On the Proof Theory of Property-Based Testing of Coinductive Specifications, or: PBT to Infinity and beyond

Roberto Blanco[1], Dale Miller[2], and Alberto Momigliano[3]

[1] INRIA Paris, France
[2] INRIA Saclay & LIX, École Polytechnique, France
[3] DI, Università degli Studi di Milano, Italy

Reasoning about infinite computations via coinduction and corecursion has an ever increasing relevance in formal methods and, in particular, in the semantics of programming languages, starting from [13]; see also [10] for a compelling example — and, of course, coinduction underlies (the meta-theory of) process calculi. This was acknowledged by researches in proof assistants, who promptly provided support for coinduction and corecursion from the early 90's on, see [16, 7] for the beginning of the story concerning the most popular frameworks.

It also became apparent that tools that searched for refutations/counter-examples of conjectures prior to attempting a formal proof were invaluable: this is particularly true in PL theory, where proofs tend to be shallow but may have hundreds of cases. One such tool is *property-based testing* (PBT), which employs automatic test data generation to try and refute executable specifications. Pioneered by *QuickCheck* for functional programming [5], it has now spread to most major proof assistants [2, 15].

In general, PBT does not extend well to coinductive specifications (an exception being Isabelle's *Nitpick*, which is, however, a counter-model generator). A particular challenge, for example, for *QuickChick* is extending it to work with Coq's notion of coinductive via *guarded* recursion (which is generally seen to be an unsatisfactory approach to coinduction).

While PBT originated in the functional programming community, we have given in a previous paper [3] a reconstruction of some of its features (its operational semantics, different flavors of generation, shrinking) in purely proof-theoretic terms employing the framework of Foundational Proof Certificates [4]: the latter, in its full generality, defines a range of proof structures used in various theorem provers (e.g., resolution refutations, Herbrand disjuncts, tableaux, etc). In the context of PBT, the proof theory setup is much simpler. Consider an attempt to find counter-examples to a conjecture of the form $\forall x[(\tau(x) \wedge P(x)) \supset Q(x)]$ where τ is a typing predicate and P and Q are two other predicates defined using Horn clause specifications. By negating this conjecture, we attempt to find a (focused) proof of $\exists x[(\tau(x) \wedge P(x)) \wedge \neg Q(x)]$. In the focused proof setting, the *positive phase* (where test cases are generated) is represented by $\exists x$ and $(\tau(x) \wedge P(x))$. That phase is followed by the *negative phase* (where conjectured counter-examples are tested) and is represented by $\neg Q(x)$. FPCs are simple logic programs that can describe potential counter-examples using different generation strategies, e.g., δ-debugging, fault isolation, explanation, etc. Such a range of generation strategies can be programmed as the *clerks and experts* predicates that decorate the sequent rules used in a FPC proof checking kernel: such a kernel is also able to do a limited amount of proof reconstruction.

As explained in [3], the standard PBT setup needs little more than Horn logic specifications. However, when addressing infinite computations, we need richer specifications. While coinductive logic programming [18] may at first seem to fit the bill, the need to model infinite *behavior* rather than infinite objects as (ir)rational trees on the domain of discourse, has lead us to adopt a much stronger logic (and associated proof theory) with explicit rules for induction and coinduction.

A natural choice for such a logic is the fixed point logic $\mathcal{G}$ [6] and its linear logic cousin μMALL [1], which are associated to the *Abella* proof assistant and the *Bedwyr* model-checker. In fact, the latter has already been used for related aims [8].

To make things more concrete, consider the usual rules for CBV evaluation in the λ-calculus with constants, but define it *coinductively* (see [10]): using Bedwyr's concrete syntax, this is written as

```
Define coinductive coeval: tm -> tm -> prop by
 coeval (con C) (con C);
 coeval (fun R) (fun R);
 coeval (app M N) V :=
        exists R W, coeval M (fun R) /\ coeval N W /\ coeval (R W) V.
```

Is evaluation still deterministic? And if not, can we find terms `E`, `V1`, and `V2` such that `coeval E V1 /\ coeval E V2 /\ (V1 = V2 -> false)`? Indeed we can, since a divergent term such as Ω co-evaluates to anything. In fact, co-evaluation is not even type sound in its generality. Our PBT tool can find such counter-examples.

Our approach can also be used to separate various notion of equivalences in λ- and process calculi: for example, separating applicative and ground similarity in PCFL [17], or analogous standard results in the π-calculus. While similar goals have been achieved for labeled transition systems and for CCS (using, for example, the *Concurrency Workbench*), it is a remarkable feature of the proof-theoretic account that is easy to generalizes PBT from a system without bindings (say, CCS) to a system with bindings (say, the π-calculus). Such ease is possible since proof theory accommodates the *λ-tree syntax* approach to treating bindings [11]: this approach includes the ∇ quantifier [12] that appears in both Abella and Bedwyr.

In our current setup, we attempt to find counter-examples, using Bedwyr to execute both the generation of test cases (controlled by using specific FPCs [3]) and the testing phase. Such an implementation of PBT has the advantages of allowing us to piggyback on Bedwyr's facilities for efficient proof search via tabling for (co)inductive predicates. There are a couple of treatments of the negation in the testing phase. One approach to eliminating negation from intuitionistic specification can be based on the techniques in [14]. Another approach identifies the proof theory behind model checking as the linear logic μMALL [9] and in that setting, negations can be eliminated by using De Morgan duality (and inequality).

References

[1] D. Baelde. Least and greatest fixed points in linear logic. *ACM Trans. Comput. Log.*, 13(1):2:1–2:44, 2012.

[2] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCoS*, volume 6989 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 2011.

[3] R. Blanco, D. Miller, and A. Momigliano. Property-based testing via proof reconstruction. In *PPDP*, pages 5:1–5:13. ACM, 2019.

[4] Z. Chihani, D. Miller, and F. Renaud. A semantic framework for proof evidence. *J. of Automated Reasoning*, 59(3):287–330, 2017.

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

[6] A. Gacek, D. Miller, and G. Nadathur. A two-level logic approach to reasoning about computations. *Journal of Automated Reasoning*, 49(2):241–273, Aug 2012.

[7] E. Giménez. Codifying guarded definitions with recursion schemes. In P. Dybjer and B. Nordström, editors, *Selected Papers 2nd Int. Workshop on Types for Proofs and Programs, TYPES'94, Båstad, Sweden, 6–10 June 1994*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer-Verlag, Berlin, 1994.

[8] Q. Heath and D. Miller. A framework for proof certificates in finite state exploration. In *PxTP@CADE*, volume 186 of *EPTCS*, pages 11–26, 2015.

[9] Q. Heath and D. Miller. A proof theory for model checking. *J. Autom. Reasoning*, 63(4):857–885, 2019.

[10] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[11] D. Miller. Mechanized metatheory revisited. *Journal of Automated Reasoning*, Oct. 2018.

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

[13] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991.

[14] A. Momigliano. Elimination of negation in a logical framework. In *CSL*, volume 1862 of *Lecture Notes in Computer Science*, pages 411–426. Springer, 2000.

[15] Z. Paraskevopoulou, C. Hritcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015.

[16] L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7(2):175–204, Apr. 1997.

[17] A. M. Pitts. Operationally Based Theories of Program Equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, 1997.

[18] L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In L. Arge, C. Cachin, T. Jurdziński, and A. Tarlecki, editors, *Automata, Languages and Programming*, pages 472–483, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.