# Property-Based Testing via Proof Reconstruction

Work-in-progress

Roberto Blanco
Dale Miller
INRIA Saclay — Île-de-France & LIX/École 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 $\lambda$-tree syntax. The $\lambda$Prolog 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-Redex.

## 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 this 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

$$\forall L\colon (list\ int)\ \forall R\colon (list\ int)\ [rev\ L\ R \supset rev\ R\ L]$$

stating that *rev* is idempotent should be a theorem. In fact, we might wish to prove a number of formulas of the form $\forall x\colon \tau\ [P(x) \supset Q(x)]$ where both $P$ and $Q$ are formulae over given relational specifications and a single variable (for brevity). Occasionally, it can be important in this setting to move the type judgment $x\colon \tau$ into the logic by turning the type into a predicate: $\forall x[(\tau(x) \wedge P(x)) \supset Q(x)]$. 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 specifications in $P$ and/or $Q$ can contain errors. It can be therefore valuable to first attempt to find counterexamples to such formulas prior to pursue a proof. That is, we might try to prove formulas of the form $\exists x[(\tau(x) \wedge P(x)) \wedge \neg Q(x)]$ 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 in $P$ 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 and automatically.

LFMTP 2017, September 2017, Oxford, UK

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

```
nat z.                lst nl.
nat (s N) :- nat N.   lst (cns N Ns) :- nat N, lst Ns.

app nl Xs Xs.
app (cns X Xs) Ys (cns X Zs) :- app Xs Ys Zs.
```

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

$$[app\ Xs\ Ys\ Zs \supset app\ (cns\ X\ Xs)\ Ys\ (cns\ X\ Zs)].$$

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 view 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 *bipole*. 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 them as *fixed points*. For example, the Prolog-style specifications above of *nat* and *app* can be written instead as the following fixed point definitions.

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

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 $\exists x\colon \tau\ [P(x) \wedge^+ \neg Q(x)]$ 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 conceptionally 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 may 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.

In this paper, 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 $\lambda$Prolog, which 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 either 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 produced. 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 $\exists x\ [\tau(x) \wedge^+ P(x) \wedge^+ \neg Q(x)]$ can be directly translated into a description of the range of possible witness terms for this quantifier.

### 1.3 Lifting PBT to treat $\lambda$-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 calculus 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 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 $\nabla$-quantifier [12, 18]. While the $\nabla$-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 $\nabla$ is the following: if fixed point definitions do not contain implications and negations, then exchanging occurrences of $\forall$ and $\nabla$ 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 $\nabla$.

This direct treatment of $\lambda$-terms within the PBT setting allows us to apply property-based testing to a number of metaprogramming tasks. After describing some 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 drives the derivation of our object logic. In this case it is basically the usual Prolog vanilla meta-interpreter, save for interpreting $\nabla$ as $\Pi$; the "program" is represented as Horn-like clauses by a two-place predicate `prog` relating heads and bodies, built out of object-level logical constants (`tt`, `or`, `and`, `nabla`) and user-defined constructors for predicates. For example, to generate lists of as and bs and compute the reverse a list, we have the following `prog` clauses, where we omit the code for `append`:

```
prog (is_elt a) tt.
prog (is_elt b) tt.
prog (is_eltlist nl) tt.
prog (is_eltlist (cns X Xs))
     (and (is_elt X) (is_eltlist Xs)).
prog (rev nl nl) tt.
prog (rev (cns X Xs) Rs)
     (and (rev Xs Sx) (append Sx (cns X nl) Rs)).
```

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 bound, in this case the *height* of a proof of being a list. The testing phase performs deterministic computation with the meta-interpreter `interp` and then negates the conclusion using negation-as-failure (NAF):

```
cexrev Xs Ys :- check (qgen (qheight 3)) (is_eltlist Xs),
              interp (rev Xs Ys), not (Xs = Ys).
```

Note that the call to NAF is safe since, by the totality of `rev`, `Ys` will be ground.

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 `prog` and unfolded. This is driven by the meta-interpreter `interp` (omitted). To it, `check` adds a certificate term and calls to *expert* predicates on said term (except `nabla`, which is transparent to the experts). Experts decide when the computation proceeds — producing certificates for the continuations — and when

```
check Cert tt          :- tt_expert Cert.
check Cert (and G1 G2) :- and_expert Cert Cert1 Cert2, check Cert1 G1, check Cert2 G2.
check Cert (or G1 G2)  :- or_expert Cert Cert' LR, ((LR = left, check Cert' G1); (LR = right, check Cert' G2)).
check Cert (nabla G)   :- pi x\ check Cert (G x).
check Cert A           :- unfold_expert Cert Cert', prog A G, check Cert' G.

tt_expert     (qgen (qsize In In)).
tt_expert     (qgen (qheight _)).
or_expert     (qgen (qsize In Out)) (qgen (qsize In Out)) _.
or_expert     (qgen (qheight H))    (qgen (qheight H))    _.
and_expert    (qgen (qsize In Out)) (qgen (qsize In Mid)) (qgen (qsize Mid Out)).
and_expert    (qgen (qheight H))    (qgen (qheight H))    (qgen (qheight H)).
unfold_expert (qgen (qsize In Out)) (qgen (qsize In' Out)) :- In > 0, In' is In - 1.
unfold_expert (qgen (qheight H))    (qgen (qheight H'))    :- H  > 0, H'  is H  - 1.
```

**Figure 1: Kernel for expert-driven term generation**

it fails. The first argument of an expert, e.g., `and_expert`, refers to the conclusion of the corresponding rule and the remaining ones, if any, to the premises. Here the complexity of generated candidates is bound by limiting unfoldings, either by height (`qheight`, producing shallow terms), number of constructors (`qsize`, producing small terms), or both by *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:

$$\begin{array}{llll} \text{Types} & A, B & ::= & int \mid ilist \mid A \to B \\ \text{Terms} & M & ::= & x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \mid c \mid err \\ \text{Constants} & c & ::= & n \mid plus \mid nil \mid cons \mid hd \mid tl \\ \text{Values} & V & ::= & c \mid \lambda x{:}A.\ M \mid plus\ V \\ & & & \mid cons\ V \mid cons\ V_1\ V_2 \end{array}$$

The rules for the dynamic and static semantics are given in Figure 2, where the latter assumes a signature $\Sigma$ with the obvious type declarations for constants. Rules for *plus* are omitted for brevity.

The encoding in $\lambda$Prolog is pretty standard and also omitted: we declare constructors for terms, constants and types, while we carve out values via an appropriate predicate. A similar predicate characterizes the threading in the operational semantics of the *err* expression, used to model run time errors such as taking the head of an empty list. We follow this up (see the bottom of Figure 2) with the static semantics (predicate `wt`), where constants are typed via a table `tcc`. Note that we have chosen an *explicitly* contexted encoding of typing as opposed to one based on hypothetical judgments such as in [16]: this choice avoids using implications in the body of the typing predicate and, as a result, allows us to use $\lambda$Prolog's universal quantifier to implement the reasoning level $\nabla$-quantifier.

Now, this calculus enjoys the usual property of subject reduction and progress, where the latter means "being either a value, an error, or able to make a step." And in fact we can fairly easily prove those results in a theorem prover such as Abella. However, the case distinction in the progress theorem does require some care: were it to be unprovable given a mistake in the specification, it would not be immediate to localize where the problem may be. On the other hand, one could wonder whether our calculus enjoys the *subject expansion* property — the alert reader will undoubtedly realize that this is highly unlikely, but rather than wasting time in a proof attempt, we search for a counterexample and find:

```
cexsexp M M' A :- check (qgen (qsize 8 _)) (step M M'),
                  interp (wt null M' A),
                  not (interp (wt null M A)).
A  = listTy
M' = c nl
M  = app (c hd) (app (app (c cns) (c nl)) (c _))
```

Other queries we can ask: are there *untypable* 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. For example, 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:

$$\frac{\Gamma \vdash_\Sigma M_1 : A \to B \quad \Gamma \vdash_\Sigma M_2 : B}{\Gamma \vdash_\Sigma M_1\ M_2 : B}\ \texttt{T-APP-B1}$$

The given mutation makes both properties fail:

```
cexprog M A :- check (qgen (qsize 6 _)) (wt null M A),
               not (interp (progress M)).
A = intTy
M = app (c hd) (c (toInt zero))

cexpres M M' A :- check (qgen (qsize 8 _)) (wt null M A),
                  interp (step M M'),
                  not (interp (wt null M' A)).
A  = funTy listTy intTy
M' = lam (x\ c hd) listTy
M  = app (lam (x\ lam (y\ x) listTy) intTy) (c hd)
```

Table 1 reports the tests, performed under Ubuntu 16.04 on a Intel Core i7-870 CPU, 2.93GHz with 8GB RAM. We time-out the computation when it exceeds 300 seconds. We list the results obtained by $\lambda$Prolog ($\lambda$P) under Teyjus [20], the counterexample found, and a brief description of the bug together with Redex's difficulty rating (shallow, medium, unnatural). The column $\alpha$C lists the time taken by $\alpha$Check [7] using NAF, which is not always the best technique [8],

$$\frac{}{hd\ (cons\ M_1\ M_2) \longrightarrow M_1}\ \texttt{E-HD} \qquad \frac{}{tl\ (cons\ M_1\ M_2) \longrightarrow M_2}\ \texttt{E-TL}$$

$$\frac{}{\lambda x : A.\ M\ V \longrightarrow [x \mapsto V]M}\ \texttt{E-ABS} \qquad \frac{M_1 \longrightarrow M_1'}{M_1\ M_2 \longrightarrow M_1'\ M_2}\ \texttt{E-APP1} \qquad \frac{M \longrightarrow M'}{V\ M \longrightarrow V\ M'}\ \texttt{E-APP2}$$

$$\frac{}{\vdash_\Sigma err : A}\ \texttt{T-ER} \quad \frac{\Sigma(c) = A}{\vdash_\Sigma c : A}\ \texttt{T-K} \quad \frac{x : A \in \Gamma}{\Gamma \vdash_\Sigma x : A}\ \texttt{T-VAR} \quad \frac{\Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x : A.\ M : A \to B}\ \texttt{T-ABS} \quad \frac{\Gamma \vdash_\Sigma M_1 : A \to B \quad \Gamma \vdash_\Sigma M_2 : A}{\Gamma \vdash_\Sigma M_1\ M_2 : B}\ \texttt{T-APP}$$

```
prog (wt _ err _)            tt.
prog (wt _ (c M) A)          (tcc M A).
prog (wt Gamma M A)          (memb (bind M A) Gamma).
prog (wt Gamma (lam M Ax)    (funTy Ax A)) (nabla x\ wt (cons (bind x Ax) Gamma) (M x) A).
prog (wt Gamma (app M N) A) (and (wt Gamma M (funTy B A)) (wt Ga N B)).
```

**Figure 2: Static and dynamic semantics of the *Stlc* language.**

| bug | check | $\alpha$C | $\lambda$P | cex | Description/Rating |
|---|---|---|---|---|---|
| 1 | preservation | 0.3 | 0.05 | $(\lambda x{:}int.\ \lambda y{:}ilist.\ x)\ hd$ | range of function in app rule |
| | progress | 0.1 | 0.02 | *hd 0* | matched to the arg. (S) |
| 2 | progress | 0.27 | 0.06 | *(cons 0) nil* | value *(cons v) v* omitted (M) |
| 3 | preservation | 0.04 | 0.01 | $(\lambda x{:}int.\ cons)\ cons$ | order of types swapped |
| | progress | 0.1 | 0.04 | *hd 0* | in function pos of app (S) |
| 4 | progress | t.o. | 207.3 | *(plus 0) ((cons 0) nil)* | the type of cons return *int* (S) |
| 5 | preservation | t.o. | 0.67 | *tl ((cons 0) nil)* | tail reduction returns the head (S) |
| 6 | progress | 24.8 | 0.4 | *hd ((cons 0) nil)* | hd reduction on part. applied cons (M) |
| 7 | progress | 1.04 | 0.1 | $hd\ ((\lambda x{:}ilist.\ err)\ nil)$ | no eval for argument of app (M) |
| 8 | preservation | 0.02 | 0.01 | $(\lambda x{:}ilist.\ x)\ nil$ | lookup always returns int (U) |
| 9 | preservation | 0.1 | 0.02 | $(\lambda x{:}ilist.\ cons)\ nil$ | vars do not match in lookup (S) |

**Table 1: *Stlc* benchmark**

but it corresponds very closely to the architecture of the present paper. Of course, $\alpha$Check sits on top of an interpreted (prototype) language, whereas Teyjus is a compiler: however, one can argue that the two level themselves out, since we use meta-interpretation for test generation. The results are essentially indistinguishable, save for bugs 4, 5 and 6: in the first, which is surprisingly hard to find, $\alpha$Check times out, while we comfortably beat the time limit. $\alpha$Check flunks number 5, which is immediate for us. Finally in bug 6 $\alpha$Check's fixed integrative deepening strategy needs to explore the search space up to level 11, while we can leverage the FPC ability to use the `qsize` metric.

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

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).

## 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 immediate 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 proofs for one type but random for another type: thus, one could easily design a certificate that would explore randomly generated lists containing just, say, the integers 0 and 1.

While $\lambda$Prolog is used here to discover counterexamples, one does not actually need to trust the logical soundness of $\lambda$Prolog (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.

## REFERENCES


[1] D. Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1), Apr. 2012.

[2] 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, New York, 2007. Springer.

[3] D. Baelde, K. Chaudhuri, A. Gacek, D. Miller, G. Nadathur, A. Tiu, and Y. Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2), 2014.

[4] 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.

[5] R. Blanco and D. Miller. Proof outlines as proof certificates: a system description. In I. Cervesato and C. Schürmann, editors, *Proceedings First International Workshop on Focusing*, volume 197 of *Electronic Proceedings in Theoretical Computer Science*, pages 7–14. Open Publishing Association, Nov. 2015.

[6] R. Blanco, Z. Chihani, and D. Miller. Translating between implicit and explicit versions of proof. In L. de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction*, volume 10395 of *LNCS*, pages 255–273. Springer, 2017.

[7] J. Cheney and A. Momigliano. $\alpha$Check: A mechanized metatheory model checker. *Theory and Practice of Logic Programming*, 17(3):311âĂŞ352, 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, *Tests and Proofs - 10th International Conference, TAP 2016, Vienna, Austria, July 5-7, 2016, Proceedings*, volume 9762 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2016.

[9] Z. Chihani, D. Miller, and F. Renaud. A semantic framework for proof evidence. *J. of Automated Reasoning*, 2016.

[10] 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.

[11] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[12] A. Gacek, D. Miller, and G. Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*, pages 33–44. IEEE Computer Society Press, 2008.

[13] Q. Heath and D. Miller. A proof theory for model checking: An extended abstract. In I. Cervesato and M. Fernández, editors, *Proceedings Fourth International Workshop on Linearity (LINEARITY 2016)*, volume 238 of *EPTCS*, Jan. 2017.

[14] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM.

[15] C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

[16] R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.

[17] D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

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

[19] 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.

[20] G. Nadathur and D. J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of $\lambda$Prolog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 287–291, Trento, 1999. Springer.

[21] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[22] 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.