Property-Based Testing via Proof Reconstruction

Work-in-progress

Alberto Momigliano
joint work with Rob Blanco and Dale Miller

LFMTP17

Sept. 8, 2017

Off the record

- After almost 20 years of formal verification with Twelf, Isabelle/HOL, Coq, Abella, I'm a bit worn out
- I still find it a very demanding, often frustrating, day job.

Off the record

- After almost 20 years of formal verification with Twelf, Isabelle/HOL, Coq, Abella, I'm a bit worn out
- I still find it a very demanding, often frustrating, day job.
- Especially when the theorem I'm trying to prove is, ehm, wrong. I mean, *almost right*:

Off the record

- After almost 20 years of formal verification with Twelf, Isabelle/HOL, Coq, Abella, I'm a bit worn out
- I still find it a very demanding, often frustrating, day job.
- Especially when the theorem I'm trying to prove is, ehm, wrong. I mean, *almost right*:
  - statement is too strong/weak
  - there are minor mistakes in the spec I'm reasoning about
- A failed proof attempt not the best way to debug those kind of mistakes
- That's why I'm inclined to give *testing* a try (and I'm in good company!)

Off the record

- After almost 20 years of formal verification with Twelf, Isabelle/HOL, Coq, Abella, I'm a bit worn out
- I still find it a very demanding, often frustrating, day job.
- Especially when the theorem I'm trying to prove is, ehm, wrong. I mean, *almost right*:
    - statement is too strong/weak
    - there are minor mistakes in the spec I'm reasoning about
- A failed proof attempt not the best way to debug those kind of mistakes
- That's why I'm inclined to give *testing* a try (and I'm in good company!)
- Not any testing: **property-based testing**

PBT

- A light-weight validation approach merging two well known ideas:
  1. automatic generation of test data, against
  2. executable program specifications.
- Brought together in *QuickCheck* (Claessen & Hughes ICFP 00) for Haskell
- The programmer specifies properties that functions should satisfy
- QuickCheck tries to falsify the properties by trying a large number of **randomly** generated cases.

QuickCheck's Hello World! (FsCheck, actually)

```
let rec rev ls =
    match ls with
    | [] -> []
    | x :: xs -> append (rev xs, [x])

let prop_revRevIsOrig (xs:int list) =
    rev (rev xs) = xs;;

do Check.Quick prop_revRevIsOrig ;;
>> Ok, passed 100 tests.

let prop_revIsOrig (xs:int list) =
    rev xs = xs
do Check.Quick prop_revIsOrig ;;

>>  Falsifiable, after 3 tests (5 shrinks) (StdGen (518275965,..
[1; 0]
```

Not so fast/quick...

- Sparse pre-conditions:

  ```
  ordered xs ==> ordered (insert x xs)
  ```

- Random lists not likely to be ordered ... Obvious issue of *coverage*
- QC's answer:
  - monitor the distribution
  - write your own generator (here for ordered lists)
  - *Quis custodiet ipsos custodes?*
    - Generator code may overwhelm SUT. Think red-black trees.
  - We need to shrink random cex to understand them. So, with generators we need to implement (and trust) shrinkers
- Exhaustive generation up to a bound may miss corner cases
- Huge literature we skip, since...

From programming to mechanized meta-theory

- . . . We are interested in the specialized area of *mechanized meta-theory*
- Yet, even here, verification still is
  - lots of work (even if you're not burned out)!
  - unhelpful if system has a bug — only worthwhile if we already "know" the system is correct, not in the design phase!

From programming to mechanized meta-theory

- . . . We are interested in the specialized area of *mechanized meta-theory*
- Yet, even here, verification still is
  - lots of work (even if you're not burned out)!
  - unhelpful if system has a bug — only worthwhile if we already "know" the system is correct, not in the design phase!
- (Partial) "model-checking" approach to the rescue:
  - searches for counterexamples
  - produces helpful counterexamples for incorrect systems
  - unhelpfully diverges for correct systems
  - little expertise required
  - fully automatic, CPU-bound

From programming to mechanized meta-theory

- . . . We are interested in the specialized area of *mechanized meta-theory*
- Yet, even here, verification still is
  - lots of work (even if you're not burned out)!
  - unhelpful if system has a bug — only worthwhile if we already "know" the system is correct, not in the design phase!
- (Partial) "model-checking" approach to the rescue:
  - searches for counterexamples
  - produces helpful counterexamples for incorrect systems
  - unhelpfully diverges for correct systems
  - little expertise required
  - fully automatic, CPU-bound
- **PBT** for MMT means:
  - Represent object system in a logical framework.
  - Specify properties it should have.
  - System searches (exhaustively/randomly) for counterexamples.
  - Meanwhile, user can try a direct proof (or go to the pub)

Testing and proofs: friends or foes?

- Isn't testing the very thing theorem proving want to replace?
- Oh, no: test a conjecture before attempting to prove it and/or test a subgoal (a lemma) inside a proof
- The beauty (wrt general testing) is: you don't have to invent the specs, they're exactly what you want to prove anyway.
- In fact, when Isabelle/HOL broke the ice adopting *random* testing some 15 years ago, many followed suit:
    - a la QC: Agda (04), PVS (06), Coq with QuickChick (15)
    - exhaustive/smart generators (Isabelle/HOL (12))
    - model finders (Nitpick, again in Isabelle/HOL (11))
- In fact, Pierce and co. are considering a version of *Software Foundations* where proofs are completely replaced by testing!

Where is the logic (programming)?

- Given the functional origin of PBT, the emphasis is on executable specs and this applies as well to PBT tools for PL (meta)-theory (PLT-Redex, Spoofax).
- QuickChick and Nitpick handle some **inductive** definitions, QC by deriving generators that satisfy essentially for logic programs, for N. by reduction to SAT problems. . .
- An exception is αCheck, a PBT tool on top of αProlog, using nominal Horn formulas to write specs and checks
- Given a spec $\text{И}\vec{a}\forall\vec{X}.A_1 \wedge \cdots \wedge A_n \supset A$, a *counterexample* is a ground substitution θ s.t. $\mathcal{M} \models \theta(A_1) \wedge \cdots \wedge \mathcal{M} \models \theta(A_n)$ and $\mathcal{M} \not\models \theta(A)$ for model $\mathcal{M}$ of a (pure) nominal logic program.
- Two forms of negation: negation as failure and negation elimination
- System searches exhaustively for counterexamples with a fixed iterative deepening search strategy

What lies beneath

- In fact, functional approaches to PBT are rediscovering logic programming:
    - Unification/mode analysis in Isabelle's *smart generators* and in Coq's QC
    - (Randomized) backchaining in PLT-Redex
- What the last 25 years has taught us is that if we take a proof-theoretic view of LP, good things start to happen
- And this now means focusing in a sequent calculus.
- In a nutshell, the (unsurprising) message of this paper: the generate-and-test approach of PBT can be seen in terms of focused sequent calculus proof where the positive phase corresponds to generation and a single negative one to testing.

μMALL

- As the plan is to have a PBT tool for **Abella**, we have in mind specs and checks in multiplicative additive linear logic with (for the time being) least fixed points (Baelde & Miller)
- E.g. , the `append` predicate is:

$$app \equiv \mu\lambda A\lambda xs\lambda ys\lambda zs\ (xs = \mathbf{nl} \wedge^{+} ys = zs) \vee \\ \exists x'\exists xs'\exists zs'(xs = \mathbf{cns}\ x'\ xs' \wedge^{+} zs = \mathbf{cns}\ x'\ zs' \wedge^{+} A\ xs'\ ys\ zs'$$

- Usual polarization for LP: everything is positive — note, no atoms.
- Searching for a cex is searching for a proof of a formula like $\exists x{:}\ \tau\ [P(x) \wedge^{+} \neg Q(x)]$ is a single bipole — a positive phase followed by a negative one.
- Correspond to the intuition that generation is hard, testing a deterministic computation

A further step: FPC

- A flexible and general way to look at those proofs is as a proof reconstruction problem in Miller's Foundational Proof Certificate framework
- FPC proposed as a means of defining proof structures used in a range of different theorem provers
- If you're not familiar with it, think a focused sequent calculus augmented with predicates (**clerks** for the negative phase and **experts** for the positive one) that produce and process information to drive the checking/reconstruction of a proof.
- For PBT, we suggest a lightweight use of FPC as a way to describe generators by fairly simple-minded experts.

FPC for the common man

- We defined certificates for families of proofs (the generation phase) limited either by the number of inference rules that they contain, by their size, or by both.
- They essentially translate into meta-interpreters that perform bounded generation, not only of terms but of derivations.
- As a proof of concept, we implement this in λProlog and we use *NAF* to implement negation — it's a shortcut, but theoretically, think fixed point and negation as $A \rightarrow \perp$.
- We use the two-level approach: OL specs are encoded as `prog` clauses and a `check` predicates will meta-interpret them using the size/height certificates to guide the generation.
- Checking $\forall x{:}elt, \forall xs, ys{:}eltlist\ [rev\ xs\ ys \rightarrow xs = ys]$ is

```
cexrev Xs Ys :-
    check (qgen (qheight 3)) (is_eltlist Xs), % generate
    solve (rev Xs Ys), not (Xs = Ys).         % test
```

From algebraic to binding signatures

- The proof-theoretic view allows us to move seamlessly from standard first-order terms to higher-order LP with λ-tree syntax, which was the whole selling point.
  - No current tool supports proofs **and** disproofs with binders
- This means accommodating the ∇-quantifier
- Here we take another shortcut and restrict to Horn specs (no hypothetical encodings).
  - ... but we have experimented with kernels for logics such LG as well
- It's well known that in this setting nabla can be soundly encoded by λProlog's universal quantification

Case study

- A simply-typed λ-calculus with constructors for integers and lists, following a PLT-Redex benchmark:

$$
\begin{array}{llcl}
\text{Types} & A, B & ::= & \mathit{int} \mid \mathit{ilist} \mid A \rightarrow B \\
\text{Terms} & M & ::= & x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \mid c \mid \mathit{err} \\
\text{Constants} & c & ::= & n \mid \mathit{plus} \mid \mathit{nil} \mid \mathit{cons} \mid \mathit{hd} \mid \mathit{tl}
\end{array}
$$

- Encode it in the usual two-level approach, but with explicit contexts (to stay Horn).
- Insert a bunch of mutations in the static and/or dynamic semantics
- Try to catch them as a violation of type safety

Measurements

bug	check	αC	λP	Description/Rating
1	preservation	0.3	0.05	range of function in app rule
	progress	0.1	0.02	matched to the arg. (S)
2	progress	0.27	0.06	value *(cons v) v* omitted (M)
3	preservation	0.04	0.01	order of types swapped
	progress	0.1	0.04	in function pos of app (S)
4	progress	t.o.	207.3	the type of cons return *int* (S)
5	preservation	t.o.	0.67	tail reduction returns the head (S)
6	progress	24.8	0.4	hd reduction on part. applied cons (
7	progress	1.04	0.1	no eval for argument of app (M)
8	preservation	0.02	0.01	lookup always returns int (U)
9	preservation	0.1	0.02	vars do not match in lookup (S)

Our implementation using size as a bound vs. α*Prolog*

Conclusions

- PBT is now most major proof assistants to complement theorem proving with a preliminary phase of conjecture testing.
- We have shown as the FPC framework can be instantiated to give a proof-theoretic reconstruction of PBT.
- We have seen as this extends as expected to binding signature to perform meta-theory model-checking.
- We have presented a proof-of-concept implementation in λProlog using *NAF*, which, in its naivety, is already effective.

Future Work: more case studies

- Search for deeper known bugs
  - "value" restriction in ML with references and let-polymorphous
  - intersection types with computational effects
- Search for unknown bugs in (λ)Prolog code "in the wild" (e.g. Hannan's "Extended natural semantics" or even old CENTAUR stuff)
- Tackle coinductive specs, to look for
  - Two process that are similar but **not** bisimilar
  - λ-terms that are ground- but **not** applicative-bisimilar...

  Tabling could prove handy.
- Implement random generators e.g. with an `unfold` expert that may flip a coin when selecting a clause to backchain on.

Future Work: architecture

- Integrate with Abella's workflow, both at the top-level (disproving conjectures) and inside a proof attempt (disproving subgoals).
- Long-ish time view: a mini Sledgehammer protocol for Abella, by which conjectures are under the hood PB-tested: if no cex reported proof outlines are used to try and conclude the proof.
- Keeping in mind that Abella's implementation **not** immediately meant for search
- Previous attempts with FPC kernels with primitive ∇ written as inductive definition in Abella proper seems too slow for generation

The blame game

- Suppose your PBT tool reports a cex. Now what? You're not getting payed just for finding faults...
- Staring at a potentially huge spec even with a cex in hand not the best way to go. Two issues:
  1. Soundness: your spec is plain wrong and returns an answer that should not hold
  2. Completeness: you've forgotten to encode some info and some answers are not produced.
- FPC to the rescue (possibly):
  1. Use certificate distillation to restrict to a more manageable set of suspects
  2. Use abductive experts to collect sets of assumptions that should hold but don't

Thanks!