The Blame Game for Property-based Testing: work-in-progress

Alberto Momigliano,
joint work with Mario Ornaghi

DI, University of Milan

CILC 2019, Trieste
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.
Our recently (re)released tool:
https://github.com/aprolog-lang

On top of \( \alpha \)-Prolog, a simple extension of Prolog with nominal abstract syntax.

Use nominal Horn formulas to write specs and checks.

Equality coincides with \( \equiv_\alpha \), \( \# \) means “not free in”, \( \langle x \rangle M \) is \( M \) with \( x \) bound, \( \mathcal{N} \) is the fresh Pitts-Gabbay quantifier.

\( \alpha \)-Check searches exhaustively for counterexamples, using iterative deepening.

Our intended domain: the meta-theory of programming languages artifacts: from static analyzers to interpreters, compilers, parsers, pretty-printers, down to run-time systems...
A motivating (toy) example

This grammar characterizes all the strings with the same number of a's and b's:

\[
S ::= . \mid bA \mid aB \\
A ::= aS \mid bAA \\
B ::= bS \mid aBB
\]

We encode it in αProlog, inserting two quite obvious bugs, but be charitable and think of a much larger grammar:

viz., the grammar of Ocaml\textsubscript{light} consists of 251 productions

\[
\begin{align*}
ss([]) & . \\
ss([b|W]) & :- ss(W). \\
ss([a|W]) & :- bb(W).
\end{align*}
\]

\[
\begin{align*}
bb([b|W]) & :- ss(W). \\
bb([a|VW]) & :- append(V,W,VW), bb(V), bb(W).
\end{align*}
\]

\[
aa([a|W]) & :- ss(W).
\]

(an ice cream to the first who finds both bugs in the next 30 secs)
We use \( \alpha \)Check to debug it, splitting the characterization of the grammar into soundness and completeness:

\[
\text{#check "sound" 10: ss(W), count(a,W,N1), count(b,W,N2) => N1 = N2.}
\]

\[
\text{#check "compl" 10: count(a,W,N), count(b,W,N) => ss(W).}
\]

The tool dutifully reports (at least) two counterexamples:

Checking for counterexamples to
sound: \( N1 = z, N2 = s(z), W = [b] \)
compl: \( N = s(s(z)), W = [b,b,a,a] \)

Where is the bug? Which clause(s) shall we blame? Can we help the user localize the slice of program involved?
Where do bugs come from? That’s a huge problem.
Where do bugs come from? That’s a huge problem.
Did anybody say *declarative debugging*? 
The idea: 1/3

- Where do bugs come from? That’s a huge problem.
- Did anybody say *declarative debugging*? Let’s do something less heavy handed.
Where do bugs come from? That’s a huge problem.

Did anybody say *declarative debugging*? Let’s do something less heavy handed.

We do not claim to have a general approach:

- First, we’re addressing the sub-domain of *mechanized meta-theory model-checking*, where fully declarative PL models are tested against theorems these systems should obey.
- Second, we just want to give *some* practical help to the poor user debugging a model w/o exploiting her as an oracle.
The idea

- The `#check` pragma corresponds to specs of the form that we try and refute $\forall \vec{X}. G \supset A$

- Take completeness of the above grammar:

  $\exists W. \text{count}(a,W,N), \text{count}(b,W,N), \text{not}(\text{ss}(W))$.

  A *counterexample* is a grounding substitution $\theta$ that $\theta(G)$ is derivable, but $\theta(A)$ is not.

- For the above to unexpectedly succeed, two (possibly overlapping) things may go wrong:

  **MA:** $\theta(A)$ fails, whereas it belongs to the intended interpretation of its definition (*missing answer*);

  **WA:** a bug in $\theta(G)$ creates some erroneous bindings that make the conclusion fail (*wrong answer*).
Our “old-school” idea consists in coupling:

1. *abduction* to try and diagnose MA’s with
2. *proof verbalization*: presenting at various levels of abstraction *proof-trees* for WA’s to explain where the bug occurred.

Differently from declarative debugging, we ask the user only to state who she trusts:

- built-in, certainly; libraries, most likely;
- predicates that have sustained enough testing;

and which are the *abductable* predicates:

- some heuristics based on the dependency graph should help.
Proof verbalization

- Back to the *soundness* check: we trust unification and the auxiliary count predicate ...

\[
\text{ss}(W), \text{count}(a,W,N1), \text{count}(b,W,N2) \implies N1 = N2.
\]

sound: \( N1 = z, \, N2 = s(z), \, W = [b] \)

- ...hence it must be a case of WA, starring \( \text{ss}([b]) \). Verbalizing the proof tree yields:

\( \text{ss}([b]) \) for rule s2, since:

\( \text{ss}([]) \) for fact s1.

- This points to rule s2

\[
\begin{align*}
\text{ss}([b|W]) & : \text{ss}(W). \quad \% \text{BUG} \\
\text{ss}([b|W]) & : \text{aa}(W). \quad \% \text{OK}
\end{align*}
\]

- Clearly, proof trees tend to be longer than that and we *distill* them to hide information, up to showing only the skeleton of the proof (the clauses used).
Abduction

- Once we fix the previous bug, the second still looms:
  \[\text{count}(a,W,N), \text{count}(b,W,N) \Rightarrow \text{ss}(W).\]
  \[\text{compl: } N = s(s(z)), W = [b,b,a,a]\]

- It's a **MA**: putting all the grammar in the abducibles, we have:
  \[\text{ss}([b,b,a,a]) \text{ for rule s2, since:}\]
  \[\text{aa}([b,a,a]) \text{ for assumed.}\]

- We realize that there is **no** clause head \(\text{aa}([b\mid VW])\) in the program, matching the failed leaf: we have forgot the clause:
  \[\text{aa}([b\mid VW]) :- \text{append}(V,W,VW), \text{aa}(V),\text{aa}(W).\]
Once we fix the previous bug, the second still looms:
\[
\text{count}(a,W,N), \text{count}(b,W,N) \Rightarrow \text{ss}(W).
\]
compl: \( N = s(s(z)), W = [b,b,a,a] \)

It’s a MA: putting all the grammar in the abducibles, we have:
\[
\text{ss}([b,b,a,a]) \text{ for rule s2, since:}
\text{aa}([b,a,a]) \text{ for assumed.}
\]

We realize that there is no clause head \( \text{aa}([b|VW]) \) in the program, matching the failed leaf: we have forgot the clause:
\[
\text{aa}([b|VW]) :\equiv \text{append}(V,W,VW), \text{aa}(V),\text{aa}(W).
\]

I told you the bugs were silly, didn’t I?

That’s why we implemented a tool for mutation testing: plenty of unbiased faulty programs to explain away!
Mutation testing

- Change a source program in a localized way by introducing a single (syntactic) fault — have a “mutant”, hopefully not semantically equivalent.
- “Kill it” with your testing suite means finding the fault.
- A killed mutant is a good candidate for blame assignment: it contains reasonable bugs not planted by ourselves.
- We have written a mutator for αProlog by randomly applying type-preserving mutation operators
- and checking with αCheck (up to a bound of course) that the mutant is not equivalent to its ancestor;
- if so, we pass it to the blame tool for explanation.
Architecture of the tool

- The back-end consists of an $\alpha$Prolog meta-interpreter working on a reified version of the sources of an $\alpha$Prolog program.
- The front-end is written in Prolog and is responsible for everything else:
  - The reification process and syncing the latter with the sources
  - Calling $\alpha$Check, feeding the meta-interpreter with the necessary info and doing the verbalization
Conclusions

- We are close to release a tool for explanations of bugs reported by $\alpha$Check for **full $\alpha$Prolog**— whose features we have not used in this talk.

- While our approach of abduction + explanations is simple-minded it tries to find a sweet spot in helping understanding bugs in PL models w/o going full steam into declarative debugging.

- Experience (e.g., significant case studies) will tell if we succeeded.

- The mutator is of independent interest for evaluating the effectiveness of the various strategies of $\alpha$Check in finding bugs in $\alpha$Prolog specifications.
Thanks!