Property-Based Testing via Proof Reconstruction: Work-in-progress

Roberto Blanco  Dale Miller
INRIA Saclay — Ile-de-France & LIX/Ecole polytechnique Palaiseau, France
{roberto.blanco,dale.miller}@inria.fr

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

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 most major proof assistants to complement theorem proving with a preliminary phase of conjecture testing. We present a proof theoretical reconstruction of this style of testing for relational specifications (such as those used in the semantics of programming languages) and employ the Foundational Proof Certificate framework to aid in describing test generators. We do this by presenting certain kinds of "proof outlines" that can be used to describe the shape and size of the generators for the conditional part of a proposed property. Then the testing phase is reduced to 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-Redux.

1. Introduction

In this brief paper, we examine property-based testing (PBT) from a proof theory point-of-view and explore some of the advantages that result from exploiting that perspective.

1.1 Generate-and-test as bipoles

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

\[ \forall L: \text{list} \forall R: \text{list} \ [ \text{rev}(L, R) \supset \text{rev}(R, L)] \]

stating that rev is idempotent should be a theorem. More generally, we might wish to prove a number of formulas of the form \( \forall x : \tau [P(x) \supset Q(x)] \) where both \( P \) and \( Q \) are given relational specifications. Occasionally, it can be important in this setting to move the type judgment \( x : \tau \) into the logic of a formula by turning the type into a predicate: \( \forall x [ (\tau(x) \land P(x)) \supset Q(x)] \). Proving such formulas can often be difficult since their proof may involve the clever invention of prior lemmas and induction invariants. In many practical settings, such formulas are, in fact, not theorems since the relational specification of \( P \) and/or \( Q \) can contain errors. It can be therefore valuable to first attempt to find counterexamples to such formulas prior to attempting a proof. That is, we might attempt to prove formulas of the form \( \exists x [(\tau(x) \land P(x))] \land \neg Q(x) \) instead. If a term \( t \) of type \( \tau \) can be discovered such that \( P(t) \) holds while \( Q(t) \) does not, then one can return to the specifications of \( P \) and \( Q \) and revise them using the concrete evidence in \( t \) about how the specifications are wrong. The process of writing and revising relational specifications could be aided if such counterexamples are discovered quickly.

The literature contains at least two ways to view Horn clause-style relational specifications in proof theoretic terms. For example, specifications such as

\[ \text{nat} 0. \quad \text{plus} 0 M P. \]
\[ \text{nat} (s N) :- \text{nat} N. \quad \text{plus} (s N) M (s P) :- \text{plus} N M P. \]

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

\[ \forall N \forall M \forall P [\text{plus} N M P \supset \text{plus} (s N) M (s P)]. \]

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 notion 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 biple. 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 Horn clauses as fixed points. For example, the Prolog-style specification above can be written instead as the following fixed point definitions.

\[ \text{nat} = \mu \lambda N \lambda n (n = 0 \lor \exists n' (n = s n' \land + N n')) \]
\[ \text{plus} = \mu \lambda P \lambda n \lambda m \lambda p (n = 0 \land + m = p) \lor \]
\[ \exists n' \exists p' (n = s n' \land + p = s p' \land + P n' m p') \]

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 \( \exists x : \tau [P(x) \land \neg Q(x)] \) 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 conceptually 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 must 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. Here, 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 that 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 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 generated. 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 \( \exists x : \tau \ P(x) \land \lnot Q(x) \) can be directly translated into a description of the range of possible witness terms for this quantifier.

1.3 Lifting PBT to \( \lambda \)-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 calculi 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 in generalizations 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 \( \exists \) quantifier [12, 18]. While the \( \exists \)-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 \( \exists \) 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 will allow us to apply property-based testing to a number of metaprogramming tasks. After describing 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 in this case is basically the usual Prolog vanilla meta-interpreter, save for interpreting \( \exists \) as \( \Pi \), which drives the derivation of our object logic; the latter is represented as Horn-like clauses by a two-place predicate \( \text{prog} \) relating heads and bodies, built out of object-level logical constants (tt, or, and, nbla) and user-defined constructors for predicates. For example, to generate lists of a and bs and compute the reverse a list, we have the following prog clauses:

\[
\begin{align*}
\text{prog} \ (\text{is_elt} \ a) & \ \text{tt} . \\
\text{prog} \ (\text{is_elt} \ b) & \ \text{tt} . \\
\text{prog} \ (\text{is_eltlist} \ \text{null}) & \ \text{tt} . \\
\text{prog} \ (\text{is_eltlist} \ \text{Xs}) & \ \text{tt} . \\
\text{prog} \ (\text{rev} \ \text{null}) & \ \text{tt} . \\
\text{prog} \ (\text{rev} \ \text{X} \ \text{XS}) & \ \text{tt} . \\
\text{prog} \ (\text{rev} \ \text{XS} \ \text{SX}) & \ \text{tt} .
\end{align*}
\]

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 depth \( \text{qheight} \). The testing phase performs deterministic computation with the meta-interpreter \( \text{interp} \) and then negating the conclusion using negation-as-failure (NAF), where the call to NAF is safe since \( \text{YS} \) will be ground:

\[
\text{cxrev} \ \text{XS} \ \text{YS} := \ \text{check} \ (\text{ggen} \ (\text{qheight} \ 3)) \ (\text{is_eltlist} \ \text{XS}), \ \text{interp} \ (\text{rev} \ \text{XS} \ \text{YS}), \ \text{not} \ (\text{XS} = \text{YS}) .
\]

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 \( \text{nbla} \), which is transparent to the experts). Experts decide when the computation proceeds — producing certificates for the continuations — and when it fails. Here the complexity of generated candidates is bound by limiting unfoldings, either by depth (qheight, producing shallow terms), number of constructors (qsize, producing small terms), or both by \( \text{pairing} \) (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:

\[
\begin{align*}
\text{Types} & \quad A, B \quad ::= \quad \text{int} \mid \text{list} \mid A \to B \\
\text{Terms} & \quad M \quad ::= \quad x \mid \lambda x : A \ M \mid M_1 \ M_2 \mid c \mid \text{err} \\
\text{Constants} & \quad c \quad ::= \quad n \mid \text{plus} \mid \text{nil} \mid \text{cons} \mid \text{hd} \mid t1 \\
\text{Values} & \quad V \quad ::= \quad c \mid \lambda x : A \ M \mid \text{plus} \ V \\
& \quad \mid \text{cons} \ V \mid \text{cons} \ V_1 \ V_2 
\end{align*}
\]
The reasoning level result, allows us to use $\lambda$ judgments such as in [16]: this choice makes it possible to avoid encoding of typing as opposed to one based on hypothetical subject expansion could wonder whether our calculus enjoys the to be unprovable given a mistake in the specification, it would not those results in a theorem prover such as Abella. However, the case or able to make a step.” And in fact we can fairly easily prove and progress, where the latter means “being either a value, an error, characterizes the threading in the operational semantics of the err mutations proposed by the PLT-Redex benchmark, to be spotted untypable as a violation of either the preservation or progress properties. E.g.,

$$M = \text{app} (c\ \text{hd}) (\text{app} (\text{app} (c\ \text{cns}) (c\ \text{nl})) (c\ _))$$

$$M' = \text{c\ nl}$$

interp (wt null M A),
not (interp (wt null M A)).

$$A = \text{listTy}$$

$$M' = \text{c\ n1}$$

$$M = \text{app} (c\ \text{hd}) (\text{app} (\text{app} (c\ \text{cns}) (c\ \text{nl})) (c\ _))$$

Other queries we can ask: are there antotypable 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. E.g., 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:

$$\Gamma \vdash_\Sigma M_1 : A \rightarrow B \quad \Gamma \vdash_\Sigma M_2 : B \quad \Gamma \vdash_\Sigma (M_1 : A) (M_2 : B) : \text{T-APP-B1}$$

The given mutation makes both properties fail:

$$\text{cxexp} M\ M'\ A ::= \text{check} (\text{qgen} (\text{qsize} 8\ _)) (\text{step} M\ M'), \text{interp} (\text{wt null} M\ A), \not\text{interp} (\text{wt null} M\ A)).$$

$$A = \text{funTy\ listTy\ intTy}$$

$$M' = \text{lam} (x\ \text{\c\ hd}) \text{listTy}$$

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

Other tests yield:

$$\text{cxexp} M\ M'\ A ::= \text{check} (\text{qgen} (\text{qsize} 8\ _)) (\text{wt\ null} M\ A), \text{interp} (\text{step} M\ M'), \not\text{interp} (\text{wt\ null} M'\ A)).$$

$$A = \text{intTy}$$

$$M' = \text{c\ n1}$$

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

$$A' = \text{intTy}$$

$$M' = \text{c\ n1}$$

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

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 oCheck [7] comes in. The tool adds on top of the nominal logic programming language oProlog a checker for relational specifications as we do here. One of the implementation techniques is based as well on unification, as far as testing of the conjunction is concerned. The generation phase is instead “wired in” via iterative-deepening search, based on derivation height. In this sense oCheck is less flexible than the FPC-based architecture that we propose here, since it can be seen as a fixed choice of experts.

Figure 1. check is the proof checking kernel: it is parametrized by four experts.
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 possible 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 certificates can also be organized to consider only allowing small proofs and disproofs.

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.

Finally, more distant cousins in the logic programming world are declarative debugging [21] and the Logic-Based Model Checking project at Stony Brook (http://www.cs.sunysb.edu/~lmc).

References


