Property-Based Testing via Proof Reconstruction

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

Dale Miller
INRIA Saclay & LIX, École Polytechnique, France
dale.miller@inria.fr

Alberto Momigliano
DI, Università degli Studi di Milano, Italy
momigliano@di.unimi.it

ABSTRACT
Property-based testing (PBT) is a technique for validating code against an executable specification by automatically generating test-data. We present a proof-theoretical reconstruction of this style of testing for relational specifications and employ the Foundational Proof Certificate framework to describe test generators. We do this by presenting certain kinds of “proof outlines” that can be used to describe various common generation strategies in the PBT literature, ranging from random to exhaustive, including their combination. We also address the shrinking of counterexamples as a first step towards their explanation. Once generation is accomplished, the testing phase boils down to a standard logic programming search. After illustrating our techniques on simple, first-order (algebraic) data structures, we lift it to data structures containing bindings using \( \lambda \)-tree syntax. The \( \lambda \)Prolog programming language is capable of performing both the generation and checking of tests. We validate this approach by tackling benchmarks in the meta-theory of programming languages coming from related tools such as PLT-Redex.

CCS CONCEPTS
• Software and its engineering → Formal software verification;
• Theory of computation → Logic and verification; Proof theory.

1 INTRODUCTION
Property-based testing (PBT) is a technique for validating code that successfully combines two old ideas: automatic test data generation trying to refute executable specifications. Pioneered by QuickCheck for functional programming [16], PBT tools are now available for most programming languages and are having a growing impact in industry [31]. This idea now spread to several proof assistants [8, 49] to complement (interactive) theorem proving with a preliminary phase of conjecture testing. The collaboration of PBT with proof assistants is so accomplished that PBT is now a part of the Software Foundations’s curriculum (https://softwarefoundations.cis.upenn.edu/qc-current).

This tumultuous rate of growth is characterized, in our opinion, by a lack of common (logical) foundation. For one, PBT comes in different flavors as far as data generation is concerned: while random generation is the most common one, other tools employ exhaustive generation ([12, 54]) or a combination thereof ([19]). At the same time, one could say that PBT is rediscovering logic and, in particular, logic programming: to begin with, QuickCheck’s DSL is basically Horn clause logic; LazySmallCheck [54] has adopted narrowing to permit less redundant testing over partial rather than ground terms; PLT-Redex [20] contains a re-implementation of constraint logic programming in order to better generate well-typed \( \lambda \)-terms [22]. Finally, PBT in Isabelle/HOL features now a notion of smart test generators [11] and this is achieved by turning the functional code into logic programs and inferring through mode analysis their data-flow behavior. We refer to the Related Work (Section 5) for more examples of this phenomenon.

In this paper we give a uniform logical reconstruction of the most relevant aspects of PBT under the guidance of proof-theory. In particular, we will adopt ideas from the theory of foundational proof certificates (FPC [14]). In the fully general setting, FPCs define a range of proof structures used in various theorem provers (e.g., resolution refutations, Herbrand disjuncts, tableaux, etc). A key feature of this approach to proof certificates is that certificates do not need to contain all the details required to complete a formal proof. In those cases, a proof checker (for example, the specification of the check predicate Figure 5) would need to perform proof reconstruction in order to successfully check a certificate. Such proof reconstruction is generally aided by exploiting the logic programming paradigm, since unification and backtracking search aid greatly in the reconstruction of missing proof details. As we shall see in Section 2, FPCs can be used as proof outlines [10] by describing some of the general shape of a proof: checking such outlines essentially results in an attempt to fill in the missing details.

With small localized changes in the specification of relevant FPCs, we are able to account for both exhaustive and random generation. Then, we proceed to give a simple description of shrinking: this is an indispensable ingredient in the random generation approach, whereby counterexamples are minimized so as to be more easily understood by the user. On the flip side, this exercise confirms the versatility of PFC to address not only proof systems, but also refutations and counter-models, as already suggested in [29]. We give a prototype implementation of our framework in \( \lambda \)Prolog [41]. While we see it as a proof-of-concept, in so far as we do not pay any attention to efficiency, the implementation is, we argue, of some interest in several regards: as we detail in Section 4.1, our proof-theoretic stance allows us to lift our analysis to meta-programming, in particular to model-checking meta-theory [12], where PBT has
been used extensively, but with some difficulties [33]. The trouble is dealing with binding signatures, by which we mean the encoding of language constructs sensitive to naming and scooping. Here λProlog’s support for λ-tree syntax shines, allowing us to be competitive with specialized tools such as αCheck [12]. Moreover, a hot topic in the functional PBT literature is (random) data generation under invariants [15], for example, generating well-typed λ-terms or complete runs of abstract machines [30]. The same issue appears when we consider the shrinking of counterexamples since the generating and testing of arbitrary subterms can be time-consuming. The built-in features available in higher-order logic programming can have many advantages over user supplied encodings of those features.

The rest of the paper is organized as follows: in the next Section we give a gentle introduction to FPC and their use for data generation. Section 3 ties this up with PBT and showcases how we deal with its many flavors. In Section 4 we lift our approach to meta-programming and report some initial experimental results. After a review of related work (Section 5), we conclude with a list of future endeavors.

2 PBT AS PROOF RECONSTRUCTION

We now present a notion of proof certificate that can be used both as a presentation of a counterexample (to a conjectured theorem) as well as a description of the space in which to look for such counterexamples. The logic programs we use here (except for a small departure in Section 4 when we need to deal with bindings) are Horn clause programs: the logical foundations of the Prolog programming language [3]. The logic in which statements are made about what Prolog programs can prove or cannot prove is, however, a much richer logic (more akin to arithmetic since induction is needed to prove such properties more generally). We first argue, however, that for the simple task of generating and testing of counterexamples, the full nature of that strong logic is not necessary. Once that argument is made, the structure of proofs using Horn clauses are identified: it is on that structure that we attach the notion of foundational proof certificates.

2.1 Generate-and-test as a proof-search strategy

Imagine that we wish to write a relational specification for reversing lists. There are, of course, many ways to write such a specification but in every case, the formula

\[ \forall x: \tau \ [ P(x) \supset Q(x)] \]

where both \( P \) and \( Q \) are relations (predicates) of a single argument (it is an easy matter to deal with more than one argument or more complex antecedents, as well). Occasionally, it can be important in this setting to move the type judgment \( x: \tau \) into the logic by turning the type into a predicate: \( \forall x: \tau [ (\tau(x) \land P(x)) \supset Q(x)] \). Proving such formulas can often be difficult since their proof may involve the clever invention of prior lemmas and induction invariants. In many practical settings, such formulas are, in fact, not theorems since the relational specifications in \( P \) and/or \( Q \) can contain errors. It can be valuable, therefore, to first attempt to find counterexamples to such formulas prior to pursuing a proof. That is, we might try to prove formulas of the form \( \exists x [ (\tau(x) \land P(x)) \land \neg Q(x)] \) instead. If a term \( t \) of type \( \tau \) can be discovered such that \( P(t) \) holds while \( Q(t) \) does not, then one can return to the specifications in \( P \) and \( Q \) and revise them using the concrete evidence in \( t \) as a witness of how the specifications are wrong. The process of writing and revising relational specifications would go smoother if such counterexamples are discovered quickly and automatically.

Note that in order to speak of generate-and-test as a strategy for finding a proof (hence, a counterexample) for \( \exists x [ (\tau(x) \land P(x)) \land \neg Q(x)] \), we need to pick a logic in which (finite) failure is a means to actually prove a negation (such as \( \neg Q(x) \)). We survey two well-known such schemata for capturing the provability of atoms and the negation of atoms in which the meaning of those atoms is defined as Horn clauses.

2.2 Encoding logic programs into logic and proof theory

Dating back to early foundational papers on logic programming, the literature contains at least two ways to view Horn clause-style relational specifications. Following, say, Apt & van Emdem [3], the Prolog clauses displayed in Figure 1, are encoded directly as first-order Horn clauses. For example, one of the Prolog clauses in Figure 1 is the universal closure of \( \text{app} \ Xs \ Ys \ Zs \supset \text{app} \ (\text{cns} \ X \ Xs) \ Ys \ (\text{cns} \ X \ Zs) \). While that encoding of Prolog clauses into logic captured correctly the provability of atomic formulas, that approach was not able to explain negation-as-finite failure.

Following Clark [17], Prolog clauses could be viewed as containing exactly one clause per predicate, as displayed in Figure 2. Within standard first-order (classical or intuitionistic) logics, these two forms of representation are logically equivalent. The reason for writing several clauses as one clause is to consider such clauses as a logical equivalence; viewing “\( \vdash \)” as an equivalence can support the building of proofs of negated atomic formulas. To complete that picture, Clark’s completion needs the addition of some new axioms to describe both equality and inequality: in such an extended logic, negations (hopefully corresponding to negations-as-finite-failure) can be given actual proofs (in contrast to “failures to find proofs”).

These two perspectives of Prolog programs have also been echoed in the proof theoretic analysis of logic programs. The foundations of the λProlog programming language [41] views the execution of logic programs as the search for sequent calculus proofs [42]. In that development, the presentation of Prolog execution as SLD-resolution (described in, for example, [3]) was replaced by using proof-search in the sequent calculus. A second approach to the proof theory of Horn clauses (one that corresponds to the Clark completion approach [56]) involves encoding clauses as fixed points. For example, the Prolog-style specifications in Figure 2 can be written instead as the fixed point expressions in Figure 3. Using inference rules for equality due to Schroeder-Heister and Girard [28, 57] and for the treatment of inductively defined predicates [5, 36], much of
The proof search approach to encoding Horn clause computation

\[ \mu \lambda \nu \Lambda \\sigma \\\n\]

and the least fixed operator

\[ \text{Figure 3: Specifications as (least) fixed point expressions. The specifications in Figure 2 are written using standard logic notation, and for lists are } \text{nl (for the empty list) and cns (for non-empty lists).} \]

\[
\begin{align*}
nat \ z. & \quad \text{lst } \text{nl.} \\
nat \ (s \ N) & :- \ nat \ N. & \quad \text{lst} \ (\text{cns } N \ Ns) & :- \ nat \ N, \ lst \ Ns. \\
\text{app} \ \text{nl} \ Xs \ Xs. & \quad \text{app} \ (\text{cns } X \ Xs) \ Ys \ (\text{cns } X \ Zs) & :- \ \text{app} \ Xs \ Ys \ Zs. \\
\end{align*}
\]

\[\text{Figure 1: Specifications listing more than one Horn clause per predicate. The constructors for natural numbers are } z \text{ (zero) and } s \text{ (successor), and for lists are } \text{nl (for the empty list) and cns (for non-empty lists).} \]

\[
\begin{align*}
nat \ X & :- \ X = z ; \ \sigma \ X \ X' \ X = (s \ X'), \ nat \ X'. \\
lst \ Ns & :- \ Ns = \text{nl} ; \ \sigma \ Ns \ Ns' \ Ns = (\text{cns } N \ Ns'), \ nat \ N, \ lst \ Ns'. \\
\text{app} \ Xs \ Ys \ Zs & :- \ Xs = \text{nl}, \ Ys = Zs ; \\
\ & \quad \sigma \ Xs \ Xs' \ Xs' \ Xs = (\text{cns } X \ Xs'), \ Zs = (\text{cns } X \ Zs'), \ \text{app} \ Xs' \ Ys \ Zs'. \\
\end{align*}
\]

\[\text{Figure 2: Specifications listing one Horn clause per predicate. Following \text{\&}Prolog syntax, the expression } \sigma \ X \text{ denotes the existential quantifier over the variable } X \text{ and the semicolon and comma denote as usual disjunction and conjunction, respectively.} \]

\[
\begin{align*}
nat & = \mu \lambda \nu \lambda \Lambda \ \sigma \ (n = z \lor \exists n' (n = (s \ n') \land N n')) \\
lst & = \mu \lambda \Lambda \lambda (l = \text{nl} \lor \exists n' (l = (\text{cns } n I') \land nat \ n \land l')) \\
\text{app} & = \mu \lambda \lambda \lambda \lambda \lambda \lambda (xs \ys zs = \text{nl} \land ys = zs) \lor \exists x' \exists x' \exists zs' (xs = (\text{cns } x' \ x') \land zs = (\text{cns } x' \ zs') \land A x' \ ys \ zs') \\
\end{align*}
\]

\[\text{Figure 3: Specifications as (least) fixed point expressions. The specifications in Figure 2 are written using standard logic notation and the least fixed operator } \mu \text{ over higher-order abstractions written using } \lambda\text{-abstractions.} \]

model checking and Horn-clause based logic programming can be captured using sequent calculus [29, 38].

### 2.3 Focused proof systems

The proof search approach to encoding Horn clause computation results in the structuring of proofs with repeated switching between a goal-reduction phase and a backchaining phase [42]. The notion of focused proof systems [2, 35] generalizes this view of proof construction by identifying the following two phases.

1. The negative\(^2\) phase corresponds to goal-reduction: in this phase, inference rules that involve don’t-care-nondeterminism are applied. As a result, there is no need to consider backtracking over choices made in building this phase.

2. The positive phase corresponding to backchaining: in this phase, inference rules that involve don’t-know-nondeterminism are applied: here, inference rules need to be supplied with information in order to ensure that a completed proof can be built. That information can be items such as which term is needed to instantiate a universally quantified formula and which disjunct of a disjunctive goal formula should be proved.

Thus, when building a proof tree (in the sequent calculus) from the conclusion to its leaves, the negative phase corresponds to a simple computation that needs no external information, while the positive phase may need such external information to be supplied. In the literature, it is common to refer to a repository of such external information as either an oracle or a proof certificate.

When using a focused proof system for logic extended with fixed points, such as employed in Bedwyr [7] and described in [5, 29], proofs of formulas such as

\[
\exists x (\tau(x) \land P(x)) \land \lnot Q(x)
\]

are a single bipole: that is, when reading a proof bottom up, a positive phase is followed on all its premises by a single negative phase that completes the proof.\(^3\) In particular, the positive phase corresponds to the generation phase and the negative phase corresponds to the testing phase.

Instead of giving a full focused proof system of a logic including fixed points (since, as we will argue, that proof system will not, in fact, be needed to account for PBT), we offer the following analogy. Suppose that we are given a finite search tree and we are asked to prove that there is a secret located in one of the nodes of that tree. A proof that we have found that secret can be taken to be a description of the path to that node from the root of the tree: that path can be seen as the proof certificate for that claim. On the other hand, a proof that no node contains the secret is a rather different thing: here, one expects to use a blind and exhaustive search (via, say, depth-first or breath-first search) and that the result of that search

\(^2\)The terminology of negative and positive phases is a bit unfortunate: historically, these terms do not refer to positive or negative subformula occurrences but rather to certain semantic models used in the study of linear logic [27].

\(^3\)The reader familiar with focusing will understand that there are two “polarized” conjunctions, written in linear logic as \& and \&\& or in classical and intuitionalistic logics as \&\& and \&\&, respectively. In this paper, we use simply \& to denote the positive biased conjunction.
never discovers the secret. A proof of this fact requires no external information: instead it requires a lot of computation involved with exploring the tree until exhaustion. This follows the familiar pattern where the positive (generate) phase requires external information while the negative (testing) phase requires none. Thus, a proof certificate for (+) is also a proof certificate for 
\[ \exists \tau \left[ \tau(x) \land P(x) \right]. \] 

Such a certificate would contain the witness (closed) term \( t \) for the existential quantifier and sufficient information to confirm that \( P(t) \) can be proved (a proof of a typing judgment such as \( \tau(t) \) is usually trivial). Since a proof certificate for the existence-of-a-counterexample formula (+) can be taken as a proof certificate of (++) then we only need to consider proof certificates for Horn clause programs. We illustrate next what such proofs and proof certificates look like for the rather simple logic of Horn clauses.

### 2.4 Certificate checking with expert predicates

Figure 4 contains a simple proof system for Horn clause provability in which each rule is augmented with an expert predicate, a certificate \( \Xi \), and possibly continuations of certificates \( \Xi', \Xi_1, \Xi_2 \) with extracted information from certificates (in the case of \( \lor \) and \( \land \)). The premise \((A \vdash G) \in \text{grnd}(\mathcal{P})\) in the last inference rule of Figure 4 states the logic programming clause \((A \vdash G)\) is a ground instance of some clause in a fixed program \( \mathcal{P} \). Although this proof system is a focused proof system, the richness of focusing is not apparent in this simplified setting; thus, we drop the adjective “focused” from this point forward.

Figure 5 contains the λProlog implementation of the inference rules in Figure 4: here the infix turnstile \( \vdash \) symbol is replaced by the check predicate. Notice that it is easy to show that no matter how the expert predicates are defined, if the goal check Cert B is provable in λProlog then B must be a sound consequence of the program clauses stored in the prog predicate (which provides a natural implementation of the premise \((A \vdash G) \in \text{grnd}(\mathcal{P})\)).

As we mentioned in the introduction, the notion of proof certificates used here is taken from the general setting of foundational proof certificates [14]. In our case here, an FPC is a collection of λProlog clauses that provide the remaining details not supplied in Figure 5: that is, the exact set of constructors for the Cert type as well as the exact specification of the six expert predicates listed in that figure. Figure 6 displays two such FPCs, both of which can be used to describe proofs for which we bound the number of occurrences of unfoldings in a proof. For example, the first FPC provides the experts for treating certificates that are constructed using the qheight constructor. As is easy to verify, the query \((\text{check } (\text{qheight 5) B})\) (for the encoding B of a goal formula) is provable in λProlog using the clauses in Figures 5 and 6 if and only if the height of that proof is 5 or less. Similarly, the second FPC uses the constructor qsize (with two integers) and can be used to bound the total number of instances of unfoldings in a proof. In particular, the query \((\text{sigma H) check (qsize 5 H) B})\) is provable if and only if the total number of unfoldings of that proof is 5 or less.

Finally, Figure 7 contains the FPC based on the constructor max that is used to record explicitly all information within a proof: in
type qheight int -> cert.
type qsize int -> int -> cert.
ttE (qheight _).
eqE (qheight _).
ore (qheight H) (qheight H) _.
someE (qheight H) (qheight H) _.
andE (qheight H) (qheight H) (qheight H).
unfoldE (qheight H) (qheight H') :- H > 0, H' is H - 1.
eqE (qsize In In).
ttE (qsize In In).
ore (qsize In Out) (qsize In Out) _.
someE (qsize In Out) (qsize In Out) _.
andE (qsize In Out) (qsize In Mid) (qsize Mid Out).
unfoldE (qsize In Out) (qsize In' Out) :- In > 0, In' is In - 1.

Figure 6: Two FPCs that describe proofs that are limited in either height or in size.

type qheight int -> cert.
type qsize int -> int -> cert.

Figure 7: The max FPC

kind max type.
type max max -> cert.
type binary max -> max -> max.
type choose choice -> max -> max.
type term A -> max -> max.
type empty max.
ttE (max empty).
eqE (max empty).
ore (max (choose C M)) (max M) C.
someE (max (term T M)) (max M) T.
andE (max (binary M N)) (max M) (max N).
unfoldE (max M) (max M).

Figure 8: FPC for pairing

kind max type.
type max max -> cert.

Further, if we view a particular FPC as a means of restricting proofs, it is possible to build an FPC that restricts proofs satisfying two FPCs simultaneously. In particular, Figure 8 defines an FPC based on the (infix) constructor <\square>, which pairs two terms of type cert. The pairing experts for the certificate Cert1 \langle\square\rangle Cert2 simply requests that the corresponding experts also succeed for both Cert1 and Cert2 and, in the case of the orE and someE, also return the same choice and substitution term, respectively. Thus, the query

?- check (( qheight 4) <\square> ( qsize 10)) B

will succeed if there is a proof of B that has a height less than or equal to 4 while also being of size less than or equal to 10. A related use of the pairing of two proof certificates is to distill or elaborate proof certificates. For example, the proof certificate (qsize 5 \emptyset) is rather implicit since it will match any proof that used unfold exactly 5 times. However, the query

?- check (( qsize 5 \emptyset) <\square> ( max Max)) B.

will store into the \lambda Prolog variable Max more complete details of any proof that satisfies the (qsize 5 \emptyset) constraint. In particular, this forms the infrastructure of an explanation tool for attributing “blame” for the origin of a counterexample; these maximal certificates are an appropriate starting point for documenting both the counterexample and why it serves as a counterexample.

Various additional examples and experiments using the pairing of FPCs can be found in [9]. Using similar techniques, it is possible to define FPCs that target specific types for special treatment: for example, when generating integers, only (user-defined) small integers could be inserted into counterexamples.

3 PUTTING IT TOGETHER

We have explored several implementations of FPC, varying in host languages and applications [47]. For the sake of our proof-theoretic reconstruction of PBT, as we have motivated in the preceding Section, it suffices to resort to the so-called “two-level” approach [25], which should be familiar to anyone who has used meta-interpreters in logic programming: we steer the generation phase by means of appropriate FPCs, while we have the testing done by a standard vanilla meta-interpreter (Figure 9).

The interpreter back-chains on a format of reified clauses that is slightly more general than Figure 5: we adopt an implicit flattening of the disjunctive clause bodies in a list, and tag each clause body
type interp oo -> o.
type np string -> oo -> oo.
type prog oo -> list oo -> o.

interp tt.
interp (eq T T).
interp (and G1 G2) :- interp G1 , interp G2.
interp (or G1 G2) :- interp G1; interp G2.
interp A :- prog A Gs , member (np _ G) Gs , interp G.

Figure 9: The vanilla meta-interpreter.

with a unique name (possibly to be generalized to other metadata) to assist in the writing of certificates and the generation of reports for the user. For example, to generate lists of nats we write the following prog clause — compare this with Fig. 2:

prog (is_natlist L)
[(np "ni_null" (eq L null)),
 (np "ni_cons" (and (eq L (cons Hd Tl))
 (and (is_nat Hd)
 (is_natlist Tl))))].

This representation in turn induces a small generalization of the unfold expert, which is now additionally parameterized by a list of goals and by the id of the chosen alternative:

type unfoldE list oo -> cert -> cert -> string -> o.

Suppose we want to falsify the assertion that the reverse of a list is equal to itself. The generation phase is steered by the predicate check, which uses a certificate (its first argument) to produce candidate lists according to a generation strategy. The testing phase performs deterministic computation with the meta-interpreter interp and then negates the conclusion using negation-as-failure (NAF), yielding the clause:

cexrev Gen Xs :-
    check Gen (is_natlist Xs),
    interp (rev Xs Ys), not(Xs = Ys).

If we set Gen to be say qheight 3, the logic programming engine will return, among others, the answer Xs = cons zero (cons (succ zero) null). Note that the call to not is safe since, by the totality of rev, Ys will be ground at success time, which is also the reason why we choose not to return it.

The symmetry (idempotency for the functional programmer) of reverse reads as follows:

cexrev_sym Gen Xs :-
    check Gen (is_natlist Xs),
    interp (rev Xs Ys)
    not (interp (rev Ys Xs)).

Unless one’s implementation of reverse is grossly mistaken, the engine should abort searching for a counterexample according to the prescriptions of the generator.

We now see in more details how we capture in our framework various flavors of PBT.

3.1 Exhaustive generation

While PBT is traditionally associated with random generation, several tools now rely on exhaustive data generation up to a bound — in fact, such strategy is now the default in Isabelle/HOL’s PBT suite [8]:

(1) (Lazy)SmallCheck [54] views the bound as the nesting depth of constructors of algebraic data types.

(2) αCheck [13] employs the derivation height.

Our qsize and qheight FPCs in Figure 6 respectively match (1) and (2) and therefore can accommodate both. But, as mentioned in the previous Section, we can go further. A small drawback of our approach is that, while check Gen P, for Gen using one of qsize,qheight, will generate terms up to the given bound, in a PBT query the logic programming engine will enumerate them from that bound downwards. For example, a query such as

?- cexrev (qheight 10) Xs will return larger counterexamples first, starting here with a list of nine 0 and a 1. This means that if we do not have a good estimate of the dimension of our counterexample, our query may take an unnecessary long time or even loop.

A first fix, as we have seen, is certificate pairing. The query

?- cexrev ((qheight 10) <c> (qsize 6)) Xs will converge quickly, quicker in fact that with the separate bounds, to the usual minimal answer. However, we still ought to have some idea about the dimension of the counter-example beforehand and this is not realistic. Yet, it is easy, thanks to logic programming, to implement a simple-minded form of iterative deepening, where we backtrack over an increasing list of bounds:

cex_revIt Bound L :-
    mk_list Bound Range , memb H Range ,
    check (qheight H) (is_natlist L),
    interp (rev L R), not(L = R).

Paring can still fits, where we may choose to express size as a function of height — of course this can be tuned by the user, depending on the data she is working with:

cex_revIt Bound L :-
    mk_list Bound Range , memb H Range ,
    check (qheight H) (is_natlist L),
    interp (rev L R), not(L = R).

While this is immediate, it has the drawback of recomputing candidates at each level. A better approach is to introduce an FPC for a form of iterative deepening for exact bounds, where we output only those candidates requiring that precise bound. This has some similarity with the approach inFeat [19]. The interested reader can peruse this FPC in the code accompanying our paper.
### 3.2 Random generation

The FPC setup can be extended to support random generation of candidates. The idea is to implement a form of randomized backtracking, as opposed to the usual chronological one: when backchaining on an atom, we do not pick the first matching possibility, but flip a coin. As expected, most of the action in Figure 10 occurs in the unfolding expert. A certificate for random generation must instruct the kernel to select candidate constructors according to certain probability distributions. The user can specify such a distribution in a table assigning a weight to each clause in the generators of interest (referred to by their unique names). If no such table exists, the certificate assumes a uniform distribution. Upon unfolding a generator predicate, the kernel transmits the names of the constructors to the expert, which looks up their weights in the certificate and uses them to select exactly one. We use the recently added primitives for random generation in ELPI [18] to seed and retrieve a random number less than $Sum$.

At the top level, random generation must be wrapped inside another FPC (not shown here) whose only role is prescribing a given number of $qtries$ to generate candidates and verify whether any of them is a valid counterexample. The outcome is the equivalent of QuickCheck’s highly rewarding “OK, passed 100 tests” message. QuickCheck supports various other configuration options in its $config$ record, such as returning the seed for counterexample duplication, and we could easily mimic that as well.

Consider running our favorite example with a probability distribution doubling the weight of $cons$ w.r.t. $null$ and setting the number of $tries$ to 100:

```prolog
cexrevR Xs :-
    Ws = [(qw "nl_null" 1),(qw "nl_cons" 2)],
    tries NT,
```

One answer is:

```prolog
Xs = cons zero
    (cons zero (cons (succ zero) null))
```

Note that we have used an uniform choice between zero and successor w.r.t. nats.

As we discuss in Section 5, this is but one strategy for random generation and quite possibly not the most efficient one, as the experiments in Section 4.1 indicate. However, programming random generators is an art [22, 30] in every PBT approach. Nonetheless, we have tools at our disposal that could make this simpler. For example, we can pair the $qrandom$ FPC assigning higher probability to more complex constructors with $qsize$ to constrain the size of the terms that will be generated.

### 3.3 Shrinking

Randomly generated data that raise counter-examples may be too big to be the basis of the often frustrating process of bug-fixing. For an example, look no further than the run of the information-flow abstract machine described in [30]. For our much simpler example, there is certainly a smaller counter-example for our running property, say $cons$ zero (cons (succ zero) null).

Clearly, it is desirable to find automatically such smaller counter-examples. This phase is known as shrinking and consists of creating a number of smaller variants of bug-triggering data. These variants are then tested to determine if they trigger the same failure. If that is the case, the shrinking process can be repeated until we get to a local minimum. In the QuickCheck tradition, shrinkers, as well as custom generators, are the user’s responsibility, in the sense that PBT tools offer little or no support for their formulation.
is particularly painful when we need to shrink \textit{modulo} some invariant, e.g., well-typed terms or meaningful sequences of machine instructions.

One way to describe shrinking using FPCs is to follow the following outline.

\textbf{Step 1: Collect all substitution terms in an existing proof.} Given a successful proof that a counterexample exists, use the collect FPC in Figure 11 to extract the list of terms instantiating the existentials in that proof. Note that this FPC formally collects a list of terms of different types, in our running example \texttt{nat} and \texttt{list nat}: we accommodate such a collection by providing constructors (e.g., \texttt{c_nat} and \texttt{c_list_nat}) that map each of these types into the type \texttt{item}. Since the third argument of the \texttt{someE} expert predicate can be of any type, we use the \textit{ad hoc polymorphism} available in \texttt{\lambda}Prolog [46] to specify different clauses to use for this expert depending on the type of the term in that position: this allows us to chose different type coercion constructors to inject all these terms into the one type \texttt{item}.

For the purposes of the next step, it might also be useful to remove from this list any item that is a subterm of another item in that list. (The definition of the subterm relation is given also in Figure 11.)

\textbf{Step 2: Search again restricting substitution instances.} Search again for the proof of a counterexample but this time use the \texttt{huniv} FPC (Figure 12) that restricts the existential quantifiers to use subterms of terms collected in the first pass. (The name \texttt{huniv} is mnemonic for "Herbrand universe": that is, its argument is a predicate that describes the set of allowed substitution terms within the certificate.) Pairing with the FPC restricting size and/or height can additionally control the search for a new proof. Replacing the subterm relation with the proper-subterm relation can further constrain the search for proofs. For example, consider the following \texttt{\lambda}Prolog query, where \texttt{B} is a formula that encodes the counterexample property, \texttt{I} is the list of terms (items) collected from the proof of a counterexample, and \texttt{H} is the height determined for that proof.

\begin{verbatim}
  check ((huniv (T \ sigma I \ memb I Is , subterm T I))
       (qheight H) (max Max)) B.
\end{verbatim}

In this case, the variable \texttt{Max} will record the details of a proof that satisfies the height bound as well as instantiates the existential quantifiers with terms that were subterms of the original proof. One can also rerun this query with a lower height bound and by replacing the implemented notion of subterm with "proper subterm". In this way, the search for proofs involving smaller but related instantiations can be used to shrink a counterexample.

\section{PBT for MetaProgramming}

Once a topic in computational logic is described as proof search in the sequent calculus, it is often possible to generalize that topic from a treatment of first-order (algebraic) terms to a treatment of terms containing bindings. For example, once Prolog was described as proof search in sequent calculus, there was a direct line to the development of a logic programming language, for example \texttt{\lambda}Prolog, that treated terms containing bindings [41]. Similarly, once certain model checking and inductive theorem proving were described via the proof search paradigm, there were natural extensions of those tools to treat bindings in term structures: witness the Bedwyr model checker [7] and the Abella theorem prover [6].

There are two reasons why the proof search paradigm supports a natural treatment of binding within terms.

\begin{figure}
\centering
\begin{verbatim}
kind item
(type c_nat nat -> item.
type c_list_nat list nat -> item.
type subterm item -> item -> o.
type collect list item -> cert.

eqE (collect In In).
ttE (collect In In).
andE (collect In Out) (collect In Mid) (collect Mid Out).
unfoldE (collect In Out) (collect In Out).
someE (collect [(c_nat T) |In] Out) (collect In Out) (T : nat).
someE (collect [(c_list_nat T) |In] Out) (collect In Out) (T : list nat).

subterm Item Item.
subterm Item (c_nat (succ M)) :-
  subterm Item (c_nat M).
subterm Item (c_list_nat (Nat::L)) :-
  subterm Item (c_nat Nat);
  subterm Item (c_list_nat L).

Figure 11: An FPC for collecting substitution terms from proof and a predicate to compute subterms.
\end{verbatim}
\caption{An FPC for restricting existential choices.}
\end{figure}
(1) The sequent calculus of Gentzen [26] contains a notion of binding over sequents: the so-called *eigenvariables*.

(2) The sequent calculus also supports what is called the *mobility of binders*, meaning that binders within terms can move to binders within formulas (i.e., quantifiers) and these can move to binders within sequents (eigenvariables) [40].

This approach to specifying computation on terms containing bindings is generally referred to as the $\lambda$-tree syntax approach [40]. The advantage of this over many other, more explicit, methodologies is that when implementing the search for proofs involving sequents, one usually must deal with a direct treatment of bindings at sequent calculus level and this typically involves unification involving some forms of higher-order unification. In essence, once bindings are treated correctly at the proof level, bindings at the term level can be treated implicitly by having them move to the proof level.

The full treatment of $\lambda$-tree syntax in a logic with fixed points is usually accommodated with the addition of the $\nabla$-quantifier [24, 44]. That is, when fixed points are present, the sequent calculus needs to accommodate a new operator: $\nabla$ is a formula-level binder (quantifier) and the sequent calculus must permit a new binding context that is separate from the binding context provide by eigenvariables. While the $\nabla$-quantifier has had significant impact in several reasoning tasks (for example, in the formalized metatheory of the $\pi$-calculus [58] and $\lambda$-calculus [1]) an important result about $\nabla$ is the following: if fixed point definitions do not contain implications and negations, then exchanging occurrences of $\nabla$ and $\nabla$ does not affect which atomic formulas are proved [44, Section 7.2]. Since we shall be limiting ourselves to Horn-like recursive definitions (as we already argued in Section 2.3), we can use the $\lambda$Prolog implementation of $\nabla$ goal formulas in order to capture the proof search behavior of $\nabla$ in extended Horn specifications.

Thus, in order for the proof checking kernel in Figure 5 to capture our limited use of the $\nabla$-quantifier, we need to only add the following lines to its specification.

```plaintext
type nabla (A -> oo) -> oo.
check Cert (nabla G) :-
  pi x \ check Cert (G x).
```

The first item introduces the (polymorphically typed) symbol that will denote $\nabla$ and the second item is the proof checking clause for that quantifier. (The symbols $\pi x$ is the $\lambda$Prolog universal quantifier over the variable $x$.) Note that no premise involving an expert predicate (in the sense of the FPC architecture) is needed here.

In what follows, we assume that the reader has at least a passing understanding of how the mobility of binders is supported in computer systems such as $\lambda$Prolog, Twelf [51] and Beluga [52].

### 4.1 Lifting PBT to treat $\lambda$-tree syntax

To showcase the ease with which we handle searching for counterexamples in binding signatures, we encode a simply-typed $\lambda$-calculus augmented with constructors for integers and lists, following the PLT-Redex benchmark from http://docs.racket-lang.org/redex/benchmark.html. The language is as follows:

```
Types      A, B ::= int | ilist | A -> B
Terms      M ::= x | \lambda x:A. M |- M1 M2 | c | err
Constants c ::= n | plus | nil | cons | hd | tl
Values     V ::= c | \lambda x:A. M | plus V |
             | cons V | cons V1 V2
```

The rules for the dynamic and static semantics are given in Figure 13, where the latter assumes a signature $\Sigma$ with the obvious type declarations for constants. Rules for $plus$ are omitted for brevity.

The encoding of the syntax in $\lambda$Prolog is pretty standard and also omitted: we declare constructors for terms, constants and types, while we carve out values via an appropriate predicate. A similar predicate characterizes the threading in the operational semantics of the $err$ expression, used to model run time errors such as taking the head of an empty list. We follow this up (Figure 14) with the static semantics (predicate $wt$), where constants are typed via a table $tcc$. Note that we have chosen an *explicitly* context-ed encoding of typing as opposed to one based on hypothetical judgments such as in [25]: this choice avoids using implications in the body of the typing predicate and, as a result, allows us to use $\lambda$Prolog’s universal quantifier to implement the $\nabla$-quantifier.

Now, this calculus enjoys the usual property of type preservation and progress, where the latter means “being either a value, an error, or able to make a step.” And, in fact we can fairly easily prove those results in Abella.


**Theorem** prog: for all $M A$, $wt nil M A \rightarrow progress M$.

However, the case distinction in the progress theorem does require some care: were it to be unprovable due to a mistake in the specification, locating where the problem lies may be hard.

On the other hand, one could wonder whether our calculus enjoys the *subject expansion* property:

**Theorem** sexp: for all $M M A$, $step M M \rightarrow wt nil M A \rightarrow wt nil M A$.

The alert reader will undoubtedly realize that this is highly unlikely, but rather than wasting time in a proof attempt of the latter, we search for a counterexample:

```
cexsexp Bound M M’ A :- 
mk_list Bound Range, memb R Range, S is R * 2,
  check (qsize S _) 
  (and (step M M’) (is_exp null M)),
  interp (wt null M’ A),
  not (interp (wt null M A)).
```

```
A = listTy
M’ = c tl
M = app (lam(x \ err) listTy) (c tl)
```

In math notation, $(\lambda x : listTy, err) tl$ steps to $tl$, which is well-typed by $listTy$ but the type is not preserved backwards.
we ensure that the variable $M$ that occurs in the negated goal is

There are other queries we can ask: are there untypable terms, or terms that do not converge to a value? As a more comprehensive validation, we address the nine mutations (that is, purposely inserted bugs to test the PBT suite) proposed by the PLT-Redex benchmark: those are to be spotted as a violation of either the above preservation or progress properties. For example, the first mutation introduces a bug in the typing rule for application, matching the following setup we have as a default. However, bugs 2

There are other queries we can ask: are there untypable terms, or terms that do not converge to a value? As a more comprehensive validation, we address the nine mutations (that is, purposely inserted bugs to test the PBT suite) proposed by the PLT-Redex benchmark: those are to be spotted as a violation of either the above preservation or progress properties. For example, the first mutation introduces a bug in the typing rule for application, matching the following setup we have as a default. However, bugs 2

Table 1 sums up the tests performed under Ubuntu 18.04.2 on an Intel Core i5-3570K at 3.40 GHz with 16GB RAM. Rather than reporting precise timing for each benchmark this small is of doubtful relevance, we list success (✓) if we find the bug within the time out (30 seconds), otherwise we deem it a failure (✗). We list the results obtained by running λProlog under ELPI using the following four strategies: $H$ for q-height, $S$ for q-size, $H$+S for the pairing of the latter and $Rnd$. The first three strategies all use the naive iterative deepening trick we have described in Section 3.1. $Rnd$ is applied to 1000 tries. The last column has a brief description of the bug together with Redex’s (quite arbitrary) difficulty rating: shallow, medium, unnatural.

The first observation is that the exhaustive strategies perform very well catching all bugs (except for $S$ that misses 4). Random generation is less effective, as expected, in the naive uniform distribution setup we have as a default. However, bugs 2, 4 and 6 can be found with simple tweaks to the distribution of terms. Generation in bug 5 triggers a bug in the current version of ELPI and fails to finish. Bug 9, which is rather artificial in its formulation, aborts with the standard generator (the wt judgments), but is found with a simpler one.
which, however, is unable to account for random search. Similarly, the literature on PBT is too large to even try and sum up. We refer to [12] for a review with an emphasis to its interaction with proof assistants and specialized domains as programming language meta-theory.

Within the confines of meta-theory model checking, a major player is PLT-Redex [20], an executable DSL for mechanizing semantic models built on top of DrRacket with support for random testing à la QuickCheck; its usefulness has been demonstrated in several impressive case studies [33]. However, Redex has limited support for relational specifications and none whatsoever for binding signature. This is where aCheck [12, 13] comes in. That tool is built on top of the nominal logic programming language αProlog, a checker for relational specifications similar to those considered here. Arguably, aCheck is less flexible than the FPC-based architecture that we have proposed here, since it can be seen as a fixed choice of experts. Indeed, our “bounding” FPCs in Figure 6 have a clear correspondence with the way exhaustive term generation is been addressed there, as well as in (Lazy)SmallCheck [54]. In both cases, those strategies are wired-in and cannot be varied, let alone combined as we can via pairing. The notion of smart generators in Isabelle/HOL’s QuickCheck [11] is approximated by the derivation-first approach.

Using an FPC as a description of how to traverse a search space bears some resemblance with principled ways to change the depth-first nature of search in logic programming. An example is Tor [55], which, however, is unable to account for random search. Similarly to Tor, FPCs would benefit of partial evaluation to remove the meta-interpretive layer.

In the random case, the logic programming paradigm is already ahead w.r.t. to the labor-intensive QuickCheck approach of writing custom generators. In fact, we can use judgments (think typing), as generators, avoiding the issue of keeping generators and predicates in sync when checking invariant-preserving properties such as type preservation [34]. Further, viewing random generation as expert-driven random back-chaining opens up all sort of possibilities: we have chosen just one simple-minded strategy, namely what boils down to permuting the predicate definition at each unfolding, but we could easily follow others, such as the ones described in [22]; permuting the definition just once at the start of the generation phase, or even changing the weights at the end of the run so as to steer the derivation towards axioms/leaves. Of course, our default uniform distribution corresponds to QuickCheck’s oneOf combinator, while the weights table to frequency.

The last few years have shown some interest in the (random) generation of data satisfying some invariants [15]; mostly, well-typed λ-terms, with an application to testing optimizing compilers [22, 39, 48]. In particular, the most successful generator [48] consists of over 1500 lines of dense Haskell code hard-wired to generate a specific object language. Compare this to our 10 lines of readable clauses. We make no claim, at least not without trying first, about how successfully we could challenge a compiler, but we do want to remark how flexible our approach is. There also seems to be a connection with probabilistic logic programming, e.g., [23], although the inference mechanism is very different.

### 6 Conclusion and Future Work

We have described an approach that uses standard logic programming techniques and some recent developments in proof theory to design a uniform and flexible framework that accounts for many features of PBT. Given its proof-theoretic pedigree, it was immediate to extend PBT to the metaprogramming setting, inheriting the handling of λ-tree syntax, which is naturally supported by λProlog and notably absent from other environments for meta-theory model checking such as PLT-Redex.

While λProlog is used here to discover counterexamples, one does not actually need to trust the logical soundness of λProlog (negation-as-failure makes this a complex issue). Any counterexample that is discovered can be output and used within, say, Abella to formally prove that it is indeed a counterexample in its richer logic. In fact, we plan to integrate our take on PBT in Abella, in order to

### Table 1: STLC benchmark

<table>
<thead>
<tr>
<th>bug</th>
<th>check</th>
<th>αC</th>
<th>H</th>
<th>S</th>
<th>H+S</th>
<th>Rnd</th>
<th>Sizes</th>
<th>Description/Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>preservation ✓ ✓ ✓ ✓ ✓ ✓ (4,8) range of function in app rule matched to the argument (S)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>progress ✓ ✓ ✓ ✓ ✓ ✓ (4,6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>preservation ✓ ✓ ✓ ✓ ✓ ✓ (5,9) value (cons v) v omitted (M)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>progress ✓ ✓ ✓ ✓ ✓ ✓ (4,7) order of types swapped in function pos of app (S)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>preservation ✓ ✓ ✓ ✓ ✓ ✓ (4,6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>preservation ✓ ✓ ✓ ✓ ✓ ✓ (6,16) the type of cons return int (S)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>progress ✓ ✓ ✓ ✓ ✓ ✓ (6,12) tail reduction returns the head (S)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>progress ✓ ✓ ✓ ✓ ✓ ✓ (6,12) hd reduction on partially applied cons (M)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>preservation ✓ ✓ ✓ ✓ ✓ ✓ (4,9) no eval for argument of app (M)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>preservation ✓ ✓ ✓ ✓ ✓ ✓ (3,6) lookup always returns int (U)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>preservation ✓ ✓ ✓ ✓ ✓ ✓ (4,7) vars do not match in lookup (S)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The column αC lists the results αCheck [12] using NAF, which is rarely the best technique for these case studies [13], but it corresponds closely to the architecture of the present paper. The results between FPC and αCheck are essentially indistinguishable for most bugs, save for bugs 4 and 5, where αCheck times out.
support both proofs and disproofs. Although the present setup is enough to check many of Abella’s meta-logical specifications, it does not support parametric-hypothetical judgments unless under translation, as we have see for the typing judgment in Section 4.1. A natural environment instead to do PBT for every spec in Abella is Bedwyr [7], which shares the same meta-logic, but is more efficient from the point of view of proof search — after all, it was designed as a model-checker.

Once we go beyond Horn clause logic, another possibility is to extend the two-level approach to sub-structural logics, typically in the linear family; see [37] for encoding MiniML with references in linear logic or [21] for a refinement to ordered linear logic to study continuation machines. From a PBT perspective, it would make sense to move to the logic level data structures such as heaps, frame stacks etc. that could be problematic for (exhaustive) data generation. Another dimension refers to coinductive specifications, where Abella excels [45, 58]; consider for example using PBT to find programs that refute the equivalence of ground and applicative bisimilarity [53]. Again, Bedwyr seems a good choice.

One answer to the problem of generating (and shrinking) terms under some invariant is doing so in a dependently typed setting. Here, we can use encodings based on intrinsically-typed terms (see [4] for an application of the paradigm beyond the usual suspects) to rule-out by construction terms not preserving the given constraint. An immediate way to test this hypothesis is to move our FPCs to a kindred framework such as Twelf. Finally, we have just hinted at ways for localizing the origin of the bugs reported by PBT. This issue can benefit from research in declarative debugging as well as in justification for logic programs [50]. Coupled with recent results in focusing [43] this could lead us also to a reappraisal of techniques for repairing (inductive) proofs [32].

Acknowledgments This paper is a major extension of work-in-progress presented (but not published) at LFMT'17. We wish to thank Enrico Tassi for extending ELPI with support for random number generation.

REFERENCES


