# Two applications of logic programming to Coq

Matteo Manighetti ✉ 🏠
Inria, Palaiseau, France
LIX, Ecole Polytechnique, France

Dale Miller ✉ 🏠
Inria, Palaiseau, France
LIX, Ecole Polytechnique, France

Alberto Momigliano ✉ 🏠
Dipartimento di Informatica, Università degli studi di Milano, Italy

## Abstract

The logic programming paradigm provides a flexible setting for representing, manipulating, checking, and elaborating proof structures. This is particularly true when the logic programming language allows for bindings in terms and proofs. In this paper, we make use of two recent innovations at the intersection of logic programming and proof checking. One of these is the *foundational proof certificate* (FPC) framework which provides a flexible means of defining the semantics of a range of proof structures for classical and intuitionistic logic. A second innovation is the recently released Coq-Elpi plugin for Coq in which the Elpi implementation of λProlog can send and retrieve information to and from the Coq kernel. We illustrate the use of both this Coq plugin and FPCs with two example applications. First, we implement an FPC-driven sequent calculus for a fragment of the Calculus of Inductive Constructions and we package it into a tactic to perform property-based testing of inductive types corresponding to Horn clauses. Second, we implement in Elpi a proof checker for first-order intuitionistic logic and demonstrate how proof certificates can be supplied by external (to Coq) provers and then elaborated into the fully detailed proof terms that can be checked by the Coq kernel.

**2012 ACM Subject Classification** Theory of computation → Logic and verification

**Keywords and phrases** Proof assistants, logic programming, Coq, λProlog, property-based testing

**Acknowledgements** We thank Enrico Tassi for his help with Coq-Elpi. His comments and those of the anonymous reviewers on an early draft of this paper have also been very helpful.

## 1 Introduction

Recently, Enrico Tassi et al. developed the Elpi implementation [21] of λProlog [47], and more recently, Tassi has made Elpi available as the Coq-Elpi plugin [62] (`https://github.com/LPCIC/coq-elpi`) to the Coq proof assistant. This implementation of λProlog extends earlier ones in primarily two directions: First, Elpi adds a notion of constraints and constraints handling rules, which makes it more expressive than the Teyjus implementation [51] of λProlog. Second, the plugin version of Elpi comes equipped with a quotation and anti-quotation syntax for mixing Coq expressions with λProlog program elements and an API for accessing the Coq environment, including its type checker.

The logic programming interpreter that underlies Elpi provides several convenient features for the kind of meta-programming tasks that can arise within modern proof assistants. For example, λProlog provides a declarative and direct treatment of abstract syntax that contains bindings, including capture-avoiding substitution, unification, and recursive programming. Elpi spares the programmer from dealing with low-level aspects of the representation of binders (e.g., De Bruijn indexes) while still having clean and effective ways to manipulate binding structures.

Since relations (not functions) are central in λProlog, Elpi is capable of providing direct support for the many relations that have a role in implementation and usage of proof assistants. Such relations include typing (e.g., $\Gamma \vdash M : \sigma$), evaluation (e.g., natural semantics specifications [37, 33]), and interaction (e.g., structured operational semantics [49, 57]).

Felty has also made the point that LCF-style tactics and tacticals can be given an elegant and natural specification using the higher-order relational specifications provided by λProlog [23]. Some recent implementations built using Elpi support the usefulness of higher-order logic programming as a meta-programming language for proof assistants in general [20, 32] and, in particular, for Coq via the Coq-Elpi plugin [18, 63].

In this paper, we present two applications of Elpi within Coq. With these applications, we shall illustrate that Elpi is useful not only because of its meta-programming features but also because it soundly implements a higher-order intuitionistic logic: such implementations of higher-order logic have long been known to provide powerful and flexible approaches to implementing many different logics and their proof systems [24, 53]. Following that tradition, the Elpi system makes it possible to encode the proofs and proof theory of various subsets of the logic behind Coq (see also [22]).

While other meta-programming frameworks based on functional programming such as *MetaCoq* [61] can and have been used for related endeavors, we believe (together with [20]) that they would require much more boilerplate code.

Before we can present these examples, we first highlight the rather striking differences in notions of computing and reasoning that arise on each side of the Coq-Elpi API. We will also present a quick summary of the key proof theory concepts that are used by our example applications.

## 2 Two cultures

When studying structural proof theory, one learns quickly that many concepts come in pairs: negative/positive, left/right, bottom-up/top-down, premises/conclusion, introduction/elimination, etc. When we examine the larger setting of this project of linking a logic programming engine with Coq and its kernel, we find a large number of new pairings that are valuable to explicitly discuss.

### 2.1 Proof theory vs type theory

In many ways, proof theory is more elementary and low-level than most approaches to type theory. For example, type theories usually answer the question "What is a proof?" with the response "a (dependently) typed λ-term". That is, when describing a type theory, one usually decides that a proof is a certain kind of λ-term within the system. In contrast, proof theory treats logical propositions and proofs as separate. For example, proof theory does not assume that there are terms within the logic that describe proofs.

Gentzen's discovery that the key to treating classical and intuitionistic logics in the same proof system was identifying the structural rules of the weakening and contraction and placing them on the right-side of sequents [29]. This discovery led him away from natural deduction to multiple-conclusion sequent calculus. This same innovation of Gentzen's also opened the way to another key proof-theoretic discovery, that of linear logic and linear negation [30]. While sequent calculus provides an elegant presentation of full linear logic, most treatments of linear logic in type theory have been limited to single-conclusion sequents [5] or restricted, multiple-conclusion variants [36].

As is often observed, however, sequent calculus is too low-level to be used explicitly as capturing the "essence of a proof". Fortunately, the notion of *focused proof systems* [1, 42], makes it possible to collect and join the micro-level inference rules of the sequent calculus into large-scale, synthetic inference rules. As a result, using such proof systems, it is possible to extract from sequent calculus not only natural deduction proofs [56], but also proof nets [13], and Herbrand-style expansion trees [12]. The role of focused proof systems to characterize classes of proof structures is described in the next section and used in our two example applications of Elpi with Coq.

## 2.2 Proof search vs proof normalization

Gentzen-style proofs are used to model computation in at least two different ways. The functional programming paradigm, following the Curry-Howard correspondence, models computation abstractly as $\beta$-reduction within natural deduction proofs: that is, computation is modeled as *proof normalization*. On the other hand, the logic programming paradigm, following the notion of goal-directed proof search, models computation abstractly as a regimented search for cut-free sequent calculus proofs: that is, computation is modeled as *proof search*.

Proof search has features that are hard to capture in the proof normalization setting. In particular, Gentzen's *eigenvariables* are a kind of proof-level binder. In the proof normalization setting, such eigenvariables are instantiated during proof normalization. However, during the search for cut-free proofs, eigenvariables do not vary, and they are part of the syntax of terms and formulas. As a result, they can be used in the treatment of bindings in data structures more generally. Such eigenvariables can be used by the logic programming paradigm to provide a natural and powerful approach to computing with bindings within syntax.

It is worth noting that the role of the cut rule and cut-elimination is different in these two approaches to computing. In the proof normalization paradigm, the cut rule can be used to model $\beta$ reduction, especially via the explicit substitution approach [40]. In the proof search paradigm, since computing involves the search for cut-free proofs, the cut rule plays no part in the performance of computation. However, cut and cut-elimination can be used to reason about computation: for example, cut-elimination can be used to prove "substitution lemmas for free" that arise in the study of operational semantics [28].

## 2.3 $\lambda$Prolog vs Coq

Given that $\lambda$Prolog and Coq both result from combining the $\lambda$-calculus with logic, it is important to understand some of their differences. The confusion around the term *higher-order abstract syntax* (HOAS), is a case in point. In the functional programming setting, including the Coq system, the HOAS approach leads to encoding binding structures within terms using functions. The earliest such encodings were unsatisfactory since they would allow for *exotic terms* [19] and for structures on which induction was not immediately possible [59]. Later approaches yielded non-canonical and complex encodings [17, 35], as well as sophisticated type theories [55]. All of these could support inductive and coinductive reasoning. In the logic programming setting, particularly $\lambda$Prolog, HOAS is well supported since bindings are allowed with terms ($\lambda$-bindings), formulas (quantifiers), and proofs (eigenvariables). (In fact, the original paper on HOAS [54] was inspired by $\lambda$Prolog.) For this reason, the term *$\lambda$-tree syntax* was introduced to name this particular take on HOAS [46]. The Abella proof assistant [3] was designed, in part, to provide inductive and co-inductive inference involving specifications using the $\lambda$-tree syntax approach.

Another difference between $\lambda$Prolog and functional programming can be illustrated by considering how they are used in the specification of tactics. The origin story for the ML

$$\frac{\mathrm{Ind}[p]\ (\Gamma_I := \Gamma_C) \in E \quad (\mathit{head}\ A) : T \in \Gamma_I \quad k : D \in \Gamma_C \quad \Xi_1 \Downarrow l : D \vdash A \quad \mathit{decide}_e(\Xi, \Xi_1)}{\Xi \vdash k\ l : A}$$

$$\frac{E[\,] \vdash_{CIC} A : s \quad \mathit{sort}_e(\Xi,)}{\Xi \vdash A : s} \qquad \frac{E[\,] \vdash_{CIC} A =_{\beta\delta\iota\xi} A' \quad \mathit{initial}_e(\Xi)}{\Xi \Downarrow [\,] : A \vdash A'}$$

$$\frac{\Xi_1 \vdash t : B \quad \Xi_2 \Downarrow l : D[t/x] \vdash A \quad \mathit{prod}_e(\Xi, \Xi_1, \Xi_2, t)}{\Xi \Downarrow (t\ ::\ l) : (x\colon B)D \vdash A}$$

**Figure 1** Specification of a core calculus.

functional programming language was that it was the meta-language for implementing the LCF suite of tactics and tacticals [31]. To implement tactics, ML adopted not only novel features such as polymorphically typed higher-order functional programming but also the non-functional mechanisms of failure and exception handling. While $\lambda$Prolog is also based on ML-style polymorphically typed higher-order relational programming, it also comes with a completely declarative version of failure and backtracking search. Combining those features along with its support of unification (even in the presence of term-level bindings), $\lambda$Prolog provides a rather different approach to the specification of tactics [23].

## 3 Proof theory and proof certificates

In this section, we introduce the main ideas from *focused proof systems*, *foundational proof certificates*, and the *Coq-Elpi* plugin that we need for this paper.

### 3.1 Proofs for the Horn fragment

A *Horn clause* is a formula of the form $\forall\bar{x}_1.A_1 \supset \forall\bar{x}_2.A_2 \supset \forall\bar{x}_n.A_n \supset A_0$ where $\forall\bar{x}_i$ denote a list of universal quantifiers ($i \in \{1, \dots, n\}$) and $A_0, \dots, A_n$ are atomic formulas. It is well known that the following set of sequent calculus proof rules are complete for both classical and intuitionistic logic when one is attempting to prove that a given atomic formula $A$ is provable from a set $\mathcal{P}$ of Horn clauses.

$$\frac{D \in \mathcal{P} \quad \mathcal{P} \Downarrow D \vdash A}{\mathcal{P} \vdash A}\ \mathit{decide} \qquad \frac{}{\mathcal{P} \Downarrow A \vdash A}\ \mathit{init}$$

$$\frac{\mathcal{P} \Downarrow D[t/x] \vdash A}{\mathcal{P} \Downarrow \forall x.D \vdash A}\ \forall L \qquad \frac{\mathcal{P} \vdash B \qquad \mathcal{P} \Downarrow D \vdash A}{\mathcal{P} \Downarrow B \supset D \vdash A}\ \supset L$$

Here, we use two different styles of sequents. The sequent $\mathcal{P} \vdash A$ is the usual sequent which we generally use as the end sequent (the conclusion) of a proof. The sequent $\mathcal{P} \Downarrow D \vdash A$ is a *focused* sequent in which the formula $D$ is the *focus* of the sequent. The two left introduction rules and the initial rule can only be applied to the focused formula. This latter point is in contrast to Gentzen's sequent calculus where these rules can involve *any* formula on the left of the $\vdash$. The fact that this proof system is complete for both classical and intuitionistic logic (when restricted to the Horn clause fragment) follows from rather simple considerations of Horn clauses [50] and from the completeness of uniform proofs [48] or LJT proofs [34]. The use of the term *focus* comes from Andreoli's proof system for linear logic [1].

Figure 1 contains an annotated version of these proof rules: the annotations help us connect to elements of the Coq proof system.

```
kind   goal         type.
type   go           term → term → goal.
type   bc           term → term → list term → goal.
type   check        cert → goal → o.
```

```
check Cert (go (sort S) A):-
  sortE Cert,
  coq.typecheck A (sort S) ok.
check Cert (go A Tm) :-
  coq.safe-dest-app A (global (indt Prog)) _,
  coq.env.indt Prog _ _ _ _ Kn KTypes,
  decideE Kn Cert Cert' K,
  std.zip Kn KTypes Clauses,
  std.lookup Clauses K D,
  check Cert' (bc D A L),
  Tm = (app [global (indc K)|L]).
check Cert (bc (prod _ B D) A [Tm|L])  :-
  prodE Cert Cert1 Cert2 Tm,
  check Cert1 (bc (D Tm) A L),
  check Cert2 (go B Tm).
check Cert (bc A A' [])  :-
  initialE Cert,
  coq.unify-eq A A' ok.
```

**Figure 2** Implementation of the core calculus.

1. Instead of having separate connectives for ∀ and ⊃, we have the dependent product connective $(x : A)D$.
2. We account for *computation* inside atoms by generalizing the *init* rule to allow type-level conversion.
3. We have incorporated *proof certificates* [16] (using the schematic variable $\Xi$) along with *expert predicates* (predicates with the $e$ subscript). We explain these in Section 3.2.
4. The inference rules are annotated by terms structures that can be given directly to the Coq kernel for checking.
5. We have added various premises which are responsible for interacting with Coq.

The proof system described in Figure 1 (and implemented in Figure 2) corresponds to a subset of the Calculus of Inductive Constructions in which inductive definitions are limited to Horn clauses. This system is inspired by the calculus for proof search in Pure Type Systems introduced in [41], based in turn on ideas stemming from focusing (in particular, uniform proofs [48] and the LJT calculus [34]). Similar to that calculus, we have a term language that includes terms and lists of terms, and two typing judgments for the two categories. This style of proof terms coincides with the idea behind the *spine calculus* [11]. The main novelty of our proof system here is that proof terms and proof certificates are used simultaneously in all inference rules.

The proof system is parameterized by Coq's global environment $E$, here a set of constant and inductive definitions; following Coq's reference manual, inductive definition are denoted by $\mathrm{Ind}[p](\Gamma_I := \Gamma_C)$, where $\Gamma_I$ determines the names and types of the (possibly mutually) inductive type and $\Gamma_C$ the names and types of its constructors; finally $p$ denotes the number of parameters and plays here no role. The local context is empty, since we are only dealing with types that correspond to Horn clauses, and atomic types are inductively defined. In fact, we do not have a ∀ rule on the right, although the proof theory would gladly allow it. This means that there are no bound variables in our grammar of terms. Terms are always applied to a (possibly empty) list of arguments. We delegate to Coq's type checking the enforcement of the well-sortedness of inductive types. The decide rule, as in the previous proof system for Horn logic, given an atom, selects a clause on which to backchain on: we lookup the constructors of an inductive definition from the global environment, one that matches the head symbol of the atom we aim to backchain on, and then call the latter judgment that will find a correct instantiation, if any. The rules for backchaining include the (conflation of the) left introduction rules for ∀ and ⊃, as well as the *init* rule, which incorporates Coq's conversion.

It may be at first surprising that there are no introduction rules for propositional connectives, nor equality for that matter. However, one of the beauties of the Calculus of Inductive Construction is that they are, in fact, defined inductively and therefore the *decide* rule will handle those. Thus, the syntax of proof-terms is rather simple.

## 3.2 Proof certificate checking

Figure 2 contains the Elpi implementation of the inference rules in Figure 1: $\Xi \vdash t\colon A$ corresponds to `check Cert (go A T)` and $\Xi \Downarrow l\colon D \vdash A$ corresponds to `check Cert (bc D A L)`, The code in that figure mixes both Coq-specific and FPC-specific items. We describe both of these separately.

### 3.2.1 Coq-specific code

Coq terms are accessed through the Coq-Elpi API, and their representation in λProlog takes advantage of native λProlog constructs such as lists and binders. The following is part of the Coq-Elpi API signature of constants that we use.

```
kind term   type.                                % reification of Coq terms
kind gref   type.                                % reification of refs to globals
type global gref → term.                         % coercion to term
type indt   inductive → gref.                    % reification of inductive types
type indc   constructor → gref.                  % reification of their constructors
type app    list term → term.                    % reification of nary application
type prod   name → term → (term → term) → term.  % reification of dependent product
```

Note that `prod` encodes dependent products by taking a name for pretty printing, a term and a λProlog abstraction from terms to terms: i.e., $(x : B)D$ is encoded by `prod ‘‘x’’ B (x\ D x)`; when, in the implementation of the product-left rule, we apply `D` to the variable `Tm`, we get a new term that can be used to continue backchaining. This application is obtained via meta-level substitution, in the style of HOAS. In this sense, our calculus uses *implicit* substitutions, rather than explicit ones as in the LJT and PTSC's tradition; this is consistent with proof search in our application being cut-free, whereas explicit substitutions are linked to cuts. The decide rule makes use of the Coq-Elpi primitives `coq.safe-dest-app` to obtain the head term of a (possibly nested) application, and `coq.env.indt` to access the global environment of inductive definitions and query for information about them. The `decideE` predicate, has, among others, the role of selecting which constructor to focus on from the inductive type. The kernel will successively obtain the type of the selected constructor, and initiate the backchaining phase. The latter is terminated when the focused atom unifies with the current goal, via Elpi's primitive `coq.unify-eq`.

### 3.2.2 FPC-specific code

The *foundational proof certificates* (FPC) framework was proposed in [16] as a flexible approach to specifying a range of proof structures in first-order classical and intuitionistic logics. Such specifications are also executable using a simple logic programming interpreter. As a result of using logic programming, proof certificates in this framework are allowed to lack details that can be reconstructed during the checking phase. For example, substitution instances of quantifiers do not need to be explicitly described within a certificate since unification within the logic programming checker is often capable of reconstructing such substitutions.

In this and the next section, we shall only use a much reduced subset of the FPC framework: in essence, an FPC will be used as a simple mechanism for bounding the search for proofs. In our examples, a proof certificate, denoted by the schematic variable $\Xi$, is a particular term that is threaded throughout a logic programming interpreter. For example, the inference rules in Figure 1 are augmented with an additional premise that invokes an *expert predicate* with is responsible for extracting relevant information from a proof certificate $\Xi$ as well as constructing continuation certificates, such as, $\Xi_1$ and $\Xi_2$. For example, the premise $prod_e(\Xi, \Xi_1, \Xi_2, t)$ calls the expert for products which should extract from the certificate $\Xi$ a substitution term $t$ and two continuation certificates $\Xi_1$ and $\Xi_2$ for the two premises of this rule. If the certificate $\Xi$ does not contain an explicit substitution term, the expert predicate can simply return a logic variable which would denote any term that satisfies subsequent unification problems arising in completing the check of this certificate.

In our case here, an FPC is a collection of λProlog clauses that provide the remaining details not supplied in Figures 1 and 2: that is, the exact set of constructors for the type of certificates `cert` as well as the specification of the expert predicates listed *ibidem*. The top of Figure 3 displays two FPCs, both of which can be used to describe proofs where we bound the dimension of a proof. For example, the first FPC dictates that the query `(check (qheight 5) A)` is provable in the kernel using the clauses in Figures 2 and 3 if and only if the height of that proof is 5 or less. Similarly, the second FPC can be used to bound the total number of instances of the decide rule in a proof. (Obviously, such proof certificates do not contain, for example, substitution terms.)

As it has been described in [6], it is also possible to pair together two different proof certificates, defined by two different FPC definitions, and do the proof checking in parallel. This means that we can build an FPC that *restricts* proofs satisfying two FPCs simultaneously. In particular, the infix constructor `<c>` in Figure 3 forms the pair of two proof certificates and the pairing experts for the certificate `Cert1 <c> Cert2` simply request that the corresponding experts also succeed for both `Cert1` and `Cert2`. Thus, the query `check ((qheight 4) <c> (qsize 10)) A` will succeed if there is a proof of `A` that has a height less than or equal to 4 while also being of size less than or equal to 10.

## 3.3 A Prolog-like tactic

Thanks to the Coq-Elpi interface, in particular to the “main” procedure **solve**, we can package the λProlog code for the checker as a tactic that can be called as any other tactic in a Coq script.

```
Elpi Tactic dprolog.
Elpi Accumulate lp:{{
  solve [str ”height”, int N] [goal _ Ev G _] _ :-
    coq.say "Goal:" {coq.term→ string G},
type  qheight int → cert.
type  qsize   int → int → cert.
type  <c>     cert → cert → cert.  infixr  <c> 5.
```

```
ttE   (qheight _).
sortE (qheight _).
prodE (qheight H) (qheight H) (qheight H) T.
decideE Kn (qheight H) (qheight H’) K :- std.mem Kn K, H  > 0, H’ is H  - 1.
%
ttE   (qsize In In).
sortE (qsize In In).
prodE (qsize In Out) (qsize In Mid) (qsize Mid Out) T.
decideE Kn (qsize In Out) (qsize In’ Out) K :- std.mem Kn K, In > 0, In’ is In - 1.
%
ttE   (A <c> B) :- ttE A,  ttE B.
sortE (A <c> B) :- sortE A, sortE B.
prodE (C1 <c> C2) (D1 <c> D2) (E1 <c> E2) T :- prodE C1 D1 E1 T, prodE C2 D2 E2 T.
decideE Kn (A <c> B) (C <c> D) K :- decideE Kn A C K, decideE Kn B D K.
```

**Figure 3** Sample FPCs

```
    check (qheight N) (go G Term),
    Ev = Term,
    coq.say "Proof:" {coq.term→ string Ev}.
...  (* Other clauses for different fpc omitted *)
}}.
```

The glue code between Coq-Elpi and the implementation of our calculus is straightforward: the goal consists of a quadruple of a (here inactive) context, an *evar*, a type (goal) and a list of extra information, also inactive. In addition, we supply the certificate: it consists of an integer (or two in the case of pairing) and a string to identify the “resource” FPC that we will use in this case. We just need to call check on the goal G, together with the certificate, in order to obtain a reconstructed proof term. We do not call the reconstruction directly on the evar because Coq-Elpi ensures that evars manipulated by λProlog are well-typed at all times; since we cannot guarantee that, as we work with partially reconstructed term, we get around this by an explicit unification.

The following example shows how we can use the above tactic to do FPC-driven logic programming modulo conversion in Coq and return a Coq proof-term:

```
Inductive insert (x:nat) : list nat → list nat → Prop :=
| i_n : insert x []  [x]
| i_s : ∀ y: nat, ∀ ys, x <= y → insert x (y :: ys) (x ::  y ::  ys)
| i_c : ∀ y: nat, ∀ ys rs, y <= x → insert x ys rs → insert x (y :: ys) (x :: rs).
Lemma i1: ∃ R, insert 2 ([0] ++ [1]) R.
elpi dprolog height 10.
Qed.
Print i1.
ex_intro (fun R : list nat ⇒ insert 2 ([0] ++ [1]) R) [0; 1;  2]
  (i_c 2 0 [1]  [1;  2]  (le_S 0 1 (le_S 0 0 (le_n 0)))
    (i_c 2 1 []  [2]  (le_S 1 1 (le_n 1)) (i_n 2)))
```

The dprolog tactic implements some of the features of Coq's eauto; it is programmable and as such not restricted to depth-first search, since it follows the dictates of the given FPC; for example we could easily add iterative-deepening search. Furthermore, FPCs can provide a *trace* that may be more customizable than the one offered by (e)auto's hard-wired Debug facility.

## 4 Revisiting property-based testing for Coq

We have presented in a previous paper [9] a proof-theoretical reconstruction of property-based testing (PBT) [26] of relational specifications, adopting techniques from foundational proof certificates to account for several features of this testing paradigm: from various *generation* strategies, to *shrinking* and fault localization.

Given the connection that Coq-Elpi offers between logic programming and the internals of Coq, it is natural to extend the FPC-driven logic programming interpreter of the previous section to perform PBT over **Inductive** types.

While nothing prevents us from porting all the PBT features that we have accounted for in [9], for the sake of this paper we will implement only FPC corresponding to different flavors of *exhaustive* generation, as adopted, e.g., in SmallCheck [60] and $\alpha$Check [14, 15], and their combination. Note however that it would take no more than two lines of code in the decideE expert to implement a form of random data generation in the sense of randomized backtracking [25].

Of course, Coq already features QuickChick [52] (https://softwarefoundations.cis.upenn.edu/qc-current), which is a sophisticated and well-supported PBT tool, based on a different perspective: being a clone of Haskell's QuickCheck, it emphasizes testing executable (read *decidable*) specifications with random generators. While current research [39] aims to increase automation, it is fair to say that testing with QuickChick, in particular relational specifications, is still very labor intensive. We do not intend to compete with QuickChick at this stage, but we shall see that we can test immediately **Inductive** definitions that corresponds to pure Horn programs, without having to provide a decidability proof for those definitions. Furthermore, we are not committed to a fixed random generation strategy, which, in general, requires additional work in the configuration of generators and shrinkers.

### 4.1 PBT as proof reconstruction

If we view a Horn property (in uncurried form) as a many-sorted logical formula

$$\forall x_1 : \tau_1 \ldots x_n : \tau_m, A_1 \wedge \cdots \wedge A_m \supset B \tag{*}$$

where the $A_i$ and $B$ are predicates defined using Horn clause specifications, a counter-example to this conjecture consists of a witness of the negated formula

$$\exists x_1 : \tau_1 \ldots x_n : \tau_n, A_1 \wedge \cdots \wedge A_m \wedge \neg B \tag{**}$$

In our Coq setting $A_i$ and $B$ will be propositions, while the $\tau_j$ are honest-to-goodness datatypes. We will treat the two quite differently, in so far as elements of those datatypes will be *generated*, while predicates will only be *checked.* This distinction is reminiscent of bi-directional type-checking and plays also a part in interpreting the negation sign. In a proof system for intuitionistic logic extended with fixed points [2], negation corresponds to the usual intuitionistic interpretation, which is what Coq supports. However, for the sake of PBT and as we argued in [9] , we can identify a proof certificate for (∗∗) simply with a proof certificate for $\exists x_1\colon \tau_1 \dots x_n\colon \tau_n, A_1 \wedge \cdots \wedge A_m$ and we can resort to negation-as-failure to check that the conclusion does not hold without caring for any evidence for the latter. This also means that we do not produce a Coq proof term for the refutation of our property — and neither does QuickChick, which runs at the OCaml level — although we can return the witness for the existential.

## 4.2 An Elpi tactic for PBT

We will invoke the tactic in a proof environment where the overall goal is the property that the system-under-test (SUT) should meet. This means that, after **intro** has been used to introduce the relevant hypotheses, the user specifies which variables of the environment should be used for generating data and which for executing the specification. In addition to this, the user should specify all the certificate information that will guide the data generation phase. Concretely, for the property (∗) and the specification of a FPC, the call to the tactic will be:

```
elpi dep_pbt <fpc> (A_1 ∧ ··· ∧ A_m) (x_1)...(x_n).
```

The tactic calls `check` with the given FPC on the dependent variables and delegates to a vanilla meta-interpreter (see Section A.1 in the appendix) the test of the hypotheses and of the negation of the goal:

```
interp (A_1 ∧ ··· ∧ A_m), not (interp B)
```

where `not` is λProlog's negation-as-failure operator.

To exemplify, let us add to the previous specification of list insertion a definition of ordered list:

```
Inductive ordered : list nat → Prop :=
| o_n : ordered []
| o_s : ∀ x : nat, ordered [x]
| o_c : ∀ (x y : nat), ∀ xs,  ordered xs → x <= y → ordered (x :: y :: xs).
```

A property we may wish to check before embarking on a formal proof is whether insertion preserves ordered-ness:

```
Conjecture ins_ord: ∀ (x : nat) xs rs, ordered xs → insert x xs rs → ordered rs.
intros x xs rs Ho Hi.
elpi dep_pbt (height 5) (Ho ∧ Hi) (x) (xs).
Abort.
```

In this query the tactic tests the hypotheses `Ho` and `Hi` against data `x,xs` generated exhaustively up to a height at most 5 from the library **Inductive** definitions of `nat` and `list`. We do not generate values for `rs`, since by (informal) mode information we know that it will be computed. Since we did slip in an error, our tactic reports a counter-example, namely `Proof Term: [0, [0; 1; 0]]`, which unpacks to `x = 0` and `xs = [0; 1; 0]`. As the latter is definitely not an ordered list, this points to a quite evident bug in the definition of `ordered`. We leave the fix to the reader.

In order to generate the PBT query, some pre-processing is needed. In particular, we turn variables inhabiting datatypes into λProlog logic variables when they appear inside a specification, and generate queries for each of these logic variables in association with the type of the data variable it corresponds to. In order to realize this pre-processing step, we leverage extensively λProlog's higher order programming features. The substitutions are handled with the technique of `copy` clauses [44].

Note that testing the above conjecture with QuickChick would have required much more setup: if we wished to proceed relationally as above, we would have had to provide a proof of decidability of the relevant notions. Were we to use functions, then we would have to implement a generator and shrinker for ordered lists, since automatic derivation of the former does not (yet) work for this kind of specification.

For a more significant case study, let us turn to the semantics of programming languages, where PBT has been used extensively and successfully [38]. Here we will consider a far simpler example, a typed arithmetic language featuring numerals with predecessor and test-for-zero, and Booleans with if-then-else, which comes from the *Software Foundations* book series (`https://softwarefoundations.cis.upenn.edu/plf-current/Types.html`). Whereas this example is admittedly quite simple-minded, it has, among others, been used as a benchmark for evaluating QuickChick's automation of generators under invariants [39], and to be amenable to the tool, the specification had to be massaged non-insignificantly.

```
Inductive tm : Type :=
 | ttrue : tm | tfalse : tm | tif : tm → tm → tm → tm | tzero : tm | tsucc : tm → tm
 | tpred : tm → tm | tiszero : tm → tm.
 Inductive typ : Type := | TBool : typ | TNat : typ.
```

The completely standard static and small step dynamic semantics rules are reported in appendix A.2.

While it is obvious that *subject expansion* fails for this calculus, it is gratifying to have it confirmed by our tactic, with counterexample `e = tif ttrue tzero ttrue`:

```
Conjecture subexp: ∀ e e' t, step e e' → has_type e' t → has_type e t.
 intros e e' t HS HT.
 elpi dep_pbt (height 2) (HS ∧ HT) (e).
 Abort.
```

Another way to asses the fault detection capability of a PBT setup is via *mutation analysis* [10], whereby localized bugs are purposely inserted, with the view that they should be caught ("killed") by a "good enough" testing suite. Following on an exercise in the aforementioned chapter of SF, we modify the typing relation by adding the following (nonsensical) clause:

```
Module M1.
Inductive has_type : tm → typ → Prop :=
...
  | T_SuccBool : ∀ t, has_type t TBool → has_type (tsucc t) TBool.
end M1.
```

Some of the desired properties for our SUT now fails: not only type uniqueness, but also progress with counterexample `e = tsucc ttrue`:

```
Definition progress (e : tm) (Has_type : tm → typ → Prop) (Step : tm → tm → Prop):=
     ∀ t, Has_type e t → notstuck e Step.
Conjecture progress_m1: ∀ e, progress e M1.has_type step.
 unfold progress.
 intros e t Ht.
 elpi dep_pbt (height 2) (Ht) (e).
 Abort.
```

To make the example slightly more interesting, we now move to an *intrinsically-typed* representation [4] of our object language, where by indexing terms with object types, we internalize the typing judgment into the syntax:

```
Inductive tm : typ → Type :=
| ttrue : tm TBool | tfalse : tm TBool | tzero : tm TNat | tsucc : tm TNat → tm TNat
| tpred : tm TNat→ tm TNat | tiszero : tm TNat→ tm TBool
| tif: ∀ (T : typ), tm TBool→ tm T → tm T → tm T.
```

Now, the operational semantics is by construction type-preserving, but bugs can still occur, see variations 3 in the same chapter that falsifies determinism of evaluation:

```
Module M3.
Inductive step : ∀ {T: typ}, tm T → tm T → Prop :=
...
  | ST_Funny2 : ∀ T t1 t2 t2' t3,          (*bug*)
  t2 ⟹ t2' → (tif T t1 t2 t3) ⟹ (tif T t1 t2' t3)
End M3.
 Goal ∀ (T : typ) (x y1 y2 : tm T ), M3.step x y1 → M3.step x y2 → y1 = y2.
 intros T x y1 y2 H1 H2.
 elpi dep_pbt pair 3 5 (H1 ∧ H2) (x).
 Abort.
 Counterexample:
 x = (tif TBool ttrue (tiszero tzero) ttrue
```

While we can deal with this encodings seamlessly, QuickChick's automatic derivation of generators is not applicable to dependent types, forcing us again either to provide decidability proofs for all judgments affected by the mutation or to embark in some non-trivial dependent functional programming, possibly based on monad transformers.

## 5 Elaboration of external proof certificates for the Coq kernel

The trusted base of Coq is its kernel, which is a type-checking program that certifies that a dependently typed $\lambda$-term has a given type. If type checking succeeds, the formula corresponding to that type is, in fact, accepted by Coq users as a theorem of intuitionistic logic (along with any axioms that have been asserted). The rest of the Coq system, especially its tactic language, is designed to help a human user build proofs-as-$\lambda$-terms that can be checked by the kernel.

There are many theorem provers for intuitionistic logic [58] for which a successful proof is not the kind of detailed $\lambda$-term required by the Coq kernel. Often, such provers provide no information about the proofs they discover. To the extent that some evidence is output after a successful run, such evidence is usually just a trace of some key aspects of a proof, where some details are often not included. For example:

1. Substitution instances of quantifiers might not be recorded in a proof since such instances can, in principle, be reconstructed using unification.
2. Detailed typing information might not need to be stored within a proof since types can often be reconstructed during proof checking [45].
3. Some simplifications steps might be applied within a proof without recording which rewrites were used. A simple non-deterministic proof-search engine might be expected to reconstruct an equivalent simplification.

A majority of the external and automatic theorem provers for intuitionistic logic do not involve induction. Instead, they go beyond Horn clause by permitting formulas with no restriction on occurrences of $\forall$ and $\supset$. In that case, we need to modify the focused proof

```
kind deb                          type.
type lambda                  deb → deb.
type apply         int → list deb → deb.
type idx                   int → index.
type lc              int → deb → cert.
type args       int → list deb → cert.
```

```
impC      (lc C (lambda D)) (lc C D).
impE      (args C (A:: As)) (lc C A) (args C As).
initialE  (args C []) .
decideE   (lc C (apply H A)) (args C A) (idx V) :- V is C - H - 1.
storeC    (lc C D) (x\ lc C' D) (x\ idx C) :- C' is C + 1.
```

**Figure 4** The FPC definition of De Bruijn notation as proof evidence.

rules that we have seen in Section 3.1 by adding the following rules.

$$\frac{\mathcal{P}, B \vdash D}{\mathcal{P} \vdash B \supset D} \supset R \qquad \frac{\mathcal{P} \vdash D[y/x]}{\mathcal{P} \vdash \forall x.D} \forall R$$

As usual, the $\forall R$ rule has the restriction that the eigenvariable $y$ is not free in its conclusion.

As has been detailed in earlier work on foundational proof certificates, this richer notion of proof system can provide for richer proof certificates. The main differences with what we have seen before is that the left-hand sides of sequents can now grow during the proof checking process. When reading the right introduction rule for $\supset$ from conclusion to premise, we shall say that the antecedent of the implication is *stored* in the left side of the context. When this store action occurs, an *index* is used by the store command to name that new, left-hand formula occurrence. In this extended situation, the decide expert uses the index of an assumption in order to enter a focus phase of inference. A full proof checking kernel for first-order intuitionistic logic has been given in [16] so we do not reproduce it here.

To give an example, consider using untyped $\lambda$-terms encoded using De Bruijn's notation as proof certificates for propositional intuitionistic logic over just $\supset$. The fact that such terms can be used as proof certificates for such formulas (denoting simple types) can be formally defined using the FPC description in Figure 4. Using the constants provided in that figure, the untyped $\lambda$-term $\lambda x(x(\lambda y(y(\lambda z(x(\lambda u\, z))))))$ can be encoded as the following Elpi term of type `deb`.

```
(lambda (apply 0 [lambda (apply 0 [lambda (apply 2 [lambda (apply 1 [])])])]) )
```

Using the constructor `lc` and `args`, terms in De Bruijn syntax (terms of type `deb`) are incorporated into proof certificates (terms of type `cert`) along with other integer arguments that are needed to compute offsets to address bound variables.

We describe here briefly how to use the technology behind FPCs and logic programming in order to provide a flexible approach to connecting external provers of first-order intuitionistic logic to Coq. Following the general outline that has been described in [6, 7], we assume that the following steps are taken.

1. Modify an external prover to output some form of proof evidence (proof certificate) for formulas it claims are theorems.
2. Develop a formal definition of the semantics behind such proof certificates using the FPC framework. The FPC for De Bruijn expressions given in Figure 4 is an example of this step.

3. Check proofs by executing the logic programming checker that is parameterized by the particular FPC definition.

As we have mentioned in Section 3.2.2, the logic programming setting allows parallel checking and synthesizing of a pair of certificates. That is, during the checking of one certificate, it is possible to synthesize, for example, a fully detailed term that is appropriate for handing to the Coq kernel. If one is interested only in building Coq kernel proof structures, we can bypass the use of an explicit pairing operation and build the synthesis of such proof structures directly into the FPC proof checker. We took exactly this step in Figure 1 where proof checking involved both proof certificates as well as Coq terms. If one is interested in checking only one kind of external proof structure, then the FPC for that structure could also be built into the checker (via, say, partial evaluation of logic programs [43]).

Continuing with the previous example, consider an external theorem prover for propositional intuitionistic logic which returns proof structures as untyped $\lambda$-terms using De Bruijn's notation. Using the FPC provided in Figure 4 and the proof certificate checker in the file `ljf-dep.mod`, of the repository `https://github.com/proofcert/fpc-elpi`, the De Bruijn term displayed above can be elaborated into a proper proof for the following Coq theorem.

```
Theorem dneg_peirce_mid : ∀ P Q: Prop, (((((P → Q) → P) → P) → Q) → Q).
```

We note that this proof certificate checker and Coq proof synthesizer is rather compact, comprising less than 90 lines of Elpi code.

## 6 Conclusion and future work

This paper follows a line of research starting in the late 1980s and gaining more steam in the last five years, which advocates the usefulness of proof theory and higher-order logic programming for the many tasks concerning the development, enrichment, and even formal verification of proof assistants. The development of the Coq-Elpi plug-in has made this connection tighter.

We have presented two applications of this synergy: one supporting an out-of-the-box way to do property-based testing for inductive relations; the other geared towards providing a flexible approach to connecting external provers of first-order intuitionistic logic to Coq

The code reported in Fig. 1 is a simplification for exposition purposes of the real implementation of the kernel. Following ideas from bidirectional type checking, we have factored out the product left rule in $\forall - L$ and $\supset -L$, where the former delegates to Coq the *check* that the instantiation term $t$ is well-typed w.r.t. $B$, while in the latter, proof search will *generate* such a term, given the type $B$. There are also other minor tweaks, such as a rule performing weak-head reduction, allowing us to handle directly existential goals.

There are many avenues of development for this line of research. We would like to exploit one of the distinguishing features of Elpi: the *delay* mechanism. The use of constraints for data generation is well developed [27] and we could try to leverage it to improve our PBT tactic to generate partially instantiated terms, without recurring to needed narrowing as in LazySmallCheck [60]. On the more practical side, it would be worthwhile to investigate *random* generation, following the ideas in [25, 9].

Finally, it makes sense to tie together the two threads of this paper and provide a way of checking and elaborating proof evidence for intuitionistic logic *over* (inductively) defined atoms using previously proved lemmata, that is capturing most of the features of `eauto`. This can be pushed further up to FPC for (co)inductive proofs [8].

The source files mentioned in this paper are available at `https://github.com/proofcert/fpc-elpi`.

## References

1 Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992. doi:10.1093/logcom/2.3.297.

2 David Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1):2:1–2:44, April 2012. URL: http://tocl.acm.org/accepted/427baelde.pdf, doi:10.1145/2071368.2071370.

3 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. doi:10.6092/issn.1972-5787/4650.

4 Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in Coq. *J. Autom. Reason.*, 49(2):141–159, 2012. doi:10.1007/s10817-011-9219-0.

5 G. Bierman. *On Intuitionistic Linear Logic.* PhD thesis, Computing Laboratory, University of Cambridge, 1994.

6 Roberto Blanco, Zakaria Chihani, and Dale Miller. Translating between implicit and explicit versions of proof. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 255–273. Springer, 2017. doi:10.1007/978-3-319-63046-5_16.

7 Roberto Blanco, Matteo Manighetti, and Dale Miller. FPC-Coq: Using Elpi to elaborate external proof evidence into Coq proofs. Technical Report hal-02974002, Inria, July 2020. Presented at the Coq Workshop 2020. URL: https://hal.inria.fr/hal-02974002.

8 Roberto Blanco and Dale Miller. Proof outlines as proof certificates: a system description. In Iliano Cervesato and Carsten Schürmann, editors, *Proceedings First International Workshop on Focusing*, volume 197 of *Electronic Proceedings in Theoretical Computer Science*, pages 7–14. Open Publishing Association, November 2015. doi:10.4204/EPTCS.197.2.

9 Roberto Blanco, Dale Miller, and Alberto Momigliano. Property-based testing via proof reconstruction. In E. Komendantskaya, editor, *Principles and Practice of Programming Languages 2019 (PPDP '19)*, October 2019. doi:10.1145/3354166.3354170.

10 Matteo Cavada, Andrea Colò, and Alberto Momigliano. MutantChick: Type-preserving mutation analysis for Coq. In *CILC*, volume 2710 of *CEUR Workshop Proceedings*, pages 105–112. CEUR-WS.org, 2020. URL: http://ceur-ws.org/Vol-2710/short2.pdf.

11 Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003. doi:10.1093/logcom/13.5.639.

12 Kaustuv Chaudhuri, Stefan Hetzl, and Dale Miller. A multi-focused proof system isomorphic to expansion proofs. *J. of Logic and Computation*, 26(2):577–603, 2016. URL: http://hal.inria.fr/hal-00937056, doi:10.1093/logcom/exu030.

13 Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In G. Ausiello, J. Karhumäki, G. Mauri, and L. Ong, editors, *Fifth International Conference on Theoretical Computer Science*, volume 273 of *IFIP*, pages 383–396. Springer, September 2008. doi:10.1007/978-0-387-09680-3_26.

14 James Cheney and Alberto Momigliano. $\alpha$Check: A mechanized metatheory model checker. *Theory and Practice of Logic Programming*, 17(3):311–352, 2017. doi:10.1017/S1471068417000035.

15 James Cheney, Alberto Momigliano, and Matteo Pessina. Advances in property-based testing for \alpha prolog. In *TAP@STAF*, volume 9762 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2016. doi:10.1007/978-3-319-41135-4_3.

16 Zakaria Chihani, Dale Miller, and Fabien Renaud. A semantic framework for proof evidence. *J. of Automated Reasoning*, 59(3):287–330, 2017. doi:10.1007/s10817-016-9380-6.

17 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008. doi:10.1145/1411204.1411226.

18 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: Algebraic hierarchies made easy in coq with elpi (system description). In *FSCD*, volume 167 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl, 2020. doi:10.4230/LIPIcs.FSCD.2020.34.

19 Joëlle Despeyroux, Amy Felty, and Andre Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138. Springer, April 1995. doi:10.1007/BFb0014049.

20 Cvetan Dunchev, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing HOL in an higher order logic programming language. In *LFMTP*, pages 4:1–4:10. ACM, 2016. doi:10.1145/2966268.2966272.

21 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λprolog interpreter. In *LPAR*, volume 9450 of *Lecture Notes in Computer Science*, pages 460–468. Springer, 2015. URL: https://doi.org/10.1007/978-3-662-48899-7_32, doi:10.1007/978-3-662-48899-7_32.

22 Amy Felty. Encoding the calculus of constructions in a higher-order logic. In M. Vardi, editor, *8th Symp. on Logic in Computer Science*, pages 233–244. IEEE, June 1993. doi:10.1109/LICS.1993.287584.

23 Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993. doi:10.1007/BF00881900.

24 Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, number 310 in Lecture Notes in Computer Science, pages 61–80, Argonne, IL, May 1988. Springer. doi:10.1007/BFb0012823.

25 Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In *ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 383–405. Springer, 2015. doi:10.1007/978-3-662-46669-8_16.

26 George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, pages 74–80, July 1997. doi:10.1145/263244.263267.

27 Fabio Fioravanti, Maurizio Proietti, and Valerio Senni. Efficient generation of test data structures using constraint logic programming and program transformation. *J. Log. Comput.*, 25(6):1263–1283, 2015. doi:10.1093/logcom/ext071.

28 Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012. URL: http://arxiv.org/abs/0911.2993, doi:10.1007/s10817-011-9218-1.

29 Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935. doi:10.1007/BF01201353.

30 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

31 Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. doi:10.1007/3-540-09724-4.

32 Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing type theory in higher order constraint logic programming. *Mathematical Structures in Computer Science*, 29(8):1125–1150, 2019. doi:10.1017/S0960129518000427.

33 John Hannan. Extended natural semantics. *J. of Functional Programming*, 3(2):123–152, April 1993. doi:10.1017/S0956796800000666.

34 Hugo Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *CSL*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1994. doi:10.1007/BFb0022247.

35 Furio Honsell, Marino Miculan, and Ivan Scagnetto. $\pi$-calculus in (co)inductive type theories. *Theoretical Computer Science*, 2(253):239–285, 2001. doi:10.1016/S0304-3975(00)00095-5.

36 Martin Hyland and Valeria de Paiva. Full intuitionistic linear logic (extended abstract). *Annals of Pure and Applied Logic*, 64(3):273–291, 11 November 1993. doi:10.1016/0168-0072(93)90146-5.

37 Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, March 1987. doi:10.1007/BFb0039592.

38 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM. doi:10.1145/2103656.2103691.

39 Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL):45:1–45:30, 2018. doi:10.1145/3158133.

40 Stéphane Lengrand. *Normalisation & Equivalence in Proof Theory & Type Theory*. PhD thesis, University of St. Andrews, dec 2006. URL: https://tel.archives-ouvertes.fr/tel-00134646.

41 Stéphane Lengrand, Roy Dyckhoff, and James McKinna. A sequent calculus for type theory. In *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2006. doi:10.1007/11874683_29.

42 Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009. doi:10.1016/j.tcs.2009.07.041.

43 J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3):217 – 242, 1991. URL: http://www.sciencedirect.com/science/article/pii/074310669190027M, doi:https://doi.org/10.1016/0743-1066(91)90027-M.

44 Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991. doi:10.1093/logcom/1.4.497.

45 Dale Miller. Proof checking and logic programming. *Formal Aspects of Computing*, 29(3):383–399, 2017. URL: http://dx.doi.org/10.1007/s00165-016-0393-z, doi:10.1007/s00165-016-0393-z.

46 Dale Miller. Mechanized metatheory revisited. *Journal of Automated Reasoning*, 63(3):625–665, October 2019. doi:10.1007/s10817-018-9483-3.

47 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012. doi:10.1017/CBO9781139021326.

48 Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991. doi:10.1016/0168-0072(91)90068-W.

49 Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

50 Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990. doi:10.1145/96559.96570.

51 Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of $\lambda$Prolog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 287–291, Trento, 1999. Springer. doi:10.1007/3-540-48660-7_25.

52 Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In Christian Urban and Xingyuan Zhang, editors, *ITP 2015*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015. URL: `https://doi.org/10.1007/978-3-319-22102-1_22`, `doi:10.1007/978-3-319-22102-1_22`.

53 Lawrence C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, 1989. `doi:10.1007/BF00248324`.

54 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988. `doi:10.1145/53990.54010`.

55 Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, number 6173 in Lecture Notes in Computer Science, pages 15–21, 2010. URL: `https://doi.org/10.1007/978-3-642-14203-1_2`, `doi:10.1007/978-3-642-14203-1_2`.

56 Elaine Pimentel, Vivek Nigam, and Jo ao Neto. Multi-focused proofs with different polarity assignments. In Mario Benevides and Rene Thiemann, editors, *Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015)*, volume 323 of *Electronic Notes in Theoretical Computer Science*, pages 163–179, July 2016. `doi:10.1016/j.entcs.2016.06.011`.

57 Gordon D. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

58 Thomas Raths, Jens Otten, and Christoph Kreitz. The ILTP problem library for intuitionistic logic. *Journal of Automated Reasoning*, 38(1):261–271, 2007. `doi:10.1007/s10817-006-9060-z`.

59 C. Röckl, D. Hirschkoff, and S. Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts. In F. Honsell and M. Miculan, editors, *Proc. FOSSACS'01*, volume 2030 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2001. URL: `https://doi.org/10.1007/3-540-45315-6_24`, `doi:10.1007/3-540-45315-6_24`.

60 Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In Andy Gill, editor, *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 37–48. ACM, 2008. `doi:10.1145/1411286.1411292`.

61 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The metacoq project. *J. Autom. Reason.*, 64(5):947–999, 2020. `doi:10.1007/s10817-019-09540-0`.

62 Enrico Tassi. Elpi: an extension language for Coq. CoqPL 2018: The Fourth International Workshop on Coq for Programming Languages, January 2018. URL: `https://hal.inria.fr/hal-01637063/`.

63 Enrico Tassi. Deriving proved equality tests in Coq-Elpi: Stronger induction principles for containers in Coq. In *ITP*, volume 141 of *LIPIcs*, pages 29:1–29:18. Schloss Dagstuhl, 2019. URL: `http://drops.dagstuhl.de/opus/volltexte/2019/11084`, `doi:10.4230/LIPIcs.ITP.2019.29`.

# A Appendix

In this appendix we list some definitions and pieces of code that we have mentioned in the main paper.

## A.1 The vanilla meta-interpreter

We report below the encoding of the vanilla meta-interpreter used in the testing phase of the `dep_pbt` tactic. Differently from Fig. 2 we appeal, via quotations, to Coq's defined connectives. An `atomic` proposition is one defined **Inductive**ly.

```
type   interp    term → o.
type   backchain   term → term → o.
```

```
interp {{True}}.
interp (sort _).
interp {{lp:G1 ∧ lp:G2}} :- interp G1, interp G2.
interp {{lp:G1 ∨ lp:G2}} :- interp G1; interp G2.
interp {{lp:T1 = lp:T2}} :- coq.unify-eq T1 T2 ok.
interp {{ex (lp:G)}} :- interp (G X).
interp Atom :-
  atomic Atom,
  coq.safe-dest-app Atom (global (indt Prog)) _,
  coq.env.indt Prog _ _ _ _ _ KTypes,
  std.mem KTypes D, backchain D Atom.
backchain A A' :- atomic A,  coq.unify-eq A A' ok.
backchain D A :- is_imp D A D', !,  backchain D' A, interp Ty.
backchain D A :- is_uni D D',  backchain (D' X) A.
```

## A.2 Semantics of the typed arithmetic language

We list the rules for static and dynamic semantics of the language mentioned in Section 4.2 and related notions:

```
Inductive has_type : tm → typ → Prop :=
    | T_Tru : has_type ttrue TBool
    | T_Fls : has_type tfalse TBool
     | T_Test : ∀ t1 t2 t3 T,
         has_type t1 TBool → has_type t2 T → has_type t3 T → has_type (tif t1 t2 t3) T
    | T_Zro : has_type tzero TNat
    | T_Scc : ∀ t1,  has_type t1 TNat → has_type (tsucc t1) TNat
    | T_Prd : ∀ t1,  has_type t1 TNat → has_type (tpred t1 ) TNat
    | T_Iszro : ∀ t1,  has_type t1 TNat → has_type (tiszero t1) TBool.
```

```
Inductive nvalue : tm → Prop :=
  | nv_zero : nvalue tzero
  | nv_succ : ∀ t, nvalue t → nvalue (tsucc t).
Inductive bvalue : tm → Prop :=
  | bv_t : bvalue ttrue
  | bv_f : bvalue tfalse.
Reserved Notation "t1 '⟹ ' t2" (at level 40).
Inductive step : tm → tm → Prop :=
    | ST_IfTrue : ∀ t1 t2,  (tif ttrue t1 t2) ⟹ t1
    | ST_IfFalse : ∀ t1 t2, (tif tfalse t1 t2) ⟹ t2
    | ST_If : ∀ t1 t1' t2 t3,
        t1 ⟹ t1' → (tif t1 t2 t3) ⟹ (tif t1' t2 t3)
    | ST_Succ : ∀ t1 t1',
        t1 ⟹ t1' → (tsucc t1) ⟹ (tsucc t1')
    | ST_PredZero : (tpred tzero) ⟹ tzero
    | ST_PredSucc : ∀ t1,
        nvalue t1 → (tpred (tsucc t1)) ⟹ t1
    | ST_Pred : ∀ t1 t1',
        t1 ⟹ t1' → (tpred t1) ⟹ (tpred t1')
    | ST_IszeroZero : (tiszero tzero) ⟹ ttrue
    | ST_IszeroSucc : ∀ t1,
         nvalue t1 → (tiszero (tsucc t1)) ⟹ tfalse
    | ST_Iszero : ∀ t1 t1',
        t1 ⟹ t1' → (tiszero t1) ⟹ (tiszero t1')
  where "t1 '⟹ ' t2" := (step t1 t2).
Inductive notstuck (e : tm) (Step : tm → tm → Prop) : Prop :=
   | pn : nvalue e  → notstuck e Step
   | pb : bvalue e → notstuck e Step
   | ps e' : Step e e' → notstuck e Step.
```