Defining inference rules by certificate checking

Roberto Blanco and Dale Miller

Inria & LIX, École Polytechnique

Abstract. We apply the foundational proof certificate (FPC) framework to the problem of designing high-level outlines of proofs. The original idea of the FPC framework is that proof evidence from a wide range of theorem provers can be formally defined in such a way that a proof checker could check the proof evidence emitted from those various theorem provers. In this paper, we illustrate how the FPC framework can be used to design high-level inference rules or, essentially, small proof outlines. The proof checker could then be given the job of seeing if it is possible to expand such outlines into the performance of a fully detailed proof. If such a performance is successful, the theorem is proved and the certificate can become the proof evidence. In order to validate this approach to designing large scale inference rules, we have built the ACheck system that allows us to take a sequence of theorems from the Abella theorem prover and apply the proof outline “do the obvious induction and close the proof using previously proved lemmas”. We also illustrate how the logic programming engine that underlies our proof checker provides our scheme with a great deal of flexibility, including the reconstruction and elaboration of proof certificates.

1 Introduction

Inference rules, as used in, say, Frege proofs (a.k.a. Hilbert proofs) are usually greatly restricted by limitations of human psychology and by what skeptical people are willing to trust. Typically, checking the application of such inference rules involves simple syntactic checks, such as deciding on whether or not two premises have the structure $A$ and $A \supset B$ and the conclusion has the structure $B$. The introduction of automation into theorem proving has allowed us to engineer inference steps that are more substantial and include both computation and search. In recent years, a number of proof theoretic results allow us to extend that literature from being a study of minuscule inference rules (such as modus ponens or Gentzen’s introduction rules) to a study of large scale and formally defined “synthetic” inference rules. In this paper, we describe a particular way to specify and check such synthetic inference rules as a way to inductively prove lemmas from previous lemmas.

Frege proofs are lists of formulas, each one of which is either an axiom or the conclusion of an inference rule whose premises come earlier in the list. We shall not speak of axioms since these can be modeled as inference rules that depend on zero previous lemmas. Also, when we speak of a formula that is a member of such a list of formulas, we shall usually refer to it as a lemma.
Consider defining the addition of natural numbers using the following inductive relational specification

Define plus : nat -> nat -> nat -> prop by
  plus z N N ;
  plus (s N) M (s P) := plus N M P.

where \( z \) and \( s \) denote zero and successor, resp. (Examples will be displayed using the syntax of the Abella theorem prover [2]: this syntax should be familiar to users of other systems, such as Coq.) When this definition is introduced, we should establish several properties immediately, e.g., that the addition relation is determinate and total.

Theorem plustotal :
  forall N, nat N -> forall M, nat M -> exists S, plus N M S.

Theorem plusdeterm : forall N, nat N -> forall M, nat M ->
  forall S, plus N M S -> forall T, plus N M T -> S = T.

Anyone familiar with proving such lemmas knows that their proofs are simple: basically, the obvious induction leads quickly to a final proof. Of course, if we wish to prove more results about addition, one may need to invent and prove some lemma before simple inductions will work. For example, proving the commutativity of addition makes use of two additional lemmas.

Theorem plus0com : forall N, nat N -> plus N z N.

Theorem plusscom : forall M, nat M -> forall N, nat N ->
  forall P, plus M N P -> plus M (s N) (s P).

Theorem pluscom : forall N, nat N -> forall M, nat M ->
  forall S, plus N M S -> plus M N S.

Each of these three lemmas have essentially the same kind of proof: use the obvious induction statement, apply some previously proved lemmas and the inductive hypothesis, and deal with any remaining case analysis.

The fact that many theorems can be proved by using induction-lemmas-cases is well-known and built into existing theorem provers. For example, the waterfall model of the Boyer-Moore theorem prover [7] will prove such theorems in a similar fashion (but for inductive definitions of functions). Twelf [18] can often prove automatically that some relations are total and functional using a series of similar steps [19]. Also, LCF tactics and tacticals have been used to implement procedures that attempt to find proofs using these steps [20]. Finally, the TAC procedure in [5] attempts to follow such a procedure as well but in a rather fixed and inflexible fashion.

In this paper, we present an approach to describing the simple rules that can prove a given lemma based on previously proved lemmas. Specifically, we define simple proof certificates that describe the structure of the intended proof outlines that we expect and then we run a proof checker on that certificate to see if the certificate can be elaborated into a full proof or not of the lemma.
Since the design of the certificate language is based on the proof theory of synthetic connectives and since the proof checker we use employs both unification and backtracking search, this approach to describing high-level inference rules is flexible and natural.

2 A focused proof system

Consider the following two, familiar introduction rules in Gentzen’s LJ sequent calculus [14] for first-order intuitionistic logic.

\[
\frac{\Gamma \vdash B}{\Gamma \vdash B_1} \quad \frac{\Gamma \vdash B[t/x]}{\Gamma \vdash \exists x.B}
\]

If one attempts to prove sequents by reading these rules from conclusion to premises, then these rules need either information from some external source (e.g., an oracle providing the \( i \in \{1, 2\} \) or the term \( t \)) or some implementation support for non-determinism (e.g., unification and backtracking search).

It is difficult to meaningfully use Gentzen’s sequent calculus to directly support proof automation. Consider, for example, attempting to prove the sequent

\[
\Gamma \vdash \exists x \exists y[(p x y) \lor ((q x y) \lor (r x y))]
\]

where \( \Gamma \) contains, say, a hundred formulas. The search for a (cut-free) proof of this sequent can confront the need to choose from among a hundred-and-one introduction rules. If we choose the right-side introduction rule, we will then be left with, again, a hundred-and-one introduction rules to apply to the premise. Thus, reducing this sequent to, say, \( \Gamma \vdash (q t s) \) requires picking one path of choices in a space of \( 10^{14} \) choices.

Focused proof systems address this explosion by organizing introduction rules into two distinct phases. In particular, the introduction rules for \( \lor \) and \( \exists \) are written instead as

\[
\frac{\Gamma \vdash B \downarrow \Uparrow \Theta \vdash R}{\Gamma \vdash B_1 \lor B_2 \downarrow \Uparrow \Theta \vdash R} \quad \frac{\Gamma \vdash B[t/x] \downarrow \Uparrow \Theta \vdash R}{\Gamma \vdash \exists x.B \downarrow \Uparrow \Theta \vdash R}
\]

Here, the formula on which one is doing introduction rules is marked with the \( \downarrow \Uparrow \) (the formula under focus): as a result, it is easy to see that reducing proving the sequent

\[
\Gamma \vdash \exists x \exists y[(p x y) \lor ((q x y) \lor (r x y))] \downarrow \Uparrow
\]

to \( \Gamma \vdash (q t s) \downarrow \Uparrow \) involves only those choices related to the formula marked for focus: no interleaving of other choices needs to be considered.

While the \( \downarrow \Uparrow \) phase involves rules that may not be invertible, the \( \uparrow \) phase involves introduction rules that are only invertible. For example, the left-introduction rules for \( \lor \) and \( \exists \) are invertible and their introduction rule is listed as

\[
\frac{\Gamma \uparrow B_1, \Theta \vdash R \quad \Gamma \uparrow B_2, \Theta \vdash R \quad \Gamma \uparrow \{y/x\} B, \Theta \vdash R \quad \Gamma \uparrow \exists x. B, \Theta \vdash R}{\Gamma \uparrow B_1 \lor B_2, \Theta \vdash R}
\]
These rules need no external information (in particular, any new variable \( y \) will work in the \( \exists \) introduction rule). In these last two rules, the zone between \( \uparrow \) and \( \vdash \) contains a list of formulas. When there are no more invertible rules that can be applied to that first formula, that formula is moved to (stored in) the zone written as \( \Gamma \), using the following rule

\[
\frac{C, \Gamma \uparrow \Theta \vdash R}{\Gamma \vdash C, \Theta \vdash R} S_l
\]

Finally, when the zone between the \( \uparrow \) and the \( \vdash \) is empty (i.e., once all invertible inference rules have been completed), it is time to select a (possibly non-invertible) introduction rule to attempt. For that, we have the two decide rules:

\[
\frac{\Gamma, N \downarrow N \vdash E}{\Gamma, N \uparrow \vdash \uparrow E} D_I \quad \frac{\Gamma \vdash P \downarrow}{\Gamma \uparrow \vdash \uparrow P} D_r
\]

Although we cannot show all focused inference rules, we will present those that deal with the least fixed point operator. Formally speaking, when we define a predicate, such as \( \text{plus} \) in the previous section, we are actually naming a least fixed point expression. In the case of \( \text{plus} \), that expression is

\[
\mu \lambda P \lambda n \lambda m \lambda p. (n = z \land m = p) \lor \exists n' \exists p'. [n = (s \cdot n') \land p = (s \cdot p') \land (P \cdot n' \cdot m \cdot p')]
\]

For the treatment of least fixed points, we follow the \( \mu \text{LJ} \) proof system and its focused variant (see [1, 4]). The treatment of least fixed point expressions in the \( \uparrow \) phase and the \( \downarrow \) phases is given by the three rules

\[
\frac{\mu Bt, \Gamma \uparrow \Theta \vdash R}{\Gamma \uparrow \mu Bt, \Theta \vdash R} \quad \frac{\Gamma \uparrow S t, \Theta \vdash N \uparrow BS \bar{t} \vdash S \bar{t}}{\Gamma \uparrow S t, \Theta \vdash N} \quad \frac{\Gamma \vdash B(\mu Bt) \downarrow}{\Gamma \vdash \mu Bt \downarrow}
\]

Notice that the right introduction rule is just an unfolding of the fixed point. There are two ways to treat the least fixed point on the left: one can either perform a store operation or one can do an induction using, in this case, the invariant \( S \). The left premise of the induction rule shows that \( S \) is a prefixed point (i.e., \( BS \subseteq S \)). In general, always supplying an invariant can be tedious so we shall also identify two consequences of general induction, namely unfolding (also on the left) and obvious induction, meaning that the invariant to use is nothing more than the immediately surrounding sequent as the invariant \( S \). In the case of the obvious induction, the left premise sequent will be trivial.

### 3 Foundational proof certificates

We briefly describe the foundational proof certificate (FPC) [11, 16] approach to defining proof evidence. The main idea behind FPCs is that anyone who builds a proof in the process of executing a theorem prover should be able to output that proof in such a form that an independently written proof checker will be able to check that proof for validity. In order for such a scheme to work, the form
of such proof evidence must be formally defined: the foundational approach to proof certificates provides a flexible scheme for defining that proof semantics formally and in such a way that the specification can be interpreted on top of a suitable logic engine.

There are four key ingredients to providing an FPC and they are all described via their relationship to focused proof systems. In fact, consider the following augmented versions of inference rules we have seen in the previous section.

\[
\Xi_1 \vdash \Gamma \vdash B_i \Downarrow \\
\Xi_0 : \Gamma \vdash B_1 \lor B_2 \Downarrow
\]

\[
\Xi_1 : \Gamma \vdash B[t/x] \Downarrow \\
\Xi_0 : \Gamma \vdash \exists B.B \Downarrow
\]

These two augmented rules contain two of the four ingredients of an FPC: the schema variable \( \Xi \) ranges over terms that we shall think of as being the actual "certificate". The addition premises involve experts which are predicates that relate the concluding certificate \( \Xi_0 \) to a continuation certificate \( \Xi_1 \) and some additional information. The expert predicate for the disjunction can provide an indication of which disjunct to pick and the expert for the existential quantifier can provide an indication of which instance of the quantifier to use. Presumably, these expert predicates are capable of digging into a certificate and extracting such information. However, it is not required for an expert to be always expert in the sense that the disjunctive expert might chose to return both values of 1 and 2 for the value of \( i \): the proof checker will thus need to handle such non-determinism during the checking of certificates.

The two other key ingredients to an FPC are illustrated by examining the following augmented inference rules. The store-left (\( S_l \)) inference rule is augmented to be

\[
\Xi_1 : \langle l, C \rangle, \Gamma \vdash \Theta \vdash R \\
\Xi_0 : \Gamma \vdash \Theta \vdash R
\]

Here, the extra premise invokes a clerk predicate which is responsible for computing an index \( l \) that is associated to the stored formula (here, \( C \)). The decide-left (\( D_l \)) inference rule is augmented with an expert predicate that returns the index of the formula that is to be selected for focus.

\[
\Xi_1 : \Gamma, \langle l, N \rangle \Downarrow N \vdash E \\
\Xi_0 : \Gamma, \langle N, l \rangle \vdash \vdash \vdash E
\]

The indexing mechanism does not need to be functional (i.e., different formulas can have the same index) in which case the decide rule must also be interpreted as non-deterministic. In earlier work [10, 11], indexes have been identified with structures as diverse as formula occurrences and de Bruijn numerals. In this paper, indexes will be simply lemma names and a token marking a hypothesis (that is, a formula that is stored on the left using the \( S_l \) inference rule).

An FPC is then defined as follows: one provides the term constructors for certificates and for indexes and then defines clerk and expert predicates needed for all (augmented) focused sequents (clerk predicates are used within the \( \Uparrow \) phase and expert predicates are used within the \( \Downarrow \) phase). Given these definitions,
we then check whether or not a sequent of the form $\Xi : \Gamma \vdash B$ is provable. This latter check can be done using a logic programming engine since such an engine should support both unification and backtracking search (thereby allowing one to have a trade-off between large certificates with a great deal of explicit information versus small certificates where details are elided and reconstructed as needed.

4 Certificate design

Imagine telling a colleague “The proof of this theorem follows by a simple induction and the three lemmas we just proved.” You may or may not be correct in such an assertion since (a) the proposed theorem may not be provable and (b) the simple proof you describe may not exist. In any case, it is clear that there is a rather simple, high-level algorithm to follow that will search for such a proof. In this section, we show how the FPC framework can formally specify what that algorithm is and in the the next section, we describe a logic engine that can execute that algorithm.

Following the paradigm of focused proof systems for first-order logic, there is a clear, high-level outline to follow for doing proof search for cut-free proofs: first do all invertible inference rules and then, select a formula on which to do a series of non-invertible choices. This latter phase ends when one encounters invertible inference rules again or the proof ends. In the setting we describe here, there are two significant complicating features with which to be concerned.

4.1 Treating the induction rule

The invertible ($\triangleright$) phase is generally a place where no important choices in the search for a proof appears. When dealing with a formula that is a fixed point, however, this is no longer true. We can choose to “freeze” the fixed point (meaning that we choose not to induct on it) or we can set up an inductive step (these two choices correspond to the two $\triangleright$ phase inference rules at the end of Section 3). The proof theory of fixed points described in [1, 4] provides the justification of designing a focused proof system in this way. We divide this second choice into three such choices. First, we can choose to simply unfold a fixed point definition (this is a consequence of applying induction). Second, we can select the surrounding sequent context to be the actual inductive invariant. Third, we can take an explicit induction offered from the author of the certificate: in the context of this paper, if the author of that certificate is a human, we do not use this option. However, the author of a certificate can also be the proof checker itself (see Section 6) and in that case, an actual invariant may be inserted into a certificate.

4.2 Lemmas must be invoked

The application of lemmas into a proof outline is critical to the kind of linear proof development we have in mind. Although the focusing framework does not
restrict the shape of lemmas, we consider here the effect of focused proof construction with a lemma that is a Horn clause. For example, the three lemmas addressing the commutativity of addition at the end of Section 2 are Horn clauses. Consider, for example, applying a lemma of the form $\forall \bar{x} [A_1 \supset A_2 \supset A_3]$ in proving the sequent $\Gamma \vdash E$. Since the formulas $A_1$, $A_2$, and $A_3$ are polarized positively, we can design the proof outline (simply by only authorizing fixed points to be frozen during this part of the proof) so that $\Gamma \vdash \forall \bar{x} [A_1 \supset A_2 \supset A_3] \vdash E$ is provable if and only if there is a substitution $\theta$ for the variables in the list of variables $\bar{x}$ such that $\theta A_1$ and $\theta A_2$ are in $\Gamma$ and the sequent $\Gamma, \theta A_3 \vdash E$ is provable. The application of such a lemma is then easily seen as a kind of forward chaining: if the context $\Gamma$ contains two atoms (frozen fixed points) then add a third. Augmenting contexts in this way are critical for eventually enabling obvious inductions to succeed in completing a proof. In this way, the focused proof system can easily be used to apply lemmas.

The main issue that a certificate-as-proof-outline therefore needs to provide is some indication of what lemmas should be used during the construction of a proof. The following specifications of collections of supporting lemmas—starting from the least explicit to the most explicit—are easily written within our framework.

- a bound on the number of lemmas that can be used to finish the proof
- a list of possible lemmas to use in finishing the proof
- a tree of lemmas, indicating which lemmas are applied following the conjunctive structure of the remaining proof

5 The Proof Checker

The most direct and natural way to implement the FPC approach to proof checking is to use an appropriate logic programming language. When the logic is (either classical or intuitionistic) first-order logic without fixed points, then (as argued in [10, 11]) the $\lambda$Prolog programming language [17] is a good choice since its treatment of bindings allows for elegant and effective implementations of first-order quantification in formulas and of eigenvariables in proofs. When the logic itself contains fixed points, as is the case in this paper, $\lambda$Prolog is no longer a natural setting for such a checker: instead, a stronger logic that incorporates some aspects of the closed world assumption is needed. Our ACheck system employs the Bedwyr model checking system [3, 6]. In either the $\lambda$Prolog or Bedwyr implementations, one takes the augmented inference rules as displayed in Section 3 and essentially turns them sideways: the conclusion becomes the head of a logic programming clause and the premises become the body of the clause. In that way, the kernel of the checker—the interpreter for the augmented inference rules—is an immediate and natural encoding of inference rules. If one is willing to trust the Bedwyr logic engine, then it will be easy to trust the kernel encoded into it. (One exception to this design is the implementation of the rule that generates the obvious invariant: that code is more involved but does not need to be trusted since we can always arrange to check the generated invariant.)
The source code for the system we are calling ACheck is composed of three items. The first two are standard systems, namely, Bedwyr and Abella (although an extension to Abella is needed to support the exporting of theorems and certificates for checking). The third, new component is called FPCcheck and is located at

https://github.com/proofcert/fpccheck

The documentation at that address explains where to find and load the other two systems. To illustrate here the kinds of examples available on the web page, the Abella theory files can have a ship command that is followed by a string describing a certificate to use to prove the proposed theorem: checking of this certificate is shipped to the Bedwyr-based kernel for checking. In this particular case, the induction certificate constructor is given three arguments: the first is the maximal number of decides that can be used in the proof, the second and third are bounds on the number of unfoldings in the \(\uparrow\) and \(\downarrow\) phases respectively.

Theorem plus0com : forall N, is_nat N -> plus N zero N.
ship "(induction_{1\downarrow0\uparrow1})".

Theorem plusscom : forall M, is_nat M -> forall N, is_nat N ->
forall P, plus M N P -> plus M (succ N) (succ P).
ship "(induction_{1\downarrow0\uparrow1})".

Theorem pluscom : forall N, is_nat N -> forall M, is_nat M ->
forall S, plus N M S -> plus M N S.
ship "(induction_{2\downarrow1\uparrow0})".

The bound on the number of decide rules (first argument) is also a bound on the number of lemmas that can be used on any given branch of the reconstructed proof.

6 Certificate elaboration

Proof checking has been implemented many times over the past decades, ranging from Automath [12] to the Edinburgh LCF [15] to Dedukti [13]. Although logic programming engines have seldom been used for such purposes, they make for rather natural and direct implementations of proof checkers. Logic programming foundations make it possible to naturally perform certain tasks that might be harder to do in these other proof checking frameworks. One such task is to do “proof reconstruction” as part of proof checking: the designer of a proof certificate format can leave out details of a proof from a certificate if that designer feels confident that the missing details can be reconstructed with acceptable costs. In that way, there is an easy trade-off between the size of certificates and the costs of checking certificates.

In a similar vein, it is possible to do what might be called “proof elaboration”: that is, if we are given a certificate structure that leaves out details and another one that contains some missing details, then it should be possible to elaborate
the implicit certificate into the more explicit certificate during the process that checks the implicit certificate. For example, consider the existential expert as defined for two different certificate constructors (Section 3).

\[ \exists e((\text{instan } t \, \Xi), \Xi, t) \]

\[ \exists e((f \, \Xi), (f \, \Xi), t) \]

The first constructor \texttt{instan} provides an explicit instantiation term for the existential quantifier while the second constructor does not contain such information.

We can have a pairing constructor, say \texttt{⟨·,·⟩}, of certificates that is defined for all clerks and expects by simply invoking the same clerk and expert on the components of the pair. For example, the existential expert would be defined as

\[ \exists e(\langle \Xi_1, \Xi_2 \rangle, \langle \Xi_1', \Xi_2' \rangle, t) \]

\[ \exists e(\Xi_1, \Xi_1', t) \land \exists e(\Xi_2, \Xi_2', t) \]

In this way, if we pair an implicit proof certificate with the more explicit version of that certificate, we can use the underlying logic programming engine to record into the more explicit certificate information that was discovered during the proof reconstruction of the implicit certificate.

Following this strategy, it is possible for us to elaborate the induction certificate that simply provides a bound on the number of decide rules that are used to build a proof into an explicit tree structure of indexes that are used to build the proof. Using this elaboration device, once an implicit certificate is used to find a proof, a more explicit form can be immediately synthesized and output. This more explicit certificate could contain useful information and could be checked by a checker that might want more explicit information.

7 Conclusion

We have briefly described a methodology for defining high-level inference rules within an existing interactive inductive theorem prover. This methodology applies the notion of proof certificate in a surprising fashion: we use the certificate to describe the high-level structure of a proof and then ask the proof checker to attempt a proof reconstruction of that certificate for the given theorem.

We have focused on addressing “simple inductive proofs” that are applicable in a wide range of situations involving inductively defined datatypes. Since our method is a direct implementation of simple proof theory concepts and since those proof theory concepts are also known to work for co-induction and for the \(\lambda\)-tree approach to syntax (including the \(\nabla\)-quantifier [2, 3]) we will be able to generalize this work to that richer setting.

Finally, it would be interesting to see how our use of high-level descriptions of proofs and proof reconstruction might be related to the work of Bundy and his colleagues on proof plans and rippling [8, 9].

Acknowledgments: We thank Kaustuv Chaudhuri for some useful discussions on this topic. This work was funded by the ERC Advanced Grant ProofCert.
References