expansion trees [12, 38] can also be accounted for by first admitting quantificational formulas and then storing in certificates the instantiations for the existential quantifiers.

4.3 Resolution refutations

An FPC defining binary resolution refutations has been given in [16] and we describe it briefly here since the experimental results described in Section 7 build on this example. A *clause* is a formula of the form $\forall x_1 \ldots \forall x_p.\ [L_1 \vee \cdots \vee L_q]$, where $p, q \geq 0$ and $L_1, \ldots, L_q$ are all literals (i.e., atoms or negated atoms). As polarized formulas, disjunctions in clauses are polarized negatively. A resolution refutation is essentially two lists of clauses $C_1, \ldots, C_n$ and $C_{n+1}, \ldots, C_m$ where each element of the second list is also accompanied with a *justification* which is a triple of indexes $\langle i, j, k\rangle$ that carries the claim that C_k is the result of resolving C_i and C_j. We also assume that the last clause C_m is the empty clause, written as f^-. The first list of clauses is used to form the theorem to be proved, namely, $\vdash \neg C_1 \vee \cdots \vee \neg C_m$, where by $\neg C_i$ we mean the negation normal form of the negation of clause C_i.

The main element of a resolution proof is the claim that two clauses, say, C_i and C_j *resolve* to yield a third clause C_k: that is, that the triple $\langle i, j, k\rangle$ is the

justification associated to C_k. If that claim is correct, then it is the case that the sequent $\vdash \neg C_i, \neg C_j \Uparrow C_k$ must be provable in *LKF* with a focused proof of decide depth three or less. Also, every resolution triple corresponds to a cut, as illustrated by the inference rule of *LKF*. In particular, this figure is part of the translation of the claim that C_i and C_j resolve to yield clause C_{n+1} where both i and j are members of $\{1, \ldots, n\}$.

$$\dfrac{\vdash \neg C_i, \neg C_j \Uparrow C_{n+1} \qquad \dfrac{\vdash \neg C_1, \ldots, \neg C_n, \neg C_{n+1} \Uparrow \cdot}{\vdash \neg C_1, \ldots, \neg C_n \Uparrow \neg C_{n+1}}\, store}{\vdash \neg C_1, \ldots, \neg C_n \Uparrow \cdot}\, cut$$

Here, the left premise is a small proof that involves at most three decide rules (one on both i and j and one on an unspecified literal): a certificate can easily be written that describes how such a proof might be constructed. The right premise leads to yet another use of cut in order to check the next claimed resolution triple. Such proof construction ends when $\neg C_m$ appears in the sequent on the extreme right branch of the proof: since that formula is t^+, that branch is finished.

We shall not present a formal definition of resolution refutations as an FPC here in order to save space: the interested reader can find such definitions in [13, 16, 17]. There are, of course, a lot of choices as to how much information is placed into a certificate for resolution. For example, the exact instantiations used to compute resolvents could be explicitly added or not. If the instantiations are not part of the certificate, then checking the certificate would require the checker to reconstruct those substitution terms: a kernel based on a logic programming engine (as described in Section 3) is capable of applying unification and backtracking search in order to produce such instantiations. If one is not willing to trust an implementation of unification and backtracking search, it is possible (as we describe later) to design a proof certificate format that includes such substitution information.

Another piece of information that is not explicitly captured in the usual definition of resolution is the order in which the clauses C_i and C_j are applied in order to build the subproof justifying the resolution triple $\langle C_i, C_j, C_m \rangle$. In this polarized setting, this order is important and certificates can be designed to attempt both orders or to use the explicit order given in the certificate. This difference in design will not affect the size of certificates but can affect the time required to check certificates (see Section 7).

5 Pairing certificates

Because FPC definitions of proof evidence are declarative (in contrast to procedural), some formal manipulations of proof certificates are enabled easily. We illustrate how the formal *pairing* of two certificates can be used to transform proof certificates into either more or less explicit certificates.

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

Fig. 2. The pairing FPC

5.1 The pairing FPC

Consider checking a proof certificate for a resolution refutation that does not contain the substitutions used to compute a resolvent. Since the checking process computes a detailed focused sequent in the background, that process must compute all the substitution terms required by sequent calculus proofs. If we could check *in parallel* a second certificate that allows for storing such substitution terms, then those instances could be inserted into the second, more explicit certificate. Fortunately, it is a simple matter to do just such parallel checking of two proof certificates.

Let `<c>` be an infix constructor of type `cert -> cert -> cert` and let `<i>` be an infix constructor of type `index -> index -> index`. The full specification (using λProlog syntax) of the FPC for pairing is given in Figure 2. This pairing operation allows for the parallel checking of two certificates: clearly, both certificates must eventually expand into the same underlying sequent calculus proof but those certificates could retain different amounts of detail from each other. Note that the definition of pairing for the existential expert ensures that both certificates *agree* on the same information (here a witness t). Of course, one (or both) of those certificates do not need to actually contain the witness information. While paired certificates must be able to agree on substitution terms, choices for (positive) disjunctions, and cut formulas, they will not need to agree on the notion of index. Instead, we use the pairing constructor `<i>` to form an index out of two indexes.

While the transformations between proof certificates that can take place using this pairing FPC are useful (as we argue in the following sections), such transformations are also limited. For example, pairing cannot be used to transform a proof certificate based on, say conjunctive normal forms, into one based on resolutions, since the former makes no use of cut and the latter contains cuts. The pairing of two such certificates will (almost) always fail to succeed. Pairing is really limited to transforming within the spectrum of "many details, fewer details" and not between two different styles of proof. Thus, it is possible to transform a proof certificate encoding resolution that does not contain substi-

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

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

Fig. 3. A certificate format including maximal details

tution terms to one that does contain substitution terms. The reverse is also possible.

5.2 A maximally explicit FPC

We can define a *maximally explicit* FPC that contains all the information that is explicitly needed to fill in all details in the augmented inference rules in Figure 1. In principle, this certificate format records the full trace of the underlying sequent calculus proof computed during the execution of the kernel. The FPC in Figure 3 is capable of storing all such details. Note that the natural number argument of `max` is used by the store clerk to choose a fresh index for every stored formula. The constructors of type `max` are different nodes of a symbolic proof tree, holding all information needed by the clerks and experts without recording the actual proof derivation. The constructors are as follows: `max0` is a leaf node, `max1` is a unary node, `max2` is a binary node, `maxv` is used to bind an eigenvariable to the rest of the tree, `maxt` is annotated with a term, `maxf` with a cut formula, `maxc` with a (disjunctive) choice, and `maxi` with an index.

Such a proof certificate can be automatically obtained through elaboration of any other proof certificate and the use of the pairing of certificates. For example, if the sequent `dd (s (s z))` $\vdash \cdot \Uparrow F$ is provable then calling the checker with the sequent `dd (s (s z)) <c> (max z` X`)` $\vdash \cdot \Uparrow F$, where X is a logic variable of type `max`, will build a fully explicit proof object.

5.3 Elaboration and distillation of certificates

The kernel is building a formal sequent proof which is not explicitly stored but is, in a sense, performed by the kernel. It is the performance of such a sequent calculus proof that helps to provide trust in the kernel. If a certificate is lacking necessary details for building such a sequent calculus proof (such as substitution instances), a kernel could attempt to reconstruct those details. The formal pairing of certificates described above links two certificates that lead to the same performance of a sequent calculus proof: in the logic programming setting, it is completely possible to see such linking of certificates as a means to transform one certificate to another certificate. The term *elaboration* will be used to refer to the process of transforming an implicit proof certificate to a more explicit proof certificate. The converse operation, called *distillation*, can also be performed: during such distillation, certain proof details can be discarded.

Since a given proof certificate can be elaborated into a number of different sequent calculus proofs, certificates can be used to provide high-level descriptions of *classes* of proofs. For example, FPCs have been used to describe *proof outlines* [8]: using a logic programming based kernel to check such a proof outline means that the kernel will attempt to reconstruct a complete proof based on the information given in the outline. If such a reconstruction is possible, pairing the proof checking of a proof outline with an explicit form of FPC would mean that the missing proof details could be recorded. In a similar fashion, Martin Davis's notion of "obvious logical inference" [19] can be described easily as an FPC: here, an inference is "obvious" if all quantifiers are instantiated at most once. Thus, using a kernel to attempt to check such an FPC against a specific formula essentially implements the check of whether or not an "obvious inference" can complete the proof.

Since we shall focus on certificate elaboration in the rest of this paper, we conclude this section with a few comments about certificate distillation. Consider, for example, a proof certificate that contains substitution instances for all quantifiers that appear within a proof. In some situations, such terms might be large and their occurrences within a certificate could make a certificate large. In the first-order logic setting, however, if a certificate stores instead linkage or mating information between literals in a proof, then the implied unification problems can be used to infer the missing substitutions (assuming that the kernel contains a trusted implementation of unification). The resulting certificate could well be much smaller: checking them could, however, involve a possibly large unification problem to be performed. Besides such approaches to proof compression, distilling can provide an elegant way to answer questions such as: What lemmas have been used in this proof? How deep (counting decide rules) is a proof? What substitution terms were used in a certain subproof? Certificates that retain only some coarse information such as this could be used to provide some high-level insights into the structure of a given proof.

6 The kernel as a functional program

Given that the maximally explicit certificate contains all the information needed to build a (focused) sequent calculus proof, a proof checker for only that FPC does not need to perform unification or backtracking search. Such a checker may be simple and easy to analyze and trust. To demonstrate this possibility, we have implemented in OCaml a proof checker for the maximal FPC in Figure 3.

MaxChecker is an OCaml program of about 200 lines of code (available online at `proofcert.github.io`). Separate from the kernel is a parser that reads from an input that contains three items: (*i*) a collection of non-logical constants and their (simple) types; (*ii*) a polarized version of a formula (the proposed theorem); and (*iii*) a proof certificate in the maximal FPC format. The kernel is then asked to check whether or not the given certificate yields a proof of the proposed theorem. If this check is successful, the kernel prints out the (unpolarized) theorem as a means to confirm what formula it has actually checked.

As Pollack has argued in [45], the printer and parser of our system must be trusted to be faithfully representing the formulas that they input and output. Here, we assume that that concern is addressed in standard ways: in our particular tool, we have used standard parser generating tools in order to link trust in our tool with trust in a well engineered and frequently used tool.

It is now an easy matter to describe the architecture of a proof checker that we can use to check *any* FPC-defined proof certificate while only needing to trust MaxChecker. First, use the flexible λProlog based (or equivalent) interpreter to do the formal checking of any proof certificate accompanied by its FPC definition. If we do that checking using both the maximal and pairing FPCs then the maximal certificate (the most explicit form of the input certificate) can be extracted. Second, run MaxChecker on this final and explicit certificate.

We can push this issue of trust another step. Since the MaxChecker is a simple terminating functional program, it should be a simple matter to implement it within, say, the Coq proof assistant, and formally prove in Coq that a successful check leads to a formal proof in, say, Gentzen's LK and LJ proof systems. By reflecting [11, 26] these weaker proof systems into Coq (including the axiom of excluded-middle for classical logic proofs), the chaining of a flexible certificate elaborator with the Coq based MaxChecker can then be used to get Coq to accept proofs from a range of other proof systems. The first author plans such a Coq implementation as part of his Ph.D. dissertation.

It is possible (at least in some logical settings) to leave out some details from a proof certificate while still providing for determinant proof checking. For example, consider the variant of the maximal FPC in which no substitution terms are stored: specifically, redefine the type as well as the clerk and expert predicates in Figure 3 for the `maxv` and `maxt` constructors as follows.

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

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

Certificates of this modified format will not contain any reference to eigenvariables or to substitution terms (existential witnesses). A proof checker for such certificates can, however, use so-called *logic variables* instead of explicit witness terms and then perform unification during the implementation of the initial rule. Since the unification of first-order terms (even in the presence of eigenvariables and their associated constraints) is determinate, such proof checking will not involve the need to perform backtracking search. The main downside for this variant of the maximally explicit certificate is that checking will involve the somewhat more complex operation of unification. Of course, such unification must deal with either Skolem functions or eigenvariables in order to address quantifier alternation. (λProlog treats eigenvariables directly since it implements unification under a mixed quantifier prefix [33].)

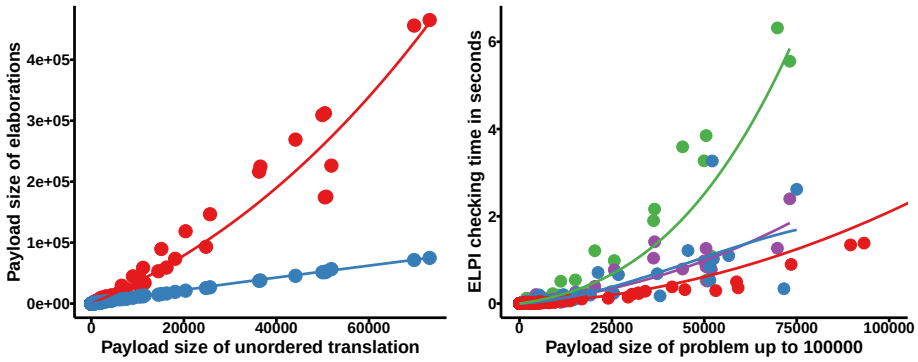
7 Some experiments with certificate elaboration

We have experimented with various uses of certificate pairing and we report briefly on some of those experiments here. The code, data, and results from these experiments are available at `proofcert.github.io`.

We have used pairing in our λProlog checker in order to distil and elaborate a number of matrix-style (cut-free) proofs: for example, we have elaborated the `cnf` proof certificates (Section 4.2) into matings [3] and elaborated the decide depth FPC into an FPC based on oracle strings (see [17, Section 7]). Furthermore, these various certificate formats can be elaborated to the maximally explicit certificate. Since these certificate formats are seldom used in actual theorem provers, we describe below our more extensive experiments with resolution refutations.

We have defined three variations on the FPC definition of resolution with factoring that is given in Section 4.3. Let us call the FPC given above in Section 4.3 `unordered-without`, meaning that that format does not store substitution information and that when the certificate contains the triple $\langle i, j, k\rangle$, the order in which one decides on i and j is unknown. (Existing resolution systems might not offer to order these indexes.) We also defined the `ordered-without` format: in that case, the triple $\langle i, j, k\rangle$ means that i must be decided on before j. This certificate format is a simple modification of the one in Section 4.3: just one line of the `decide` expert is deleted from the `unordered-without` FPC definition. Finally, a third variant `ordered-with` was also defined: this certificate retains substitution and eigenvariable information as well.

Our goal is to certify the output of a *bona fide*, complex proving tool, that is sufficiently powerful to provide us with reasonably sized and publicly available *proof corpora*. To that end, we have selected Prover9 [32], a legacy, automated theorem prover of modest capabilities: an important feature for our experiment is that Prover9's output exposes a relatively simple and well-documented resolution calculus. We have taken the full set of Prover9 refutations in the TPTP library [49]—a total of 2668 in version 6.4.0—and excluded 52 files with irregular formatting (the resulting set of examples is precisely that of version 6.3.0). Of these, 978 fall in the fragment supported by the resolution FPCs; 27 are empty



Data series: unordered-without, ordered-without, ordered-with, maximal.

Fig. 4. Complexity of certificate elaboration

proofs that refute false. The two largest problems are extreme outliers, also excluded since they would be of limited utility to establish or confirm trends. Each problem is expanded into a detailed proof with Prover9's `Prooftrans` tool. This proof is parsed and a proof certificate for the unordered FPC is extracted, along with type signatures for atoms and terms. The λProlog runtime uses pairing to elaborate and check the more explicit certificates, and it outputs the formula and the maximally explicit certificate to MaxChecker.

Figure 4 shows a summary of our experiments with this output from Prover9. The size of a formula or term is simply a count of the number of constructors in that formula or term. The size of resolution certificates is defined here to be the sum of the sizes of the initial and derived clauses along with their justifications. The size of maximally explicit certificates is defined as the size of the actual certificate term plus the size of the original set of clauses. Certificate sizes grow as they are made more explicit, but the blowup here is bounded by small constants. Elaborating from `unordered-without` to `ordered-without` causes no change in size while elaborating further to `ordered-with` generally grows certificates by 16%. Finally, elaborating to the maximally explicit certificate causes an increase by an average factor of 2.8 (although that factor ranges from 1.02 to 6.54). Here, a natural number is counted as one symbol; the unary representation of numbers causes a blowup in size (the average factor being 5.8 with range from 1.2 to 361).

The second graph in Figure 4 shows that the more detailed a certificate is, the faster it is to check. For example, a certificate in the `unordered-without` format of 75000 symbols can be checked in 6 or more seconds: a similarly sized certificate in the maximally explicit format can be checked in less than a second.

The choice of Teyjus [39] or ELPI [20] as λProlog runtime yields qualitatively similar results, but shows significant performance differences and asymmetries, especially in the substantial elaboration overhead; in general, ELPI is faster. The checking times for the MaxChecker on the large, maximally explicit certificates running in OCaml are negligible compared to elaboration times within λProlog:

in particular, MaxChecker always ran in less than 0.01 seconds on each example displayed in Figure 4.

We have successfully checked all resolution refutations produced by Prover9 that involved binary resolution and factoring. In order to capture all of Prover9's proofs in the TPTP repository we need to add support for paramodulation: the FPC for paramodulation given in [14] is a starting point.

8 Conclusions

In this paper, we have analyzed the nature of some simple proof structures whose definitions are established using the Foundational Proof Certificate (FPC) framework. We have illustrated several versions of such proofs that occupy different positions on the spectrum between implicit and explicit proof. Both extremes are possible with the FPC setting. Of course, the nature and effectiveness of proof checkers can be greatly impacted by how implicit or explicit such proof formats are. As we illustrated, it is possible for implicit proof structures to be rather small but expensive to check: for example, constant sized with exponential checking time (Section 4.2). On the other hand, they can also contain more details and be much easier to check. We have also noted that logic programming provides a simple, immediate, and sound proof checker for any formal FPC definition.

We then introduced the notion of formally pairing two certificates into one: when such a paired certificate is checked, it is possible for information to flow between proof certificates (which may store different aspects of a proof) with the implementation of the kernel (which must ultimately generate all details of a proof). In this way, checking an implicit certificate can lead to the construction of a more explicit certificate. In fact, we illustrated how it was possible to define a maximally explicit proof certificate in which enough details are present that a simple functional program (in our case, written in OCaml) is able to check the proof without needing backtracking search and unification. As such, if one is not willing to trust a logic programming checker, it is possible to use the logic programming checker to expand an implicit proof to a maximally explicit proof certificate and then certify the answer using the simpler (presumably) trusted functional program.

The pairing of proof certificates can be used with other tasks elaborating certificates. Distilling of proofs, the converse of elaboration, might also be useful in the analysis of proofs. For example, pairing can be used to extract from any certificate the tree of cut formulas used within it or to compute its decide depth.

While the discussion in this paper has been limited to treating classical first-order logic, focusing proof systems and the FPC framework have also been proposed for first-order intuitionistic logic [17, 29] as well as logics extended with least and greatest fixed points [5, 6]. As a result, most of the points described in this paper can also be applied to those settings as well.

Acknowledgement. We thank the anonymous reviewers for their comments on an earlier version of this paper. This work was funded, in part, by the ERC Advanced Grant ProofCert.

References

1. J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.
2. P. B. Andrews. Refutations by matings. *IEEE Trans. Computers*, 25(8):801–807, 1976.
3. P. B. Andrews. Theorem proving via general matings. *J. ACM*, 28(2):193–214, 1981.
4. A. Assaf and G. Burel. Translating HOL to Dedukti. In C. Kaliszyk and A. Paskevich, editors, *Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015*, volume 186 of *EPTCS*, pages 74–88, 2015.
5. D. Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1), Apr. 2012.
6. D. Baelde and D. Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, volume 4790 of *LNCS*, pages 92–106, 2007.
7. H. Barendregt and E. Barendsen. Autarkic computations in formal proofs. *J. of Automated Reasoning*, 28(3):321–336, 2002.
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*, 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. P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Third Annual Symposium on Software Development Environments (SDE3)*, pages 14–24, Boston, 1988.
11. S. Boutin. Using reflection to build efficient and certified decision procedures. In *International Symposium on Theoretical Aspects of Computer Software*, pages 515–529. Springer, 1997.
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. Z. Chihani. *Certification of First-order proofs in classical and intuitionistic logics*. PhD thesis, Ecole Polytechnique, Aug. 2015.
14. 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)*, number 9323 in LNCS, pages 201–210. Springer, 2015.
15. 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.*, number 323 in ENTCS, 2016.
16. Z. Chihani, D. Miller, and F. Renaud. Foundational proof certificates in first-order logic. In M. P. Bonacina, editor, *CADE 24: Conference on Automated Deduction 2013*, number 7898 in LNAI, pages 162–177, 2013.
17. Z. Chihani, D. Miller, and F. Renaud. A semantic framework for proof evidence. *J. of Automated Reasoning*, 2016. doi:10.1007/s10817-016-9380-6.

18. D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.
19. M. Davis. Obvious logical inferences. In A. Drinan, editor, *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, pages 530–531, Los Altos, CA, Aug. 1991. William Kaufmann.
20. 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*, volume 9450 of *LNCS*, pages 460–468. Springer, 2015.
21. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. of Symbolic Logic*, 57(3):795–807, Sept. 1992.
22. G. Ebner, S. Hetzl, G. Reis, M. Riener, S. Wolfsteiner, and S. Zivota. System description: GAPT 2.0. In N. Olivetti and A. Tiwari, editors, *Proceedings of the 8th International Joint Conference on Automated Reasoning, IJCAR 2016*, volume 9706 of *LNCS*, pages 293–301. Springer, 2016.
23. G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935.
24. J. Hannan. Extended natural semantics. *J. of Functional Programming*, 3(2):123–152, Apr. 1993.
25. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
26. J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical report, Citeseer, 1995.
27. J. Harrison, J. Urban, and F. Wiedijk. History of interactive theorem proving. In J. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 135–214. North Holland, 2014.
28. 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.
29. C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.
30. T. Libal and M. Volpe. Certification of Prefixed Tableau Proofs for Modal Logic. In *the Seventh International Symposium on Games, Automata, Logics and Formal Verification (GandALF 2016)*, number 226 in EPTCS, pages 257–271, Catania, Italy, Sept. 2016.
31. S. Marin, D. Miller, and M. Volpe. A focused framework for emulating modal proof systems. In L. Beklemishev, S. Demri, and A. Máté, editors, *11th Conference on Advances in Modal Logic*, number 11 in Advances in Modal Logic, pages 469–488, Budapest, Hungary, Aug. 2016. College Publications.
32. W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2010.
33. D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.

34. D. Miller. Formalizing operational semantic specifications in logic. In *Proceedings of the 17th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2008)*, volume 246, pages 147–165, Aug. 2009.
35. D. Miller. Proof checking and logic programming. *Formal Aspects of Computing*, 29(3):383–399, 2017.
36. D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.
37. 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)*, number 9450 in LNCS, pages 266–280, Nov. 2015.
38. D. A. Miller. Expansion tree proofs and their conversion to natural deduction proofs. In R. E. Shostak, editor, *Seventh Conference on Automated Deduction*, volume 170 of *LNCS*, pages 375–393, Napa CA, May 1984. Springer.
39. G. Nadathur and D. J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 287–291, Trento, 1999. Springer.
40. G. C. Necula. Proof-carrying code. In *Conference Record of the 24th Symposium on Principles of Programming Languages 97*, pages 106–119, Paris, France, 1997. ACM Press.
41. F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *4th Symp. on Logic in Computer Science*, pages 313–321, Monterey, CA, June 1989.
42. F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
43. B. Pientka and J. Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, number 6173 in LNCS, pages 15–21, 2010.
44. G. D. Plotkin. A structural approach to operational semantics. *J. of Logic and Algebraic Programming*, 60-61:17–139, 2004.
45. R. Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 1998.
46. J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12:23–41, Jan. 1965.
47. 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.
48. 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.
49. G. Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
50. A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 2 edition, 2000.