Property-Based Testing of the Meta-Theory of Abstract Machines: an Experience Report

Francesco Komauli and Alberto Momigliano

DI, Università di Milano, Italy

Abstract. 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. 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 [ADL12], which we tackle with αCheck, the simple model-checker on top of the nominal logic programming αProlog. 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 carry out some *mutation testing* on the given model, to asses the trade-off between exhaustive and randomized data generation, using for the latter the PBT library *FSCheck* for *F#*. Spoiler alert: αProlog 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, e.g. [BBN11,PHD+15].

In this paper we are not concerned with the issue of verification vs. validation in general, but in a particular domain: the mechanization in a logical framework of the meta-theory (MMT) of programming languages (PL) and related calculi. To fix ideas, think about the formal verification of compiler correctness [Ler09]. As most practitioners may testify, the main properties of interest are well known and have mathematically shallow proofs. The difficulty lies in the potential magnitude of the cases that one must consider and in the trickiness that some encodings require.

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 designer of PL artifacts in her work in *developing*, and eventually formally proving correct, such calculi, rather than concentrating on representations that we already know to be correct.

The technique we adopt is *property-based testing* (PBT), which successfully combines two very old ideas: automatic test data generation and refuting executable specifications. Pioneered by *QuickCheck* for functional programming [CH00], PBT tools are now available for pretty much every programming language and having a growing impact in industry [Hug07][1]. To make this concrete, consider the following snippet of a (faulty) Prolog specification of an ordered list and a check validating the preservation of order by insertion (we are using the concrete syntax of αCheck, more on this later):

```
ordered([]).
ordered([X]).
ordered([X,Y|Xs]):-less_equal(X,Y),ordered(Xs). % should be ordered([Y|Xs]).

insert(X,[],[X]).
insert(X,[Y|Ys],[X,Y|Ys]):- less_equal(X,Y).
insert(X,[Y|Ys],[Y|Zs])  :- greater(X,Y), insert(X,Ys,Zs).

#check "ins_corr" 15: ordered(Xs), insert(X,Xs,Ys) => ordered(Ys).
Checking depth 1 2 3 4 5 6 7 8 9 10. Total: 0.012 s:
X = z
Xs = [z,s(z),z]
Ys = [z,z,s(z),z]
```

The tool reports a minimal counterexample to the conjecture, namely a substitution that verifies the antecedent but not the consequent.

PBT is also applicable at the meta-programming level, that is where the properties of interest are the theorems that some PL artifact (think again about a compiler) must satisfy. Significant examples in this area can be found in [Kle12,FM17].

αCheck [CM17] is a light-weight PBT tool built on top of αProlog [CU08], a logic programming language based on *nominal* logic. The latter is particularly suited to represent object logics where *binders* and associated notions such as α-equivalence, (declarative) generation of *new* names, capture-avoiding substitutions are paramount. In contrast to QuickCheck, αCheck uses its logic programming engine to perform *exhaustive symbolic* search for counterexamples, as we elaborate further in Section 3.1.

In this paper, we take αCheck beyond the comfort of PBT of high-level object languages, which have been investigated extensively (`https://github.com/aprolog-lang/checker-examples`), and address the validation of *abstract machines* and (typed) assembly languages [Mor05]. This domain seems more challenging from a validation standpoint, since instructions operating at a low level provide less "structure" while searching for counterexamples, which then tend to be substantially more complex. Further, it seems hard to generate meaningful se-

[1] An exception is the logic programming world, where the porting of QuickCheck to SWI-Prolog (`http://www.swi-prolog.org/pack/list?p=quickcheck`) seems broken, nor has the work in [AFSC14] been released.

quences of machine states and machine runs so that properties may be validated with a reasonable coverage. Similar remarks have appeared in the context of PBT of C programs [YCER11]. Not coincidentally, the authors of [Hca13] suggests that "[counterexamples to properties such as dynamic non-interference in abstract machines may be] well beyond the scope of naive exhaustive testing" (op.cit. pag. 12).

We validate with αCheck Appel and al.'s list-machine benchmark [ADL12] (CIVmark, **C**ompiler **I**mplementation **V**erification). We concentrate mostly on its base version 1.0, but we also touch on 2.0, see Section 5.1. CIVmark is conceived as a benchmark for "machine-checked proofs about real compilers". Two implementations, including sketch of proofs of type soundness in the proof assistants Twelf and Coq come with the paper [ADL12].

PBT's data generation strategy comes in other flavours, beyond the exhaustive one taken by αCheck: random [CH00] and the combination of the two based on functional enumeration [DJW12], with an ever increasing emphasis on coroutining the generation and testing phases [Fca15,LPP18]. αCheck is unique in taking the logic programming way, which naturally supports many of the optimizations currently researched in the functional setting; the latter is re-discovering logic programming features such as mode inference [Bul12b] or CLP [Fca15]. Another contribution of this paper is to compare the merits of exhaustivity-based PBT in a logic programming style versus the more usual randomized functional setting popularized 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.

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 [ADL12], caused by an incorrect specification of typing for values, see Section 4.1. This is particularly striking, considering the paper comes with two *formalized* proofs of the main theorems claimed for the list-machine model. The mystery disappears once we realized that the Coq implementation of that judgment was different from what was reported in the paper. 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.

That was encouraging, but to assess further the efficacy of our approach, we resorted to *mutation testing* [PY07] of the given debugged model. To gauge the trade-off between exhaustive and randomized data generation, we encoded the same model in the strict functional language *F#* and tested with its checker *FsCheck* (`https://fscheck.github.io/FsCheck`). As we will see in Section 4, 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 [Hca13]. Here again, exhaustive PBT shows its colours.

2 The list-machine

We present here the highlights of the syntax, static and dynamic semantics of CIVmark. It consists of a basic pointer machine with instructions that build and destruct a list, and (un)conditional jumps to iterate them. The dynamics of the machine is

described with a standard SOS, accompanied by a reasonably sophisticated type system which, being able to make static predictions about lists being empty or not, guarantees that a well typed machine's run does not get stuck. Full listings of the rules are in the electronic appendix [KM18].

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

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

Given a set of *variables* v and *labels* l, 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

Programs are sequences of labelled instructions and *stores* map variables to values.

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

We use a functional notation such as $p(l)$ for look up into such structures. Writing $r[v \mapsto a]$ 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 behaviour of the machine is described by a small-step relation $(r,\ \iota) \overset{p}{\mapsto} (r',\ \iota')$ that, given a fixed program p, works on store/instruction configurations (r, i) in a continuation-passing style. We comment on some of the rules:

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

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

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

If we are executing a **fetch-field** $v\ 0\ v'$, we look up in the store the value of v, which better be a cons cell, and we update the store assigning to v' its head. If the tag is 1, we assign to v' its tail and in both cases return a new configuration

consisting of the updated store and the continuation instruction. Similarly, **cons** v_0 v_1 v' will build a cons cell of the values of the first two variables in the store and assign it to v'.

A chain of computations is built as the Kleene closure of the small-step relation, with the instruction **halt** signaling the end of a program execution. A program p is said to *run* if it runs in the Kleene closure, starting from the instruction at $p(l_0)$ with an initial store containing only the binding $v_0 \mapsto$ nil, until a **halt** instruction is reached. For example, this program from [ADL12] consists of three blocks that builds a list of length 3 and iterates over it until it is empty.

L_0 : **cons** v_0 v_0 v_1; **cons** v_0 v_1 v_1; **cons** v_0 v_1 v_1; **jump** L_1,
L_1 : **branch-if-nil** v_1 L_2; **fetch-field** v_1 1 v_1; **branch-if-nil** v_0 L_1; **jump** L_2,
L_2 : **halt**,
end

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

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

The type system includes therefore the expected subtyping relation and a notion of *least common super-type* $\tau \sqcup \tau'$. 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.

$$\text{typing env } \Gamma ::= \{\ \} \mid \Gamma, v : \tau$$
$$\text{program typing } \Pi ::= \{\ \} \mid \Pi, l : \Gamma$$

Type-checking is stratified in several judgments following the structure of a program as a labeled sequence of blocks. At the bottom, instruction typing $\Pi \vdash_{\text{instr}} \Gamma\{\iota\}\Gamma'$ can be seen as a sort of Hoare triple by which an instruction transforms a typing environment (a precondition) Γ into post-condition Γ' under the fixed program typing Π. We list here some of those rules:

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

$$\frac{\Gamma(v) = \text{listcons } \tau \quad \Gamma[v' := \text{list } \tau] = \Gamma}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{fetch-field } v\ 0\ v'\}\Gamma'}\ \text{check-instr-fetch-1}$$

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

Typing **fetch-field** v 0 v' in an environment where v is a non-empty list of τ will yield a new environment where v' has that very type τ. If we fetch the tail, then v' will have some list type. In the **cons** case, we want v to have a non empty list type in the resulting environment, but of which carrier type? The smallest one in the subtyping order compatible with the types of v_0, v_1.

The above judgment is the basis for checking *blocks* $(\Pi; \Gamma \vdash_{\text{block}} \iota)$, that is sequences of instructions terminated by a **halt** or a **jump**. Finally, the top-level relation $p : \Pi$ checks a program p against a program typing Π when blocks and environments have matching labels, every block typechecks with its corresponding environment, and the initial environment is $\Gamma_0 = (v_0 : \text{nil})$.

The following is a program typing compatible with the sample program previously described: initially only v_0 is present in the environment, with type nil; in the L_1 environment, v_0 keeps the same type because it may not be altered by the program, while v_1 has type *list nil* because it can be either a listcons coming from L_0 or an empty list coming from the second **branch-if-nil** in the L_1 block. The last environment is empty because it does not refer to any variable.

$$L_0 : \{v_0 \mapsto \text{nil}\},\ L_1 : \{v_0 \mapsto \text{nil},\ v_1 \mapsto \text{list nil}\},\ L_2 : \{\ \}$$

2.3 Properties

One of the crucial aspects 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 concentrate on the top-level soundness results:

Progress: given a well-typed instruction and a well-typed store, the machine is not stuck.

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

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

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

Soundness: a well-typed program, started in the initial configuration will not get stuck.

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

Note that all the above properties are *existential* — not being stuck is defined as the existence of a reachable configuration — and this 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 [CM17] suggests that testing intermediate lemmas is beneficial, while systems such as PLT-Redex [FKF15] tend to go for the bull's eye. We report in Section 4.2 some partial answers to this dilemma.

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

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

Fig. 1. Sample mutations

2.4 Mutation testing

How do we know that PBT is effective in catching bugs in MMT? There are very few Prolog implementation of PL calculi in the wild that we can try to validate. One way to asses the effectiveness of PBT (and more in general of other testing techniques) is via *mutation testing* [PY07]. 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 recognize the faulted code, which is known as "killing" the mutant.

While mutation testing is widely applied to programming languages [JH11], there is no theory for mutations of representations of PL artifacts that we know of. Standard mutation operators from (imperative) programming languages largely do not apply; hence we took some inspiration from the very limited literature about logic programming [RTV06]; 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: exchanging variable with (anonymous) variable.
Constant mutations: exchanging constant with constant or with (anonymous) variable.

We thus manually produced two dozens mutations of the list-machine: in Fig. 1 we show two sample: the first one 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 [FKF15] and that a general theory and implementation of a tool for mutation testing of PL calculi is beyond the scope of this paper.

3 αProlog implementation

αProlog [CU08] 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 [ADL12]. The only place one would naturally try and use *name types*, and hence inherit α-equivalence, is in the 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 lists in 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. While αProlog supports polymorphism, the checker does not, as term generations is type-driven. Note, this is a good thing: any PBT tool for Prolog will have to come up with some type information, see [AFSC14], which struggles exactly in this regard.

We show some highlights, while 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 operational semantics encoding is an unsurprisingly translation of the rules reported in the electronic appendix into a predicate `step (program, store, instr, store, instr)`, where the first three arguments are inputs. We show the rules mentioned in the previous Section.

```
pred step (program, store, instr, store, instr).
step(P, R, seq(instr_fetch_field(V1, zf, V2), I), R', I) :-
    value_cons(A, _) = var_lookup(R, V1),
    R' = var_set(R, V2, A).
step(P, R, seq(instr_fetch_field(V1, sf, V2), I), R', I) :-
    value_cons(_, A) = var_lookup(R, V1),
    R' = var_set(R, V2, A).
step(P, R, seq(instr_cons(V1, V2, V3), I), R', I) :-
    A1 = var_lookup(R, V1),
    A2 = var_lookup(R, V2),
    R' = var_set(R, V3, value_cons(A1, A2)).
...
```

Type checking is slightly more interesting: we summarize the main type definitions with a sample of typechecking instructions. We add their mode declarations,

which will become relevant as soon as we try to test existential properties such as *preservation*. Note that mode checking is not implemented yet, but it is in Twelf, which we *mutate mutandis* inherited from.

```
pred check_instr (program_typing,env,instr,env).
% mode check-instr +Pi +G +I -G.
check_instr(Pi, G, instr_fetch_field(V, zf, V'), G') :-
    (ty_listcons(T), _) = env_lookup(G, V),
    G' = env_set(G, V', T).
check_instr(Pi, G, instr_fetch_field(V, sf, V'), G') :-
    (ty_listcons(T), _) = env_lookup(G, V),
    G' = env_set(G, V', ty_list(T)).
check_instr(Pi, G, instr_cons(V0, V1, V), G') :-
    (T0, _) = env_lookup(G, V0),
    (T1, _) = env_lookup(G, V1),
    ty_list(T) = lub(ty_list(T0), T1),
    G' = env_set(G, V, ty_listcons(T)).
...
pred check_block (program_typing,env,instr).
%mode check-block +Pi +G +I.
pred check_blocks (program_typing,program).
%mode check-blocks +Pi +P.
pred check_program (program,program_typing).
%mode check-program +P +Pi.
```

3.1 PBT with αCheck

αCheck [CM17] is a tool for checking desired properties of formal systems implemented in αProlog. The idea is to test properties of the form $H_1 \wedge \cdots \wedge H_n \to A$ by searching *exhaustively* (up to a bound) for a substitution θ such that $\theta(H_1), \ldots, \theta(H_n)$ 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 based on negation elimination [CMP16]. The concrete syntax for a check is

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

where `G` is a goal and `A` an atom or constraint — since in this paper we make no use of nominal features, the latter means syntactic equality. 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

$$\exists \boldsymbol{X} : \boldsymbol{\tau}.\ G \wedge\ \mathrm{gen}_{\tau_1}(X_1) \wedge\ \ldots \wedge\ \mathrm{gen}_{\tau_n}(X_n) \wedge\ \neg A \tag{1}$$

where $\mathrm{gen}_{\tau_i}(X_i)$ are type directed exhaustive generators automatically compiled by the tool to ground the free variables of 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 5.2.

A query such as (1) builds all "finished" derivations of the hypothesis G up to the 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 be used in the derivation of each of the premises.

Let's look more closely at a check, *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)`, mode analysis for `step` suggests 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 that makes the checker loop. 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.

```
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: ... % as before
                        => exists_env_b(4,Pi,R',I').
```

4 Experimental results

Reasons of space suggest to present only a selection of all the experiments that we have carried out. Full details can be found in [Kom18], available at `https://doi.org/10.13140/RG.2.2.27992.39681`.

4.1 A cautionary tale

A testing approach is any good only if it uncovers bugs; still, 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 αCheck came up with this counterexample to the type preservation check `pres_b` (pretty-printed here for better reading):

$$p = (l_0 : \mathbf{cons}(v_0, v_0, v_0);\ \text{jump } l_1);\ (l_1 : \mathbf{fetch\text{-}field}(v_0, 0, v_0);\ \mathbf{jump}\ l_2);\ (l_2; \mathbf{halt})$$

$$\Pi = (l_0 : [v_0 \mapsto \text{nil}]);\ (l_1 : [v_0 \mapsto \text{listcons nil}]);\ (l_2 : [v_0 \mapsto \text{nil}])$$

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

$$\iota = \mathbf{fetch\text{-}field}(v_0, 0, v_0);\ \mathbf{jump}\ l_2 \qquad (\textit{block referenced by } l_1)$$

This configuration after a single step from block l_1 yields:

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

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

After some soul searching, we zeroed on the encoding of the judgment *value-has-ty*, (page 481 of [ADL12]).

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

This is an essential component of the definition of store typing $r : \Gamma$, which is the case iff for all $v \in \text{dom}(r)$, it holds $r(v) : \Gamma(v)$. The *listcons* case looks 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, which defines *value-has-ty* exactly like that — the Twelf "proof" is left as an exercise and the above typing judgment undefined.

We found more issues with the paper: less dramatically, the *check-instr-branch-listcons* typing rule contains additional preconditions regarding the jump target environment similar to those in the *check-instr-branch-list* rule; however, they are absent in the Coq implementation, as they should, considering the proof of equivalence with the algorithmic version of type checking, where they are notably absent. Several typos are 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 [Kle12].

	Progress			Preservation			Soundness			UT
Mutant	⊗	Depth	Time (s)	⊗	Depth	Time (s)	⊗	Depth	Time (s)	⊗
none		13	1771		13	15180		25	8.098	
1	⊘	13	451.0		13	1933	⊗	19	0.2910	⊗
2		13	1260	⊗	13	5.517	⊗	23	1.794	
3		13	1330	⊗	13	4.632	⊗	23	1.811	⊗
4		13	1524		13	2147s		25	31.20	
5		13	1370		13	2000	⊘	25	4356	
6		13	1188	⊗	13	2.168	⊗	23	1.881	⊗
7	⊘	13	265.0		13	1571	⊗	23	1.751	⊗
8		13	1275		13	1851	⊗	23	1.786	
9	⊗	13	0.01350		13	1867	⊗	13	0.01367	
10		13	1461		13	2411		25	30.88	
11	⊘	13	223.0		13	1478		25	7.755	⊗
12		13	1229	⊘	13	1345		25	7.496	⊗
13		13	1238		13	1777		25	7.673	⊗
14		13	980.0		13	1798		25	7.691	⊗
15		13	1085	⊗	13	4.422		25	7.602	
16		13	1252	⊗	13	21.66		25	7.655	⊗
17		13	14.03		13	258.0		25	7.171	⊗
18		interrupted		⊗	12	3.142	⊗	12	0.3680	
19		13	1269		13	4853	⊗	19	0.4871	
20		13	37020		interrupted		⊗	13	0.06200	

Table 1. Mutation testing for the list-machine

4.2 Mutation analysis

We used our "home-baked" mutation testing to assess αCheck's ability to kill mutants. Checks are divided into three groups: top-level theorems as in our Section 2.3, intermediate lemmas collected in Section 8 of [ADL12] and auxiliary checks regarding low-level details of the machine specification. We have also ported from PLT-Redex [Fin16] a few dozens unit tests, as an additional baseline in our analysis. Since those are just Prolog queries exercising the static and dynamic semantics of the machine, we have made them *parametric*, so that exhaustive generation of those parameters would be more far-reaching. The experimental results are collected in Table 1: rows list the mutations considered and whether they were killed by our test suite (marked by ⊗). There are columns for the top-level checks, detailing the time spent (in seconds) and the maximal depth we allowed, while the rightmost column gives information about unit tests. We configured the tool so as to make the execution of each check stays within ten seconds[2], as it is good practice to try and keep the execution time of a test suite reasonably short, so that it can be run

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

after every change in the model. Exception to this discipline are the *progress* and *preservation* checks, whose search space size tend to explode. Checks that found a counterexample but exceeded a further one minute time-out are marked with $\oslash$ instead of $\otimes$. Unit tests (UT) of course, even in a parametric fashion, execute in just few seconds.

As better shown in a table in the electronic appendix, top-level checks kill 12 mutants (within the strict time out), 10 with the *soundness* check alone. The score raises to 15/20 if lemmas are included and to 18/20 with lower level lemmas. In this sense PBT outperforms unit tests, which have a 1/2 killing ratio.

We are also very interested in comparing exhaustive and *random* generation for MMT of abstract machines. We thus ported the benchmark to *F#* and its PBT tool *FsCheck*. We have no space to detail this implementation, for which we refer again to [Kom18]. We simply note that, as by now well-known in the literature [Hca13,Kle12], random PBT requires a lot of ingenuity in crafting custom-made generators to ensure even a basic level of coverage in testing; further, we had to write *shrinkers* to provide small readable counterexamples. Existentials properties are also hard to code.

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.

We do not compare times to find counterexamples in αCheck vs. FsCheck, because it is a serious case of oranges and apples. What we can reasonably compare (Fig. 2) is the rate of mutants killed by the two testing approaches and the αCheck implementation comes marginally ahead. Note that we have to discard roughly half of the mutations, as they do not apply to a functional encoding of the machine. 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.

5 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 [Hca13].

5.1 List-machine 2.0

In the final part of the paper [ADL12], the model is extended to cater for *indirect* jumps. This entails some small but far-reaching changes: we coerce labels into values and replace the *nil* value with the initial label $\mathtt{l}_0$. We therefore generalize the **jump** instruction and add a way to get the current label. This yields the following changes to the operational semantics:

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

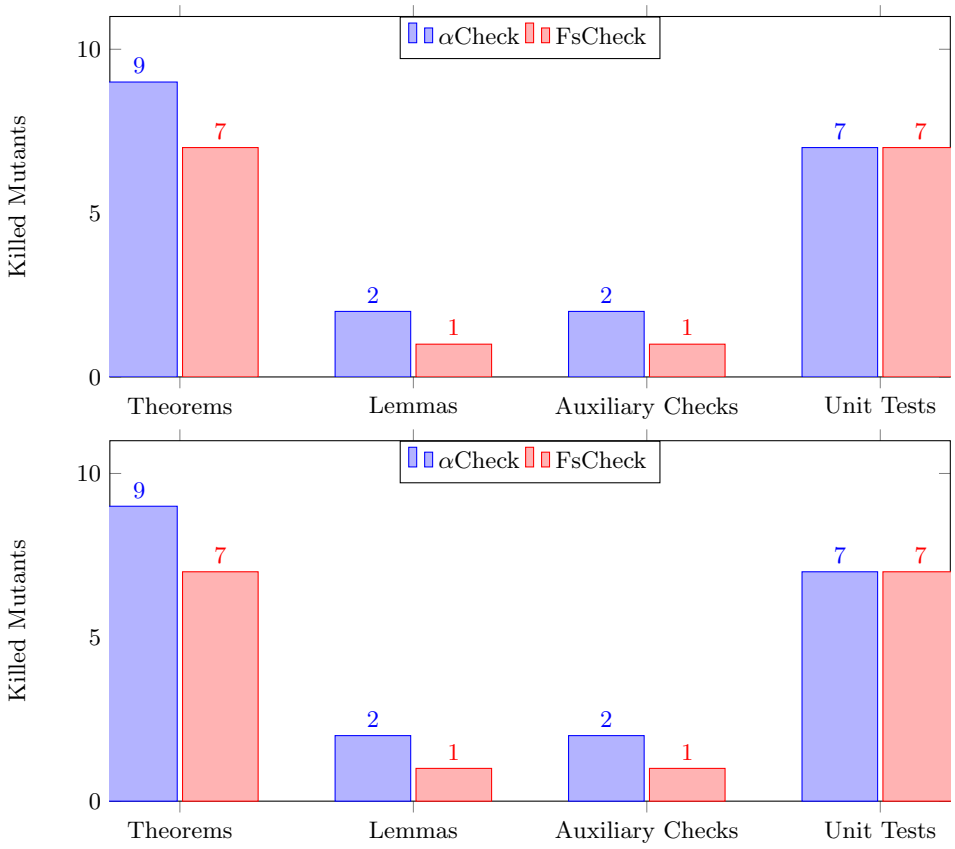

Fig. 2. Mutation analysis with αCheck and FsCheck

We also modify the type system:

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

Updating the αProlog implementation to reflect those changes is a matter of half an hour and we refer to the online documentation for all the details (`https://bitbucket.org/fkomauli/list-machine/branch/2.0`). As the paper does not spell out the soundness proof, the first thing we did was running our PBT suite to validate our implementation of the extension ... and preservation as formulated beforehand fails! The pretty-printed counterexample

$$\begin{aligned} p &= L_0 : \textbf{halt} \\ \Pi &= L_0 : [v_0 \mapsto \mathrm{nil}] \\ \iota &= \textbf{get-label}\ L_0\ v_1;\ \textbf{jump}\ v_1 \\ \iota' &= \textbf{jump}\ v_1 \end{aligned}$$

points this time not a *specification* error, but the fact that the *statement* of the theorem has to be generalized as well. In fact, in the new typing rule both instruc-

tions have to occur in sequence: 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. We modify the statement of preservation so as to try another optional step to "get over" the **jump**.

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

pred exists_env_opt(int, program, program_typing, store, instr).
exists_env_opt(N,P,Pi,R,I) :-
    step_at_most_once(P, R, I, R', I'),
    exists_env_for_store_and_block(N, Pi, R', I').

#check "preservation" 13:
 ... % as before
      => exists_env_opt(5, P, Pi, R', I').
```

After this modification, αCheck did not find any other counterexamples w.r.t. the 2.0 model.

5.2 Testing non-interference, not so quickly

It is natural to consider, in addition to the well trodden type soundness property, more intensional properties, such as *dynamic secure information-flow control* (IFC). This choice is not casual, since this is extensively studied in [Hca13], using random testing with QuickCheck. As we said, that paper challenges naive exhaustive testing in this domain and we wanted to have a say. We mainly concentrate on Section 1–4 of [Hca13] 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 $\langle pc, s, m, i\rangle$ 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, with the goal of 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*: let us call *indistinguishable* two machine states if they differ only in their private values; noninterference guarantees that in any execution starting with indistinguishable states and ending in a halted state, the final states are indistinguishable as well.

In [Hca13, Section 2.3], the authors present an operational semantics for the machine, which, while intuitive, turns out to be incorrect; then (Section 4) they detail the sophisticated testing strategies they had to program to catch the *four* bugs inserted. To give an idea, we list the buggy rule for *Load* and below the fixed one, where *i(pc) = Load* and the box signals where the bug is/was.

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

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

QuickCheck managed to locate the first two bugs with a relatively naive generation strategy by which two indistinguishable states were generated together by randomly creating the first and modifying the second in their secret part. The other two bugs necessitated a far more complex strategy for generating meaningful sequences of instructions and addresses to trigger the bugs.

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 [Hca13]. The fourth bug seemed, however, out of reach of αCheck, until we got it instead using a (bounded) αProlog query for noninterference 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.

[Hca13, Section 2] ends by introducing three additional and more outlandish bugs. We got them all with each of 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 ([Hca13, Section 5]), but the results are too preliminary to draw any conclusion.

6 Conclusions and future work

Our experience suggests that off-the-shelf PBT tools, and αCheck in particular, are already quite useful in validating the meta-theory of PL models, may they be already formalized as the list-machine, and even more if 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 realized in αCheck, seems to be a winner over the random approach. Even discounting our obvious bias, "spec'n'check" in αCheck is dead simple, requires very little effort and more than often turns out to be pretty useful. Validating low-level languages brings in more challenges, but those can be handled with the techniques we have and some work-around. Clearly, there are many other TAL models, see Chapter 4 of [Mor05], that could confirm this conclusion.

We are keenly aware that *FsCheck* and αCheck are the extremes of a (small) number of PBT-tools that we could have used for this case study. 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 [LPP18], but also Bulwahn's smart generators in Isabelle/HOL[Bul12a] are a possibility.

αCheck performs better than we hoped for, considering that its implementation is nothing more than an OCaml interpreter for nominal logic programming, explor-

ing the full search space in an iterative deepening way. 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; therefore it is natural to try and make the search strategy more modular, possibly using the notion of *hookable* disjunction from *TOR* [SDTD14].

Finally, many specifications are *coinductive* in nature [LG09]: for example, consider reasoning about *contextual equivalence* [Mom12] of pieces of code, possibly over source and target languages. It would be interesting to extend αCheck to deal with those.

Acknowledgements This research was partially supported by *INdAM/GNCS*'s project "Certificazione di verificatori automatici del sofware basati su clausole di Horn con vincoli".

References

[ADL12] Andrew W. Appel, Robert Dockins, and Xavier Leroy. A list-machine benchmark for mechanized metatheory. *J. Autom. Reasoning*, 49(3):453–491, 2012.

[AFSC14] Cláudio Amaral, Mário Florido, and Vítor Santos Costa. Prologcheck – property-based testing in prolog. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 1–17, Cham, 2014. Springer.

[BBN11] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic proof and disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCoS*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.

[Bul12a] Lukas Bulwahn. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In Chris Hawblitzel and Dale Miller, editors, *CPP*, volume 7679 of *LNCS*, pages 92–108. Springer, 2012.

[Bul12b] Lukas Bulwahn. Smart testing of functional programs in Isabelle. In Nikolaj Bjørner and Andrei Voronkov, editors, *LPAR*, volume 7180 of *LNCS*, pages 153–167. Springer, 2012.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279. ACM, 2000.

[CM17] James Cheney and Alberto Momigliano. αcheck: A mechanized metatheory model checker. *TPLP*, 17(3):311–352, 2017.

[CMP16] James Cheney, Alberto Momigliano, and Matteo Pessina. Advances in property-based testing for αProlog. In Bernhard K. Aichernig and Carlo A. Furia, editors, *TAP 2016*, volume 9762 of *LNCS*, pages 37–56. Springer, 2016.

[CU08] James Cheney and Christian Urban. Nominal logic programming. *ACM Trans. Program. Lang. Syst.*, 30(5):26:1–26:47, 2008.

[DJW12] Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: functional enumeration of algebraic types. In Janis Voigtländer, editor, *Haskell Workshop*, pages 61–72. ACM, 2012.

[Fca15] Burke Fetscher and co authors. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In Jan Vitek, editor, *ESOP 2015*, volume 9032 of *LNCS*, pages 383–405. Springer, 2015.

[Fin16] Robbie Findler. The PLT-Redex list-machine model. `https://github.com/racket/redex/tree/master/redex-benchmark/redex/benchmark/models`, 2016. Latest commit: Oct 2016.

[FKF15] Robert Bruce Findler, Casey Klein, and Burke Fetscher. Redex: Practical semantics engineering, 2015. Online at `http://docs.racket-lang.org/redex`.

[FM17] Guglielmo Fachini and Alberto Momigliano. Validating the meta-theory of programming languages. In Alessandro Cimatti and Marjan Sirjani, editors, *SEFM 2017*, volume 10469 of *LNCS*, pages 367–374. Springer, 2017.

[Hca13] Catalin Hritcu and co authors. Testing noninterference, quickly. In *ICFP*, pages 455–468. ACM, 2013.

[Hug07] John Hughes. Quickcheck testing for fun and profit. In *PADL'07*, LNCS, pages 1–32, Berlin, Heidelberg, 2007. Springer-Verlag.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.

[Kle12] Casey et al. Klein. Run your research: on the effectiveness of lightweight mechanization. In *POPL '12*, pages 285–296, New York, NY, USA, 2012. ACM.

[KM18] Francesco Komauli and Alberto Momigliano. Electronic appendix to: Property-based testing of the meta-theory of abstract machines: an experience report. `https://fkomauli.bitbucket.io/PBTAM-appendix.pdf`, 2018.

[Kom18] Francesco Komauli. Property-based testing abstract machines. Master's thesis, DI, University of Milan, April 2018.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

[LG09] Xavier Leroy and Herv Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[LPP18] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *PACMPL*, 2(POPL):45:1–45:30, 2018.

[Mom12] Alberto Momigliano. A supposedly fun thing I may have to do again: A HOAS encoding of Howe's method. In *Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'12)*, pages 33–42. ACM, 2012.

[Mor05] Greg Morrisett. *Advanced Topics in Types and Programming Languages*, chapter Typed assembly language. MIT Press, 2005.

[PHD+15] Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In *ITP*, volume 9236 of *LNCS*, pages 325–343. Springer, 2015.

[PY07] Mauro Pezzè and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

[RTV06] Juliano R Toaldo and Silvia Vergilio. Applying mutation testing in prolog programs. `http://www.lbd.dcc.ufmg.br/colecoes/wtf/2006/`, 2006.

[SDTD14] Tom Schrijvers, Bart Demoen, Markus Triska, and Benoit Desouter. Tor: Modular search with hookable disjunction. *Science of Computer Programming*, 84:101 – 120, 2014. PPDP 2012.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, pages 283–294. ACM, 2011.