Property-Based Testing of Abstract Machines
an Experience Report

Alberto Momigliano,
joint work with Francesco Komauli

DI, University of Milan

LFMTP18, Oxford
July 07, 2018
Motivation

- While people fret about program verification, I care about the study of the red meta-theory of programming languages.
- This semantics engineering addresses meta-correctness of programming, e.g. (formal) verification of the trustworthiness of the tools with which we write programs:
  - from static analyzers to interpreters, compilers, parsers, pretty-printers down to run time systems, see CompCert, seL4, CakeML, VST . . .
- Considerable interest in frameworks supporting the “working” semanticist in designing such artifacts:
  - Ott, Lem, K, PLT-Redex, the Language Workbench . . .
Why bother?

- One shiny example: the definition of SML.
Why bother?

- One shiny example: the definition of SML.
- In the other corner (infamously) PHP:
  
  “There was never any intent to write a programming language. I have absolutely no idea how to write a programming language, I just kept adding the next logical step on the way”. (Rasmus Lerdorf, on designing PHP)

- In the middle: lengthy prose documents (viz. the Java Language Specification), whose internal consistency is but a dream, see the recent existential crisis [SPLASH 16].
Meta-theory of PL

- Most of it based on common syntactic proofs:
  - type soundness
  - (strong) normalization
  - correctness of compiler transformations
  - non-interference ...

- Such proofs are quite standard, but notoriously fragile, boring, “write-only”, and thus often PhD student-powered, when not left to the reader

- mechanized meta-theory verification: using proof assistants to ensure with maximal confidence that those theorems hold
Not quite there yet

- Formal verification is lots of hard work (especially if you’re no Leroy/Appel)
- Unhelpful when the theorem I’m trying to prove is, well, wrong.
Not quite there yet

- Formal verification is lots of hard work (especially if you’re no Leroy/Appel)
- Unhelpful when the theorem I’m trying to prove is, well, wrong. I mean, *almost right*:
  - statement is too strong/weak
  - there are minor mistakes in the spec I’m reasoning about
- We all know that a failed proof attempt is not the best way to debug those mistakes
- In a sense, verification only worthwhile if we already “know” the system is correct, not in the design phase!
- That’s why I’m inclined to give *testing* a try (and I’m in good company!), in particular *property-based testing*. 
A light-weight validation approach merging two well known ideas:

1. automatic generation of test data, against  
2. executable program specifications.

Brought together in QuickCheck (Claessen & Hughes ICFP 00) for Haskell

The programmer specifies properties that functions should satisfy inside in a very simple DSL, akin to Horn logic

QuickCheck aims to falsify those properties by trying a large number of randomly generated cases.
let rec rev ls =
    match ls with
    | [] -> []
    | x :: xs -> append (rev xs, [x])

let prop_revRevIsOrig (xs:int list) =
    rev (rev xs) = xs;;

do Check.Quick prop_revRevIsOrig ;;
>> Ok, passed 100 tests.

let prop_revIsOrig (xs:int list) =
    rev xs = xs
do Check.Quick prop_revIsOrig ;;

>> Falsifiable, after 3 tests (5 shrinks) (StdGen (518275965,..)
[1; 0]
Sparse pre-conditions:
ordered xs ==> ordered (insert x xs)

Random lists not likely to be ordered ... Obvious issue of coverage. QC’s answer: write your own generator
- Writing generators may overwhelm SUT and become a research project in itself — IFC’s generator consists 1500 lines of “tricky” Haskell [JFP15]
- When the property in an invariant, you have to duplicate it as a generator and as a predicate and keep them in sync.
- Do you trust your generators? In Coq’s QC, you can prove your generators sound and even complete. Not exactly painless.

We need to implement (and trust) shrinkers, the necessary evil of random generation, transforming large counterexamples into smaller ones that can be acted upon.
Lots of current work on supporting coding or automatic derivation of (random) generators:

- **Needed Narrowing**: Classen [JFP15], Fetscher [ESOP15]
- **General constraint solving**: Focaltest [2010], Target [2015]
- A combination of the two in Luck [POPL17], a

Exhaustive data generation (**small scope hypothesis**): enumerate systematically all elements up to a certain bound:

- The granddaddy: **Alloy** [Jackson 06];
- **(Lazy)SmallCheck** [Runciman 08], **EasyCheck** [Fischer 07], **αCheck**
- Most of the testing techniques in Isabelle/HOL
What we did

- Following Robbie Findler and at.’s *Run Your Research* paper at POPL12 we want to see if we find faults in (published) PL models, but leaving the comfort of high-level object languages and addressing abstract machines and TALs.
- Comparing costs/benefits of random vs exhaustive PBT
- We take on Appel et al.’s *CIVmark*: a benchmark for “machine-checked proofs about real compilers”. No binders.
- A suicide mission for counterexample search:
  - The paper comes with two formalization, in Twelf and Coq
  - Data generation (well typed machine runs) more challenging than (single) well-typed terms.
The plumbing of the list-machine

- The list-machine works operates over an abstraction of lists, where every value is either nil or the cons of two values

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

- Instructions:

  - `jump /` jump to label /
  - `branch-if-nil v /` if \( v = \text{nil} \) then jump to /
  - `fetch-field v 0 v'` fetch the head of \( v \) into \( v' \)
  - `fetch-field v 1 v'` fetch the tail of \( v \) into \( v' \)
  - `cons v_0 v_1 v'` make a cons cell in \( v' \)
  - `halt` stop executing
  - `\iota_1; \iota_2` sequential composition

- Configurations:

\[
\text{program } p := \text{end} \mid p, l_n : \iota
\]
\[
\text{store } r := \{ \} \mid r[v \mapsto a]
\]
Operational semantics

- \((r, \iota) \xmapsto{p} (r', \iota')\) for a fixed program \(p\), in CPS-style. E.g.:

  \[
  r(v) = \text{cons}(a_0, a_1) \quad r[v' := a_0] = r' \\
  (r, (\text{fetch-field } v 0 \ v'; \iota)) \xmapsto{p} (r', \iota) \\
  \]
  \[
  r(v) = \text{cons}(a_0, a_1) \quad r[v' := a_1] = r' \\
  (r, (\text{fetch-field } v 1 \ v'; \iota)) \xmapsto{p} (r', \iota) \\
  \]
  \[
  r(v_0) = a_0 \quad r(v_1) = a_1 \quad r[v' := \text{cons}(a_0, a_1)] = r' \\
  (r, (\text{cons } v_0 v_1 v'; \iota)) \xmapsto{p} (r', \iota) \\
  \]

- Computations chained the Kleene closure of the small-step relation, with **halt** for the end of a program execution.

- A program \(p\) runs in the Kleene closure, starting from instruction at \(p(l_0)\) with an initial store \(v_0 \mapsto \text{nil}\), until a **halt**
Static semantics

- Each variable has list type then refined to empty and nonempty lists

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

- The type system includes therefore the expected subtyping relation and a notion of least common super-type

- A program typing \( \Pi \) is a list of labeled environments representing the types of the variables when entering a block

- Type-checking follows the structure of a program as a labeled sequence of blocks.

- At the bottom, instruction typing \( \Pi \vdash \text{instr } \Gamma \{ l \} \Gamma' \) where an instruction transforms a \( \Gamma \) into post-condition \( \Gamma' \) under the fixed the program typing \( \Pi \).

\[
\begin{align*}
\Gamma(v) = \text{listcons } \tau & \quad \Gamma[v' := \tau] = \Gamma' \\
\Pi \vdash_{\text{instr}} \Gamma \{ \text{fetch-field } v \ 0 \ v' \} \Gamma' & \quad \text{check-instr-fetch-0}
\end{align*}
\]

\[
\begin{align*}
\Gamma(v) = \text{listcons } \tau & \quad \Gamma[v' := \text{list } \tau] = \Gamma \\
\Pi \vdash_{\text{instr}} \Gamma \{ \text{fetch-field } v \ 0 \ v' \} \Gamma' & \quad \text{check-instr-fetch-1}
\end{align*}
\]
Question  What are the properties of interest?

Answer  The theorem the calculus satisfies:

\[ p : \Pi \quad \Pi \quad \vdash_{\text{instr}} \Gamma \{\iota\} \Gamma' \quad r : \Gamma \]

\[ \text{step-or-halt}(p, r, \iota) \quad \text{progress} \]

\[ p : \Pi \quad \vdash_{\text{env}} \Gamma \quad r : \Gamma \quad \Pi; \Gamma \vdash_{\text{block}} \iota \quad (r, \iota) \xrightarrow{p} (r', \iota') \quad \text{preservation} \]

\[ \exists \Gamma'. \quad \vdash_{\text{env}} \Gamma' \land r' : \Gamma' \land \Pi; \Gamma' \vdash_{\text{block}} \iota' \]

More questions

- What about intermediate lemmas? Do they catch more bugs?
- What are the trade off between random and exhaustive generation on low-level code?
LP implementation: $\alpha$Check 1/2

- $\alpha$Check is a PBT tool on top of $\alpha$Prolog, a variant of Prolog with nominal abstract syntax.
- Equality coincides with $\equiv_\alpha$, $\# \text{ means “not free in”}$, $\langle x \rangle M$ is an $M$ with $x$ bound, $\mathcal{U}$ is the Pitts-Gabbay quantifier.
- Use nominal Horn formulas to write specs and checks
- A check $\mathcal{U}\vec{a}\forall \vec{X}. A_1 \land \cdots \land A_n \supset A$ is a bounded query:
  $\neg \mathcal{U}\vec{a}. \exists \vec{X}. A_1 \land \cdots \land A_n \land gen(X_1) \land \cdots \land gen(X_n) \land not(A)$
  - Search via iterative-deepening for complete (up to the bound) proof trees of all hypotheses
  - Instantiate all remaining variables $X_1 \ldots X_n$ occurring in $A$ with exhaustive generator predicates for all base types, automatically provided by the tool.
  - Then, see if conclusion fails using negation-as-failure.
- Can also use negation elimination (skip for today)
The encoding is pure many-sorted Prolog: we not use the nominal machinery — not even for labels, as they have identity.

The check for progress is immediate: no set-up, the tool will add grounding generators for $P,R,I$:

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

Preservation needs some work: the conclusion is existential $\exists \Gamma'. \Gamma' \vdash \Gamma' \land \Pi; \Gamma' \vdash_{block} \iota'$ and we need custom made generator to ground $\Gamma'$.
Functional implementation: FsCheck

- We ported the machine to **F#** (adapting the Coq code, easy) and checked with **FsCheck**, its porting of QuickCheck, with automatic derivation of generators from algebraic types.
- Those are (as expected) useless: top level checks had **zero coverage**: preconditions too hard for uniform distributions;
- We had to spend a lot of effort to produce well-typed programs, while having no type-inference whatsoever;
  - for progress, this means generate simultaneously a program \( p \), a program typing \( p_i \) 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 \( l \).
- Wait, there is more: writing **shrinkers** here is non-trivial again, as we need to shrink modulo well-typing.
Proof of the pudding: validating the list-machine

- The preservation property fails! Here’s the offending program:
  
  \[(l_0 : \text{cons}(\nu_0, \nu_0, \nu_0); \text{jump } l_1);
  (l_1 : \text{fetch-field}(\nu_0, 0, \nu_0); \text{jump } l_2);
  (l_2; \text{halt})\]

- There was a major mistake in the journal paper w.r.t. assigning types to values:
  
  \[
  \text{cons}(a_0, a_1) : \text{listcons } \tau
  \]
Proof of the pudding: validating the list-machine

- The preservation property fails! Here’s the offending program:
  
  \[
  (l_0 : \text{cons}(v_0, v_0, v_0); \text{jump } l_1);
  (l_1 : \text{fetch-field}(v_0, 0, v_0); \text{jump } l_2);
  (l_2; \text{halt})
  \]

- There was a major mistake in the journal paper w.r.t. assigning types to values:
  
  \[
  \text{???
  cons}(a_0, a_1) : \text{listcons } \tau
  \]

- Mutation Analysis:
  
  1. change a program inserting a single fault
  2. see if your testing method detects it (killing a mutant)
  3. it’s as good at the killing ratio

- We adopted idea from mutation testing in Prolog to insert mutations such as:
  
  \[
  \Gamma(v) = \text{listcons } \tau \quad \Gamma[v' := [\text{list } \tau] = \Gamma'
  \]
  
  \[\prod \vdash_{\text{instr}} \Gamma\{\text{fetch-field } v 1 v'}\} \Gamma'\]
  
  check-instr-fetch*
The image shows a bar chart comparing the number of mutants killed by two tools, \( \alpha \) Check and FsCheck, across different types of checks:

- **Theorems**: Type soundness
- **Lemmas**: Intermediate (typically non-inductive) results
- **Auxiliary Checks**: Even lower checks coming from Twelf
- **Unit Tests**: Queries adapted from PLT-Redex

The chart indicates:

- Theorems: \( \alpha \) Check killed 9 mutants, FsCheck 7.
- Lemmas: \( \alpha \) Check killed 2 mutants, FsCheck 1.
- Auxiliary Checks: \( \alpha \) Check killed 2 mutants, FsCheck 1.
- Unit Tests: \( \alpha \) Check and FsCheck both killed 7 mutants.

The textual content provides additional context:

- \# of mutants killed by each tool
- “Theorems” means type soundness, “lemmas” are intermediate (typically non-inductive) results, “auxiliary” are even lower checks coming from Twelf.
- “Unit tests” are just queries adapted from PLT-Redex.
Conclusions

- PBT is a great choice for meta-theory model checking.
- Validating low-level languages is more challenging, but we can handle with the tools we have and some additional work.
- Checking specifications with $\alpha$Check is immediate.
- Bare-to-the-bone QuickCheck is a lot of work to setup.
- W.r.t. costs–benefits, exhaustive generation, even in our naive way, comes ahead over the random approach . . .
  - but we need automatic mutation testing to confirm this.
Future work: other PBT tools

- We know very well that FsCheck and $\alpha$Check are the extremes of PBT tools and we really should run this benchmark with others that have support for custom generators.
- Since the benchmark has no binders, there are many choices:
  - the new QuickChick, with automatically generated generators
  - Luck — but you still have to write gens and it's slow
  - Bulwhahn's smart generators in Isabelle/HOL, less likely
    Nitpick
Future work: $\alpha$Check

$\alpha$Check works surprisingly well, given the naivete of its implementation: basically an iterative deepening modification of the original OCaml interpreter for $\alpha$Prolog

But experiments with other abstract machines (IFC) reminds us of how naive we are w.r.t. the combinatorial explosion

Change the hard-wired notion of bound (\# of clauses used) and how it is distributed over subgoals:

- Take ideas from Tor

Bring in some random-ness by doing random backchaining: flip a coin instead of doing chronological backtracking

Prune the search space by not generating terms that exercise “equivalent” part of the spec
It’s folklore that linear logical framework are well suited to encode object logic with imperative features, e.g. Pfenning and Cervesato’s encoding of MLR in LLF;

Data structures for heaps, stores... are replaced by linear, affine, etc predicates

- This seems promising for exhaustive PBT, where every constructor counts
- Work in progress: linear version of the list-machine benchmark via the two level approach (in $\lambda$Prolog)

Sub-structural PBT can bring some form of validation to frameworks such as Celf, whose meta-theory is not there yet

Meta-interpreters not viable in the long run:
- give the $\alpha$Check treatment to languages such as LolliMon
- use program specialization to do amalgamation
Thanks for listening and have a good lunch!