Design and verification of functional proof checkers

Roberto Blanco
Inria
Paris, France
roberto.blanco@inria.fr

Abstract

Small proof checkers increase trust in the results of theorem provers by providing independent and trustworthy validation of results; even more, these checkers can be carefully architected to be reusable across proof formats and enable the communication, and sharing between systems, of proofs and their meaning. One such architecture is the FPC framework, based on sequent calculi for various logics, notably classical and intuitionistic. We describe purely functional implementations of determinate proof-checking kernels in OCaml and Gallina, and correctness of the latter is given by a formal proof of soundness in Coq, from which a verified checker can be extracted; we also discuss the integration of these tools in the context of general, not necessarily determinate, proof checking, and directly within proof assistants.

CCS Concepts • **Software and its engineering → Functional languages**; • **Theory of computation → Logic and verification**; *Interactive proof systems*; *Proof theory*;

Keywords Coq, functional programming, OCaml, proof certificates, proof checking

ACM Reference Format:
Roberto Blanco. 2018. Design and verification of functional proof checkers. In *Proceedings of ML Family Workshop.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In theorem proving, it is of paramount importance not only how to prove theorems with the assistance of computer programs, but also how to represent the proofs produced by those programs and how to express and communicate the meaning of those proofs. As any complex software, theorem provers are subject to bugs, which in this context have a crucial aggravating risk: programming errors in a theorem prover may compromise its soundness, and therefore produce false assertions about, e.g., the correctness of a critical piece of software they are tasked with verifying. A widely agreed-upon principle is that theorem provers should satisfy the *de Bruijn criterion*, i.e., their output (the proof evidence) should be easily and independently checkable by small and simple programs: these are called proof checkers.

ML Family Workshop, September 2018, Saint Louis, Missouri USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The notion of using functional languages to build and check proofs is not new: for example, the kernels of proof assistants such as Coq and the HOL family—both written in ML—are direct applications of this idea, and indeed ML itself originated as the language of the seminal LCF theorem prover. Beyond these specialized proof checkers where the format of proofs is essentially fixed, we wish to consider the design of proof-checking *kernels* which operate on two pieces of information: first, a self-contained description of a proof format (in terms of an underlying logic), which plugs into the kernel and instantiates a full proof checker; second, a proof expressed in the given proof format. A proof-checking kernel is thus a parametric proof checker.

The fundamental requirement of a proof-checking kernel is that it must be sound by construction: no matter what code it may be instantiated with (i.e., a certificate definition) and no matter the proof evidence, it must never be tricked into falsely declaring a formula to be a theorem of its logic. Recent advances in proof and type theory have enabled the design of precisely such powerful proof checking architectures as the FPC framework [5]—our present focus—and Dedukti [1]. Their interest extends well beyond the obvious value of being able to easily program correct-by-construction checkers for many different proof formats. By enabling these formats to be expressed in a common logic, communication and even interoperability between varied and otherwise isolated theorem provers becomes possible up to the point where their logics coincide.

In this paper, we explore the role of functional languages in the implementation of trusted proof checkers. In its most general form, proof checking is closely related to proof search, which can be encoded naturally in a suitable logic programming language. However, most common formats for proof evidence offer a large amount of guidance that restricts the otherwise intractable search space, to the point that proof checking can in some instances be described by decidable procedures on the payload of a given proof object. In addition, from the point of view of trust, logic programming involves certain sophisticated mechanisms (backtracking search, higher-order unification, etc.) whose full expressive power is seldom needed; avoiding unnecessary sources of complexity in trusted software is therefore very desirable. Our contribution specializes the general-purpose framework of Foundational Proof Certificates (FPC) to represent and check proof evidence that effectively embodies a functional

computation. We have considered both classical and intuitionistic logics; this presentation will use the first of these as a vehicle.

In the remainder of the paper we will design strategies for efficient and trustworthy proof checkers covering the functional fragment of proof systems based on the sequent calculus. We consider two groups of designs. First, we study purely functional (side effect-free) native implementations in OCaml, where a simple encoding, based on function spaces, models the treatment of binders in the first-order fragment of the logic. Later, we encode our proof checkers in Coq's programming language, Gallina, and prove their soundness; these can later be extracted to another functional programming language like, again, OCaml.

2 Proofs and certificates

As a preliminary approach to our discussion of functional proof checkers, we present a quick overview of the foundations of proof checking; for a more comprehensive treatment, see, e.g., [5]. The object of this study is the representation and manipulation of certain families of proofs represented inside sequent calculi that are sound and complete with respect to a given logic, and the representation and manipulation of proof evidence related to those proofs in the form of *proof certificates* for the sequent calculi.

At its core, sequent calculus manipulates formulas expressed by the usual set of logical constants: in the classical case, the nullary constants *true* (t) and *false* (f), and the binary constants *and* ($\wedge$) and *or* ($\vee$); in first-order logic, the *universal* ($\forall$) and *existential* ($\exists$) quantifiers are added. These are supplemented by a type signature of non-logical constants that function as *atoms*; a *literal* is either an atom or a negated atom. Quantifiers parameterize atoms by *terms*, which are also part of the signature. Quantifiers are instantiated by the usual substitution operation ($[t/x]A$ represents the substitution of a term t for a free variable x in a formula A). *Sequents* are built from lists of formulas organized in several zones around a turnstyle ($\vdash$). A sequent calculus consists of a collection of *inference rules* built from these elements, each relating a conclusion below the inference line and a number of premises above. The substrate of Figure 1, where all additional annotations and shadowed boxes are removed, represents such a system.

The standard sequent calculus can be refined (by reducing its potential nondeterminism) in two phases. The first step is the introduction of *focusing*, and with it the notion of *polarity*. A focused system manipulates polarized formulas: in our presentation, the propositional constants t, f, $\wedge$ and $\vee$ have two superscripted versions, positive and negative; $\forall$ is always negative and $\exists$ is always positive. Inference rules are now divided in groups based on the polarity of the top-level connective of the formula being manipulated: negative (*asynchronous*, $\Uparrow$) or positive (*synchronous*, $\Downarrow$). Focused

ASYNCHRONOUS INTRODUCTION RULES

$$\frac{}{\Xi_0 \vdash \Gamma \Uparrow t^-, \Theta}$$

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

$$\frac{\Xi_1 \vdash \Gamma \Uparrow A, B, \Theta \quad \vee_c(\Xi_0, \Xi_1)}{\Xi_0 \vdash \Gamma \Uparrow A \vee^- B, \Theta}$$

$$\frac{(\Xi_1\, y) \vdash \Gamma \Uparrow (B\, y), \Theta \quad \forall_c(\Xi_0, \Xi_1)}{\Xi_0 \vdash \Gamma \Uparrow \forall x.B, \Theta}\ \dagger$$

$$\frac{\Xi_1 \vdash \Gamma \Uparrow \Theta \quad f_c(\Xi_0, \Xi_1)}{\Xi_0 \vdash \Gamma \Uparrow f^-, \Theta}$$

SYNCHRONOUS INTRODUCTION RULES

$$\frac{t_e(\Xi_0)}{\Xi_0 \vdash \Gamma \Downarrow t^+}$$

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

$$\frac{\Xi_1 \vdash \Gamma \Downarrow B_i \quad \vee_e(\Xi_0, \Xi_1, i)}{\Xi_0 \vdash \Gamma \Downarrow B_1 \vee^+ B_2}\ i \in \{1, 2\}$$

$$\frac{\Xi_1 \vdash \Gamma \Downarrow (B\, t) \quad \exists_e(\Xi_0, \Xi_1, t)}{\Xi_0 \vdash \Gamma \Downarrow \exists B}$$

IDENTITY RULES

$$\frac{\langle l, \neg P_a \rangle \in \Gamma \quad init_e(\Xi_0, l)}{\Xi_0 \vdash \Gamma \Downarrow P_a}\ init$$

$$\frac{\Xi_1 \vdash \Gamma \Uparrow B \quad \Xi_2 \vdash \Gamma \Uparrow \neg B \quad cut_e(\Xi_0, \Xi_1, \Xi_2, B)}{\Xi_0 \vdash \Gamma \Uparrow \cdot}\ cut$$

STRUCTURAL RULES

$$\frac{\Xi_1 \vdash \Gamma \Downarrow P \quad \langle l, P \rangle \in \Gamma \quad decide_e(\Xi_0, \Xi_1, l)}{\Xi_0 \vdash \Gamma \Uparrow \cdot}\ decide$$

$$\frac{\Xi_1 \vdash \Gamma \Uparrow N \quad release_e(\Xi_0, \Xi_1)}{\Xi_0 \vdash \Gamma \Downarrow N}\ release$$

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

† (eigenvariable restriction): y is not free in the premise

Figure 1. The augmented LKF^a focused sequent calculus for classical logic [4].

proofs are organized into *phases* of rules of the same polarity, with *structural* and *initial* rules at the boundaries.

The second extension is the FPC framework, which augments the focused sequent calculus with the shadowed boxes of Figure 1. This family of augmentations is always sound. Our particular instance, which is also complete, will now be introduced alongside the functional fragment of the system.

3 Functional proof checkers

Consider the standard inference rule for conjunction in classical sequent calculus:

$$\frac{\vdash A, \Gamma \qquad \vdash B, \Gamma}{\vdash A \wedge B, \Gamma} \wedge$$

Here, a proof of $A \wedge B$ is decomposed into two separate proofs of A and B, respectively. In the FPC framework, each inference rule is augmented with a parametric predicate (defined by each proof format) which inspects the proof evidence available at a point in the proof, and does one of two things: either it allows the application of the inference rule, producing continuations of the proof evidence for each premise; or it disallows its application. The decorated inference rule in the *focused* sequent calculus looks like this:

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

The sole additions to the standard proof system are the representations of the proof evidence Ξ_i (called certificate terms), and the parametric predicate $\wedge_c(\cdot, \cdot, \cdot)$ that relates the evidence in the conclusion to the evidence in the premises, where applicable. This treatment, extended to all the inference rules in the proof system, admits a crisp declarative encoding. In the determinate fragment of the proof system, this encoding is purely functional. In OCaml, the example rule can be expressed by a main case analysis on a sequent—which determines the applicable inference rule—, and a simple recursive manipulation on the components of the sequent, as shown in Figure 2 (here, without thunking).

That is, inference rule encodings are embedded in a top-level, recursive checking function that computes whether a certificate can guide the kernel towards a proof of the formula that is the object of the recursion. The certificate definition is supplied by the user of the kernel, here encapsulated in the module `Lkf_bureau`. This module is responsible—using a proof certificate as input—for all decisions concerning the checking of a proof: whether an inference rule is allowed to proceed, what disjunct to select, what term to use to instantiate an existential quantifier, what formula to use in a cut, etc. Any determinate proof format can be encoded by furnishing the following pieces of information, all carefully encapsulated by interfaces:

- A type of certificates defining the constructors of proof evidence and the information they contain.

```
let rec check Ξ0 = function
  | Unfocused(Γ, workbench) ->
    (match workbench with
    | hd :: Θ ->
      (match hd with
      | NegativeAnd(A, B) ->
        and_clerk A B Ξ0 Γ Θ
(*...*)
and and_clerk A B Ξ0 Γ Θ =
  match Lkf_bureau.and_clerk Ξ0 with
  | None -> false
  | Some (Ξ1, Ξ2) ->
    let check_left =
      check Ξ1 (Unfocused(Γ, A :: Θ))
    and check_right =
      check Ξ2 (Unfocused(Γ, B :: Θ))
    in check_left && check_right
(*...*)
```

Figure 2. OCaml encoding of top-level pattern matching on sequent (partial) and example inference rule.

- A type of indexes used to specify how we can refer symbolically to formulas and what information is associated with them.
- An implementation of the functions of the bureau, one for each inference rule, to control the evaluation and flow of proof evidence.
- A term type and an atom type over which formulas can be written.

When we restrict ourselves to propositional logic, the term type is superfluous and can be omitted. In full first-order logic, quantifiers need to be modeled. In the logic programming encoding, the quantifiers are modeled by term abstractions over formulas, from which application of a term yields a formula: an eigenvariable for the universal quantifier and a term for the existential quantifier. In OCaml, we use function spaces to model both quantifiers. For example, the constructor for the universal quantifier is defined as `ForAll of (Lkf_term.t -> t)`. The encoding of its inference rule in the kernel is charged with generating a fresh eigenvariable, modeled as a kernel-side term separate from client-side terms, which cannot contain kernel-side terms.

A common argument made in favor of this design is that a clear intuition of its correctness is self-evident, being a direct encoding of the sequent calculus (in a logic programming language in the general case, but also in a functional language in the determinate case). The computational behavior of the functional kernel is moreover very simple and paves the way to formally stating and verifying these claims. To this end, we use the Coq proof assistant. Using its purely functional programming language, Gallina, we reimplement

the functional kernel; the result is virtually identical to the OCaml version. Without loss of generality, and for the purposes of the proof of soundness, this kernel is integrated with the most general form of determinate certificate format, known as *maximal elaboration* and briefly discussed below. The main result is the soundness of the checker, which is proved by structural induction on the certificate with the aid of a few simple lemmas about the context of a sequent; sequents of the focused sequent calculus are polarized and can be depolarized by application of a depolarization function (see [2] for details).

Theorem 3.1. *Let Ξ be a maximally elaborate certificate and S be a sequent. If Ξ successfully certifies the sequent S via* check*, then the depolarization of the sequent $[\![S]\!]^0$ holds.*

Corollary 3.2. *Let Ξ be a maximally elaborate certificate and B be a formula. If Ξ certifies the initial sequent $\vdash \cdot \Uparrow [B]$, then the unpolarized formula $[\![B]\!]^0$ holds.*

In the propositional case, the proofs present no obstacles. The treatment of quantifiers and binders in Coq deserves special attention. In the OCaml version, function spaces were used in a very restricted form to represent the notion of term substitution by application, and that notion only. While this encoding is equally possible in Gallina, without further information about well-formedness a formal proof must reject this encoding because function spaces are much more expressive than our extremely limited informal interpretation. Either external assumptions or an explicit treatment of substitution, together with a clear model of eigenvariables, are required to make progress in the first-order case. That is, the treatment of higher-order abstract syntax (HOAS) that is built into languages like λProlog must be reconstructed in Coq to enforce the proper encoding of object-level quantification, and that fact used to prove soundness.

4 Discussion

How can we integrate a specialized checker in an arbitrary workflow to increase our trust in the correctness of proofs? Whenever FPCs are involved in the representation of those proofs, a full checker based on a logic programming kernel can be used. In this general setting, we can combine a proof certificate with a *pairing* combinator that allows us to obtain versions of the same certificate with more (or less) detail, and use it to, basically, record a full trace of the sequent calculus proof: we call this a maximal elaboration [3]. Since any proof can be expressed as a maximally elaborate certificate, all we need to do is add a final checking stage to the workflow: first, check the proof certificate *and*, at the same time, output its maximal elaboration; finally, use the latter as input to a functional checker on the same formula.

We have validated this refinement of the verification workflow by performing end-to-end, fully automatic checking of results produced by automated theorem provers integrating the OCaml-based functional checker as the final component in the chain. Thanks to its excellent performance, especially compared to the logic programming-based checkers that precede it, the overhead of this final assurance is negligible. So far, reassuringly, we have found no false proofs originating from unverified theorem provers, and no false checks originating from unverified proof-checking kernels. An important and related question is: how can we substitute the verified checker for the unverified checker? Given our formalization in Coq, its extraction mechanism can generate OCaml code corresponding to the verified checker written in Gallina, although the correctness of parsers and the mappings of certain types and functions to native OCaml code are not part of the verification and are a possible source of shallow errors.

The integration of the proof checker directly inside a system like Coq is another interesting possibility. We have experimented with this in the classical propositional setting by selecting the type of propositions, `Prop`, as the type of atoms, defining a polarized version of these and a depolarization to regular propositions. An advantage of this choice is that, if the checker proves a (polarized) formula, its depolarization yields a Coq proposition that can be considered proven and used normally inside Coq. This opens the door to exciting possibilities to program proofs in Coq. The tradeoff of `Prop` with respect to a type of atoms with decidable equality is that the checker must output not, say, a boolean, but another proposition that accumulates the equalities observed in the initial rules of a potential proof—this output proposition will be trivially true if the proof was successful.

A concern to be addressed is the fact that the soundness proof of the checker for classical logic is (very mildly) nonconstructive: under the described assumptions, it makes a single appeal to the excluded middle in its treatment of the cut rule. However, this does not invalidate the correctness claims of the code written in Gallina proper, and is in fact a condition shared with other developments that make careful, limited use of classical reasoning, such as the CompCert verified compiler.

Finally, an interesting area of further study would be in the development of a verified implementation of a general proof checker for arbitrary, not necessarily determinate, certificate definitions. This would involve the formalization of the mechanisms (backtracking search, unification, etc.) whose complexity is avoided by the functional checker. In parallel with these considerations, metaprogramming [6] could be used to directly port a (possibly existing, informally trusted) logic programming-based proof-checking kernel into Coq.

Acknowledgments

I am grateful to Dale Miller and the two anonymous reviewers for their careful reading and suggestions. This work was partly supported by the ERC Advanced Grant ProofCert.

References

[1] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. 2016. Dedukti: a Logical Framework based on the $\lambda\Pi$-Calculus Modulo Theory. (2016). http://www.lsv.ens-cachan.fr/~dowek/Publi/expressing.pdf Unpublished.

[2] Roberto Blanco. 2017. *Applications of Foundational Proof Certificates in theorem proving*. Ph.D. Dissertation. École polytechnique.

[3] Roberto Blanco, Zakaria Chihani, and Dale Miller. 2017. Translating between implicit and explicit versions of proof. In *CADE-26: 26th International Conference on Automated Deduction (LNCS)*, Leonardo de Moura (Ed.), Vol. 10395. Springer, Gothenburg, Sweden, 255–273. https://doi.org/10.1007/978-3-319-63046-5_16

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

[5] Zakaria Chihani, Dale Miller, and Fabien Renaud. 2016. A semantic framework for proof evidence. *J. of Automated Reasoning* (2016). https://doi.org/10.1007/s10817-016-9380-6 Published electronically doi:10.1007/s10817-016-9380-6.

[6] Enrico Tassi. 2018. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). (2018). https://hal.inria.fr/hal-01637063 Unpublished.