An interactive assistant for the definition of proof certificates

Roberto Blanco, Zakaria Chihani

An interactive assistant for the definition of proof certificates

Roberto Blanco[1] and Zakaria Chihani[2]

[1] Inria and LIX/École polytechnique
Palaiseau, France
roberto.blanco@inria.fr
[2] CEA LIST, Software Security Lab
Gif-sur-Yvette, France
zakaria.chihani@cea.fr

Abstract

The *Foundational Proof Certificate* (FPC) approach to proof evidence offers a flexible framework for the formal definition of proof semantics, described through its relationship to focused proof systems. The certificates thus produced by tools are executable when interpreted on top of a suitable logic engine, and can therefore be independently verified by trusted proof checkers.

The fundamental obstacle encountered here lies in translating the proof evidence produced by a tool in the terms of a formal definition in the system. These formal definitions are akin to domain-specific languages (in which proofs can be written) programmed in the assembly language of the underlying proof systems: a delicate task for which both expert knowledge and great care are needed.

To facilitate broader adoption, we begin to explore techniques that abstract away part of this complexity and bring the FPC framework closer to a user-friendly, programmable platform in which a wide range of high-level certificate definitions can be easily encoded.

Keywords: focused proof system, foundational proof certificate, proof checking

1 Introduction

The notion of proof as a document that attests to the truth of a certain mathematical property finds ample support in the ever-growing myriad of software tools that facilitate the production and verification of such entities: automated theorem provers, proof assistants, model checkers, programming languages, etc. However, there remain fundamental issues of what constitutes acceptable proof evidence of the truth of a statement; how such information should be presented and communicated; and how its foundations can be assessed, deemed to be compatible with a certain logical framework, and, possibly, reused: questions the proliferation of tools and standards, as well as reasoning frameworks, only exacerbates. Some exchange format standards have arisen and are in wide use [16], but the critical gap between the syntax and the semantics of proof evidence remains [4].

In response to this situation, Dale Miller and colleagues have developed a general framework for the expression of proof evidence in the form of *Foundational Proof Certificates* (henceforth, FPCs) [13]. Central to this paradigm is the notion of certificate: a document of which, given a suitable definition, acts as a universal representative of a proof, which exists independently of any and all tools — however central software is to the production and treatment of these. Under this discipline, a separation of concerns is established in the dichotomy of the prover, responsible for producing proofs in the form of FPCs, and the checker, in charge of their

independent verification. The proof checker, then, elaborates a certificate into a formal proof. The following four desiderata synthesize the fundamental requirements of such a program.

1. A simple checker can, in principle, check if a proof certificate denotes a proof.
2. The format for proof certificates must support a wide range of proof options.
3. A proof certificate is intended to denote a proof in the sense of structural proof theory.
4. A proof certificate can simply leave out details of the intended proof.

Harnessing recent advances in proof theory, crystallized in the form of various focused proof systems [1] and their metatheories, these requirements have been shown to be well within reach. These systems become the "rules of chemistry" which enable us to naturally and efficiently take "atoms" of inference and compose them into more complex, higher-level "molecules" of inference. FPCs offer a general framework to define and describe proofs in a certain logic, regardless of provenance, and verify their truthfulness by means of a checker that embodies the aforementioned "rules," in terms of which the certificates are ultimately expressed. The remarkable fact that universal checkers that embody these rules can and should be technically simple and their correctness easy to ascertain, even formally, naturally creates a delegation of responsibilities, in which the burden of correctness is lifted from the shoulders of the complex programs that are provers, and trust in the overall process increases, among other benefits.

There remains the practical question of adoption as a practical standard for communication used by software provers. As will be expounded in following sections, the expressive power of FPCs and the capabilities of proof checkers have been studied through a number of implementations and definitions of the semantics of a range of representative proof evidence. However, the translation of the complex, heterogeneous, ad hoc proof formats of the many provers of interest in the common language of FPCs presents us with a substantial technical challenge.

The theoretical foundations of proof certificates have been compared with the basic rules of chemistry in their proposals. We liken them, rather, to a fantastic new assembler, one that programs a machine that is based purely on logic. The definition of the semantics of a certain proof certificate — which we have previously characterized as PRICEs [8] — essentially specifies a *domain-specific language* in which proof evidence of a theorem can be expressed: the concrete certificate that contains this information acts as a program written in the language of the PRICE and interpreted on the checker that implements the assembly of the chosen architectural logic. The analogy is apt insofar as it reflects the deep ties to the intricacies of the proof systems: so far, programming these PRICEs has been the delicate domain of the expert, and while ground has been broken in the connection between concrete proof assistants and suitable logics, it is laborious to extend these results to arbitrary proving software, commonly closer to the proof normalization of functional programming than to the proof search that is native to this effort — in contrast with Dedukti, [6] which endeavors to solve similar questions through different methods.

For Foundational Proof Certificates to fulfill their potential, we believe it is necessary to develop for them a full-fledged discipline of programming. In this way, they can cease to be the exclusive domain of the logician, and be understood and used by the authors of the various families of theorem proving tools, thus becoming easily integrated in them and paving the way for the communication of proofs between tools that is one of the goals of this research effort. We find this a pragmatic and necessary goal, and set out to commence its development in this paper.

The rest of the paper is organized as follows. Section 2 furnishes an introduction to the FPC framework and presents the systems for classical logic that will be employed a means

of illustration and experimentation. Section 3 discusses the current state of FPC programming: the problem domains to which it has been successfully applied to date, and the required methodology, as well as the limitations of this approach. In response to these, Section 4 proposes an assistant to simplify, and make the creation and maintenance of PRICEs more robust and accessible to non-specialists; Section 5 describes a prototype implementation, codenamed `sunshine`, that encodes these precepts in a simple, meaningful logical framework. Finally, Section 6 discusses further developments that we consider necessary in the technical evolution of FPCs as a discipline of wide reach and practical impact.

2 Foundational Proof Certificates

2.1 Focused sequent calculus

The "rules of chemistry" and "atoms of inference" referred to in Section 1 are precisely those of the sequent calculus, in which the proof system imposes no restrictions whatsoever on the shapes of proofs. With the addition of focusing, proofs are decomposed in alternating *asynchronous* and *synchronous* phases, which constitute a first element of order in the otherwise chaotic sequent calculus. Miller's desiderata guided the design of a framework for Foundational Proof Certificates, which will be introduced now. The presentation will be centered on Gentzen's classical calculus *LK*, and its focused variant *LKF* [11].

Consider the proof system in Figure 1, where all annotations in blue are disregarded: this is the system *LKF*. A focused system operates on *polarized formulas*, obtained from regular, unpolarized formulas by replacing each logical connective with a polarized version of it, positive or negative, and by assigning a polarity, again positive or negative, to each constant. In first-order classical logic, the four connectives $\wedge$ (conjunction), $\vee$ (disjunction), t (true), f (false) exist in both positive and negative polarities; the universal quantifier $\forall$ is always negative, whereas the existential quantifier $\exists$ is always positive. The polarity of a formula is that of its top-level connective. Formulas are always in negation normal form, so that the scope of negation is exactly atomic, and thus negation corresponds to a simple polarity flip. In the description that follows, by "formula" we will mean a polarized formula.

It is visible from the inference rules that constitute the calculus that sequents can be of two, distinct types, corresponding to the two kinds of phases.

1. **Up-arrow sequents** $\vdash \Gamma \Uparrow \Theta$, related to the asynchronous phase, where Γ is a multiset of formulas and Θ is a list of formulas. The rules of this phase are invertible and involve exclusively up-arrow sequents in both conclusion and premises. Due to this property, rules can be applied to saturation; by convention, in order from head to tail of the list Θ.
2. **Down-arrow sequents** $\vdash \Gamma \Downarrow B$, related to the synchronous phase, where Γ is a multiset of formulas and B is a formula. Non-invertible rules are applied to a single formula, B, that is the focus of the phase, making local choices and introducing backtracking points as demanded by the rules, until no more actions avail.

In both classes of sequents, the left-hand side Γ represents a *storage* of formulas that exist in opposition to the *workbench* (respectively, Θ and B). Phase transitions are arbitrated through the *structural rules*, which move synchronous formulas to storage during the asynchronous phase and select ("decide on") one of these to focus on in the transition from asynchronous to synchronous (contraction), and release an asynchronous formula in the transition from synchronous to asynchronous. A final group of inference rules deals with identity (weakening) and cut. As proved by Liang and Miller, theoremhood of a formula B of *LK* is equivalent to provability of

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

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

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

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

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

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

P is a positive formula. N is a negative formula. P_a is a positive literal. C is a positive formula or a negative literal. $\neg B$ is the negation of B in negation normal form. The proviso † is the eigenvariable restriction that y must not occur free in Ξ, Θ, Γ, and B.

Figure 1: The augmented *LKF* proof system LKF^a

the entry point sequent $\vdash \cdot \Uparrow \hat{B}$, where $\cdot$ is the empty storage and $\hat{B}$ is an arbitrary polarization of B; moreover, the cut-elimination property holds in the system.

2.2 Augmenting the calculus

Consider now Figure 1 in its entirety. Each premise and conclusion in every inference rule has been enriched with a *certificate term*, Ξ. Moreover, an additional "premise" has been added to every rule — except the asynchronous rule for true. These premises relate the certificate term of the conclusion with the certificate terms, and any other necessary information, required by the premises to proceed with proof search: which branch of a disjunction to pick, which stored formula to select, how to instantiate an existential variable. In addition, the formula storage zone Γ becomes a multiset of *indexed formulas*. By furnishing declarations (syntax) and definitions (semantics) to these extensions, certification is enabled.

Because *LKF* is sound and complete and all these extensions can accomplish is either enabling or disabling certain inference rules based on the certificate term in their conclusion, one can prove directly, by erasure of these restrictions, that soundness is preserved. What we gain

is a powerful discipline that brings order into the maelstrom of indiscriminate proof search.

The semantics of an FPC definition, which describes a family of certification strategies, is what we call PRICE. There are five groups of elements that constitute a PRICE.

1. **Polarization.** A polarization strategy determines how to move from unpolarized formulas to their polarized versions (the other direction is direct by erasure). As we have seen, the choice of polarities by itself does not affect provability, but its interactions with the other members of a PRICE may. Despite this seeming leniency, the role of polarities should not be underestimated: they have a strong impact of the proofs that can be found for a given theorem, and can represent the difference between exponential, brute force search and intent, efficient navigation along a predetermined proof path.

2. **Certificate terms (the "R(egions)" of a PRICE).** Conceptually, certificate terms represent the state, or the region of a proof in which we find ourselves at a given point in the derivation. As this state changes, so do the certificates, and the information contained in them can be used to steer the derivation towards success.

3. **Indexes.** As synchronous formulas are moved to the storage zone Γ, they are annotated with data contained in an index term, supplied in the *store* rule. Like certificate terms, they may contain arbitrary information, and can later be called upon to selectively retrieve a subset of candidate formulas for selection by the *decide* rule. Lookup can be as loose or as tight as the designer chooses, offering great control over backtracking.

4. **Clerks.** These predicates, denoted by the $_c$ subscript in Figure 1, define the control semantics of asynchronous rules. While not explicitly responsible for the decisions that are characteristic of synchronous rules, they can nonetheless perform bookkeeping that will eventually enable their counterparts to make their decisions when their time comes. These can be thought of as ordinary program clauses, making full use of the power of logic programming.

5. **Experts.** The synchronous counterparts of clerks, signified by the $_e$ subscript, are like their asynchronous duals arbitrary program clauses, but in addition to possibly recording information while processing certificate terms they will be required to make the decisions demanded by their phase. How much or how little they commit to one or several possible courses of action is not dictated by the framework and remains purely a design issue.

Identity rules take place during the synchronous phase and are therefore supervised by experts. Structural rules are more nuanced: *store* occurs during the asynchronous phase and is assigned a clerk, although it is charged with the critical operation of filing formulas into storage, in the process assigning them indexes. In turn, *decide* arbitrates the transition from asynchronous to synchronous phase by using those very indexes, and its complement *release* mediates the other phase transition. Both these transitions can be seen as operating "between two worlds," and are here both declared as experts.

λProlog implementations of both the *LKF* system augmented with the FPC framework, named LKF^a, are available as kernels. These are direct translations of the proof system and, if one is willing to trust an implementation of a logic programming language, deeming such a kernel trustworthy as well should be fairly simple.

3 Programming FPCs in the small

3.1 Development in practice

Even as we guide users away from the cryptic assembly of the sequent calculus, letting them instead write abstract certificate terms based on a PRICE, the establishment of a mapping between these abstract terms and their correspondence with the assembly level is an inevitable step of significant complexity that needs to be repeated, with variations, for each FPC definition. Despite this, the applicability of the framework to a number of representative and highly varied settings has been studied with satisfactory results. Among these, proof formats for classical and intuitionistic first-order logic, are studied at length in [9], and the examples in this section are derived from here. A proof theoretic vision of model checking is possible if a richer logic is selected; $\mu MALL$, i.e., multiplicative-additive linear logic with greatest and least fixed points as logical connectives instead of exponentials, is suitable for this purpose [10]. In a similar vein, the addition of fixed points to intuitionistic logic, μLJ, serves to reason about constructive proofs and their expression as outlines, in a note closer to the connection between theorem provers and certification [5]. It is possible to treat modal logics like K as well, and a focused version of the labeled sequent system based on it, $G3K$; this system can then be utilized to certify modal tableaux [12]. Logic programming is the natural environment for these developments. Kernels and proof checkers are built in λProlog [14], or Bedwyr [3] when fixed points are needed.

Commonly, an approximate idea of the end result exists in the mind of the designer from the outset, subject to refinements and generalizations. In practice, testing plays an essential role to confirm, at least empirically, that a PRICE satisfies its intended semantics. In spite of its practical importance, the support offered by the usual programming environments is rather basic, as are debugging facilities. Even working around this, it must be remembered that to date few formal guarantees of a semantic correspondence are routinely obtained. Anecdotal evidence suggests that is is possible for seemingly small changes to break examples that previously worked, and better support to avoid these regressions would be desirable: the augmented proof systems ensure soundness, but reaching and maintaining completeness (relative to the proof strategy of interest) is the sole domain of the designer, and currently evaluated mainly by way of test cases.

The rest of the section is dedicated to two elementary examples of uses of the framework. It will become clear that this methodology of programming is still very close to the logic, and expert domain knowledge is required to harness its expressive power; even then, judging by our experience, the process is far from straightforward. We present these as motivation to develop past these early stages and into a truly formal methodology. The interactive assistant will take some first steps in that direction.

3.2 Example: a decision procedure

An extreme minimalist use of the FPC framework is studied in this subsection. Consider the propositional fragment of the logic where everything is negatively polarized. In this situation, there is no opportunity to offer guidance during the synchronous phase, and our only recourse is to exhaustive search. While completely blind and impossibly inefficient, it is clearly a sound, if empty, strategy, a fact that can be clearly represented with a PRICE, describing its parts as follows.

1. **Polarization** is "purely negative." All logical connectives are polarized negatively. By convention, formulas are in conjunction normal form, and negations translate into atoms

```
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.
```

Figure 2: A conjunctive normal form PRICE: full decision procedure

of complementary polarities, e.g., negated atoms are given negative polarity; non-negated atoms are given positive polarity. This is the only allowed use of positive polarity.

2. **Certificate terms** carry no information. A singleton, nullary constructor (`cnf`) leaves no option but to propagate it from conclusion to premises.
3. **Indexes** carry no information, either. A singleton, nullary constructor (`idx`) is assigned to all formulas in storage and decided on. The storage zone acts as a bucket devoid of any clues to help us decide on the appropriate formula: each decision will be attempted on each and every formula in Γ.
4. **Clerks** are declared for the negative connectives to enable proof search to proceed through them and store positive formulas and negative literals, i.e., atoms, without distinction.
5. **Experts** are declared only for phase transitions, vacuously, and for the *init* rule, to allow an atom to be matched with its stored complement and close the branch. It can be seen that, by design, the other inference rules cannot occur, and therefore their definition would have no effect in proof search.

Adopting a simple naming standard and encoding of arguments from conclusion to premises and left to right, we obtain the PRICE in Figure 2.

With patience, the theoremhood of small propositional formulas can now be checked by using `cnf` with the entry point sequent.

3.3 Example: binary resolution

It is instructive to see how the core of a standard proof technique can be encoded in the framework. To this end we turn to resolution refutations. Here we want to attempt to prove a formula of the form $\neg C_1 \vee \ldots \vee \neg C_n$, with each clause C_i a disjunction of literals closed by universal quantifiers. The key idea of the technique lies in the binary resolution rule (here, Ls represent literals of either polarity).

$$\frac{L_{1,1} \vee \cdots \vee L_{1,i-1} \vee P_a \vee L_{1,i+1} \vee \cdots \vee L_{1,m} \quad L_{2,1} \vee \cdots \vee L_{2,j-1} \vee \neg P_a \vee L_{2,j+1} \vee \cdots \vee L_{2,n}}{L_{1,1} \vee \cdots \vee L_{1,i-1} \vee L_{1,i+1} \vee \cdots \vee L_{1,m} \vee L_{2,1} \vee \cdots \vee L_{2,j-1} \vee L_{2,j+1} \vee \cdots \vee L_{2,n}}$$

A proof operates by attempting to apply the rule to a pair of clauses to generate a new clause, until the empty clause f is reached, at which point the refutation is concluded with success. By assigning names to the clauses that compose the formula to be proved, as well

```
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.
```

Figure 3: A binary resolution PRICE (1): store the negated clauses of the alleged theorem

as the new clauses that result from applications of the rule, it becomes possible to represent compactly a proof by resolution by a list of triples denoting the two premises and the conclusion of each application of the rule, respectively.

The initial certificate of a proof attempt by resolution must encode this information. Assuming clause names are natural numbers assigned incrementally and starting from one, an obvious possibility is this triple.

1. A list of clauses corresponding to the $C_1, \ldots, C_n$ of the formula whose proof will be attempted. These will receive identifiers 1 through n.
2. A list of clauses used in the proof but not included in the input clauses, i.e., derived by applications of the resolution rule on previous clauses. These will receive identifiers starting from $n + 1$.
3. A list of triples of numeric indexes i, j, k, where i and j are the indexes of the premises and k the index of the conclusion of the rule.

In order to design the PRICE in detail, we determine one possible shape of a general proof by resolution in LKF^a, and work around to sculpt it during proof search. A detailed description can be found, e.g., in [9]. Conceptually, a proof is divided in three main region types:

1. The first phase operates asynchronously storing the clauses that configure the potential theorem under their respective indexes. The encoding is shown in Figure 3.
2. The second phase connects each use of the resolution rule with a use of cut. One of the branches of this cut will continue on to the next instance of the resolution rule, forming a backbone of cuts that culminates in the empty clause. The encoding is shown in Figure 4.
3. The third phase (type) constitutes the proof of the other branch of each cut of the second phase, and is responsible for checking that the resolution rule on the triple of indexes specified by the certificate is, indeed, correct. The encoding is shown in Figure 5.

It is clear from even a cursory analysis that an interesting certificate such as this, compact and relatively simple, depends deeply on the proof system and the conventions it establishes during the design of a general "proof scaffold." Preconditions must be carefully documented, but irrespective of this, and while the framework provides facilities for abstraction, the process remains delicate and difficult to scale. Let this illustration motivate the sections that follow.

4 Towards programming in the large

4.1 Patterns of abstraction

As exemplified in Section 3, PRICE definitions have been the result of careful analysis of the logical structure of proofs in relation to a suitable, focused proof system expressed in terms

```
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.
```

Figure 4: A binary resolution PRICE (2): a backbone of cuts that lead to the selection of the empty clause

```
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).
```

Figure 5: A binary resolution PRICE (and 3): check an individual resolution step by a shallow proof

of the sequent calculus. They range in complexity from the simplest decision procedures — exponential on the size of the input — to sophisticated methods of proof reconstruction that encode precise instructions to guide proof search; the size of certificates, too, varies widely. The programming environment of choice is generally that of higher-order logic, e.g., λProlog or Bedwyr.

A common observation that arises from most of the more sophisticated certificate definitions, and many moderately complex ones, is that they can be fairly brittle in their behavior: they will certainly work under the exact conditions for which they were designed, but will fail unexpectedly if presented with seemingly minor variations on the basic problem structure considered during programming; of the examples considered, resolution is a particularly apt example. Even simple definitions, such as imposing bounds to the complexity of proofs based on depth can present limited, if unexpected, complexity in managing the regions of the proof in an exact manner. These issues originate in the conceptual distance between the high-level states and translations determined by the algorithms that define decision procedures, and the

low-level sequents and inference rules of the sequent calculus. These latter must be orchestrated by a PRICE to produce an equivalent behavior. This, embedded in a powerful logic programming environment, results in systems whose behavior can be laborious to diagnose and debug.

The keys to reducing the complexity involved in the handcrafting of a certificate definition are *abstraction* and *automation*. There are two fundamental, compatible strategies to hiding the complexity that underlies the framework: one can attempt to expose the logic minimally, hiding aspects of the inner workings of the proof system that are inessential to the end user; on the other hand, there has to be a core of the programming interface, describing the instructions that guide proof search, that can never be neglected: simplifying its use is the second alternative. A minimal proficiency with logic must be assumed, and in the early stages of the development of a system like we propose this level must be, as a minimum, close to the logic of choice, while the minutiae of its underlying proof system become less critical with successive refinements. Two measures to guide access to the logic itself, not the framework, are apparent.

1. **Choice of logic.** If the user has selected a logic to target their problems, they will be prompted to state so. Otherwise, an assistant will ask a few questions about their formulas to determine if the system supports a suitable logic, either directly or by means of a mapping. Other options, such as inferred or explicit polarities, below, will have a bearing on the dialog with the user.
2. **Choice of polarity.** Among the five components of a PRICE, only polarity is part of the proof system proper; the other four parameters instantiate the FPC framework and are considered separately. The choice of polarity depends on various considerations. Especially common among these are the following:
   (a) All connectives involved in a clause representing a deterministic computation must be negative.
   (b) Formulas to which a contraction rule may be applied, i.e., through the *decide* rule, must be therefore positively polarized.
   (c) Polarization criteria for atoms (and, conversely, for their negations) are not strictly necessary in classical logic, but have a profound influence on the shape of the proofs whose derivation they may allow.

   As in the choice of logic, a user who has knowledge of polarities and their behavior can choose to select the allowed subset of polarized connectives. Otherwise, an attempt at inferring the adequate polarities can be made from such questions about the nature of the envisioned proofs as have are being considered here. Where there are degrees of freedom left, the various possibilities could be explored through interactive examples.

Once these parameters are fixed, the user needs to define indexes to store and decide on formulas, certificate constructors to represent the state of proof reconstruction, and the clauses of clerks and experts that encode the transitions between states for each inference rule allowed under the constraints imposed by the above steps. The facilitation of this labor is principally technical in nature. Constructor parameters can be declared and supplied to the clauses, choices restricted to syntactically correct values, and some decisions made automatically based on the current signature. On top of all this syntactic assistance, the definition of semantics falls ultimately on the user. Therefore, whereas the benefits of abstraction are limited to a simplified interface to the FPC API, those of automation can significantly simplify programming, especially for non-specialists.

4.2 Workflow overview

`sunshine` is an interactive PRICE definition assistant. Its purpose is to guide and supervise the design of a certificate definition, from its contact surface with logic to the domain-specific language in which certificates will be written. It does not assume or require familiarity with the FPC framework, although acquiring this will be a natural consequence of controlled exposure to the sequent calculus.

The model of computation for the assistant is an adaptation of the REPL pattern that incrementally builds a PRICE, modified in part by some static restrictions of the logic, and by those programmed dynamically by the user. The system starts with a blank slate and gradually builds a functional FPC definition, while never allowing false steps to be taken. It is an assistant in that it maintains and makes explicit all possible decisions that can be taken at any one time in the context of the logical framework, helps the user to state their intent and requests the necessary information while shielding them as much as possible — or desired — from implementation details. To do this, it is structured along the following components integrated in an activity loop.

1. **Logic selection.** Subject to an optional guidance procedure, selects the corresponding kernel implementing the proof system for the logic of choice, along with the logic programming environment that forms the basis for the runtime. In addition, an empty signature for the proof system is loaded and made available to the refinement loop.
2. **Restrictions on focusing.** Any restrictions imposed either manually or through an inference procedure limit some of the options that are offered through the refinement loop, namely those regarding the available inference rules and, with them, the types of clerk and expert clauses that may be defined. (In particular if there are unresolved ambiguities, a related procedure could be integrated in the miscellaneous processing of the loop, although we will not pursue this possibility in the present treatment.)
3. **Refinement loop.** Once a logical frame of reference has been established, and the logical component of the PRICE determined, the user enters a REPL loop within which they can proceed to iteratively define the components of an FPC and experiment with them interactively. Each step offers several possibilities to extend the definition, while the system ensures that the PRICE remains in a consistent state, if possibly incomplete, thus minimizing the inadvertent accumulation of errors. A number of basic steps can be chosen from.
   (a) **Definition of certificate or index terms.** Before any clerk or expert clauses can be defined, the signature must contain some constructors for certificates and, in all likelihood, indexes. Each of these will have an arity, and each argument a type from the signature.
   (b) **Ancillary signature declarations.** In order to supply information to both certificate and index constructors, and manipulate this information in the clerk and expert clauses, it will be necessary in all but the most trivial certificates (cf., e.g., Figure 2) to extend the signature with new kinds and type constructors, in much the same way as certificate and index terms are defined.
   (c) **Definition of clerk or expert clauses.** Based on the contents of the signature (types, constructors) and the restrictions placed by focusing on the sequent calculus, the user can select from a list of possible end-sequents corresponding to each possible inference rule, and edit (create, update, delete) the clauses assigned to the clerk or expert for the inference rule. Pure transformations based on pattern matching can be

fully guided with ease, and the user need not be exposed to the programming language and kernel under the hood. However, clerks and experts in complex PRICEs may carry out computation, and for this "dropping down" to the programming language level is the only option that offers sufficient generality to be currently considered.

(d) **Ancillary clauses.** Complex FPC definitions of the kind described in the last category may require the declaration of arbitrary propositions and clauses outside the interface of the framework. For this, as above, the only possibility that is currently envisioned is through direct programming in the core language.

(e) **Validation and miscellaneous subprocesses.** After each step, the evaluation must include various checks to ensure that the revised PRICE remains in a consistent state, alerting the user of any problematic attempts, as well as serving as a watchdog against bugs in the assistant itself. We consider these facilities in following subsections. (Additionally, the tool could evolve to offer an extension interface to allow the writing of customized modules to enrich the functionality of the software.)

(f) **Import/export.** At any moment, the user must be capable of exporting the current state of the assistant into a full-fledged, executable PRICE, and restore the state from a previous session, or from any other means through which an FPC definition can be created, including programming directly against a kernel without an assistant. Specially formatted annotations are used to communicate information specific to the assistant, such as any desired restrictions on focusing, as already discussed. These must include any extrinsic source of information used by the validation subprocesses that does not directly depend on the FPC framework itself.

The process continues for as long as the user desires, and can be interrupted or restored at any time. There are some dependencies on runtime systems that must be maintained, with relative ease. More interesting are the internal dependencies of the growing PRICE. The interactive loop encourages a certain incremental approach that maintains consistency at all times. However, changes in definitions, like the parameters of a certificate constructor which has been used in the definition of clerks and experts, cannot be performed in one basic step of the loop. As in the general practice of programming, there are no trivial solutions for cascading errors, and if substantial alterations are required it is always possible to export the PRICE, perform the modifications and re-import and validate the modified version. Other alternatives will be discussed shortly.

4.3 Static analysis

The fundamental component of the validation sub-step of the main activity loop is charged with ensuring that user changes introduce no inconsistent states. In first instance, this requires validating the signature and typechecking the "program," i.e., the instantiation of the kernel where calls to clerks and experts are filled in with the definitions given by the PRICE. It is possible, although inefficient, to "export" the FPC definition on the background, load it in the kernel, compile it against the core programming environment, and report any resulting errors. Moreover, error messages would need to be deciphered and references to the positions of errors back-translated and correlated with the user's input.

This general-purpose treatment covers those checks that are shared with the logic programming environment, but the assistant is not limited to these simple properties. An internal model of the signature and the general structure of the kernel enables more sophisticated processing that goes beyond the validation phase.

1. **Recommendations and auto-fill.** When writing a clause for a clerk or expert, the checker can explore the signature for compatible type constructors, and offer or default to filling out arguments automatically if a single possibility is found, otherwise it can offer a list of compatible constructors, and recurse into any arguments of these.
2. **Flow analysis.** The clerk and expert clauses correlate the transformations between certificate constructors, as well as the storage and retrieval of formulas through index constructors. This program does not exist in a vacuum: each clerk and expert is embedded in a specific inference rule of the kernel implementing the focused sequent calculus for the logic. This, coupled with the restrictions on focusing, which are, in fact, restrictions on the logic itself and its formulas, gives way to a rich category of analyses, of which the following general representative has been studied.
   (a) **Dependency graph.** From the end-sequents and the possible transitions between them and to the *init*-like rules that end a branch of the derivation — noting that some of these rules inspect the certificate, while others discard it —, a dependency graph can conservatively determine whether certain certificate constructors constitute dead ends that cannot possibly lead to a successful proof. Parameters that are unused, or are passed without transformations or effects can be identified, as can index families that will become stale and not contribute to the process, based on their use by the *store* and *decide* rules.
3. **Refactoring assistance.** The static analyzer can present the user with additional operations, specifically consistent transformations on the current state of the PRICE, such as renaming, and introduction or removal of constructor parameters, with the possibility to supply or recommend default values for all affected instances, insert dummy placeholders, globally or locally, or traverse the program in an input sub-loop.

The `lint`-like tool that results can find useful applications not only embedded in the assistant, but as part of a reactive development environment that is decoupled from the framework, once the scaffolding of the FPC framework is factored in.

In addition to exploiting the restrictions on the program imposed by the FPC framework and the evolving FPC definition, further constraints can be coordinated with user-defined tests, which are the topic of the next subsection.

4.4 Testing

A final and no less critical component of the assistant facilitates the crucial requirement that the user be capable of writing certificates in the defined language that enable them to prove properties of interest. As observed in Section 3, tests give confidence that the defined certificates guide proof search along the desire paths. The FPC framework ensures the soundness of each and every PRICE, and the main task of the designer is to restrict the indiscriminate completeness of blind proof search to reflect the semantics of the proof format under consideration.

This feature reflects the mental process, akin to test-driven development, that guides the design of the PRICE and offers additional information to the static analyzer, pinpointing the sources of shortcomings in the program and even suggesting when it may be more lenient than necessary. We cannot assume tests to be satisfiable by simple changes: this is to say that adding a new test that covers unimplemented functionality is not considered an inconsistent step. Failing tests in the suite do not cause a failure in validation, but must nevertheless report that the current scope of a certificate definition is not yet complete.

Runtimes and programming languages support tests in varying degrees, and the assistant must therefore provide a unified interface to each targetable system. As with the simpler flavor of static analysis, it becomes necessary to instantiate the runtime system with the current state of the PRICE, execute it under controlled circumstances, and parse the output to report back on test results through the interface. Tests can be of two types.

1. **Positive tests.** Polarized formulas and certificates that must lead to a successful proof of the formula. They serve to ensure that the kernel does enough to find a proof.
2. **Negative tests.** Polarized formulas and certificates that must exhaust the search for a proof, given the constraints of the definition, without success. These are more natural to a model checking system like Bedwyr, but can be built with some extra work into the other environments (esp., λProlog), and are concerned with preventing the kernel from doing too much by limiting its expressive power.

With these facilities in place, the assistant has all the basic elements it needs to guide the user through the process of defining a PRICE from start to finish.

5 Architecture and design of the assistant

5.1 Backend

This section gives a brief overview of a prototype implementation of `sunshine`. The system can be neatly decomposed in a frontend, in charge of interactions with the user, and a backend that implements the core of the system and interfaces it with the FPC framework. This last piece of software is considered separate, trustable, and endowed with a standard API (exposed through a logic programming language) against which to program: this is prepared to encompass the set of supported runtime systems.

For the purposes of the present treatment, we make a number of simplifying assumptions that allow us to focus on the key aspects of the system.

1. **Logic selection.** A single proof system, LKF^a, described in Section 2, is supported in its λProlog implementation. This determines the programming environment, so that while a layer of abstraction can be foreseen, efforts in modularity can be directed to other, more interesting areas. For this reason, it is possible to lift the restrictions of the "user language" (i.e., keywords, variable naming conventions, typing, etc.) directly from the "object language," in this case λProlog.
2. **Restrictions on focusing.** It will be assumed that the user has a basic knowledge of focusing and the focused proof system, so that inference of polarity becomes unnecessary. This will simplify somehow the static analysis, given that no external restrictions must be accounted for, and the full system is studied by default. This is an interesting feature to have, but its complexities are beyond the scope of a first incarnation of the software.
3. **Refinement loop.**
   (a) **Ancillary clauses.** The full programming environment will not be exposed at this point, except through the importing of PRICEs processed by an external editor. External definitions will not be given through the assistant, though they may be imported, and clauses will be centered around pure, if possibly redundant, pattern matching.
   (b) **Validation and miscellaneous subprocesses.** Only static analysis and testing will be considered, with no support for external modules at this point.

Given these fortified requirements, the assistant jumps directly into the refinement loop, with all the necessary data structures to maintain the signature and state of the program, and the classes implementing the functionality of the various possible steps, fully instantiated with the logic, the kernel and the runtime of its implementation. Within the application, a main controller accepts commands from the user and dispatches them to the appropriate handlers, thus updating the state of the PRICE and performing all necessary checks. The user interacts with the controller through an interface. It can be observed that this application model fits well in the common model-view-controller (MVC) architectural design pattern, with only small adjustments.

As noted above, the language in which identifiers, program clauses, etc., will be written is standard λProlog, as implemented in Teyjus [15]. Use of predefined types, predicates, and operators from the standard library is not allowed in the current, purely declarative setting based on inductive types and pattern matching, although a prefixed subset of identifiers is reserved for internal use by the assistant due to the rudimentary support for namespaces in the language. Translation from the data structures of the assistant to the target language, modular composition (i.e., instantiation of the kernel, execution of test cases) and management of batch jobs is handled through a separate utility class that is accessed by command modules as needed.

The implementation of such a proof of concept is not particularly sensitive to technological choices, and the basic algorithms it requires do not hide any extraneous sources of complexity. We think the merit of this design relies on its elegant integration in a conceptually simple but powerful framework.

5.2 Frontend and user interface

The frontend performs the function of views in the MVC model, and connect the actions of the user with the commands defined by the various controller classes. A local installation of the assistant and its dependencies can run in any Unix-like machine through a simple command-line interface. This CLI will present an overview of the current PRICE definition and allow display of all further details; it will display available options and query for choices; and it will compose the calls to the controllers, and display errors and return values from the invoked methods. While a simple linear dialog is possible, the task of displaying complex information is facilitated by the use of a terminal graphics library in the tradition of the `curses` family.

An appealing alternative that remains to be explored exploits the wide availability of modern MVC frameworks such as Django (for Python) or Ruby on Rails (for Ruby), which would replace the terminal with a web application, which in turn could be hosted by an external server, made available to researchers for experimentation without installation, and would be capable of producing more responsive, powerful interfaces. The logic programming environments exist outside these frameworks and present prospective servers with some additional, less mainstream requirements. However, recent experiments cross-compiling the proof assistant Abella — closely related to Bedwyr — to JavaScript and running it on the browser [7] show promise in relaxing these demands.

With this note, we conclude the description of the proof of concept. Assistant implementations will be made available from the ProofCert repositories.[1]

[1] Code repositories will be made publicly available after the celebration of DaleFest on 16 December 2016.

6 Discussion

In foregoing pages we have proposed a definition assistant that streamlines the creation and maintenance of FPC definitions and improves existing analysis and testing techniques for the typical programming environments in which this activity takes place. It can serve as a learning tool not only applied to the FPC framework, but for the sequent calculus and logic programming more generally. By way of exercise, we may recommend to understand what it would take to recreate (redesign?) the example PRICEs in Section 3, and reflect on what this teaches us about the problems and the challenges presented to make this technology more usable and widely adopted.

Some obvious candidates for improvements come in the form of extensions to the assistant, i.e., both removing the simplifications and adding the possibilities for refinement that have been discussed in Sections 4 and 5. Some, like polarity inference, are of some theoretical interest, while many others are principally needed to evolve a mature tool that more users may be interested in using. These incremental advances push away from the details of the logic, but the disentanglement is never complete, and we do not think we should be satisfied with this state of affairs. One issue in particular remains unanswered: how to guarantee that a PRICE implements a semantics defined in terms other than those of the proof system? The semantics we refer to is known beforehand and corresponds to the proof method we wish to certify; it is not an artifact that may be discarded lightly. A further qualitative step to greater abstraction would pursue to hide the clockwork of sequent calculus as much as possible, and from the point of view of the user this would be ideal, but is dependent on establishing a more indirect link between semantics. This is an avenue we would like to explore in the future.

Also related to trust and correctness, scrupulous developers may want to consider whether there may be bugs lurking in the kernels that constitute the cornerstone of the system, and whose correctness has been taken as a matter of fact throughout the paper. Certified implementations of the various kernels, and even a more general framework for certification of these critically important pieces of code, would be steps in a right direction orthogonal to that of the correctness of semantics considered before.

Less lofty but useful goals also avail. Given that we would like to see greater support for the FPC framework in the corpus of verification software, it would be beneficial to explore adding total or partial support to representatives of this category, such as theorem provers. Abella [2], a proof assistant based on logic that is closely related to Bedwyr, is a promising candidate that could pave the way for new developments in the interaction between standard proving tools and certification, including but not limited to adopting certificates as tactics, exporting certificates from native proof scripts.

To conclude, we reiterate our conviction that the theorem proving community would benefit greatly from the approach to certification advocated through the FPC framework. Protected by firm proof theoretical foundations, it remains to make these available to the larger scientific community, not only through advocacy, but by example as well. Making the framework attractive and easy to use is key to acquiring a critical mass of users that could bring it, and its benefits, closer to widespread use.

Acknowledgments. The authors cannot fail to express their gratitude to Dale Miller as the spring and inspiration for this project. They furthermore wish to thank Tomer Libal in special for helpful discussions and comments. This work has been funded by the ERC Advanced Grant ProofCert.

References

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[2] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: a system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014.

[3] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The Bedwyr system for model checking over syntactic expressions. In Frank Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction*, pages 391–397. Springer, 2007.

[4] Roberto Blanco, Tomer Libal, and Dale Miller. Defining the meaning of TPTP formatted proofs. In Boris Konev, Stephan Schulz, and Laurent Simon, editors, *Proceedings of the 11th International Workshop on the Implementation of Logics*, pages 78–90. EPiC Computing, 2015.

[5] Roberto Blanco and Dale Miller. Proof outlines as proof certificates: a system description. In Iliano Cervesato and Carsten Schürmann, editors, *Proceedings of the 1st International Workshop on Focusing*, pages 7–14. EPTCS, 2015.

[6] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, *Proceedings of the 2nd Workshop on Proof Exchange for Theorem Proving*, pages 28–43. CEUR Workshop Proceedings, 2012.

[7] Kaustuv Chaudhuri. Abella in your browser, 2016. http://abella-prover.org/tutorial/try/.

[8] Zakaria Chihani. *Certification of first-order proofs in classical and intuitionistic logics*. PhD thesis, École polytechnique, 2015.

[9] Zakaria Chihani, Dale Miller, and Fabien Renaud. A semantic framework for proof evidence. To appear in *Journal of Automated Reasoning*, 2017.

[10] Quentin Heath and Dale Miller. A framework for proof certificates in finite state exploration. In Cezary Kaliszyk and Andrei Paskevich, editors, *Proceedings of the 4th Workshop on Proof Exchange for Theorem Proving*, pages 11–26. EPTCS, 2015.

[11] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

[12] Tomer Libal and Marco Volpe. Certification of prefixed tableau proofs for modal logic. In Domenico Cantone and Giorgio Delzanno, editors, *Proceedings of the 7th International Symposium on Games, Automata, Logics and Formal Verification*, pages 257–271. EPTCS, 2015.

[13] Dale Miller. Communicating and trusting proofs: the case for foundational proof certificates. In Peter Schroeder-Heister, Gerhard Heinzmann, Wilfrid Hodges, and Pierre Edouard Bour, editors, *Proceedings of the 14th Congress of Logic, Methodology and Philosophy of Science*, pages 323–342. College Publications, 2015.

[14] Dale Miller and Gopalan Nadathur. *Programming with higher-order logic*. Cambridge University Press, 2012.

[15] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In Harald Ganziger, editor, *Proceedings of the 16st International Conference on Automated Deduction*, pages 287–291. Springer, 1999.

[16] George Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43:337, 2009.