Property-Based Testing via Proof Reconstruction

ABSTRACT

Property-based testing is a technique for validating code against an executable specification by automatically generating test data. From its original use in programming languages, this technique has now spread to major proof assistants to complement theorems proving with a preliminary phase of conjecture testing. We present a proof-theoretical reconstruction of this style of testing for relational specifications in proof-theoretic terms. For example, specifications such as

\[ \forall x. \exists y. \left( P(x) \rightarrow Q(y) \right) \]

can be viewed as a set of first-order Horn clauses: one of these formulas would be the universal closure of

\[ \forall \alpha. \forall \beta. \forall \gamma. \left( \alpha \left( X, Y, Z \right) \rightarrow \beta \left( X, Y, Z \right) \right) \]

The proof search approach to encoding Horn clause computation results in the structuring of proofs with repeated switchings between a goal-reduction phase and a backchaining phase [19]. The notion of focused proof systems generalizes this view of proof construction in the sense that goal-reduction corresponds to the negative phase: during this phase, the conclusion-to-premise construction of proofs proceeds without needing to make any choices (no backtracking). At the same time, the backchaining phase corresponds to the positive phase: during this phase, proof construction generally needs to consume some information from, say, an oracle or to allow for some nondeterminism. The combination of a positive phase and a negative phase is called a bipole. In this view of logic programming, proof search involves proofs with arbitrary numbers of bipoles. Comprehensive focusing systems exist for linear, intuitionistic and classical logics [15].

A different approach to the proof theory of Horn clauses involves encoding them as fixed points. For example, the Prolog-style specifications above of \( \text{nat} \) and \( \text{app} \) can be written instead as the following fixed point definitions.

\[ \text{nat} = \mu A. \lambda n. \left( n = z \lor \exists n'. \left( n = s \cdot n' \land n' + n = z \right) \right) \]

\[ \text{app} = \mu A. \lambda x. \lambda y. \lambda z. \left( \left(x = n \land y = z\right) \lor \exists s. \left( s = n \land x = s \land y = z \lor z = \text{cns} \cdot x \cdot y \cdot s \land \text{app} = \text{app} \cdot x \cdot y \cdot z \right) \right) \]

When using a focused proof system for logic extended with fixed points, such as is employed in Bedwyr [2] and described in [1, 13], proofs of formulas such as \( \exists x. \left( \tau \left( P(x) \land -Q(x) \right) \right) \) are a single bipole:
when reading a proof bottom up, a positive phase is followed on all its premises by a single negative phase that completes the proof. In particular, the positive phase corresponds to the generation phase and the negative phase corresponds to the testing phase. From this description, it is conceptionally easy (as one would expect) to construct an implementation of the testing phase while it can be difficult to steer the generation phase through a (possibly) great deal of nondeterminism. For example, the blind exhaustive enumeration of possible counterexamples is generally known to be ineffective. Significant sophistication may go into crafting generators and assembling them.

1.2 Flexible test case generation via proof reconstruction

The foundational proof certificate (FPC) framework was proposed in [9] as a means of defining proof structures used in a range of different theorem provers (e.g., resolution refutations, Herbrand disjuncts, tableaux, etc). The FPC framework was designed using focused proof systems as a kind of protocol: during the construction of a positive phase, the proof checker could request specific information from a proof certificate. In the general setting, proof certificates do not need to contain all the details required to complete a formal proof. In those cases, a proof checker would need to perform proof reconstruction. For example, FPCs can be used as proof outlines [5] since they can describe some of the general shape of a proof: e.g., apply the obvious induction invariant and complete the proof via the enumeration of all remaining cases. The proof checker would attempt to fill in the missing details, either obtaining a proof of the described shape or failing to do so.

In this paper, we propose to use FPCs as a language for describing generators. We have experimented with writing proof checkers in both OCaml (as an extension to Abella [3]) and λProlog, which could be used to check proof certificates and in the process steer the proof of the expression \( P(x) \), and the corresponding typing expression, say, \( \tau(x) \).

As we shall illustrate, we have defined certificates that describe families of proofs that are limited either by the number of inference rules that they contain, by their height, or by both. Using similar techniques, it is possible to define FPCs that target specific types for specific treatment: for example, when generating integers, only (user-defined) small integers would be produced. Using a proof reconstructing checker (such as is easy to do with a logic programming system), the search space of proofs that a FPC describes for a specific formula of the form \( \exists x \left( \tau(x) \land P(x) \land \neg Q(x) \right) \) can be directly translated into a description of the range of possible witness terms for this quantifier.

1.3 Lifting PBT to treat \( \lambda \)-tree syntax

Describing a computational task using proof theory often allows researchers to lift descriptions based on first-order (algebraic) terms to descriptions based on \( \lambda \)-tree syntax (a specific approach to higher-order abstract syntax). For example, once logic programming was given a proof search description, it was natural to generalize the usual approaches to logic programming from the manipulation of first-order terms (Prolog) to the manipulation of \( \lambda \)-terms (\( \lambda \)Prolog) [17]. Similarly, once certain model checking and inductive theorem provers were presented using sequent calculus in a first-order logic with fixed points [1, 13], it was possible to incorporate \( \lambda \)-terms syntax in generalizations of model checkers, as in the Bedwyr system [2], and of theorem provers, as in Abella [3].

The full treatment of \( \lambda \)-tree syntax in a logic with fixed points is usually accommodated with the addition of the \( \forall \)-quantifier [12, 18]. While the \( \forall \)-quantifier has had significant impact in several reasoning tasks (for example, in the formalized metatheory of the \( \pi \)-calculus and \( \lambda \)-calculus) an important result about \( \forall \) is the following: if fixed point definitions do not contain implications and negations, then exchanging occurrences of \( \forall \) and \( \exists \) does not affect what atomic formulas are proved [18, Section 7.2]. Since we shall be limiting ourselves to Horn-like recursive definitions, the \( \lambda \)Prolog implementation of \( \forall \) will also implement \( \exists \).

This direct treatment of \( \lambda \)-terms within the PBT setting allows us to apply property-based testing to a number of metaprogramming tasks. After describing some more details of how PBT can be encoded in proof theory (and logic programming) in the next section, we discuss in Section 3 the treatment of metaprogramming.

2 BASIC APPROACH

The setup follows [16]: we introduce a simple specification logic, which drives the derivation of our object logic. In this case it is basically the usual Prolog vanilla meta-interpreter, save for interpreting \( \forall \) as \( \Pi \); the “program” is represented as Horn-like clauses by a two-place predicate \( \text{prog} \) relating heads and bodies, built out of object-level logical constants (\( \text{tt} \), \( \text{or} \), and, \( \text{nabla} \)) and user-defined constructors for predicates. For example, to generate lists of \( \text{as} \) and \( \text{bs} \) and compute the reverse a list, we have the following \( \text{prog} \) clauses, where we omit the code for append:

\[
\text{prog} \left( \text{is_elt} \ a \right) \ \text{tt}.
\text{prog} \left( \text{is_elt} \ b \right) \ \text{tt}.
\text{prog} \left( \text{is_eltlist} \ n1 \right) \ \text{tt}.
\text{prog} \left( \text{is_eltlist} \ \left( \text{cns} \ X \ Xs \right) \right).
\quad \left( \text{and} \left( \text{is_elt} \ X \right) \left( \text{is_eltlist} \ Xs \right) \right).
\text{prog} \left( \text{rev} \ n1 \ n1 \right) \ \text{tt}.
\text{prog} \left( \text{rev} \ \left( \text{cns} \ X \ Xs \right) \ Rs \right).
\quad \left( \text{and} \left( \text{rev} \ X \ Sx \right) \left( \text{append} \ Sx \ \left( \text{cns} \ X \ n1 \right) \ Rs \right) \right).
\]

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 up to a certain bound, in this case the height of a proof of being a list. The testing phase performs deterministic computation with the meta-interpreter \( \text{interp} \) and then negates the conclusion using negation-as-failure (NAF):

\[
\text{celexv} \ Xs \ Ys \ :- \ \text{check} \ \left( \text{qgen} \ \left( \text{qheight} \ 3 \right) \right) \left( \text{is_eltlist} \ Xs \right),
\quad \text{interp} \ \left( \text{rev} \ Xs \ Ys \right), \ \text{not} \ \left( \text{Xs} = Ys \right) \text{.}
\]

Note that the call to NAF is safe since, by the totality of \( \text{rev} \), \( Ys \) will be ground.

The FPC kernel is presented in Figure 1. Each object-level connective is interpreted as \( \lambda \)Prolog code, and user-defined constructors are looked up in \( \text{prog} \) and unfolded. This is driven by the meta-interpreter \( \text{interp} \) (omitted). To it, check adds a certificate term and calls to \text{expert} predicates on said term (except \text{nabla}, which is transparent to the experts). Experts decide when the computation proceeds — producing certificates for the continuations — and when...
check Cert tt :- tt_expert Cert.
check Cert (and G1 G2) :- and_expert Cert G1 Cert2, check Cert G1, check Cert G2.
check Cert (or G1 G2) :- or_expert Cert G LR, ((LR = left, check Cert G1); (LR = right, check Cert G2)).
check Cert (nabla G) :- pi x\ check Cert (G x).
check Cert A :- unfold_expert Cert', prog A G, check Cert G.

\[
\begin{align*}
tt\_expert & \quad \text{(agen (qsize In In)).} \\
\text{tt\_expert} & \quad \text{(agen (qheight _)).} \\
\text{or\_expert} & \quad \text{(agen (qsize In Out)) (agen (qsize In Out)) _.} \\
\text{or\_expert} & \quad \text{(agen (qheight H)) (agen (qheight H)) _.} \\
\text{and\_expert} & \quad \text{(agen (qsize In Out)) (agen (qsize In Mid)) (agen (qsize Mid Out)).} \\
\text{and\_expert} & \quad \text{(agen (qheight H)) (agen (qheight H)) (agen (qheight H)).} \\
\text{unfold\_expert} & \quad \text{(agen (qsize In Out)) (agen (qsize In’ Out)) :- In > 0, In’ is In - 1.} \\
\text{unfold\_expert} & \quad \text{(agen (qheight H)) (agen (qheight H’)) :- H > 0, H’ is H - 1.}
\end{align*}
\]

Figure 1: Kernel for expert-driven term generation

it fails. The first argument of an expert, e.g., and_expert, refers to the conclusion of the corresponding rule and the remaining ones, if any, to the premises. Here the complexity of generated candidates is bound by limiting unfoldings, either by height (qheight, producing shallow terms), number of constructors (qsize, producing small terms), or both by pairing (not shown here, but see [6]).

3 PBT FOR METAPROGRAMMING

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 | \(A, B\) ::= \(\text{int} | \text{list} | A \rightarrow B\) |
|-------|------------------------------------------------|
| Terms | \(M\) ::= \(x | \lambda x : A. M | M_1 M_2 | c | \text{err}\) |
| Constants | \(c\) ::= \(n | \text{plus} | \text{nil} | \text{cons} | \text{hd} | \text{tl}\) |
| Values | \(V\) ::= \(c | \lambda x : A. M | \text{plus} V | \text{cons} V | \text{cons} V_1 V_2\) |

The rules for the dynamic and static semantics are given in Figure 2, where the latter assumes a signature \(\Sigma\) with the obvious type declarations for constants. Rules for \(\text{plus}\) are omitted for brevity.

The encoding 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 \(\text{err}\) expression, used to model run time errors such as taking the head of an empty list. We follow this up (see the bottom of Figure 2) with the static semantics (predicate \(\text{wt}\)), where constants are typed via a table \(\text{tcc}\). Note that we have chosen an explicitly contexted encoding of typing as opposed to one based on hypothetical judgments such as in [16]: 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 reasoning level \(V\)-quantifier.

Now, this calculus enjoys the usual property of subject reduction 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 a theorem prover such as Abella. However, the case distinction in the progress theorem does require some care: were it to be unprovable given a mistake in the specification, it would not be immediate to localize where the problem may be. On the other hand, one could wonder whether our calculus enjoys the \textit{subject expansion} property — the alert reader will undoubtedly realize that this is highly unlikely, but rather than wasting time in a proof attempt, we search for a counterexample and find:

\[
\begin{align*}
\text{cexsexp} M M' A & :- \text{agen (qsize 8 _)} (\text{step } M M'), \text{interp (wt null } M') \text{,}
\text{not (interp (wt null } M)).
\end{align*}
\]

\(A = \text{listTy}\)
\(M' = c\ \text{nl}\)
\(M = \text{app } (c\ \text{hd}) (\text{app } (c\ \text{cns}) (c\ \text{nl})) (c\ _))\)

Other queries we can ask: are there \textit{untypable} terms, or terms that do not converge to a value?

As a more comprehensive validation we addressed the nine mutations proposed by the PLT-Redex benchmark, to be spotted as a violation of either the 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:

\[
\frac{\Gamma \vdash x : M_1 : A \rightarrow B \quad \Gamma \vdash y : M_2 : B}{\Gamma \vdash x y : M_1 M_2 : B} \quad \text{T-APP-B1}
\]

The given mutation makes both properties fail:

\[
\begin{align*}
\text{cexprog} M A & :- \text{agen (qsize 6 _)} (\text{wt null } M A),
\text{not (interp (progress } M)).
\end{align*}
\]

\(A = \text{intTy}\)
\(M = \text{app } (c\ \text{hd}) (c\ (\text{toInt } \text{zero}))\)

\[
\begin{align*}
\text{cexprs} M M' A & :- \text{agen (qsize 8 _)} (\text{wt null } M A), \text{interp (step } M M'),
\text{not (interp (wt null } M').
\end{align*}
\]

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

Table 1 reports the tests, performed under Ubuntu 16.04 on a Intel Core i7-870 CPU, 2.93GHz with 8GB RAM. We time-out the computation when it exceeds 300 seconds. We list the results obtained by \(\lambda\)Prolog (\(\lambda P\)) under Teyjus [20], the counterexample found, and a brief description of the bug together with Redex’s difficulty rating (shallow, medium, unnatural). The column \(\alpha\)C lists the time taken by \(\alpha\)Check [7] using NAF, which is not always the best technique [8],
but it corresponds very closely to the architecture of the present paper. Of course, αCheck sits on top of an interpreted (prototype) language, whereas Teyjus is a compiler: however, one can argue that the two level themselves out, since we use meta-interpretation for test generation. The results are essentially indistinguishable, save for bugs 4, 5 and 6: in the first, which is surprisingly hard to find, αCheck times out, while we comfortably beat the time limit. From its original version, testing à la QuickCheck; its usefulness has been demonstrated in several impressive case studies [14]. However, Redex has limited support for relational specifications and none whatsoever for binding signature. This is where αCheck [7] comes in. The tool adds on top of the nominal logic programming language αProlog a checker for relational specifications as we do here. One of the implementation techniques is based as well on NAF, as far as testing of the testing signature. This is where our approach stands out. Finally in bug 6 αCheck’s fixed integrative deepening strategy needs to explore the search space up to level 11, while we can leverage the FPC ability to use the qsize metric.

4 RELATED WORK

Property-based testing is a technique for validating code against an executable specification by automatically generating test-data, typically in a random and/or exhaustive fashion. From its original use in programming languages [10], this technique has now spread to most major proof assistants [4, 22] to complement theorem proving with a preliminary phase of conjecture testing. We do not have the space for a comprehensive review, for which we refer to [7], but we mention two of the main players w.r.t. metatheory model checking: PLT-Redex [11] is 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 [14]. However, Redex has limited support for relational specifications and none whatsoever for binding signature. This is where αCheck [7] comes in. The tool adds on top of the nominal logic programming language αProlog a checker for relational specifications as we do here. One of the implementation techniques is based as well on NAF, as far as testing of the testing signature. This is where our approach stands out. Finally in bug 6 αCheck’s fixed integrative deepening strategy needs to explore the search space up to level 11, while we can leverage the FPC ability to use the qsize metric.

5 CONCLUSION AND FUTURE WORK

We have described some work-in-progress that uses standard logic programming techniques and some recent developments in proof theory to design a flexible framework for PBT. Given its proof theoretic pedigree, it was immediate to extend PBT to the metaprogramming setting.
Figure 1 specifies only two certificate formats: one that limits the size and one that limits the height of a proof. We have also implemented another certificate format that implements both restrictions at the same time. It is easy to code other certificates: by reading random bits from an external source of entropy, certificates can describe randomly organized proofs (and, hence, witness terms). Certificates can also be organized to consider only allowing small proofs for one type but random for another type: thus, one could easily design a certificate that would explore randomly generated lists containing just, say, the integers 0 and 1.

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 fact, we plan to integrate our take on PBT in Abella, in order to support both proofs and disproofs.

Acknowledgments The work of Blanco and Miller was funded by the ERC Advanced Grant ProofCert.

REFERENCES