# Why Proof-Theory Matters in Specification-Based Testing

Alberto Momigliano

Dipartimento di Informatica, Università degli Studi di Milano
momigliano@di.unimi.it

**Abstract.** We survey some recent and ongoing developments in giving a logical reconstruction of specification-based testing via the lenses of structural proof-theory.

## 1 Introduction

Formal verification of software properties is still a labor intensive endeavor, notwithstanding recent advances: automation plays only a partial role and the engineer is heavily involved not only in the specification stage, but in the proving one as well, even with the help of a proof assistant. This effort is arguably misplaced in the *design* phase of a software artifact, when mistakes are inevitable and even in the best scenario the specification and its implementation may change. A *failed* proof attempt is hardly the best way to debug either.

These remarks are, of course, not novel, as they lie at the basis of model checking and other counter-examples generation techniques, where the emphasis is on *automatically refuting*, rather than proving, that some code respects its spec. Contrary to Dijkstras diktat, testing, and more in general validation, has found an increasing niche in formal verification, prior or even in alternative to theorem proving [5, 19].

The message of this brief summary is that, somewhat surprisingly, structural proof-theory [18] offers a unifying approach to the field. While the ideas that I am going to sum up here may have a wider applicability, I am going to narrow it to:

– *Specification-based testing* (SBT, also known as *property*-based testing [12]), a lightweight validation technique whereby the user specifies executable properties that the code should satisfy and the system tries to refute them via automatic (typically random) data generation.
– One specific domain of interest: the mechanization of the semantics of programming languages and related artifacts, where proofs tend to be shallow, but may have hundreds of cases and are therefore a good candidate to SBT.

## 2 SBT as Proof-Search

We use as a running examples a call-by-value $\lambda$-calculus (where values are lambdas and numerals) whose static and big-step dynamic semantics follows — readers should substitute it with a more substantial specification of a programming language whose meta-theory they wish to investigate:

$$\frac{}{\vdash n : \text{nat}}\ \texttt{T-N} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \texttt{T-VR} \qquad \frac{x \notin \text{dom}(\Gamma) \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B}\ \texttt{T-AB}$$

$$\frac{\Gamma \vdash M_1 : A \to B \quad \Gamma \vdash M_2 : B}{\Gamma \vdash M_1 \cdot M_2 : B}\ \texttt{T-AP}$$

$$\frac{\text{value } V}{V \Downarrow V}\ \texttt{E-V} \qquad \frac{M_1 \Downarrow \lambda x.\, M \quad M_2 \Downarrow V_2 \quad M\{V_2/x\} \Downarrow V}{M_1 \cdot M_2 \Downarrow V}\ \texttt{E-AP}$$

Consider now the type preservation property for closed terms:

$$\forall M\, M'\, A.\ M \Downarrow M' \longrightarrow M : A \longrightarrow M' : A$$

In fact, the result does *not* hold for this calculus, since we have managed to slip a typo in one of the rules. One counter-example is $M = (\lambda x.\, x \cdot n) \cdot n, M' = n \cdot n, A =$ nat. How to go from it to the origin of the bug is another research topic in itself [17]; suffices to say that it points to a mistake in rule `T-AP`, namely the type in the minor premise should be $A$. A tool that automatically provides such a counter-example would save us from wasting time on a potentially very long failed proof attempt.

While this issue can and has been successfully tackled in a functional programming setting [13], at least two factors make a proof-theoretic reconstruction fruitful: 1) it fits nicely with a (co)inductive reading of rule-based presentations of our system-under-test 2) it easily generalizes to logics that intrinsically handle issues that are pervasive in the domain of programming languages semantics, such as naming and scoping. In fact, as argued in [6], the SBT literature is rediscovering (constraint) logic programming ideas such as narrowing, mode checking, random back-chaining etc.

If we view a specification (property) as a logical formula $\forall x[(\tau(x) \wedge P(x)) \supset Q(x)]$ where $\tau$ is a typing predicate and $P$ and $Q$ are two other predicates defined using Horn clause specifications (to begin with), providing a counter-example consists of negating the property, and searching for a proof of $\exists x[(\tau(x) \wedge P(x)) \wedge \neg Q(x)]$.

Stated in this way the problem points to a logic programming solution, and since the seminal work of Miller et al. [15], structural proof-theory formulates it as a proof-search problem in the sequent calculus, where the specification is a fixed set of assumptions (typically sets of clauses) and the negated property is the goal.

A first solution that I was involved with is $\alpha$Check [7], which supplements a logic programming interpreter [9] to account for counter-example search. The tool uses 1) nominal logic as a logical foundation, which is particularly apt at encoding binding structures, 2) automatically derived type-driven *exhaustive* generators for data enumeration, 3) two approaches to implementing negation: negation as failure and negation elimination [8], 4) a fixed search strategy, namely iterative-deepening based on the height of partial proof trees.

## 3 SBT via FPC

While $\alpha$Check turned out to be quite effective (see the case studies at `https://github.com/aprolog-lang/checker-examples`), the approach was unnecessarily rigid, in particular wiring-in a fixed data generation and search strategy, and did not

$$\frac{\Xi_1 \vdash G_1 \qquad \Xi_2 \vdash G_2 \qquad \wedge_e(\Xi, \Xi_1, \Xi_2)}{\Xi \vdash G_1 \wedge G_2} \qquad \frac{tt_e(\Xi)}{\Xi \vdash tt} \qquad \frac{\Xi' \vdash G[t/x] \qquad \exists_e(\Xi, \Xi', t)}{\Xi \vdash \exists x.G}$$

$$\frac{\Xi' \vdash G \quad (A \text{ :- } G) \in grnd\ (\mathcal{P}) \quad unfold_e(\Xi, \Xi')}{\Xi \vdash A}$$

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

$$\frac{n \vdash G_1 \qquad n \vdash G_2}{n \vdash G_1 \wedge G_2} \qquad \frac{}{n \vdash tt} \qquad \frac{n \vdash G[t/x]}{n \vdash \exists x.G}$$

$$\frac{n \vdash G \quad (A \text{ :- } G) \in grnd\ (\mathcal{P}) \quad n \geq 0}{n+1, \vdash A}$$

**Fig. 1.** Clerks and expert presentation of the sequent calculus for Horn logic

reflect the richness of features that SBT offers. However, being the proof-theory of $\alpha$Check based on the notion of *uniform proofs* [15], it is easy to generalize it via the subsuming theory of *focused proof systems* [2]. We can roughly characterize focusing as a complete strategy to organize the rules of the sequent calculus into two phases: 1) a *negative* phase corresponding to goal-reduction, where we apply rules involving *don't-care-nondeterminism*; as a result, there is no need to consider backtracking, and 2) a *positive* phase corresponding to back-chaining (*don't-know-nondeterminism)*: here, inference rules need to be supplied with external information (e.g., which clause to back-chain on) in order to ensure that a completed proof can be built. Thus, when building a proof tree from the conclusion to its leaves, the negative phase corresponds to a simple deterministic computation, while the positive phase may need to be guided by an oracle.

The connection with SBT is that in a query the *positive phase* (which corresponds to the generation of possible counter-examples) is represented by $\exists x$ and $(\tau(x) \wedge P(x))$. That is followed by the *negative phase* (which corresponds to counter-example testing) and is represented by $\neg Q(x)$. This formalizes the intuition that generation may be hard, while testing is just computation.

The final ingredient is how to supply the external information to the positive phase: this is where the theory of *foundational proof certificates* [10] (FPC) comes in. In their fully generality, FPCs can be seen as a generalization of proof-terms in the Curry-Howard tradition, able to define a range of proof structures used in various theorem provers (e.g., resolution refutations, Herbrand disjuncts, tableaux, etc). They can be programmed as *clerks and experts* predicates that decorate the sequent rules used in an FPC proof checking kernel. An FPC system is a specification of the certificate format together with the clerks and experts processing it. In our setting, we can view FPCs as simple logic programs that guide the search for potential counter-examples using different generation strategies.

Figure 1 contains a simple proof system for a fragment of Horn clause provability in which each inference rule is augmented with an additional premise involving an expert predicate, a certificate $\Xi$, and possibly continuations of certificates ($\Xi'$, $\Xi_1$, $\Xi_2$), if one reads the rules from conclusion to premises. The logic programmers among us will recognize it as an instrumented version of the vanilla meta-interpreter over a fixed Horn program $\mathcal{P}$. For example, the $\exists$-expert may be in charge of extracting from $\Xi$ the term $t$ with which to instantiate $G$ so that we can build the rest of the proof according to the resulting certificate $\Xi'$. In the bottom part of the figure, we instantiate the framework with the simplest form of proof certificate, namely a positive integer, where the only active expert is a simple non-zero check while back-chaining: this characterizes exhaustive generation bounded by *height*, which happens to be the generation strategy of $\alpha$Check. As detailed in [6], different FPCs capture random generation, via randomized backtracking, as well as diverse features such as $\delta$-debugging, bug-provenance, etc.

## 4 To infinity and beyond

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, (see [14] for a compelling example) and, of course, coinduction underlies (the meta-theory of) process calculi. To our knowledge, there are no SBT approaches for coinductive specifications, save for the quite limited features provided by Isabelle/HOL's *Nitpick* [5].

When addressing potentially infinite computations, where in our setup we strive to model infinite *behavior* (think divergence of a finite program) rather than infinite *objects* (i.e., infinite terms such as streams), we need to go significantly beyond the simple proof-theory of [6] and adopt a much stronger logic with explicit rules for induction and coinduction.

A natural choice is the fixed point linear logic $\mu$MALL [3], which is associated to the *Bedwyr* model-checker [4]. In fact, this logic has already shown its colors in the proof-theoretic reconstruction and certification of model checking problems such as (non)-reachability [11]. $\mu$MALL consists of a sequent calculus presentation of *multiplicative additive linear logic* with least and greatest fixed points operators in lieu of exponentials, over a simply-typed term language.

Continuing with our running example, let us now consider a *coinductive* definition of CBV evaluation following [14]. In other terms, we take the same rules as in Section 2, but we read them as the greatest fixed point of the defined relation. This is represented by the following formula (reminiscent of a linearization of Clark's completion), where $\nu$ is the greatest fixed point operator, $\Lambda$ is the abstractor in the meta-logic and *val* stands for the $\mu$-formula characterizing values:

$$\begin{aligned}\mathrm{coeval} \equiv \nu(\Lambda CE.\Lambda m.\Lambda m'.&(\exists V.\ m = V \otimes m' = V \otimes \mathrm{val}\ V) \oplus \\ &(\exists M_1 M_2 M V_2 V.\ m = M_1 \cdot M_2 \otimes m' = V \otimes (CE\ M_1\ (\lambda x.\, M)) \otimes \\ &(CE\ M_2\ V_2) \otimes (CE\ (M\{V_2/x\})\ V))\end{aligned}$$

$\otimes$ and $\oplus$ are multiplicative conjunction and additive disjunction and for the sake of space we do not make explicit the encoding of the object-level syntax and of substitution inside the meta-logic.

Whether or not this notion of co-evaluation makes sense (see for a fair criticism [1]), we would like to investigate if standard properties such as type soundness or determinism of evaluation hold: they do not — to refute the latter, just note that a divergent term such as $\Omega$ co-evaluates to anything. Type preservation, for a *correct* version of the rules in Sec. 2, is falsified by a variant of the $Y$-combinator.

We have a prototype implementation of SBT for coinductive specs on top of Bedwyr, which we use both for the generation of test cases (controlled using specific

FPCs [6]) and for the testing phase. Such an implementation has the advantage of allowing us to piggyback on Bedwyr's facilities for efficient proof-search via *tabling* for (co)inductive predicates, thus avoiding costly meta-interpretation.

To make it more concrete, let me report the query refuting determinism of co-evaluation. We use Bedwyr's concrete syntax, where `check`, implementing the kernel rules in the top of Fig. 1, is in charge of controlling the generation of lambda terms (predicate `is_exp` parameterized over a context of bound variables), here in the exhaustive fashion detailed *ibidem*(generator `height 4`); `coeval` encodes the coinductive CBV operational semantics using higher-order abstract syntax and the last conjunct corresponds to ground disequality.

```
?= check (height 4) ((is_exp void M) && (is_exp void M1) && (is_exp void M2))
   /\ coeval M M1 /\ coeval M M2 /\ (M1 = M2 -> false).

Found a solution (+ 4173ms):
 M2 = con one
 M1 = con zero
 M =  app (fun (x\ app x x)) (fun (x\ app x x))
```

The system finds in reasonable time the expected counter-example, for `M` being the encoding of $\Omega$. Note that we only generate *finite* terms, but the testing phase now appeals (twice) to the coinductive hypothesis.

Other applications of SBT w.r.t. infinite behavior are in separating various notion of equivalences in lambda and process calculi: for example, applicative and ground similarity in PCFL [21], or analogous standard results in the $\pi$-calculus.

These examples put forward another challenge: the specification of a coinductive notion such as applicative similarity goes beyond the Horn fragment, to wit:

$$\begin{aligned} \mathrm{asim} \equiv \nu(\Lambda AS.\Lambda m.\Lambda n.\forall M'.\ \mathrm{eval}\ m\ (\lambda x.\, M') \multimap \exists N'.\ \mathrm{eval}\ m\ (\lambda x.\, N') \otimes \\ \forall R.\ (AS\ (M'\{R/x\})\ (N'\{R/x\}))) \end{aligned}$$

This makes the treatment of negation in the testing phase of a SBT query problematic since the interpretation of finite failure as provability of falsehood breaks down, at least in an intuitionistic setting [16]. Here, the adoption of linear logic as a meta-logic comes to the rescue, since in linear logic occurrences of negations can be eliminated by using De Morgan duality and inequality.

## 5 Conclusion

I have tried to delineate a path where structural proof-theory reconstructs, unifies and extends current trends in SBT, with a particular emphasis to ongoing work on extending the paradigm to infinite computations. A natural next step is *concurrency*: logical framework such as *CLF* (and its implementation *Celf* [22]) based on sub-structural logics have been designed to encode concurrent calculi, e.g., session types [20]. However, the meta-theory of such frameworks is still in the workings and this precludes so far any reasoning about them. On the other hand, a FPC approach to the validation of those properties seems a low hanging fruit.

*Acknowledgment* A shout-out to my co-authors in this line of work: Rob Blanco, James Cheney, Francesco Komauli, Dale Miller and Matteo Pessina.

## References


1. D. Ancona, F. Dagnino, and E. Zucca. Reasoning on divergent computations with coaxioms. *Proc. ACM Program. Lang.*, 1(OOPSLA):81:1–81:26, 2017.
2. J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.
3. D. Baelde. Least and greatest fixed points in linear logic. *ACM Trans. Comput. Log.*, 13(1):2:1–2:44, 2012.
4. D. Baelde, A. Gacek, D. Miller, G. Nadathur, and A. Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *21th Conf. on Automated Deduction (CADE)*, number 4603 in LNAI, pages 391–397. Springer, 2007.
5. 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.
6. R. Blanco, D. Miller, and A. Momigliano. Property-based testing via proof reconstruction. In *PPDP*, pages 5:1–5:13. ACM, 2019.
7. J. Cheney and A. Momigliano. $\alpha$Check: A mechanized metatheory model checker. *Theory and Practice of Logic Programming*, 17(3):311352, 2017.
8. J. Cheney, A. Momigliano, and M. Pessina. Advances in property-based testing for $\alpha$Prolog. In B. K. Aichernig and C. A. Furia, editors, *TAP 2016*, volume 9762 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2016.
9. J. Cheney and C. Urban. Nominal logic programming. *ACM Transactions on Programming Languages and Systems*, 30(5):26, August 2008.
10. Z. Chihani, D. Miller, and F. Renaud. A semantic framework for proof evidence. *J. of Automated Reasoning*, 59(3):287–330, 2017.
11. 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.
12. J. Hughes. Quickcheck testing for fun and profit. In M. Hanus, editor, *PADL 2007*, volume 4354 of *LNCS*, pages 1–32. Springer, 2007.
13. C. Klein and coauthors. Run your research: on the effectiveness of lightweight mechanization. POPL '12, pages 285–296. ACM, 2012.
14. X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.
15. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
16. A. Momigliano. Elimination of negation in a logical framework. In *CSL*, volume 1862 of *Lecture Notes in Computer Science*, pages 411–426. Springer, 2000.
17. A. Momigliano and M. Ornaghi. The blame game for property-based testing. In *CILC*, volume 2396 of *CEUR Workshop Proceedings*, pages 4–13. CEUR-WS.org, 2019.
18. S. Negri, J. von Plato, and A. Ranta. *Structural Proof Theory.* Cambridge University Press, 2001.
19. Z. Paraskevopoulou, C. Hritcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In C. Urban and X. Zhang, editors, *ITP 2015*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015.
20. F. Pfenning and D. Griffith. Polarized substructural session types. In *FoSSaCS*, volume 9034 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.
21. A. M. Pitts. Operationally Based Theories of Program Equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, 1997.
22. A. Schack-Nielsen and C. Schürmann. Celf - A logical framework for deductive and concurrent systems (system description). In *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 320–326. Springer, 2008.