Property-Based Testing to Infinity and beyond!

Alberto Momigliano
joint work with Roberto Blanco and Dale Miller

LFMTP 2020

June 29, 2020
Take away from the talk

- Standard PBT is just search in a focused sequent calculus akin to uniform proofs
- PBT’s data generation strategies (exhaustive, random) and other features (shrinking, bug provenance) can be programmed as instantiations of the Foundational Proof Certificates paradigm.
- Co-PBT requires a stronger logic with fixed points, preferably linear.

Now you can go (back) to sleep.
Property-based Testing

- A light-weight validation approach to software correctness
- The programmer specifies executable properties the code should satisfy
- PBT tries to refute them by trying a large number of automatically generated cases.
- Brought together in *QuickCheck* (Claessen & Hughes ’00) for Haskell . . .
- . . . and now available for pretty much every PL (and commercialized for Erlang).
Does insertion in an ordered list preserve order?

let rec ordered = function
  |[] -> true
  |[x] -> true
  |x::y::ys -> x <= y && ordered ys

let rec insert x = function
  |[] -> [x]
  |c::cs ->
    if x <= c then x::c::cs else c::insert x cs

let prop_insert (x:int, xs) =
  ordered xs ==> ordered (insert x xs)

do Check.Quick prop_insert

Falsifiable, after 16 tests ... (0, [0; 0; -1])
Well, it would, had I the correct encoding of ordered

```ocaml
let rec ordered = function
  |[]  -> true
  |[x] -> true
  |x::y::ys -> x <= y && ordered (y :: ys)

// was ordered ys
// rest stays the same
```

do Check.Quick prop_insert

OK. Arguments exhausted after 37 tests.

Since this is a conditional property with a sparse pre-condition, it warns me of bad coverage, but this is another story...
One of PBT’s success stories is the integration with proof assistants

Isabelle/HOL broke the ice adopting random testing some 15 years ago

- 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])

But, wait! Isn’t testing the very thing theorem proving wants 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.
Consider the verification of the meta-theory of programming languages and related calculi with a proof assistant. Type soundness, proofs by logical relations, non-interference. . .

Proofs are mostly standard, but

- lots of work, mostly wasted in the design phase
- only worthwhile if we already “know” the system is correct

PBT to the rescue:

- produces helpful counterexamples for incorrect systems
- little expertise required, fully (well, sort of) automatic
What about infinite computations?

- I don’t have to argue (but I will) about the relevance of coinduction/corecursion in PL theory:
  - divergence/co-evaluation
  - recursive (sub)typing
  - program equivalence
  - pretty much the whole meta-theory of process calculi
  - … [add your own]
- Lots of work on the **proving** side in most proof assistants
  - guarded recursion, greatest fixed points, co-patterns, co-datatypes …
- Much less on the **testing** side:
  - a little with Haskell’s QC (but basically using the *take* lemma)
  - a little with *Nitpick*, possibly with (co)datatypes, I have to check
  - seems hard for Coq’s QuickChick.
Consider the CBV big step semantics of the lambda calculus, but coinductively [Leroy & Grall '07]:

\[
\begin{array}{c}
\lambda x. M \Downarrow \lambda x. M \\
E-L \\
M_1 \Downarrow \lambda x. M \quad M_2 \Downarrow V_2 \quad M\{V_2/x\} \Downarrow V \\
E-A \\
M_1 \cdot M_2 \Downarrow V
\end{array}
\]

Is this deterministic? Is it type preserving?
No, on both counts:
- a divergent term such as \( \Omega \) co-evaluates to anything.
- a variant of the Y-combinator falsifies preservation [Filinski].

In fact, one can argue that this notion of co-evaluation makes little sense [Ancona et al. '15] and wouldn’t it be nice if your PBT tool would help you with that?
Motivating examples: separating equivalences

- Consider **PCF** with lazy lists and observational equivalence via **bisimilarity** [Pitts ’98]
- There are several notions of equivalence, e.g., depending if you observe convergence at any type (**applicative**) or at base type (**ground**). Can we separate them?
  - \( \lambda x. \bot \) is **ground** but not **applicative** bisimilar to \( \bot \)
  - Same for the eta and surjective pairing laws.
- Similar results in the \( \pi \) calculus:
  - Ground bisimilarity not a congruence
  - Early and late bisimilarity not preserved by inputs
Non-examples

- What about **streams**? All those nice Haskell-like equations, as in Louise Dennis’ thesis?
- Thanks, but no thanks
- Here we concentrate on infinite behavior (e.g., is a finite program diverging) rather than infinite objects (e.g., is 2 the last element of the infinite stream 1 :: 1 :: 1 ... ?).
- In a logical view of coPBT, we would need to **construct** such infinite terms as cex, and the literature is not satisfactory:
  - coinductive LP works with rational terms: problematic and hopelessly incomplete
  - “Coinduction in uniform” uses a fixed point term constructor: does not sit well within a logical framework (adequacy of encodings, canonical forms etc.).
PBT: from FP to LP

- PBT was born and raised functional, but is rediscovering logic programming:
  - mode analysis in Isabelle/HOL and QuickChick’s automatic derivation of generators
  - (Randomized) backchaining in PLT-Redex
  - Narrowing in LazySmallCheck . . .

- What the last 30 years have taught us is that if we take a proof-theoretic view of LP, good things start to happen

- In particular: focusing and a treatment of bindings
Specifications and code (think ordered or more interestingly the operational semantics of a PL) are logical theories.

Trying to refute a property of the form

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

means searching for a (focused) proof of

$$\exists x [ (\tau(x) \land P(x)) \land \neg Q(x)]$$

yielding a a $t$ of type $\tau$ s.t. $P(t)$ holds and $Q(t)$ does not.

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.

Intuition: generating is hard (lots of backtracking), testing is easy (deterministic computation).
Going deeper: PBT via FPC

- A flexible way to look at those proofs is as a proof reconstruction problem in the Foundational Proof Certificate framework [Chihani, Miller & Renaud 2017]
- FPC proposed as a means of defining proof structures used in a range of different theorem provers
- Think of 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, use of FPC as a way to describe generation strategies and as a way to combine them.
Proof system for Horn logic with certificates and experts

- $\Xi_1 \vdash G_1$  $\Xi_2 \vdash G_2$  $\wedge_e(\Xi, \Xi_1, \Xi_2)$  $tt_e(\Xi)$
  $\Xi \vdash G_1 \land G_2$
  $\Xi \vdash tt$

- $\Xi' \vdash G_i$  $\vee_e(\Xi, \Xi', i)$
  $\Xi \vdash G_1 \lor G_2$

- $\Xi' \vdash G[t/x]$  $\exists_e(\Xi, \Xi', t)$
  $\Xi \vdash \exists x. G$

- $=_{e}(\Xi)$
  $\Xi' \vdash G$  $(A : - G) \in grnd(P)$  $unfold_{e}(\Xi, \Xi')$
  $\Xi \vdash t = t$
  $\Xi \vdash A$

- An FPC is an instantiation of $\Xi$ and experts predicates
We capture exhaustive generation by building proofs bounded by their size – many others in [PPDP ’19] paper.

Certificates are pairs of integers and the only active expert is the non-zero check while backchaining:

\[ n, m \vdash G \]

\[
\frac{n, n_1 \vdash G_1 \quad n_1, m \vdash G_2}{n, m \vdash G_1 \land G_2}
\]

\[
\frac{n, n \vdash tt}{n, m \vdash \exists x. G}
\]

\[
\frac{n, m \vdash G_i}{n, m \vdash G_1 \lor G_2}
\]

\[
\frac{n, m \vdash G \quad (A : - G) \in \text{grnd} (P)}{n + 1, m \vdash A}
\]

\[
\frac{n, n \vdash t = t}{n, n \vdash t = t}
\]
From PBT to coPBT

- Standard PBT requires only finite computations and can be accounted for with logic programming (Horn) queries.
- Recall failure of determinism of co-evaluation:
  \[ \exists M V_1 V_2 \ [ \text{istm}(M) \land \text{istm}(V_1) \land \text{istm}(V_2) \land M \Downarrow V_1 \land M \Downarrow V_2 \land V_1 \neq V_2 \] 
- Generation (pred istm) is still Horn and finitary: it will be driven by FPC.
- If you want to capture its infinite behavior of \( M \Downarrow V_i \), you need a proof-theory with rules for fixed points, such as the ones underlying *Abella* and *Bedwyr*.
- Other specs such as bisimilarity are Harrop and will also need proofs by case analysis, which is readily available with fixed points.
Extends multiplicative additive linear logic with fixed points and free equality [Baelde & Miller ’07]:
⇒ co-evaluation is encoded (using λ-tree syntax) as this greatest fixed point (see linear version of Clark’s completion):

\[
\nu(\lambda CE.\lambda m.\lambda m'.(\exists M. m = (\text{fun } M) \otimes m' = (\text{fun } M) \oplus \\
(\exists M_1 M_2 M V_2 V. m = (\text{app } M_1 M_2) \otimes m' = V \otimes (CE M_1 (\text{fun } M)) \\
\otimes (CE M_2 V_2) \otimes (CE (M V_2) V))
\]

Left sequent rule of \(\nu\) is unfolding (case analysis), right is coinduction via simulation:
\( \mu \text{MALL} \) provides a proof-search interpretation for some aspects of model checking, see [Heath & Miller '19] for the finite case.

\( \mu \text{MALL} \) provides a proof-search interpretation for some aspects of model checking, see [Heath & Miller '19] for the finite case.

- **Polarization**: Prolog programs = least fixed points = purely positive. Dually, greatest fixed points:
  - Often in model checking problems we just need arbitrary fixed points to be unrolled and polarization is neutral

- Fixed points build in contraction, but for coPBT that’s the only place we need it.
Recall the PBT query: $\exists x[(\tau(x) \land R(x)) \land \neg Q(x)]$

With negation and logic (programming), things get hairy:

In our account of PBT, we managed to see $\neg$ as negation-as-failure, no certificate needed

$\mu$MALL builds in for Horn spec the CWA: $\neg Q$ as $Q \rightarrow \perp$

Otherwise, we can use de Morgan and inequalities to eliminate negation. E.g., non applicative simulation $m \not\leq_a n$:

$$\mu(\lambda NAS. \lambda m. \lambda n. \exists M'. m \Downarrow (fun M') \otimes \forall N'. n \Downarrow (fun N') \rightarrow \exists R. (NAS (M' R) (N' R)))$$
A Proof-of-concept implementation: **Bedwyr**

We piggy-back our prototype on top of the *Bedwyr* model-checker, which:

- implements a depth-first fully automatic search for focused proofs of a fragment of $\mu$MALL
- supports $\lambda$-tree syntax: binding in terms, pattern unification, binder mobility, $\nabla$-quantification
- implements (co)induction as loop detection via *tables* (modulo equivariance).
A Proof-of-concept implementation: architecture

- We use a “multi-level” approach where FPC drive generation and Bedwyr does the rest.
- Generators are reified in \( \text{prog} \) clauses over object level formulae of type \( \text{oo} \);

\[
\text{Define prog : } \text{oo} \to \text{oo} \to \text{prop} \text{ by }
\]
\[
\text{prog (istm } \text{Ctx (app Exp1 Exp2))}
\]
\[
(\text{istm } \text{Ctx Exp1}) \land (\text{istm } \text{Ctx Exp2}); \ldots
\]

- For each certificate format \( \text{cert} \), generation is driven by a meta-interpreter \( \text{check} \) parameterized by a \( \text{prog} \);

\[
\text{Define check : } \text{cert} \to \text{oo} \to (\text{oo} \to \text{oo} \to \text{prop}) \to \text{prop} \text{ by }
\]
\[
\text{check Cert A Prog := unfoldE Cert Cert'} \land
\text{Prog A G} \land \text{check Cert'} G \text{ Prog}.
\]
Our fav example

Plain Bedwyr is in charge of precondition and testing phase:

Define coinductive coeval : i \to i \to prop by
  coeval (fun R) (fun R);
  coeval (app M N) V := coeval M (fun R) /\ 
  coeval N W /\ coeval (R W) V.

% the query: gen by height
=? check (qheight 4) (istm nl M) & (istm nl M1) 
  & (istm nl M2) /\ coeval M M1 /\ coeval M 
M2 /\ (M1 = M2 -> false).

Found a solution + 45ms:
  M2 = app (fun (x\ x)) (fun (x\ x))
  M1 = fun (x\ x)
  M = app (fun (x\ app x x)) (fun (x\ app x x))
Conclusions

- PBT successfully complements theorem proving with a preliminary phase of conjecture testing, but its support for checking coinductive spec is unsatisfactory.
- We have shown as the FPC-based proof-theoretic reconstruction of PBT extends to such specs by relying on stronger logics.
- We have presented a proof-of-concept implementation in Bedwyr.
Current and Future Work: implementation

- PBT uses **mode** information to restrict and delay term generation:
  - w.r.t. standard evaluation $M \downarrow V$, need to generate only $M$ and often can use the judgment as a generator.
- Not the case for coinductive judgments that typically do not compute, only check (see also bisimilarity etc.)
  - refuting non-det of $\text{coeval}$ requires the unbounded, orthogonal generation of 3 terms. Does not scale.
- deal with the combinatorial explosion via **fuzzying**:
  - Generate one term and obtain the others by (random) mutations, possibly preserving global invariants (viz., typing)
- Pair it with the idea of **pre-computing** equivalence classes of terms of given depth 'a la Tarau ['18]
Current and Future Work: extensions

- Integrate with Abella’s workflow, both at the top-level (disproving conjectures) and inside a proof attempt (disproving subgoals).
- Investigate an explanation tool for attributing “blame” for the origin of a counterexample.
- Look into sub-structural object logics for PBT-ing specs about state and concurrency.
Thank you!