Property-Based Testing of Abstract Machines: an Experience Report

Francesco Komauli, Alberto Momigliano

Property-Based Testing of Abstract Machines: an Experience Report

Francesco Komauli Alberto Momigliano
DI, Università di Milano, Italy
francesco.komauli@studenti.unimi.it momigliano@di.unimi.it

Contrary to Dijkstra's diktat, testing, and more in general, validation, has found an increasing niche in formal verification, prior or even in alternative to theorem proving. Validation, in particular, property-based testing (PBT) is quite effective in mechanized meta-theory of programming languages, where theorems have shallow but tedious proofs that may go wrong for fairly banal mistakes. In this report, we abandon the comfort of high-level object languages and address the validation of abstract machines and typed assembly languages. We concentrate on Appel et al.'s list-machine benchmark [2], which we tackle both with αCheck, the simple model-checker on top of the nominal logic programming αProlog and the PBT library *FSCheck* for *F#*. This allows us to compare the relative merits of exhaustive-based PBT in a logic programming style versus the more usual randomized functional setting. We uncover one major bug in the published version of the paper plus several typos and ambiguities thereof. This is particularly striking, as the paper is accompanied by two full formalizations, in Coq and Twelf. Finally, we do a bit of *mutation testing* on the given model, to asses further the trade-off between exhaustive and randomized data generation. Spoiler alert: the former performs better.

1 Introduction

Does this sound familiar? You are in the middle of a long but supposedly straightforward formal proof trying to drag your favourite proof assistant to confirm the blindingly obvious truth of the result, when you get stuck in an unprovable part of the derivation; you realize that the statement needs to be adjusted or more commonly that there is something afoul, typically a banality, in your specification. If only you had a way to realize that the theorem was unprovable before wasting precious time in a doomed proof attempt, perhaps some kind of *testing* ...

This is where, with due respect to Dijkstra and his "thou shall not test" commandment, *validation* prior to theorem proving comes in: as a matter of fact, such tools have been available for some time in the leading proof assistants: to cite only two of them, the *NitPick/QuickCheck* combination [4] (and descendants) in Isabelle/HOL and *QuickChick* [21] in Coq.

Our angle is not verification vs. validation in general, but in a particular domain: the mechanization of the meta-theory (MMT) of programming languages (PL) and related calculi. As any practitioner may testify, the main properties of interest are well known and have mathematically shallow proofs. The difficulties lies in the potential magnitude of the cases one must consider and in the trickiness that some encodings require, typically those having to deal with binding signatures. Here, very minor mistakes in the specification, even at the level of what we would consider a typo, may severely frustrate the verification effort, to the point to make it not cost-effective. Further, we aim to support the "working semantics" in her work in *developing*, and eventually formally proving correct, such calculi, rather than concentrating on models that we already know to be correct.

Submitted to:
LFMTP 2018

The model checking technique we have adopted is *property-based testing* (PBT): while it originated as a testing techniques for concrete (functional) programs, it is also applicable at the meta-theoretical level, especially in the PL domain, as shown by the success [17] of tools such as *PLT-Redex* (`https://redex.racket-lang.org`) and, "si parva licet componere magnis", αCheck [7]. PBT's data generation strategy comes in several flavours: random [11], exhaustive [27], a combination of the two [12], with an ever increasing emphasis on coroutining the generation and testing phases [14, 19]. An analysis of some of those strategies as applied to MMT was carried out in [13].

In this paper, we abandon the comfort of high-level object languages, which have been investigated extensively, and address the validation of abstract machines and typed assembly languages. That seems more challenging, since instructions operating at a low level provide less "structure" for counterexamples, which then tend to be substantially more complex. Further, it is hard to generate sequences of machine states that yield meaningful machine runs. Similar remarks have appeared in the context of random generation of C programs [29]. Not coincidentally, the authors of [1] suggest that "[counterexamples to properties such as non-interference may be] well beyond the scope of naive exhaustive testing" (pag. 12).

We take on Appel et al.'s list-machine benchmark [2], dubbed by the authors CIVmark, standing for **C**ompiler **I**mplementation **V**erification. We concentrate mostly on its version 1.0, as originally presented at LFMTP'06, but we also touch on 2.0, see Section 6.1. CIVmark was conceived as a benchmark for "machine-checked proofs about real compilers", spurred by a civilized criticism of POPLMark as being too biased towards issues of binders. It consists of a basic pointer machines with instructions such as `car`, `cdr`, `cons` and (un)conditional jump, endowed with a standard SOS, but with a reasonably sophisticated type system: the latter, being able to make static predictions about lists being empty or not, guarantees that a well typed machine's run does not get stuck. Two implementations, including sketch of proofs of type soundness in Twelf and Coq come with the paper.

A version of this model is also part of the benchmarks distribution of PLT-Redex [15] and as such being subjected to some PBT. All the more reasons to tackle with two very distinct PBT's approaches, which fits well with the existing formalizations:

1. An encoding in the nominal logic programming αProlog [9], to be tested with its model checker αCheck [7];
2. A functional encoding with the PBT library *FSCheck* for *F#*.

This allows us to compare the relative merits of exhaustivity-based PBT in a logic programming style versus the more usual randomized functional setting polarized by QuickCheck. It is also a stress test for αCheck, since CIVmark does not exploit any of the features offered by nominal logic that makes αProlog effective, see the case studies at `https://github.com/aprolog-lang/checker-examples`, which are all devoted to high-level languages, including some studied in [17]; instead, the CIVmark benchmark brings to the fore the naivete of αProlog's search strategy.

A testing approach is "good" only if it uncovers bugs, and so we did: we falsified the type preservation property as presented in the paper, caused by an incorrect specification of typing for values. This is particularly striking, considering the paper comes with two *formalized* proof of type soundness. The mystery disappears once we realized that the Coq implementation of that judgment was different from the paper and coincided with the definition that we had reverse-engineered from the counter-example to preservation[1]. We also found several typos and ambiguities in the typing rules in the published paper, but this is not unusual where there is no connection between the text in the paper and the model verified by the proof assistant.

[1] The Twelf "proof" is left as an exercise and the above typing judgment undefined.

That was encouraging, but to asses further the trade-off between exhaustive and randomized data generation, we resorted to some *mutation testing* [22] on the given debugged model. As we will see in Section 5, exhaustive generation, in all its naivety, tends to be more cost effective than pure random PBT. Finally, we report on some ongoing work on the list-machine 2.0 and on replaying the first section of [1], which deals with an abstract machine for dynamic information-flow control. Here again, exhaustive PBT shows its colours.

2 The list machine

We present here the syntax, static and dynamic semantics and finally meta-theory of CIVmark[2].

2.1 The plumbing of the machine

The list-machine, as the name suggests, operates over an abstraction of lists, where every value is either nil or the cons of two values, in other terms S-expressions with only nil as atom.

$$\text{value } a \ ::= \ \text{nil} \mid \text{cons}(a_1,\ a_2)$$

Given a set of *variables* and *labels*, the machine features the following set of instructions:

$\iota_1,\ \iota_2,\ \ldots$	$: I$	instructions
jump l	$: I$	jump to label l
branch-if-nil $v\ l$	$: I$	if $v = \text{nil}$ then jump to l
fetch-field $v\ 0\ v'$	$: I$	fetch the head of v into v'
fetch-field $v\ 1\ v'$	$: I$	fetch the tail of v into v'
cons $v_0\ v_1\ v'$	$: I$	make a cons cell in v'
halt	$: I$	stop executing
$\iota_1;\ \iota_2$	$: I$	sequential composition

We rely on several association lists: *programs* map labels to instructions and *stores* variables to values.

$$\begin{array}{rcl} \text{program } p & ::= & \textbf{end} \mid p, l_n : \iota \\ \text{store } r & ::= & \{\ \} \mid r[v \mapsto a] \end{array}$$

We use a functional notation such as $p(l)$ for look-up into such structures. The notation $r[v \mapsto a]$ assumes that the variable v is not in the domain of r and we use $r[v := a] = r'$ for functional update.

2.2 Dynamic and static semantics

The operational semantics is summarized in the inference rules in Fig. 1: given a fixed program, the small-step relation $(r,\ \iota) \overset{p}{\mapsto} (r',\ \iota')$ works on store/instruction configurations in a continuation-passing style. Then, a big-step semantics is defined as the Kleene closure of the small-step relation, with the instruction **halt** that signals the end of a program execution. A program p is said to *run* if it runs in the big-step relation, starting from the instruction at $p(l_0)$ with an initial store containing only the binding $v_0 = \text{nil}$, until a **halt** instruction is reached.

The type system assigns to each variable a list type that is then refined to empty and nonempty lists, to guarantee safety of certain operations, e.g. **fetch-field**.

$$\text{type } \tau \ ::= \ \text{nil} \mid \text{list } \tau \mid \text{listcons } \tau$$

$$\frac{}{(r,\ (\iota_1;\ \iota_2);\ \iota_3) \overset{p}{\mapsto} (r,\ \iota_1;\ (\iota_2;\ \iota_3))}\ \text{step-seq}$$

$$\frac{r(v) = \text{cons}(a_0,\ a_1) \quad r[v' := a_0] = r'}{(r,\ (\textbf{fetch-field}\ v\ 0\ v';\ \iota)) \overset{p}{\mapsto} (r',\ \iota)}\ \text{step-fetch-field-0}$$

$$\frac{r(v) = \text{cons}(a_0,\ a_1) \quad r[v' := a_1] = r'}{(r,\ (\textbf{fetch-field}\ v\ 1\ v';\ \iota)) \overset{p}{\mapsto} (r',\ \iota)}\ \text{step-fetch-field-1}$$

$$\frac{r(v_0) = a_0 \quad r(v_1) = a_1 \quad r[v' := \text{cons}(a_0,\ a_1)] = r'}{(r,\ (\textbf{cons}\ v_0\ v_1\ v';\ \iota)) \overset{p}{\mapsto} (r',\ \iota)}\ \text{step-cons}$$

$$\frac{r(v) = \text{cons}(a_0,\ a_1)}{(r,\ (\textbf{branch-if-nil}\ v\ l;\ \iota)) \overset{p}{\mapsto} (r,\ \iota)}\ \text{step-branch-not-taken}$$

$$\frac{r(v) = \text{nil} \quad p(l) = \iota'}{(r,\ (\textbf{branch-if-nil}\ v\ l;\ \iota)) \overset{p}{\mapsto} (r,\ \iota')}\ \text{step-branch-taken}$$

$$\frac{p(l) = \iota'}{(r,\ \textbf{jump}\ l) \overset{p}{\mapsto} (r,\ \iota')}\ \text{step-jump}$$

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

$$\frac{(r,\ \iota) \overset{p}{\mapsto} (r',\ \iota') \quad (p, r', \iota') \Downarrow}{(p, r, \iota) \Downarrow}\ \text{run-step}$$

$$\frac{}{(p, r, \textbf{halt}) \Downarrow}\ \text{run-halt}$$

$$\frac{[v_0 = \text{nil}] = r \quad p(l_0) = \iota \quad (p, r, \iota) \Downarrow}{p \Downarrow}\ \text{run-prog}$$

Figure 1: Small and big step operational semantics

The type system includes therefore the expected subtyping relation and a notion of *least common supertype* $\tau \sqcup \tau'$. A *type environment* Γ is a mapping between variables and types and subtyping is extended to environments width and depth-wise. A *program typing* Π is an association list of labeled environments, where $\Pi(l) = \Gamma$ represents the types of the variables when entering a block labeled with l.

$$\begin{array}{rcl} \text{typing env } \Gamma & ::= & \{\,\} \mid \Gamma, v : \tau \\ \text{program typing } \Pi & ::= & \{\,\} \mid \Pi, l : \Gamma \end{array}$$

Type-checking is stratified in several judgments following the structure of a program as a labeled sequence of blocks. At the bottom, instruction typing $\Pi \vdash_{\text{instr}} \Gamma\{\iota\}\Gamma'$ transforms a precondition Γ into post-condition Γ' under the program typing Π. The *check-block* relation handles terminal instructions **halt** and **jump**. Block typing is extended to all blocks of a program p with the *check-blocks* relation $\Pi \vdash_{\text{blocks}} p$, whereby for each label l in program p, the associated block $\iota = p(l)$ is type-checked with its environment $\Gamma = \Pi(l)$. Finally, the top-level *type-checking* relation $\models_{\text{prog}} p : \Pi$ states that a program p type-checks with a program typing Π when blocks and environments have matching labels, every block type-checks with its corresponding environment, and the initial environment is $\Gamma_0 = (v_0 : \text{nil})$.

$$\frac{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1\}\Gamma' \quad \Pi \vdash_{\text{instr}} \Gamma'\{\iota_2\}\Gamma''}{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1;\ \iota_2\}\Gamma''}\text{check-instr-seq}$$

$$\frac{\Gamma(v) = \text{list } \tau \quad \Pi(l) = \Gamma_1 \quad \Gamma[v := \text{nil}] = \Gamma' \quad \Gamma' \subset \Gamma_1}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil } v\ l\}(v : \text{listcons } \tau,\ \Gamma')}\text{check-instr-branch-list}$$

$$\frac{\Gamma(v) = \text{listcons } \tau}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil } v\ l\}\Gamma}\text{check-instr-branch-listcons}$$

$$\frac{\Gamma(v) = \text{nil} \quad \Pi(l) = \Gamma_1 \quad \Gamma \subset \Gamma_1}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil } v\ l\}\Gamma}\text{check-instr-branch-nil}$$

$$\frac{\Gamma(v) = \text{listcons } \tau \quad \Gamma[v' := \tau] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{fetch-field } v\ 0\ v'\}\Gamma'}\text{check-instr-fetch-0}$$

$$\frac{\Gamma(v) = \text{listcons } \tau \quad \Gamma[v' := \text{list } \tau] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{fetch-field } v\ 1\ v'\}\Gamma'}\text{check-instr-fetch-1}$$

$$\frac{\begin{array}{c}\Gamma(v_0) = \tau_0 \quad \Gamma(v_1) = \tau_1 \\ (\text{list } \tau_0) \sqcup \tau_1 = \text{list } \tau \quad \Gamma[v := \text{listcons } \tau] = \Gamma'\end{array}}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{cons } v_0\ v_1\ v\}\Gamma'}\ \text{check-instr-cons}$$

. . . . . .

$$\frac{}{\Pi;\Gamma \vdash_{\text{block}} \textbf{halt}}\text{check-block-halt}$$

$$\frac{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1\}\Gamma' \quad \Pi;\Gamma' \vdash_{\text{block}} \iota_2}{\Pi;\Gamma \vdash_{\text{block}} \iota_1;\iota_2}\text{check-block-seq}$$

$$\frac{\Pi(l) = \Gamma_1 \quad \Gamma \subset \Gamma_1}{\Pi;\Gamma \vdash_{\text{block}} \textbf{jump } l}\text{check-block-jump}$$

Figure 2: Instruction and block typing

2.3 Properties

One of the few drawbacks of PBT as a testing technique for programming units is coming up with meaningful properties. In MMT, this is a non-issue, as PL calculi come equipped with the meta-theorems they must satisfy, when they are not variants of standard results. As far as the list-machine is concerned, we are spoiled with choices: the paper (pages 467–8) lists more than a dozen theorems ranging from basic properties of subtyping to type soundness. We report the top-level ones:

Progress: given a well-typed instruction and a well-typed store, the machine steps or halt.

$$\frac{\models_{\text{prog}} p : \Pi \quad \Pi \vdash_{\text{instr}} \Gamma\{\iota\}\Gamma' \quad r : \Gamma}{\text{step-or-halt}(p,\ r,\ \iota)}$$

Preservation: if a well-typed block steps, there is an environment type-checking the next instruction.

$$\frac{\models_{\text{prog}} p : \Pi \quad \vdash_{\text{env}} \Gamma \quad r : \Gamma \quad \Pi;\Gamma \vdash_{\text{block}} \iota \quad (r,\ \iota) \overset{p}{\mapsto} (r',\ \iota')}{\exists \Gamma'.\ \vdash_{\text{env}} \Gamma' \ \wedge\ r' : \Gamma' \ \wedge\ \Pi;\Gamma' \vdash_{\text{block}} \iota'}$$

Soundness: a well-typed program, after a finite amount of steps, either halts or can make another step.

$$\frac{\models_{\text{prog}} p : \Pi \quad \Gamma = \Pi(l_0) \quad \iota = p(l_0) \quad (r,\ \iota) \overset{p}{\mapsto}{}^{*} (r',\ \iota')}{\text{step-or-halt}(p,\ r',\ \iota')}$$

Note that all the above properties are *existential* and while this is a non-issue for (constructive) theorem proving, it may be a challenge for testing, not only in the functional setting, but also when logic programming is concerned, as we shall see in Section 3.1.

The Twelf implementation adds more properties, mostly linked to its own peculiar meta-theory. So, what shall we test? Past experience with αCheck [7] suggests that testing intermediate lemmas is beneficial, while systems such as PLT-Redex tend to go for the bull's eye. We report in Section 5.2 some partial answers to this dilemma.

2.4 Mutation testing

How do we know that PBT is effective in catching bugs in MMT? There are very few F#/Prolog implementation of PL calculi in the wild that we can sink our metaphorical teeth in. In truth, PLT-Redex has collected a small set of bugged models, but for one, you really have to fancy Lisp[2], and secondly, it is not clear how meaningful those (very few) bugs are. Case in point, their list-machine model comes with only 3 (three) bugs that are caught by counterexamples with at most two instructions.

One way to asses the effectiveness of PBT is via *mutation testing* [22]. As well known, the latter is a form of white box testing, whereby a program is changed in a localized way by introducing a *fault*. The resulting program is called a "mutant" and the aim of a testing suite is to fail the faulted code, which is known as "killing" the mutant.

While mutation testing is widely applied to programming languages [16], the fact is that we do not have a theory for mutations of PL models. Standard mutation operators from (imperative) programming languages largely do not apply; hence we took some inspiration from operators for the closest we can get to models specified by derivation rules, Prolog [24]; this resulted in the following mutation operators, adapted to a strongly-typed setting:

Clause mutations: predicate deletion and swap, replacement of conjunction by disjunction.

Operator mutations: arithmetic and relational operator mutation.

Variable mutations: replacing variable by (anonymous) variable and vice versa

Constant mutations: replacing constant by constant, by (anonymous) variable and vice versa.

We thus manually produced two dozens mutations of the list-machine, half of which are specific to a relational implementation of part of the model. In Fig. 3 we show two sample mutations: the first one, ported from PLT-Redex, is a variable mutation, by which the *step-cons* rule fails to update the store forgetting r'. The second one is a constant mutation where, removing the list type constructor, v' gets the wrong type.

While we readily acknowledge that a manual approach to mutation testing is seriously limited, we point out that this is common in the field and that a general theory and implementation of a tool for mutation testing of PL calculi is beyond the scope of this paper.

[2] It is not just a question of syntax: the code for the only check in [15], namely *progress*, is so procedural to make it hard to relate to the formal statement in the paper.

$$\frac{r(v_0) = a_0 \quad r(v_1) = a_1 \quad r[v' := \mathrm{cons}(a_0,\, a_1)] = r'}{(r,\, (\textbf{cons}\ v_0\ v_1\ v';\ \iota)) \overset{p}{\mapsto} (\boxed{r},\, \iota)}\ \text{step-cons*}$$

$$\frac{\Gamma(v) = \mathrm{listcons}\ \tau \quad \Gamma[v' := \boxed{\cancel{\mathrm{list}}}\ \tau] = \Gamma'}{\Pi \vdash_{\mathrm{instr}} \Gamma\{\textbf{fetch-field}\ v\ 1\ v'\}\Gamma'}\ \text{check-instr-fetch*}$$

Figure 3: Sample mutations

3 αProlog implementation

αProlog [9] is a logic programming language particularly suited to encoding PL calculi and related systems due to its support for nominal logic. However, since this case study is purposely first order only, we did not get to use any of those goodies and the encoding is many sorted pure Prolog code. In fact, it follows quite closely the reference Twelf implementation by Appel [2]. The only place one would naturally try and use *name types*, and hence inherit α-equivalence, is encoding of variables and labels. Yet, the machine model calls for *distinguished* (initial) variable and label $\mathtt{v}_0$ and, $\mathtt{l}_0$ and this suggests an encoding based on an enumeration of constants — remember, we are in the business of bounded model checking, so we tend to avoid using infinite types as integers. The only downside is the need for an explicit inequality predicate. We use association lists for all the environments the list-machine uses (note the Haskell like syntax for type abbreviations) and standard Prolog predicates for related operations such as looking up or updating (non-destructively) such a list. Do not be fooled by the functional notation, it is flattened at compile time to the equivalent relation. The complete code can be found at `https://bitbucket.org/fkomauli/list-machine`.

```
var : type.
v0 : var.
v1 : var.
...
pred not_same_var (var, var).
not_same_var(v0, v1).
not_same_var(v0, v2).
...
type block =   (label,instr).
type program = [(block)].
type store =   [(var,value)].
...
func var_lookup (store,var) = value.
var_lookup([(V,A)|R],V) = A.
var_lookup([(V,A)|R],V1) = A1 :-
  not_same_var(V,V1), A1 = var_lookup(R,V1).
```

The small-step semantics is an unsurprisingly translation of the rules in Fig. 1 into a predicate `step (program, store, instr, store, instr)`, where the first three arguments are inputs. Type checking is slightly more interesting: we summarize the main type definitions:

```
pred check_instr   (program_typing,env,instr,env).
pred check_block   (program_typing,env,instr).
pred check_blocks  (program_typing,program).
pred check_program (program,program_typing).
```

Note that `check_instr(Pi,G,I,G')` will instantiate `G'` during execution. All other checks expect fully grounded inputs, i.e. no type inference is possible.

3.1 PBT with αCheck

αCheck [7] is a tool for checking desired properties of formal systems implemented in αProlog. The idea is to test properties/specifications of the form $H_1 \wedge \cdots \wedge H_n \rightarrow A$ by searching *exhaustively* (up to a bound) for a substitution θ such that $\theta(H_1), \ldots, \theta(H_n)$ all hold but the conclusion $\theta(A)$ does not. In this paper, we identify negation with *negation-as-failure*, but the tool includes also another strategy [8]. The concrete syntax for a check is

```
#check "name" depth:  G => A.
```

where `G` is a goal and `A` an atom or constraint[3]. As usual in Prolog the free variables are implicitly universally quantified and `depth` is the user-given bound. The above pragma is translated to the formula

$$\exists \vec{X} : \vec{\tau}.\ G \wedge\ \mathrm{gen}_{\vec{\tau}}(\vec{X}) \wedge\ \neg A \tag{1}$$

where $\mathrm{gen}_{\vec{\tau}}$ are type directed exhaustive generators automatically compiled by the tool to ground $\vec{X}$, the intersection of the free variables of G and A, needed to make the use of NF sound. The user can, alternatively, specify her own generators as αProlog code, if she feels she needs to implement a smarter generation strategy, as we will see shortly and in Section 6.2.

A query such as (1) amounts to a *derivation-first* approach, which generates all "finished" derivations of the hypothesis G up to a given depth, considers all sufficiently ground instantiations of variables, and finally tests whether the conclusion finitely fails for the resulting substitution. αCheck implements a simple-minded iterative deepening search strategy over a hard-wired notion of bound, which roughly coincides with the number of clauses that can used in the derivation of each of the premises

Let's look at a check to be more concrete, *progress*. While αProlog does not have first class existentials and disjunction, it is a basic Prolog exercise to code it:

```
pred step_or_halt (program,store,instr).
step_or_halt(P,R,instr_halt).
step_or_halt(P,R,I) :- step(P,R,I,R',I').

#check "progress" 10: check_program(P,Pi), check_block(Pi,G,I), store_has_type(R,G)
                        => step_or_halt(P,R,I).
```

Keeping in mind that the tool add generators for `P,R,I` before trying to refute `step_or_halt(P, R, I)`, informal mode analysis for `step` tells us that `R',I'` will be ground when the partial proof tree for the check is built.

The same approach will not work for *preservation*:

```
pred exists_env (program_typing,store,instr).
exists_env(Pi,R,I) :-
  store_has_type(R,G), env_ok(G), check_block(Pi,G,I).

#check "pres" 20 : check_program(P,Pi), step(P,R,I,R',I'), env_ok(G),
                   store_has_type(R,G),check_block(Pi,G,I)
                       => exists_env(Pi,R',I').
```

This because `store_has_type(R,G)` expects `G` to be ground and we are in effect trying to use αProlog for an impossible type *inference* task. The solution is to write a specialized generator `build_env` for type environments (and recursively, for types and variables). The peculiarity is that we need to add a *local* depth bound, so that smallish environments can be built independently from the hard-wired bound, which is additively distributed along all the atoms in the check.

[3] Since in this paper we make no use of nominal features, this means syntactic equality.

```
pred exists_env_b (int,program_typing,store,instr).
exists_env_b(N,Pi,R,I) :-
  N > 0, exists_env_b(N - 1,Pi,R,I).
exists_env_b(N,Pi,R,I) :-
  N >= 0, build_env(N,G), store_has_type(R,G), env_ok(G), check_block(Pi,G,I).
#check "pres_b" 20 : ... => exists_env_b(4,Pi,R',I').
```

Please see Section 5 for empirical results. Even discounting our obvious bias, specifying and validating properties ("spec'n'check") in αCheck is dead simple, requires very little effort and more than often turns out to be pretty useful.

4 F# implementation

The granddaddy of PBT being QuickCheck, it is natural to look into a functional implementation of the benchmark to establish a baseline for αCheck's efficacy. We chose *FsCheck* (`https://fscheck.github.io/FsCheck`), partly because we were familiar with it, but mostly because, differently from other libraries, we hoped we could leverage FsCheck's ability to provide automatic data generation for any datatype via .Net reflection to start spec'n'check without further ado. How wrong we were.

The functional implementation of the machine is unremarkable: we use F#'s maps for all sorts of environments and integers for variables and labels. The `step` function is total, since type-checking see to that. The type-checking function follows the imperative specification in the paper (Section 8.2).

This is what a check such as *progress* looks like in FsCheck's DSL, keeping in mind that `==>` operationally means: (lazily) pass to the post-condition all the tests that satisfies the pre-conditions and discard the other up to a (configurable) limit:

```
(typecheck-program pi p) && (typecheck-block pi g i) && (store-has-type r g)
   ==>  lazy (step-or-halt p r i)
```

Having carefully read [1], we were expecting low coverage for checks as the above, since uniform distributions are not likely to find data that has to satisfy severe constraints; still, we were not quite ready for the number we got: *zero*. FsCheck, as typical in adaptations of QuichCheck, provides a monadic language to write your own generators and so we did. In [1], the emphasis was writing generators so that the machine would run longer without getting stuck; here we need to produce well-typed programs, and this is far from immediate, since we have no type-inference whatsoever. For the progress property, we need to generate simultaneously a program `p`, a program typing `pi` that type-checks with `p`, a store `r` compatible with a type environment `g`, a label `l` that belongs to program `p` and the instruction `i` associated to label `l`.

Rather than showing the code, which is available in the repo `https://bitbucket.org/fkomauli/list-machine/branch/fsharp`, we sketch some of the strategy. The starting point is giving more bias to `cons` instructions, so as to populate the store giving a chance for branching and fetching to do something interesting. We choose an instruction only if we know it is safe to execute in the given type environment. With the instruction itself, the generators produces the environment updated after the execution of the instruction. We then build sequence of non-terminal instructions. Jumps are delicate: a jump is always directed forward to a random label with id greater than the current block. If the chosen landing block has an incompatible typing environment or there is no label to jump forward to, then an **halt** instruction is generated instead. And this is just part of the reasoning behind it.

The smart generators developed for this benchmark consists of over a hundred lines of very dense F# code. Even if it sounds like a small number, the time spent behind it was dozens of hour dedicated

to studying data correlation and distribution to generate meaningful objects. The process continued by improving distributions and fixing bugs that were found out by analyzing coverage. Though not a formal proof, we also validated with *FsCheck* itself the soundness of generators, as far as providing data that satisfies the constraints we imposed. Completeness is out of the question, but even in systems such as QuickChick [21] where such proofs are expressible, this is far from automatic.

Shrinkers are a necessary evil of random generation, transforming large counterexamples into smaller ones that can be understood and acted upon. The QuickCheck philosophy across the board is to put in the hand of the user all the hard choices and implementing shrinkers is one. While shrinking blocks can be achieved by removing non-terminal instructions, we must safeguard several sources of *correlation* between data: for example, programs and program typings cannot be shrunk independently, as we must ensure that they still type-check. For the latter we wrote a type inference function to generate them together from scratch. This strategy worked well enough to reduce counterexamples to sizes sufficiently close to the minimal ones easily found with the iterative deepening approach of αCheck.

5 Experimental results

Reasons of space and the desire not to bore the reader silly suggest to present only a selection of all the experiments that we have carried out. Full details can be found in [18].

5.1 A cautionary tale

A testing approach is any good only if it uncovers bugs, but we were not expecting to find any in the model presented in the paper, which came equipped with a type soundness proof formalized in two proof assistants. So, imagine our surprise when FsCheck, and later on αProlog came up with this counterexample to type preservation:

$$p = (l_0 : \text{cons}(v_0, v_0, v_0);\ \text{jump}\ l_1);\ (l_1 : \text{fetch-field}(v_0, 0, v_0);\ \text{jump}\ l_2);\ (l_2; \text{halt})$$
$$\Pi = (l_0 : [v_0 \mapsto \text{nil}]);\ (l_1 : [v_0 \mapsto \text{listcons nil}]);\ (l_2 : [v_0 \mapsto \text{nil}])$$
$$r = [v_0 \mapsto \text{cons}(\text{cons}(\text{nil}, \text{nil}), \text{nil})]$$
$$i = \text{fetch-field}(v_0, 0, v_0);\ \text{jump}\ l_2 \qquad (\textit{instruction at}\ l_1)$$

which after a single step yields:

$$r' = [v_0 \mapsto \text{cons}(\text{nil}, \text{nil})]$$
$$i' = \text{jump}\ l_2$$

However, there can be no Γ' satisfying the postconditions: any such type environment must contain either the binding $[v_0 \mapsto \text{listcons}\ \tau]$ or $[v_0 \mapsto \text{list}\ \tau]$ to accommodate r', but both would not be compatible with what is required by the jump.

After a fair amount of soul searching, we zeroed on the encoding of the the judgment *value-has-ty*, (page 481 of [2]).

$$\frac{}{\text{nil} :\ \text{nil}} \qquad \frac{}{\text{nil} :\ \text{list}\ \tau} \qquad \frac{}{\text{cons}(a_0,\ a_1) :\ \text{listcons}\ \tau} \qquad \frac{a :\ \text{listcons}\ \tau}{a :\ \text{list}\ \tau}$$

This is an essential component of the definition of $r : \Gamma$, which can be stated as for all $v \in \text{dom}(r)$, it holds $r(v) : \Gamma(v)$. The *listcons* case sounds fishy, since it makes no assumption about the types of a_0 and

a_1. Once we changed that case to:

$$\frac{a_0 : \tau \quad a_1 : \text{list } \tau}{\text{cons}(a_0,\ a_1) :\ \text{listcons } \tau}$$

the counterexample failed to show up. This change was also confirmed by inspecting the Coq implementation (it is left as an exercise in the Twelf one), which defines *value-has-ty* exactly like that.

Less dramatically, the *check-instr-branch-listcons* typing rule in the original paper contains additional preconditions similar to those in the *check-instr-branch-list* rule, regarding the jump target environment; however in the Coq implementation they are not present, and they could not be, considering the proof of equivalence with the algorithmic version of type checking, where they are notably absent. Several typos also present, viz. rule *check-instr-cons0* (Section 8.1) and in the type checking algorithm (instruction `typecheck_instr` $\Pi\ \Gamma\ \iota$). Typos were spotted through formalization, not PBT.

The moral is, of course, that if there is no formal connection between a formalization and the paper reporting it, Isabelle/HOL and literate Agda being the precious exceptions, you may want to be skeptical of the latter. Similar findings appear in [17].

5.2 Mutation analysis

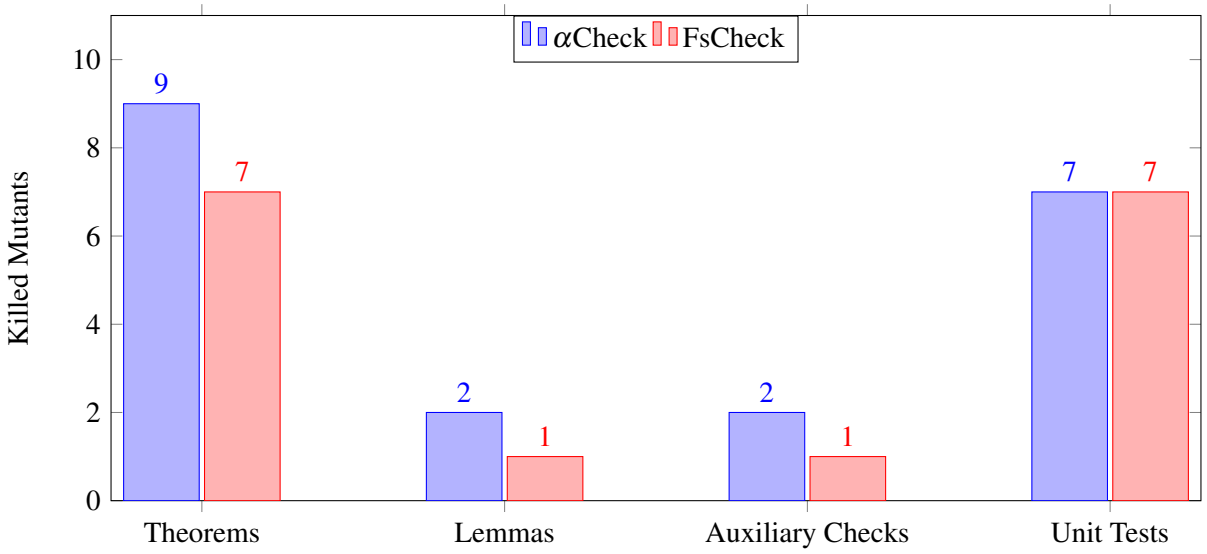

Figure 4: Mutation analysis for all properties and unit tests

We used our "home-baked" mutation testing to assess the efficacy of PBT suites implemented with αCheck and FsCheck w.r.t. the 11 mutations applicable to both implementation. Checks are divided in three groups: top-level theorems as in our Section 2.3, intermediate lemmas collected in Section 8 of [2] and auxiliary checks regarding low-level details of the machine specification. We have also ported from PLT-Redex to both αProlog and F# few dozens unit tests, as an additional baseline for mutant killing. In doing, so we have made some of them *parametric*, so that exhaustive and random generation of those parameters could make those tests more far-reaching.

Checks that took more than a minute to find a counterexample are considered timed-out[4]. We do

[4]Checks executed on machine with Intel Core i5–4–200U CPU,a clock speed of 1.60GHz and 8GB of RAM, with 64 bit Ubuntu 17.10 Artful and Linux kernel 4.13.0

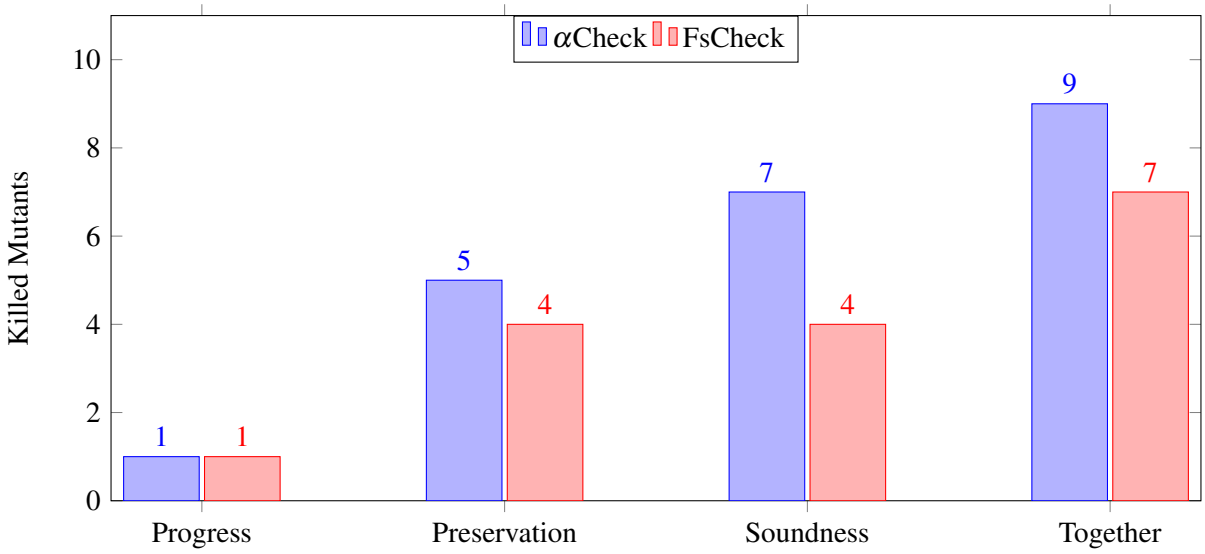

Figure 5: Mutation analysis for top-level theorems

not compare times to find counterexamples in αCheck vs. FsCheck, because it is a serious case of oranges and apples. We did take timing information's separately, and again we refer to [18] for the full breakdown. In summary, the FsCheck implementation of PBT is quick (as promised, although not as much as you would expect), not only because of the efficiency of the host language, but also thanks to the flexibility of the configuration of how checks are executed. The whole FsCheck BPT suite completes in around 5 minutes: roughly 10 seconds are taken by top-level checks and about 40 seconds by the intermediate lemmas, while the remaining time is taken by the auxiliary checks. Instead, execution time of the αCheck suite is not as predictable: we manually set the depth bound so as to make the execution stay withing 10 seconds. Exception to this discipline are the *progress* and *preservation* checks, whose search space size tend to explode several minutes to complete. Unit tests, even in a parametric fashion, execute in just few seconds in both implementations.

What we can reasonably compare is the rate of mutants killed by the two testing approaches (Fig. 4) and the αCheck implementation comes marginally ahead. Lemmas and low-level checks did not perform well, while top-level theorems were capable of killing most mutants. In fact, the *soundness* property was the one which killed most, as we can see in Fig. 5 and it is a good countermeasure against possible errors contained in the model. This is consistent to the PLT-Redex model, where soundness is the only property checked.

While we do not want to read too much in such a limited experimentation, it is safe to say that αCheck keeps its ground, considering how inexpensive it is to set up.

6 Extensions

Here we very briefly report on extending the list machine model to its 2.0 version, see `http://www.cs.princeton.edu/~appel/listmachine/2.0/` and on replicating some of the results in [1]. In both cases, we fixed αCheck as the testing approach: in the first case, to showcase how nice it is not having to adapt generators and shrinkers and in the second one because the random approach has already been

thoroughly investigated by much smarter people than us.

6.1 List-machine 2.0

In the final part of the paper [2], the model is extended to cater for *indirect* jumps. This entails some small but far-reaching changes: labels are now coerced into values and the *nil* value is replaced by the initial label $\mathtt{l}_0$. We generalize the **jump** instruction and add a way to get the current label. This yields the following changes to the operational semantics:

$$\frac{r(v) = l \quad p(l) = \iota'}{(r,\ \textbf{jump}\ v) \overset{p}{\mapsto} (r,\ \iota')}\ \text{step-jump} \qquad \frac{r[v := l] = r'}{(r,\ \textbf{get-label}\ l\ v; \iota) \overset{p}{\mapsto} (r,\ \iota)}\ \text{step-get-lab}$$

We modify the type system for the new instruction(s):

$$\frac{\Pi(l) = \Gamma_1 \quad \Gamma \subset \Gamma_1 \quad v \notin \mathrm{dom}(\Gamma_1)}{\Pi; \Gamma \vdash_{\text{block}} \textbf{get-label}\ l\ v;\ \textbf{jump}\ v}\ \text{check-block-jump2}$$

Updating the αProlog implementation to reflect those changes is a matter of half an hour and we refer to the online documentation for details (`https://bitbucket.org/fkomauli/list-machine/2.0`). As the paper does not detail the soundness proof, the first thing we did was running our PBT suite to validate our implementation of the extension.

Preservation as formulated beforehand fails! This time, it is not a specification error, but the fact that the statement of the theorem has to be generalized. In fact, in the new typing rule both instructions have to occur in sequence and if we take a single step after **get-label**, no typing rule will match the remaining **jump**, thus making the typing requirement in the existential conclusion fail. One solution is to modify the latter so as to try another optional step to "get over" the **jump**. After this modification, αCheck did not find any other counterexamples w.r.t. the 2.0 model.

6.2 Testing non-interference, not so quickly

It is natural to consider, in addition to type soundness, more intensional properties, such as *dynamic secure information-flow control* (IFC). This choice is not casual, since this is the topic of [1], which uses classic QuickCheck. The paper claims that, for the hardest-to-find bugs, minimal counterexamples are well beyond the scope of naive exhaustive testing. Thus, we want to test this. We mainly concentrate on Section 1–4 of [1] and refer the reader to the paper for full details of the model and their testing outcome.

The starting point is a simple, but not simplistic abstract stack-and-pointer machine with instructions such as *push*, *pop*, *load* and *store*. A machine state consists of a program counter, a stack, a random-access memory, and a fixed instructions sequence. In dynamic IFC, security levels (labels, here only public $\bot$ and private $\top$) are attached to run-time values and propagated during the execution, making sure that private data does not leak. In this model, the values manipulated by the machines are labeled integers $x@L$. The property that the abstract machine must satisfy is *end-to-end noninterference*: if we call *indistinguishable* two machine states if they differ only in their private values, non interference guarantees that in any execution starting with indistinguishable states and ending in a halted state, the final states are indistinguishable as well.

In [1, Section 2.3], the authors present an operational semantics for the machine, which, while intuitive, turns out to be bugged; then they detail (Section 4) the sophisticated testing strategies they had to program to catch the *four* bugs inserted. To give an idea, on the left we report the buggy rule for *Load*

and on the right the fixed one, where *i(pc) = Load* and the box signals where the issue is.

$$\frac{}{\langle pc,(x@L:s),m\rangle \Longrightarrow \langle pc,(\boxed{m(x)}:s),m\rangle}\ Load^{*} \qquad \frac{}{\langle pc,(x@L:s),m\rangle \Longrightarrow \langle pc,(\boxed{m(x)@L}:s),m\rangle}\ Load^{OK}$$

QuickCheck managed to locate the first two bugs with a relatively naive generation strategy by which indistinguishable states where generated together by randomly creating the first and modifying the second in their secret part. The other two bugs necessitated a far more complex strategies for generating meaningful sequences of instructions and addresses, akin to the ones we used for the *FsCheck* implementation of the list machine.

αCheck found the first two bugs in less than a minute without any setup. The third one required writing a generator that yields more structured programs, such as sequences of *push* and *store* so that values in memory can change during execution, a necessary condition to find distinguishable states. This generator is a simplification of the *weighted* and *sequence* strategy in [1]. The fourth bug seemed, however, out of reach of αCheck, until we got it instead using a plain αProlog query for non-interference, rather than a check, the difference being the search strategy: depth-first vs. un-optimized iterative deepening. After the last bug has been fixed, the query did not find any counterexample and completes in about 5 minutes.

Section 2 in the paper ends by introducing three additional and more outlandish bugs. We got them all with the previous techniques (vanilla check, check with a generator and query with a generator) with queries being the more efficient one. Additionally, we have also carried out some experiments with a second version of the machine that includes a *jump* instruction [1, Section 5], but the results are too preliminary to draw any conclusion.

7 Conclusions and future work

Our experience suggests that off-the-shelf PBT tools are already quite useful in validating the metatheory of PL models, may they be already formalized as the list machine and their evolution, or the more if under under development. PBT helps in finding errors in the specifications and also in adjusting the statements of theorems when the model changes, as in the case of list-machine 2.0. From the costs/benefits perspective, exhaustive generation, even in the naive way we have considered, seems to be a winner over the random approach. Validating low-level languages brings in more challenges, but those can be handled with the tools we have and some additional work. Clearly, there are many other TAL models, see Chapter 4 of [23], that could confirm this conclusion.

We are keenly aware that *FsCheck* and αProlog are the extremes of a small number of PBT-tools that we could have used for this case study, starting from running PLT-Redex on our mutations, to other logic programming model-checkers such as *Bedwyr* [3]. The present paper is not meant to be an exhaustive (sorry for the pun) survey of any applicable tool. Still, a gap that should be filled is replaying the benchmark with approaches that goes beyond pure generate-and-test and try to automatically derive (random) generators that intrinsically satisfy certain pre-conditions. The obvious candidate is the new QuickChick [19], but also Bulwahn's smart generators in Isabelle/HOL[5] are a possibility.

αCheck performs better than we hoped for, considering that its implementation is nothing more than a OCaml interpreter for nominal logic programming, exploring the full search space in an iterative deepening way up to a bound. In other terms, it is probably orders of magnitude slower than standard Prolog and it makes no effort to prune the search space or to explore it in more flexible ways. As the preservation example shows, a hard-wired additive bound is, to say the least, inconvenient and therefore

it is natural to try and make the search strategy more modular, possibly using the notion of *hookable* disjunction from *TOR* [28]. Orthogonally, model-checking is all about pruning the search space and in the context of MMT, ideas from [25] could be valuable.

Finally, it is folklore that *linear* logical framework are well suited to represent in an elegant and concise way PL calculi with imperative features, as initially suggested by [6]. However, when it comes to *reason* about them, the offering is quite meagre, with a few exceptions [20]. This is due to the (current) lack of meta-reasoning facilities in the leading (well, the only) sub-structural logical framework, *Celf*. An alternative worth pursing hence is validation via PBT in such a setting. In [26], the authors have presented a proof theoretical reconstruction of PBT for relational specifications: this leverages the Foundational Proof Certificate framework [10] to describe both the generation and the testing phase in terms of focused search. The idea translates directly to linear logic as well and can be immediately prototyped by encoding what it boils down to very simple *Lolli*-like meta-interpreters in a host language such as λProlog. The conjecture is that a linear encoding would help particularly exhaustive data generation by making structures such as stores and type environments *implicit* and turning them into (atomic) logical assertions.

References

[1] Catalin Hritcu et al. (2013): *Testing noninterference, quickly*. In: *ICFP*, ACM, pp. 455–468.

[2] Andrew W. Appel, Robert Dockins & Xavier Leroy (2012): *A List-Machine Benchmark for Mechanized Metatheory*. *J. Autom. Reasoning* 49(3), pp. 453–491.

[3] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur & Alwen Tiu (2007): *The Bedwyr System for Model Checking over Syntactic Expressions*. In Frank Pfenning, editor: *CADE*, *LNCS* 4603, Springer, pp. 391–397. Available at `http://dx.doi.org/10.1007/978-3-540-73595-3_28`.

[4] Jasmin Christian Blanchette, Lukas Bulwahn & Tobias Nipkow (2011): *Automatic Proof and Disproof in Isabelle/HOL*. In Cesare Tinelli & Viorica Sofronie-Stokkermans, editors: *FroCoS*, *LNCS* 6989, Springer, pp. 12–27. Available at `http://dx.doi.org/10.1007/978-3-642-24364-6_2`.

[5] Lukas Bulwahn (2012): *The New Quickcheck for Isabelle - Random, Exhaustive and Symbolic Testing under One Roof*. In Chris Hawblitzel & Dale Miller, editors: *CPP*, *LNCS* 7679, Springer, pp. 92–108. Available at `http://dx.doi.org/10.1007/978-3-642-35308-6_10`.

[6] Iliano Cervesato & Frank Pfenning (2002): *A Linear Logical Framework*. *Information and Computation* 179(1), pp. 19 – 75, doi:https://doi.org/10.1006/inco.2001.2951.

[7] James Cheney & Alberto Momigliano (2017): *αCheck: A mechanized metatheory model checker*. *TPLP* 17(3), pp. 311–352.

[8] James Cheney, Alberto Momigliano & Matteo Pessina (2016): *Advances in Property-Based Testing for αProlog*. In Bernhard K. Aichernig & Carlo A. Furia, editors: *TAP 2016*, *LNCS* 9762, Springer, pp. 37–56, doi:10.1007/978-3-319-41135-4_3.

[9] James Cheney & Christian Urban (2008): *Nominal logic programming*. *ACM Trans. Program. Lang. Syst.* 30(5), pp. 26:1–26:47.

[10] Zakaria Chihani, Dale Miller & Fabien Renaud (2013): *Foundational Proof Certificates in First-Order Logic*. In Maria Paola Bonacina, editor: *Automated Deduction - CADE-24*, *LNCS* 7898, Springer, pp. 162–177, doi:10.1007/978-3-642-38574-2_11.

[11] Koen Claessen & John Hughes (2000): *QuickCheck: a lightweight tool for random testing of Haskell programs*. In: *ICFP*, ACM, pp. 268–279.

[12] Jonas Duregård, Patrik Jansson & Meng Wang (2012): *Feat: functional enumeration of algebraic types*. In Janis Voigtländer, editor: *Haskell Workshop*, ACM, pp. 61–72, doi:10.1145/2364506.2364515.

[13] Guglielmo Fachini & Alberto Momigliano (2017): *Validating the Meta-Theory of Programming Languages (Short Paper)*. In Alessandro Cimatti & Marjan Sirjani, editors: *SEFM 2017*, *LNCS* 10469, Springer, pp. 367–374, doi:10.1007/978-3-319-66197-1_23.

[14] Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes & Robert Bruce Findler (2015): *Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System*. In Jan Vitek, editor: *ESOP 2015*, *LNCS* 9032, Springer, pp. 383–405, doi:10.1007/978-3-662-46669-8_16.

[15] Robbie Findler: *The PLT-Redex list-machine model*. `https://github.com/racket/redex/tree/master/redex-benchmark/redex/benchmark/models`. Latest commit: Oct 2016.

[16] Yue Jia & Mark Harman (2011): *An Analysis and Survey of the Development of Mutation Testing*. *IEEE Trans. Software Eng.* 37(5), pp. 649–678.

[17] Casey et al. Klein (2012): *Run your research: on the effectiveness of lightweight mechanization*. In: *POPL '12*, ACM, New York, NY, USA, pp. 285–296, doi:10.1145/2103656.2103691.

[18] Francesco Komauli (2018): *Property-Based Testing Abstract Machines*. Master's thesis, DI, University of Milan, doi:10.13140/RG.2.2.27992.39681.

[19] Leonidas Lampropoulos, Zoe Paraskevopoulou & Benjamin C. Pierce (2018): *Generating good generators for inductive relations*. *PACMPL* 2(POPL), pp. 45:1–45:30, doi:10.1145/3158133.

[20] Alberto Momigliano & Jeff Polakow (2003): *A formalization of an Ordered Logical Framework in Hybrid with applications to continuation machines*. In: *MERLIN*, ACM.

[21] Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos & Benjamin C. Pierce (2015): *Foundational Property-Based Testing*. In: *ITP*, *LNCS* 9236, Springer, pp. 325–343.

[22] Mauro Pezzè & Michal Young (2007): *Software testing and analysis - process, principles and techniques*. Wiley.

[23] Benjamin C. Pierce, editor (2005): *Advanced Topics in Types and Programming Languages*. MIT Press.

[24] Juliano R Toaldo & Silvia Vergilio (2016): *Applying Mutation Testing in Prolog Programs*.

[25] Michael Roberson, Melanie Harries, Paul T. Darga & Chandrasekhar Boyapati (2008): *Efficient software model checking of soundness of type systems*. In Gail E. Harris, editor: *OOPSLA*, ACM, pp. 493–504. Available at `http://doi.acm.orberg/10.1145/1449764.1449803`.

[26] Dale Miller Roberto Blanco & Alberto Momigliano (2017): *Property-Based Testing via Proof Reconstruction: Work-in-progress*. Available at `http://www.dimi.uniud.it/assets/preprints/5-2017-miculan.pdf`. LFMTP'17, Oxford.

[27] Colin Runciman, Matthew Naylor & Fredrik Lindblad (2008): *Smallcheck and lazy smallcheck: automatic exhaustive testing for small values*. In: *Haskell*, ACM, pp. 37–48.

[28] Tom Schrijvers, Bart Demoen, Markus Triska & Benoit Desouter (2014): *Tor: Modular search with hookable disjunction*. *Science of Computer Programming* 84, pp. 101 – 120, doi:https://doi.org/10.1016/j.scico.2013.05.008. PPDP 2012.

[29] Xuejun Yang, Yang Chen, Eric Eide & John Regehr (2011): *Finding and understanding bugs in C compilers*. In: *PLDI*, ACM, pp. 283–294.