Proof Outlines as Proof Certificates: A System Description

Roberto Blanco
Inria & LIX, École Polytechnique

Dale Miller
Inria & LIX, École Polytechnique

We apply the *foundational proof certificate* (FPC) framework to the problem of designing high-level outlines of proofs. The FPC framework provides a means to formally define and check a wide range of proof evidence. A focused proof system is central to this framework and such a proof system provides an interesting approach to proof reconstruction during the process of proof checking (relying on an underlying logic programming implementation). Here, we illustrate how the FPC framework can be used to design proof outlines and then to exploit proof checkers as a means for expanding outlines into fully detailed proofs. In order to validate this approach to proof outlines, we have built the ACheck system that allows us to take a sequence of theorems and apply the proof outline "do the obvious induction and close the proof using previously proved lemmas".

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 skeptics are willing to trust. Typically, checking the application of 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 briefly describe the ACheck system in which we build and check *proof outlines* as combinations of such synthetic rules.

Consider the following inductive specification of addition of natural numbers

```
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, respectively. (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 theorems 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.

To appear in EPTCS.

```
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.
```

These three lemmas have the same high-level proof outline: use the obvious induction invariant, 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 prover [6] proves such theorems in a similar fashion (but for inductive definitions of functions). Twelf [11] can often prove that some relations are total and functional using a series of similar steps [12]. The tactics and tacticals of LCF have been used to implement procedures that attempt to find proofs using these steps [13]. Finally, TAC [4] 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 proof certificates that describe the structure of a proof outline 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 of the lemma.

2 A focused proof system

Consider the two introduction rules in Figure 1. 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).

$$\frac{\Gamma \vdash B_i}{\Gamma \vdash B_1 \vee B_2} \qquad \frac{\Gamma \vdash B[t/x]}{\Gamma \vdash \exists x.B}$$

Figure 1: From the LJ calculus

$$\frac{\Gamma \vdash B_i \Downarrow}{\Gamma \vdash B_1 \vee B_2 \Downarrow} \qquad \frac{\Gamma \vdash B[t/x] \Downarrow}{\Gamma \vdash \exists x.B \Downarrow}$$

Figure 2: Focusing annotations

It is meaningless to use Gentzen's sequent calculus to directly support proof automation. Consider, for example, attempting to prove the sequent $\Gamma \vdash \exists x \exists y[(p\ x\ y) \vee ((q\ x\ y) \vee (r\ x\ y))]$, where Γ contains, say, a hundred formulas. The search for a (cut-free) proof of this sequent can confront the need to choose from among a hundred-and-one introduction rules. If we choose the right-side introduction rule, we will then be left with, again, a hundred-and-one introduction rules to apply to the premise. Thus, reducing this sequent to, say, $\Gamma \vdash (q\ t\ s)$ requires picking one path of choices in a space of 101^4 choices.

Focused proof systems address this explosion by organizing rules into two phases. For example, the rules in Figure 1 are written instead as Figure 2. Here, the formula on which one is introducing a connective is marked with the $\Downarrow$: as a result, reducing the sequent $\Gamma \vdash \exists x \exists y[(p\,x\,y) \vee ((q\,x\,y) \vee (r\,x\,y))] \Downarrow$ to $\Gamma \vdash (q\ t\ s) \Downarrow$ involves only those choices related to the formula marked for focus: no interleaving of other choices needs to be considered.

While the $\Downarrow$ phase involves rules that may not be invertible, the $\Uparrow$ phase involves rules that must be invertible. For example, the left rules for $\vee$ and $\exists$ are invertible and their introduction rule is listed as

$$\frac{\Gamma \Uparrow B_1, \Theta \vdash \mathscr{R} \quad \Gamma \Uparrow B_2, \Theta \vdash \mathscr{R}}{\Gamma \Uparrow B_1 \vee^+ B_2, \Theta \vdash \mathscr{R}} \qquad \frac{\Gamma \Uparrow [y/x]B, \Theta \vdash \mathscr{R}}{\Gamma \Uparrow \exists x.B, \Theta \vdash \mathscr{R}}$$

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 (i.e., stored in) the zone written as Γ, using the following *store-left* rule

$$\frac{C,\Gamma \Uparrow \Theta \vdash \mathscr{R}}{\Gamma \Uparrow C,\Theta \vdash \mathscr{R}}\ S_l.$$

Finally, when the zone between the $\Uparrow$ and the $\vdash$ is empty (i.e., 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

$$\frac{\Gamma,N \Downarrow N \vdash E}{\Gamma,N \Uparrow \cdot \vdash \cdot \Uparrow E}\ D_l \qquad \text{and} \qquad \frac{\Gamma \vdash P \Downarrow}{\Gamma \Uparrow \cdot \vdash \cdot \Uparrow P}\ D_r.$$

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 `plus` in the previous section, we are actually naming a least fixed point expression. In the case of `plus`, that expression is

$$\mu\lambda P\lambda n\lambda m\lambda p.(n = z \wedge m = p) \vee \exists n'\exists p'.[n = (s\ n') \wedge p = (s\ p') \wedge (P\ n'\ m\ p')].$$

For the treatment of least fixed points, we follow the μLJ proof system and its focused variant [1, 3]. The treatment of least fixed point expressions in the $\Uparrow$ phase and the $\Downarrow$ phase is given by the three rules

$$\frac{\mu B\bar{t},\Gamma \Uparrow \Theta \vdash \mathscr{R}}{\Gamma \Uparrow \mu B\bar{t},\Theta \vdash \mathscr{R}} \qquad \frac{\Gamma \Uparrow S\bar{t},\Theta \vdash N \qquad \Uparrow BS\bar{x} \vdash S\bar{x}}{\Gamma \Uparrow \mu B\bar{t},\Theta \vdash N} \qquad \frac{\Gamma \vdash B(\mu B)\bar{t} \Downarrow}{\Gamma \vdash \mu B\bar{t} \Downarrow}$$

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 right premise of the induction rule shows that S is a prefixed point (i.e., $BS \subseteq S$). In general, supplying an invariant can be tedious so we shall also identify two admissible rules for 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

The main idea behind using the *foundational proof certificate* (FPC) [9] approach to defining proof evidence is that theorem provers output their proof evidence to some persistent memory (e.g., a file) and that independent and trustable proof checkers examine this evidence for validity. In order for such a scheme to work, the semantics of the output proof evidence must be formally defined. The FPC framework provides ways to formally define such proof semantics which is also executable when interpreted on top of a suitable logic engine.

There are four key ingredients to providing such a formal definition 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.

$$\frac{\Xi_1 : \Gamma \vdash B_i \Downarrow \qquad \vee_e(\Xi_0,\Xi_1,i)}{\Xi_0 : \Gamma \vdash B_1 \vee B_2 \Downarrow} \qquad \frac{\Xi_1 : \Gamma \vdash B[t/x] \Downarrow \qquad \exists_e(\Xi_0,\Xi_1,t)}{\Xi_0 : \Gamma \vdash \exists x.B \Downarrow}$$

These two augmented rules contain two of the four ingredients of an FPC: the schema variable Ξ ranges over terms that denote the actual proof evidence comprising a certificate. The additional premises involve *experts* which are predicates that relate the concluding certificate Ξ_0 to a continuation certificate Ξ_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. It is not required, however, for an expert to definitively extract such information. For example, the disjunction expert might guess both 1 and 2 for i: the proof checker will thus need to handle such non-determinism during the checking of certificates.

$$\frac{\Xi_1 : \langle l,C\rangle, \Gamma \Uparrow \Theta \vdash \mathscr{R} \qquad store_c(\Xi_0,\Xi_1,l)}{\Xi_0 : \Gamma \Uparrow C, \Theta \vdash \mathscr{R}}\ S_l$$

$$\frac{\Xi_1 : \Gamma, \langle l,N\rangle \Downarrow N \vdash E \qquad decide_e(\Xi_0,\Xi_1,l)}{\Xi_0 : \Gamma, \langle l,N\rangle \Uparrow \cdot \vdash \cdot \Uparrow E}\ D_l$$

Figure 3: Two augmented rules

Another ingredient of an FPC is illustrated by the augmented inference rules in Figure 3. The store-left (S_l) inference rule is augmented with an extra premise that invokes a *clerk* predicate which is responsible for computing an index l that is associated to the stored formula (here, C). The augmented decide-left (D_l) rule is given an extra premise that uses an expert predicate: that premise can be used to compute the index of the formula that is to be selected for focus. 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 [9], indexes have been identified with structures as diverse as formula occurrences and de Bruijn numerals. In this paper, the only role planned by indexes will be as the names given to lemmas and to hypotheses (i.e., formulas that are stored on the left using the S_l inference rule).

As indicated above, there are essentially three operations that we can perform to treat a least fixed point formula in the left-hand context. In fact, we shall expand these into the following four augmented inference rules.

1. The fixed point can be "frozen" in the sense that the store-left (S_l) rule is applied to it. As a result of such a store operation, the stored occurrence of the fixed point will never be unfolded and will not be the site of an induction. Such a frozen fixed point can only be used later in proof construction within an instance of the initial rule.

2. The fixed point can be unfolded just as it can be on the right-hand side of the sequent. (Unfolding on the left is a consequence of the more general induction rule.) The following augmented inference rule can be used to control when such a fixed point is unfolded.

$$\frac{\Xi_1 : \mathscr{N} \Uparrow B(\mu B)\bar{t}, \Gamma \vdash \mathscr{R} \qquad unfold(\Xi_0,\Xi_1)}{\Xi_0 : \mathscr{N} \Uparrow \mu B\bar{t}, \Gamma \vdash \mathscr{R}}$$

3. The most substantial inference rule related to the least fixed point is the induction rule. In its most general form, this inference rule involves proving premises that involve an invariant. A proof certificate term Ξ_0 could include such an invariant explicitly and the following augmented rule could be used to extract and use that invariant.

$$\frac{\Xi_1 : \mathscr{N} \Uparrow S\bar{t}, \Gamma \vdash \mathscr{R} \qquad \Xi_2\,\bar{y} : \mathscr{N} \Uparrow B S \bar{y} \vdash S\bar{y} \qquad ind(\Xi_0,\Xi_1,\Xi_2,S)}{\Xi_0 : \mathscr{N} \Uparrow \mu B\bar{t}, \Gamma \vdash \mathscr{R}}$$

4. In general, it appears that invariants are often complex, large, and tedious structures to build and use. Thus, it is most likely that we need to develop a number of techniques by which invariants

are not built directly but are rather implied by alternative reasoning principles. For example, the Abella theorem prover [2], allows the user to do induction not by explicitly entering an invariant but rather by performing a certain kind of guarded, circular reasoning. In the context of this paper, we consider, however, only one approach to specifying induction and that involves taking the sequent $\mathcal{N} \Uparrow \mu B\bar{t}, \Gamma \vdash \mathcal{R}$ and abstracting out the fixed point expression to yield the "obvious" invariant $\hat{S}$ so that the premise $\mathcal{N} \Uparrow S\bar{t}, \Gamma \vdash \mathcal{R}$ has an easy proof. As a result, only the second premise related to the induction rule needs to be proved. The following augmented rule is used to generate and check whether or not the obvious induction invariant can be used.

$$\frac{\Xi_1\,\bar{y} : \mathcal{N} \Uparrow B\hat{S}\bar{y} \vdash \hat{S}\bar{y} \qquad \mathit{obvious}(\Xi_0, \Xi_1)}{\Xi_0 : \mathcal{N} \Uparrow \mu B\bar{t}, \Gamma \vdash \mathcal{R}}$$

An FPC definition is then a collection of the following: (i) the term constructors for the term encoding proof evidence (the Ξ schema variable above), (ii) the constructors for building indexes, and (iii) the definitions of predicates for describing how the clerks and experts behave in different inference rules. 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 and small certificates where details are elided and reconstructed as needed). In our own work, we have used both λProlog [10] and Bedwyr [5] as that logic-based engine.

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 translate that algorithm into an FPC.

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.

Treating the induction rule. The invertible ($\Uparrow$) phase is generally a place where no important choices in the search for a proof appear. When dealing with a formula that is a fixed point, however, this is no longer true. As described in the previous section, we treat that fixed point expression either by freezing (see also [1]), unfolding, or using an explicitly supplied invariant or the "obvious" induction.

Lemmas must be invoked. Although the focusing framework can work with any provable formula as a lemma, we shall only consider lemmas that are Horn clauses (for example, the lemmas at the end of Section 2). Consider 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 $\Gamma \Downarrow \forall\bar{x}[A_1 \supset A_2 \supset A_3] \vdash E$ is provable if and only if there is a substitution θ for the variables in the list of variables $\bar{x}$ such that θA_1 and θA_2 are in Γ and the sequent $\Gamma, \theta A_3 \vdash E$ is provable. The application of such a lemma is then seen as forward chaining: if the context Γ contains two atoms (frozen fixed points) then add a third.

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 collections of supporting lemmas—starting from the least explicit to the most explicit—have been tested within our framework: (*i*) a bound on the number of lemmas that can be used to finish the proof; (*ii*) a list of possible lemmas to use in finishing the proof; and (*iii*) a tree of lemmas, indicating which lemmas are applied following the conjunctive structure of the remaining proof.

5 The proof checker

A direct and natural way to implement the FPC approach to proof checking is to use a logic programming language: by turning the augmented inference rules sideways, one gets logic programming clauses. In this way, the rule's conclusion becomes the head of the clause and the rule's premises become the body of the clause. When proof checking in (either classical or intuitionistic) first-order logic without fixed points, the λProlog programming language [10] 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 [9]. When the logic itself contains fixed points, as is the case in this paper, λProlog is no longer a good choice: instead, a stronger logic that incorporates aspects of the *closed world assumption* is needed. In particular, the ACheck system has two parts. The first is a theorem prover; we have used Abella since it was easy to modify it for exporting theorems and certificates for use by the second part. The second part is the proof checker, FPCcheck, that verifies the output of a session of the theorem prover, suitably translated. This checker is written in the Bedwyr model checking system [5] and is the new component underlying this particular paper: its code is available from `https://github.com/proofcert/fpccheck`. The documentation at that address explains where to find Bedwyr and our modified Abella system.

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: the 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 and 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␣0␣1)".
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␣0␣1)".
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␣1␣0)".
```

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 proof.

6 System architecture

Our system for producing and checking proof outlines follows a simple linear work-flow: first, state a theorem and obtain a proof outline, next, attempt to check the theorem with the outline given as a proof certificate. We have structured this process in three computation steps, involving a theorem prover, a translator, and a proof checker. Their roles are given here.

The theorem prover. An existing theorem prover is principally the source of the concrete syntax of definitions and theorems. It may not be directly involved in the work-flow, particularly if the proof language is extended to support `ship` tactics that enable the omission of detailed proof scripts. At the end of the phase a *theorem file* with all relevant definitions, theorem statements and proof scripts is obtained.

The translator. For each theorem prover, we need to furnish a translator that can convert the concrete syntax of the theorem prover into that of the proof checker. It may export explicit certificates given by the `ship` tactic or derive certificates automatically, possibly making use of proof scripts and execution traces in the theorem file. These translation facilities may be built into the theorem prover itself, or an instrumented version of it, thus encapsulating the first two stages. The translator outputs translation files usable directly by a general-purpose, universal proof checker.

The proof checker. The proof checker, as described in Section 5, implements a focused version of the μLJ logic in the Bedwyr model checking system, augmented to further implement the FPC framework. The translated theorems and their certificates plug into this kernel and constitute a full instantiation of the system, which Bedwyr can execute to verify that the certificates can be elaborated into complete proofs of their associated theorems.

Translators are the only addition needed to enable a new theorem prover to use the FPC framework. Such translators are free to implement sophisticated mechanisms to produce efficient certificates from proof scripts but, in fact, only a more shallow syntactic translation is strictly required: in this latter case, sophistication must be built into the FPC definitions. For the present study with the Abella interactive theorem prover, the translator has been integrated in an instrumented version of Abella, a system whose proximity with Bedwyr syntax is naturally well-suited to the approach.

The work-flow structure just described makes no explicit reference to the FPCs that could be shipped, constructed, and checked. These can conform to the definition of proof outlines used throughout this paper or be tailored to each specific problem. The translator module can choose to use proof outlines as the default, as is the case in our examples. It could also let a user of `ship` specify an FPC definition to be included in the resulting translation, or generate tentative certificates with rules involving other FPC definitions.

7 Conclusion

We have described a methodology for elaborating proof outlines using a framework for checking proof certificates. We have illustrated this approach with "simple inductive proofs" that are applicable in a wide range of situations involving inductively defined datatypes. We plan to scale and study significantly more complex examples in the near future. 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, 7].

Acknowledgments: We thank anonymous reviewers and K. Chaudhuri for comments on a draft of this paper. This work was funded by the ERC Advanced Grant ProofCert.

References

[1] David Baelde (2012): *Least and greatest fixed points in linear logic*. *ACM Trans. on Computational Logic* 13(1), doi:10.1145/2071368.2071370. Available at `http://tocl.acm.org/accepted/427baelde.pdf`.

[2] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu & Yuting Wang (2014): *Abella: A System for Reasoning about Relational Specifications*. *Journal of Formalized Reasoning* 7(2), doi:10.6092/issn.1972-5787/4650. Available at `http://jfr.unibo.it/article/download/4650/4137`.

[3] David Baelde & Dale Miller (2007): *Least and greatest fixed points in linear logic*. In N. Dershowitz & A. Voronkov, editors: *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, *LNCS* 4790, pp. 92–106. Available at `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/lpar07final.pdf`.

[4] David Baelde, Dale Miller & Zachary Snow (2010): *Focused Inductive Theorem Proving*. In J. Giesl & R. Hähnle, editors: *Fifth International Joint Conference on Automated Reasoning*, *LNCS* 6173, pp. 278–292, doi:10.1007/978-3-642-14203-1_24. Available at `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/ijcar10.pdf`.

[5] (2015): *The Bedwyr system*. Available at `http://slimmer.gforge.inria.fr/bedwyr/`.

[6] Robert S. Boyer & J. Strother Moore (1979): *A Computational Logic*. Academic Press.

[7] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland & A. Smaill (1993): *Rippling: A Heuristic for Guiding Inductive Proofs*. *Artificial Intelligence* 62(2), pp. 185–253, doi:10.1016/0004-3702(93)90079-Q.

[8] Alan Bundy (1987): *The use of explicit plans to guide inductive proofs*. In: *Conf. on Automated Deduction (CADE 9)*, pp. 111–120.

[9] Zakaria Chihani, Dale Miller & Fabien Renaud (2013): *Foundational proof certificates in first-order logic*. In Maria Paola Bonacina, editor: *CADE 24: Conference on Automated Deduction 2013*, *LNAI* 7898, pp. 162–177, doi:10.1007/978-3-642-38574-2_11.

[10] Dale Miller & Gopalan Nadathur (2012): *Programming with Higher-Order Logic*. Cambridge University Press, doi:10.1017/CBO9781139021326.

[11] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems*. In H. Ganzinger, editor: *16th Conf. on Automated Deduction (CADE)*, *LNAI* 1632, Springer, Trento, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[12] Carsten Schürmann & Frank Pfenning (2003): *A Coverage Checking Algorithm for LF*. In: *Theorem Proving in Higher Order Logics: 16th International Conference, TPHOLs 2003*, *LNCS* 2758, Springer, pp. 120–135, doi:10.1007/10930755_8.

[13] Sean Wilson, Jacques Fleuriot & Alan Smaill (2010): *Inductive Proof Automation for Coq*. In: *Second Coq Workshop*. Available at `http://hal.archives-ouvertes.fr/inria-00489496/en/`.