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 λ-tree syntax. The λProlog programming language is capable of performing both the generation and checking of tests. We validate this approach by tackling benchmarks in the metatheory 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 λ-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 in 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 FPC 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 λProlog [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 scoping. 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[1]

$$\forall L\colon (list\ int)\ \forall R\colon (list\ int)\ [rev\ L\ R \supset rev\ R\ L],$$

stating that *rev* is symmetric, should be a theorem. In fact, we might wish to prove a number of formulas of the form

$$\forall x\colon \tau\ [P(x) \supset Q(x)]$$

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 $x\colon \tau$ into the logic by turning the type into a predicate: $\forall x[(\tau(x) \wedge P(x)) \supset Q(x)]$. 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 $\exists x[(\tau(x) \wedge P(x)) \wedge \neg Q(x)]$ instead. If a term t of type τ 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.

[1] We use λProlog syntax [41], which is similar to Prolog syntax. One difference is that λProlog uses curried syntax and the other difference is that it is polymorphically typed: in particular, the expression $(list\ int)$ is the type of lists of integers.

Note that in order to speak of generate-and-test as a strategy for finding a proof (hence, a counterexample) for $\exists x[(\tau(x) \wedge P(x)) \wedge \neg Q(x)$, 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 $app\, Xs\ Ys\ Zs \supset app\, (cns\ X\ Xs)\ Ys\ (cns\ X\ Zs)$. 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 "`:-`" 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 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.

```
nat z.                          lst nl.
nat (s N) :- nat N.             lst (cns N Ns) :- nat N, lst Ns.

app nl Xs Xs.
app (cns X Xs) Ys (cns X Zs) :- app Xs Ys Zs.
```

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

```
nat X :- X = z ; sigma X'\ X = (s X'), nat X'.
lst Ns :- Ns = nl ; sigma Ns'\ Ns = (cns N Ns'), nat N, lst Ns'.
app Xs Ys Zs :- Xs = nl, Ys = Zs ;
                sigma Xs'\ sigma Zs'\ Xs = (cns X Xs'), Zs = (cns X Zs'), app Xs' Ys Zs'.
```

Figure 2: Specifications listing one Horn clause per predicate. Following λProlog syntax, the expression `sigma X\` denotes the existential quantifier over the variable `X` and the semicolon and comma denote as usual disjunction and conjunction, respectively.

$$
\begin{aligned}
nat &= \mu\lambda N\lambda n\ (n = \mathbf{z} \vee \exists n'(n = (\mathbf{s}\ n') \wedge N\ n'))\\
lst &= \mu\lambda L\lambda l\ (l = \mathbf{nl} \vee \exists n, l'(l = (\mathbf{cns}\ n\ l') \wedge nat\ n \wedge L\ l'))\\
app &= \mu\lambda A\lambda xs\lambda ys\lambda zs\ ((xs = \mathbf{nl} \wedge ys = zs)\ \vee \exists x'\exists xs'\exists zs'(xs = (\mathbf{cns}\ x'\ xs') \wedge zs = (\mathbf{cns}\ x'\ zs') \wedge A\ xs'\ ys\ zs'))
\end{aligned}
$$

Figure 3: Specifications as (least) fixed point expressions. The specifications in Figure 2 are written using standard logic notation and the least fixed operator μ over higher-order abstractions written using λ-abstractions.

(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

$$\exists x[(\tau(x) \wedge P(x)) \wedge \neg Q(x)] \qquad (*)$$

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 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

$$\exists x[\tau(x) \wedge P(x)]. \qquad (**)$$

Such a certificate would contain the witness (closed) term t for the existential quantifier and sufficient information to confirm that

[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 $\otimes$ and & or in classical and intuitionistic logics as $\wedge^+$ and $\wedge^-$, respectively. In this paper, we use simply $\wedge$ to denote the positive biased conjunction.

$P(t)$ 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 inference rule is augmented with an additional premise involving an *expert predicate*, a certificate Ξ, and possibly continuations of certificates (Ξ', Ξ_1, Ξ_2) with extracted information from certificates (in the case of $\vee$ and $\exists$). The premise $(A :\!-\, G) \in grnd\,(\mathcal{P})$ in the last inference rule of Figure 4 states the logic programming clause $(A :\!-\, G)$ 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 $(A :\!-\, G) \in grnd\,(\mathcal{P})$.

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 `(check (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 `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 particular, all disjunctive choices and all substitution instances for existential quantifiers are collected into a binary tree structure of type `max`. In this sense, proof certificates built with the `max` constructor are *maximally* explicit. Such proof certificates are used in [9]: it is important to note that proof checking with such maximally explicit certificates can be done with much simpler proof-checkers than those used in logic programming since backtracking search and unification are not needed.

$$\frac{\Xi_1 \vdash G_1 \qquad \Xi_2 \vdash G_2 \qquad \wedge_e(\Xi, \Xi_1, \Xi_2)}{\Xi \vdash G_1 \wedge G_2} \qquad \frac{tt_e(\Xi)}{\Xi \vdash tt}$$

$$\frac{\Xi' \vdash G_i \qquad \vee_e(\Xi, \Xi', i)}{\Xi \vdash G_1 \vee G_2} \qquad \frac{\Xi' \vdash G[t/x] \qquad \exists_e(\Xi, \Xi', t)}{\Xi \vdash \exists x.G}$$

$$\frac{=_e(\Xi)}{\Xi \vdash t = t} \qquad \frac{\Xi' \vdash G \quad (A :\!-\, G) \in grnd\,(\mathcal{P}) \quad unfold_e(\Xi, \Xi')}{\Xi \vdash A}$$

Figure 4: A proof system augmented with proof certificates and additional "expert" premises.

```
% Object-level formulas
kind oo        type.
type tt        oo.
type and, or   oo  -> oo -> oo.
type some      (A -> oo) -> oo.
type eq        A -> A  -> oo.

% Object-level Prolog clauses are
% stored as facts (prog A Body).
type prog      oo -> oo -> o.

% Certificates
kind cert           type.
kind choice         type.
type left, right    choice.

% The types for the expert predicates
type ttE, eqE                 cert -> o.
type unfoldE          cert -> cert -> o.
type someE     cert -> cert -> A -> o.
type andE   cert -> cert -> cert -> o.
type orE  cert -> cert -> choice -> o.

% Certificate checker
type    check    cert -> oo -> o.

check Cert tt          :- ttE Cert.
check Cert (eq T T)    :- eqE Cert.
check Cert (and G1 G2) :-
   andE Cert Cert1 Cert2,
   check Cert1 G1, check Cert2 G2.
check Cert (or G1 G2) :-
   orE Cert Cert' LR,
   ((LR = left,  check Cert' G1);
    (LR = right, check Cert' G2)).
check Cert (some G) :-
   someE Cert Cert1 T,
   check Cert1 (G T).
check Cert A :- unfoldE Cert Cert',
                prog A G, check Cert' G.
```

Figure 5: A simple proof checking kernel.

```
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.

```
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 7: The max FPC

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 `<c>`, which *pairs* two terms of type `cert`. The pairing experts for the certificate `Cert1 <c> 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) <c> (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

```
type    <c>      cert ->  cert ->  cert.
infixr <c>      5.

ttE      (A <c> B) :- ttE A, ttE B.
eqE      (A <c> B) :- eqE A, eqE B.
someE    (A <c> B) (C <c> D) T :-
         someE A C T, someE B D T.
orE      (A <c> B) (C <c> D) E :-
         orE A C E, orE B D E.
andE     (A <c> B) (C <c> D) (E <c> F) :-
         andE A C E, andE B D F.
unfoldE (A <c> B) (C <c> D) :-
          unfoldE A C, unfoldE B D.
```

Figure 8: FPC for pairing

use of the pairing of two proof certificates is to *distill* or *elaborate* proof certificates. For example, the proof certificate `(qsize 5 0)` is rather implicit since it will match any proof that used unfold exactly 5 times. However, the query

```
?- check ((qsize 5 0) <c> (max Max)) B.
```

will store into the λProlog variable `Max` more complete details of any proof that satisfies the `(qsize 5 0)` 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 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 "nl_null" (eq L null)),
   (np "nl_cons" (and (eq L (cons Hd Tl))
```

```
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.

```
                    (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 counterexamples 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,
  Sh is H * 3,
  check ((qheight H) <c> (qsize Sh))
      (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 in *Feat* [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

```
kind   qweight         type.
type   qw              string -> int -> qweight.
type   qrandom         list qweight -> cert.
type   qtries          int -> list qweight -> cert.
type   sum_weights     list oo -> list qweight -> int -> list qweight -> o.
type   select_clause   int -> list qweight -> string -> o.
type   tries           int -> o.

ttE   (qrandom _).
eqE   (qrandom _).
andE  (qrandom Ws) (qrandom Ws) (qrandom Ws).
orE   (qrandom Ws) (qrandom Ws) Choice :- (Choice = left;Choice = right).
someE (qrandom Ws) (qrandom Ws) T.
unfoldE Gs (qrandom Ws) (qrandom Ws) Id :-
     sum_weights Gs Ws Sum SubWs,
     random.self_init, random.int Sum Random,
     select_clause Random SubWs Id.
```

Figure 10: FPC for random generation

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`.[4] 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:

```
cexrevR Xs :-
  Ws = [(qw "nl_null" 1),(qw "nl_cons" 2)],
  tries NT,
  check (qtries NT Ws) (is_natlist Xs),
  interp (rev Xs Ys), not(Xs = Ys).
```

One answer is:

```
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 counterexamples. 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. This is particularly painful when we need to shrink *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.

[4] λProlog does not have such primitives, but we could simulate it via reading oracle streams or inter-process communication.

```
kind item                        type.
type c_nat               nat -> item.
type c_list_nat     list nat -> item.

type subterm        item -> item -> o.
type collect        list item ->
                    list item -> cert.

eqE    (collect In In).
ttE    (collect In In).
orE    (collect In Out)
       (collect In Out) C.
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.

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 `nat` and `list nat`: we accommodate such a collection by providing constructors (e.g., `c_nat` and `c_list_nat`) that map each of these types into the type `item`. Since the third argument of the `someE` expert predicate can be of any type, we use the *ad hoc polymorphism* available in λProlog [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 `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.)

Step 2: Search again restricting substitution instances. Search again for the proof of a counterexample but this time use the `huniv` FPC (Figure 12) that restricts the existential quantifiers to use subterms of terms collected in the first pass. (The name `huniv` is mnemonic for "Herbrand universe": that is, its argument is a predicate that describes the set of allowed substitution terms within the certificate.)

```
type huniv   (item -> o) -> cert.

ttE     (huniv _).
eqE     (huniv _).
orE     (huniv Pred) (huniv Pred) _.
andE    (huniv Pred) (huniv Pred)
                     (huniv Pred).
unfoldE (huniv Pred) (huniv Pred).
someE (huniv Pred) (huniv Pred)
      (T:nat) :- Pred (c_nat T).
someE (huniv Pred) (huniv Pred)
      (T:list nat) :- Pred (c_list_nat T).
```

Figure 12: An FPC for restricting existential choices.

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 λProlog query, where `B` is a formula that encodes the counterexample property, `Is` is the list of terms (items) collected from the proof of a counterexample, and `H` is the height determined for that proof.

```
check
  ((huniv
     (T\ sigma I\ memb I Is, subterm T I))
    <c> (qheight H) <c> (max Max)) B.
```

In this case, the variable `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.

4 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 λProlog, 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 reason why the proof search paradigm supports a natural treatment of binding within terms.

(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 *λ-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 λ-tree syntax in a logic with fixed points is usually accommodated with the addition of the ∇-quantifier [24, 44]. That is, when fixed points are present, the sequent calculus needs to accommodate a new operator: ∇ 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 ∇-quantifier has had significant impact in several reasoning tasks (for example, in the formalized metatheory of the π-calculus [58] and λ-calculus [1]) an important result about ∇ is the following: if fixed point definitions do not contain implications and negations, then exchanging occurrences of $\forall$ and ∇ 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 λProlog implementation of $\forall$ goal formulas in order to capture the proof search behavior of ∇ in extended Horn specifications.

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

```
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 ∇ and the second item is the proof checking clause for that quantifier. (The symbols `pi x\` is the λProlog 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 λProlog, Twelf [51] and Beluga [52].

4.1 Lifting PBT to treat λ-tree syntax

To showcase the ease with which we handle searching for counterexamples in binding signatures, we encode a simply-typed λ-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	::=	$int \mid ilist \mid A \to B$
Terms	M	::=	$x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \mid c \mid err$
Constants	c	::=	$n \mid plus \mid nil \mid cons \mid hd \mid tl$
Values	V	::=	$c \mid \lambda x{:}A.\ M \mid plus\ V$
			$\mid cons\ V \mid cons\ V_1\ V_2$

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

The encoding of the syntax in λProlog 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 λProlog's universal quantifier to implement the ∇-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 pres: forall M N A,
  step M N -> wt nil M A -> wt nil N A.
Theorem prog: forall M A,
  wt nil M A -> 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 hardish.

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

```
Theorem sexp: forall M M' A,
  step M M' -> wt nil M' A -> 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, $(\lambda x : \mathit{listTy}.\ \mathit{err})\ \mathit{tl}$ steps to *tl*, which is well-typed by *listTy* but the type is not preserved backwards.

Note that we organize the generation phase in a "derivation-first" fashion [12], where, given a generation strategy, we use judgments, here derivations in the operational semantics, as generators. Then we ensure that the variable M that occurs in the negated goal is ground. These choices are left to the user, exploiting mode analysis, if available.

$$\frac{}{hd\ (cons\ V_1\ V_2) \longrightarrow V_1}\ \text{E-HD} \qquad \frac{}{tl\ (cons\ V_1\ V_2) \longrightarrow V_2}\ \text{E-TL}$$

$$\frac{}{\lambda x : A.\ M\ V \longrightarrow [x \mapsto V]M}\ \text{E-ABS} \qquad \frac{M_1 \longrightarrow M_1'}{M_1\ M_2 \longrightarrow M_1'\ M_2}\ \text{E-APP1} \qquad \frac{M \longrightarrow M'}{V\ M \longrightarrow V\ M'}\ \text{E-APP2}$$

$$\frac{}{\vdash_\Sigma err : A}\ \text{T-ER} \quad \frac{\Sigma(c) = A}{\vdash_\Sigma c : A}\ \text{T-K} \quad \frac{x : A \in \Gamma}{\Gamma \vdash_\Sigma x : A}\ \text{T-VAR} \quad \frac{\Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x : A.\ M : A \to B}\ \text{T-ABS} \quad \frac{\Gamma \vdash_\Sigma M_1 : A \to B \quad \Gamma \vdash_\Sigma M_2 : A}{\Gamma \vdash_\Sigma M_1\ M_2 : B}\ \text{T-APP}$$

Figure 13: Static and dynamic semantics of the *STLC* language.

```
progs (wt Ga E T)
      [(np "wt-var" (memb (bind E T) Ga)),
       (np "wt-err" (eq E error)),
       (np "wt-tcc" (and (eq E (c M))
                         (tcc M T))),
       (np "wt-app" (and (eq E (app X Y))
                         (and (wt Ga X (funTy H T)) (wt Ga Y H)))),
       (np "wt-lam" (and (and (eq E (lam F Tx)) (eq T (funTy Tx T')))
                         (nabla x\ wt (cons (bind x Tx) Ga) (F x) T'))) ].
```

Figure 14: Encoding of the static semantics of the *STLC* language.

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 range of the function type to the type of the argument:

$$\frac{\Gamma \vdash_\Sigma M_1 : A \to B \quad \Gamma \vdash_\Sigma M_2 : \boxed{B}}{\Gamma \vdash_\Sigma M_1\ M_2 : B}\ \text{T-APP-B1}$$

The given mutation makes both properties fail:

```
cexprog M A :-
  mk_list Bound Range, memb R Range, S is
      R * 2,
  check (qsize S _) (wt null M A),
  not (interp (progress M)).

A = intTy
M = app (c hd) (c (toInt zero))

cexpres M M' A :-
  mk_list Bound Range, memb R Range, S is
      R * 2,
  check (qsize S _) (wt null M A),
  interp (step M M'),
  not (interp (wt null M' A)).

A  = funTy listTy intTy
M' = lam (x\ c hd) listTy
M  = app (lam (x\ lam (y\ x) listTy)
     intTy) (c hd)
```

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 that for a 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 `qheight`, *S* for `qsize`, *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.

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.

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

Table 1: *STLC* benchmark

5 RELATED WORK

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 confine 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 αCheck [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, αCheck 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 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 encondings 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 *LFMTP'17*. We wish to thank Enrico Tassi for extending ELPI with support for random number generation.

REFERENCES

[1] Beniamino Accattoli. 2012. Proof pearl: Abella formalization of lambda calculus cube property. In *Second International Conference on Certified Programs and Proofs (Lecture Notes in Computer Science)*, Chris Hawblitzel and Dale Miller (Eds.), Vol. 7679. Springer, 173–187.

[2] Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *J. of Logic and Computation* 2, 3 (1992), 297–347. https://doi.org/10.1093/logcom/2.3.297

[3] K. R. Apt and M. H. van Emden. 1982. Contributions to the theory of logic programming. *JACM* 29, 3 (1982), 841–862.

[4] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2017. Intrinsically-typed Definitional Interpreters for Imperative Languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (Dec. 2017), 34 pages. http://doi.acm.org/10.1145/3158104

[5] David Baelde. 2012. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic* 13, 1 (April 2012), 2:1–2:44. https://doi.org/10.1145/2071368.2071370

[6] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. 2014. Abella: A System for Reasoning about Relational Specifications. *Journal of Formalized Reasoning* 7, 2 (2014).

[7] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. 2007. The Bedwyr system for model checking over syntactic expressions. In *21th Conf. on Automated Deduction (CADE) (LNAI)*, F. Pfenning (Ed.). Springer, New York, 391–397.

[8] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. 2011. Automatic Proof and Disproof in Isabelle/HOL. In *FroCoS (Lecture Notes in Computer Science)*, Cesare Tinelli and Viorica Sofronie-Stokkermans (Eds.), Vol. 6989. Springer, 12–27.

[9] Roberto Blanco, Zakaria Chihani, and Dale Miller. 2017. Translating Between Implicit and Explicit Versions of Proof. In *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction (Lecture Notes in Computer Science)*, Leonardo de Moura (Ed.), Vol. 10395. Springer, 255–273.

[10] Roberto Blanco and Dale Miller. 2015. Proof Outlines as Proof Certificates: a system description. In *Proceedings First International Workshop on Focusing (Electronic Proceedings in Theoretical Computer Science)*, Iliano Cervesato and Carsten Schürmann (Eds.), Vol. 197. Open Publishing Association, 7–14.

[11] Lukas Bulwahn. 2012. The New Quickcheck for Isabelle - Random, Exhaustive and Symbolic Testing under One Roof. In *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings (Lecture Notes in Computer Science)*, Chris Hawblitzel and Dale Miller (Eds.), Vol. 7679. Springer, 92–108. https://doi.org/10.1007/978-3-642-35308-6_10

[12] James Cheney and Alberto Momigliano. 2017. αCheck: A mechanized metatheory model checker. *Theory and Practice of Logic Programming* 17, 3 (2017), 311âĂŞ352.

[13] James Cheney, Alberto Momigliano, and Matteo Pessina. 2016. Advances in Property-Based Testing for αProlog. In *Tests and Proofs - 10th International Conference, TAP 2016, Vienna, Austria, July 5-7, 2016, Proceedings (Lecture Notes in Computer Science)*, Bernhard K. Aichernig and Carlo A. Furia (Eds.), Vol. 9762. Springer, 37–56.

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

[15] Koen Claessen, Jonas Duregård, and Michal H. Palka. 2015. Generating constrained random data with uniform distribution. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000143

[16] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*. ACM, 268–279.

[17] K. L. Clark. 1978. Negation as failure. In *Logic and Data Bases*, J. Gallaire and J. Minker (Eds.). Plenum Press, New York, 293–322.

[18] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. 2015. ELPI: Fast, Embeddable, λProlog Interpreter. In *Proceedings of LPAR*. Suva, Fiji. https://hal.inria.fr/hal-01176856

[19] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: functional enumeration of algebraic types. In *Haskell Workshop*, Janis Voigtländer (Ed.). ACM, 61–72. https://doi.org/10.1145/2364506.2364515

[20] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex.* The MIT Press.

[21] Amy P. Felty and Alberto Momigliano. 2012. Hybrid - A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax. *J. Autom. Reasoning* 48, 1 (2012), 43–105. https://doi.org/10.1007/s10817-010-9194-x

[22] Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *ESOP (Lecture Notes in Computer Science)*, Vol. 9032. Springer, 383–405.

[23] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *TPLP* 15, 3 (2015), 358–401. https://doi.org/10.1017/S1471068414000076

[24] Andrew Gacek, Dale Miller, and Gopalan Nadathur. 2008. Combining generic judgments with recursive definitions. In *23th Symp. on Logic in Computer Science*, F. Pfenning (Ed.). IEEE Computer Society Press, 33–44.

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

[26] Gerhard Gentzen. 1935. Investigations into Logical Deduction. In *The Collected Papers of Gerhard Gentzen*, M. E. Szabo (Ed.). North-Holland, Amsterdam, 68–131. https://doi.org/10.1007/BF01201353

[27] Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50, 1 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

[28] Jean-Yves Girard. 1992. A Fixpoint Theorem in Linear Logic. (Feb. 1992). An email posting to the mailing list linear@cs.stanford.edu.

[29] Quentin Heath and Dale Miller. 2017. A Proof Theory for Model Checking: An Extended Abstract. In *Proceedings Fourth International Workshop on Linearity (LINEARITY 2016) (EPTCS)*, Iliano Cervesato and Maribel Fernández (Eds.), Vol. 238.

[30] Catalin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 455–468. https://doi.org/10.1145/2500365.2500574

[31] John Hughes. 2007. QuickCheck Testing for Fun and Profit. In *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007 (Lecture Notes in Computer Science)*, Michael Hanus (Ed.), Vol. 4354. Springer, 1–32.

[32] Andrew Ireland and Alan Bundy. 1996. Productive use of failure in inductive proof. *Journal of Automated Reasoning* 16 (1996), 79–111.

[33] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. 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)*. ACM, New York, NY, USA, 285–296.

[34] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Random Generators. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. https://arxiv.org/abs/1607.05443

[35] Chuck Liang and Dale Miller. 2009. Focusing and Polarization in Linear, Intuitionistic, and Classical Logics. *Theoretical Computer Science* 410, 46 (2009), 4747–4768.

[36] Raymond McDowell and Dale Miller. 2000. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science* 232 (2000), 91–119. https://doi.org/10.1016/S0304-3975(99)00171-1

[37] Raymond McDowell and Dale Miller. 2002. Reasoning with Higher-Order Abstract Syntax in a Logical Framework. *ACM Trans. on Computational Logic* 3, 1 (2002), 80–136.

[38] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. 2003. Encoding transition systems in sequent calculus. *Theoretical Computer Science* 294, 3 (2003), 411–437.

[39] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-driven QuickChecking of compilers. *PACMPL* 1, ICFP (2017), 15:1–15:23. https://doi.org/10.1145/3110259

[40] Dale Miller. 2018. Mechanized Metatheory Revisited. *Journal of Automated Reasoning* (04 Oct. 2018). https://doi.org/10.1007/s10817-018-9483-3

[41] Dale Miller and Gopalan Nadathur. 2012. *Programming with Higher-Order Logic*. Cambridge University Press.

[42] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. 1991. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic* 51 (1991), 125–157.

[43] Dale Miller and Alexis Saurin. 2006. A Game Semantics for Proof Search: Preliminary Results. *Electr. Notes Theor. Comput. Sci.* 155 (2006), 543–563. https://doi.org/10.1016/j.entcs.2005.11.072

[44] Dale Miller and Alwen Tiu. 2005. A proof theory for generic judgments. *ACM Trans. on Computational Logic* 6, 4 (Oct. 2005), 749–783.

[45] Alberto Momigliano. 2012. A supposedly fun thing i may have to do again: a HOAS encoding of Howe's method. In *Proceedings of the seventh international workshop on Logical frameworks and meta-languages, theory and practice (LFMTP '12)*. ACM, New York, NY, USA, 33–42. http://doi.acm.org/10.1145/2364406.2364411

[46] Gopalan Nadathur and Frank Pfenning. 1992. The type system of a higher-order logic programming language. In *Types in Logic Programming*, Frank Pfenning (Ed.). MIT Press, 245–283.

[47] Roberto Blanco Martínez. 2017. *Applications of Foundational Proof Certificates in theorem proving.* Ph.D. Dissertation. Université Paris-Saclay.

[48] Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, Waikiki, Honolulu, HI, USA, May 23-24, 2011*, Antonia Bertolino, Howard Foster, and J. Jenny Li (Eds.). ACM, 91–97. https://doi.org/10.1145/1982595.1982615

[49] Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Proceedings (Lecture Notes in Computer Science)*, Christian Urban and Xingyuan Zhang (Eds.), Vol. 9236. Springer, 325–343.

[50] Giridhar Pemmasani, Hai-Feng Guo, Yifei Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. 2004. Online Justification for Tabled Logic Programs. In *Functional and Logic Programming*, Yukiyoshi Kameyama and Peter J. Stuckey (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–38.

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

[52] Brigitte Pientka and Joshua Dunfield. 2010. Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description). In *Fifth International Joint Conference on Automated Reasoning (Lecture Notes in Computer Science)*, J. Giesl and R. Hähnle (Eds.). 15–21.

[53] A. M. Pitts. 1997. Operationally Based Theories of Program Equivalence. In *Semantics and Logics of Computation*, P. Dybjer and A. M. Pitts (Eds.).

[54] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In *Haskell*. ACM, 37–48.

[55] Tom Schrijvers, Bart Demoen, Markus Triska, and Benoit Desouter. 2014. Tor: Modular search with hookable disjunction. *Sci. Comput. Program.* 84 (2014), 101–120. https://doi.org/10.1016/j.scico.2013.05.008

[56] Peter Schroeder-Heister. 1993. Definitional Reflection and the Completion. In *Proceedings of the 4th International Workshop on Extensions of Logic Programming*, R. Dyckhoff (Ed.). Springer-Verlag LNAI 798, 333–347.

[57] Peter Schroeder-Heister. 1993. Rules of Definitional Reflection. In *8th Symp. on Logic in Computer Science*, M. Vardi (Ed.). IEEE Computer Society Press, IEEE, 222–232. https://doi.org/10.1109/LICS.1993.287585

[58] Alwen Tiu and Dale Miller. 2010. Proof Search Specifications of Bisimulation and Modal Logics for the π-calculus. *ACM Trans. on Computational Logic* 11, 2 (2010), 13:1–13:35. https://doi.org/10.1145/1656242.1656248