The Blame Game for Property-based Testing

Alberto Momigliano and Mario Ornaghi

DI, Università di Milano

Abstract. We report on work in progress aiming to add *blame* features to property-based-testing in logic programming, in particular w.r.t. the mechanized meta-theory model checker αCheck. Once the latter reports a counterexample to a property stated in αProlog, a combination of *abduction* and *proof explanation* tries to help the user to locate the part of the program that is responsible for the unintended behavior. To evaluate whether these explanations are in fact useful, we need an unbiased collection of faulty programs, where the bugs location is unknown to us. We have thus implemented a mutation testing tool for αProlog that generates such a set. Preliminary experiments point to the usefulness of our blame allocator. The mutator is of independent interest, allowing us to gauge the effectiveness of the various strategies of αCheck in finding bugs in αProlog specifications.

1 Introduction

Property-based testing [13] (PBT) is a lightweight validation technique by which we try to *refute* executable specifications against automatically (typically in a pseudo-random way) generated data. Once the idea broke out in the functional programming community with QuickCheck, it spread to most programming languages and turned also in a commercial enterprise [15].

PBT has also a pedagogical value and our students seem to enjoy using it during their functional programming coursework. After type-checking succeeds, the PBT message "*Ok, passed 100 tests*", whereby the tool reports the failure to refute some property, gives the students a warm fuzzy feeling of having written correct code. This feeling comes to a sudden halt when a counterexample is reported, signaling a mismatch between code and specification. In our experience, the student freezes, disconnects the brain, typically uttering something like "Prof, there's a problem here", and refrains from taking any further action.

Now, no-one likes to be wrong, not only students, and this is why testers and programmers may be different entities in a software company. While a counterexample to a property is a more useful response than, say, the *false* of a failed logic programming query — or a core dump for that matter — we can do better and go beyond the mere reporting of that counterexample; namely, try to *circumscribe* the origin of the latter. After all, the program under test may be arbitrarily large and, arguably, even partial solutions pinpointing the slice of the program involved in an error can help pursuing the painful process of bug fixing.

CEUR-WS.org/Vol-2396/paper21.pdf

"Where do bugs come from?" is a crucial question that has sparked a huge amount of research. In this short paper, we report on preliminary work on addressing a much, much smaller sub-problem: first, the domain is *mechanized meta-theory model-checking* [4], that is the validation of the mechanization in a logical framework of the meta-theory of programming languages and related calculi [11, 10]. To fix ideas, think about the formal verification of compiler correctness [21] and look no further than [18] for more impressive case studies. In this domain, the specifications under validation correspond to the theorems that the formal system under test should obey and are therefore *trusted*. Hence, we put the blame of a counterexample on the system encoding, not on a possibly erroneous property.

Secondly, and consequently, we restrict to (nominal) logic programming, that is to encoding written in αProlog [7] and properties checked by αCheck [5]. Logic programming, of course, is particularly suited to the task of blame assignment. In fact, the related notion of *declarative debugging* [3], starting with Shapiro's thesis, originated there. Our approach may be applicable outside those two boundaries, but we have not enough evidence to make any strong claim.

Our blame tool is written in Prolog, a choice taken mostly out of convenience, since it allows us the flexibility to experiment with ideas by meta-interpretation, rather then wiring them in for good in αProlog's OCAML implementation. In fact, meta-interpreters are forbidden by the interaction of the strongly-typed nature of αProlog, together with its being a first order language.

We picture the architecture of the tool in Fig. 1: we write models and specs in αProlog and we pass them to αCheck for validation. If a counterexample is found, we translate the αProlog code to standard Prolog and we feed it to our explanation tool together with the said counterexample. The tool returns an explanation that can be hopefully used to debug the αProlog model. And so on. This architecture, of course, applies only to αProlog code that can be faithfully executed in standard Prolog and therefore does not support natively nominal features.

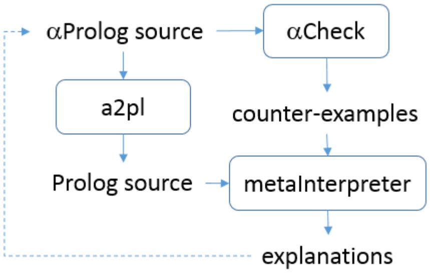

Fig. 1. Flow of the blame game

2 A motivating example

Suppose that we wish to study the following grammar (adapted from [1]), which characterizes all the strings with the same number of a's and $b's$:

```
S ::= .  | bA | aB
A ::= aS | bAA
B ::= bS | aBB
```

Here's the bulk of the αProlog code, where we have omitted the obvious definitions of list-related predicates:

```
ab  : type.     pred ss(list(ab)).     pred aa(list(ab)).
a,b : ab.       pred bb(list(ab)).     pred count(ab,list(ab),nat).

ss([]).
ss([b|W])  :- ss(W).
ss([a|W])  :- bb(W).

bb([b|W])  :- ss(W).
bb([a|VW]) :- bb(V), bb(W), append(V,W,VW).

aa([a|W])  :- ss(W).
aa([b|VW]) :- aa(V), aa(W), append(V,W,VW)
```

However, our encoding is flawed — we ask the reader to suspend her disbelief, since the bug is quite apparent. Still, it is easy to conjure an analogous scenario where the grammar has dozens of productions and the bugs harder to spot.

We shall use αCheck to debug it. We split the characterization of the grammar into soundness and completeness properties:

```
#check "sound" 10 : ss(W), count(a,W,N1), count(b,W,N2) => N1 = N2.
#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: ss(W), count(a,W,N1), count(b,W,N2) => N1 = N2
Checking depth 1 2 3  Total: 0.000329 s:
N1 = z, N2 = s(z), W = [b]

compl: count(a,W,N), count(b,W,N) => ss(W)
Checking depth 1 2 3 4 5 6 7 8 9  Total: 0.016407 s:
N = s(z), W = [b,a]
```

Now that we have the counterexamples, where is the bug? More precisely, which clause(s) shall we blame?

The `#check` pragma of αCheck corresponds to specification formulas of the form

$$\forall \boldsymbol{X}.\ G \supset A \tag{1}$$

where G is a goal and A an atomic formula (including equality and freshness constraints). A *(finite) counterexample* is a grounding substitution θ providing values for $\boldsymbol{X}$ that satisfy the negation of (1): that is, such that $\theta(G)$ is derivable, but the conclusion $\theta(A)$ is not. In this paper, we identify negation with negation-as-failure (NAF) and we will not make essential use of nominal features.

The tool synthesizes (trusted) type-based exhaustive generators for the free variables of the conclusion, so that NAF executes safely. A complete search strategy, here naive iterative deepening, searches for a proof of:

$$\exists \boldsymbol{X}{:}\boldsymbol{\tau}.\ G \wedge \mathit{gen}[\![\tau]\!](X_1) \wedge \cdots \wedge \mathit{gen}[\![\tau]\!](X_n) \wedge \mathit{not}(A) \tag{2}$$

To wit, completeness will be the query:

```
∃W. count(a,W,N), count(b,W,N), gen_ablist(W), not(ss(W)).
```

The predicate `gen_ablist(W)` corresponds to the generator formula $\mathit{gen}[\![\texttt{list(ab)}]\!](W)$ for lists of `a`'s and `b`'s. In the soundness query we know by mode analysis that we do not need generators for numerals.
For a query such as the above and more in general such as (2) to unexpectedly succeed, two (possibly overlapping) things may have gone wrong [22]:

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

WA: a bug in G creates some erroneous bindings, which again make A fail (*wrong answer*): some atom in G succeeds but it is not in the intended model.

Our "old-school" idea consists in coupling 1) *abduction* [17] to try and diagnose *MA's* with 2) *proof verbalization*, that is presenting, at various levels of abstraction, *proof-trees* for *WA's* as a means to explain where, if not why, the WA occurred.

Of course, since we do not know, in principle, which is which and the two phenomena can occur simultaneously, we are going to use some simple heuristics to drive this process. What we would rather keep our distance from is full declarative debugging (see e.g., [8]), as our experience suggests that making the user the oracle potentially at each step is just too much work.

Our notion of abduction is a simple-minded proof procedure that finds a set of assumptions Δ such that $\Gamma, \Delta \vdash G'$, but $\Gamma \not\vdash G'$ for given Γ and G'. In our limited setting here of first-order logic programming, we, as expected, internalize Γ as a fixed program P and we see Δ as a (additive) conjunction of atoms $\mathcal{A}$. An abduction procedure depends on the selection of what the *abducibles* are. This choice is related to which part of the program we trust and which we want to investigate. The system will suggest abducibles based on the dependency graph of the conclusion of the property under investigation, but we give the final say to the user, while trying to make it easy to change one's mind.

We now work through the above example. Our tool reads into standard Prolog the αProlog files, adding names to rules and treating type information as inactive predicates via appropriate operator declarations. For example, here is the `ss` predicate and related declarations:

```
ab :: type.  a  :: ab.  b  :: ab.  pred ss(list(ab)).

ss([])    :- rule(s1).
ss([b|W]) :- rule(s2), ss(W).
ss([a|W]) :- rule(s3), bb(W).
```

To begin with, we need to decide what to investigate and what to trust. Consistently with our faith in properties, we will trust `count` and `append`:

```
trusted(append(_,_,_)).  trusted(count(_,_,_)).
```

The dependency graph suggests (quite unhelpfully here) to put all of `aa,bb,ss` as abducibles.

```
assumable(ss(_)).  assumable(bb(_)).  assumable(aa(_)).
```

For exposition sake, let us investigate the completeness bug first, that is the failure of `ss([b,a])`. This is a case of MA, since that string should be accepted. The tool returns this explanation:

```
ss([b,a]) for rule s2, since:
  ss([a]) for rule s3, since:
    bb([]) for assumed
```

Once you get used to the fairly primitive verbalization, you see that *if it were the case* that `bb([])` holds, then `ss([b,a])` would have succeeded and the counterexample not found. However, there is no production for `b` accepting the empty string, so we should blame `ss`, namely either `s2` or `s3`. By comparing the clauses to the productions they are supposed to encode we see that the body of `s2` should be `aa(W)` and not `ss(W)`.

For the soundness bug, it is obvious that it is a case of WA. We trust unification, the more since here it is just syntactic equality. Hence we need to understand what went wrong in the premise `ss(W)` — remember, we trust `count`. The tool answers:

```
ss([b])  for rule s2, since:
  ss([])   for fact s1.
```

Since `s1` is OK, it is again `s2`'s fault. Once we fix that bug, αCheck does not report any more issues.

3 How it works

It is natural to split the definitions of the program under test into *builtin*, which we assume to be non problematic, and those we want to look into (*pred*, for compatibility with αProlog type declarations). By default, every user-defined predicate is under suspicion. However, once a *pred* overcomes our scrutiny, or we believe it, perhaps temporarily, to be well-behaved, we can graduate it to being *trusted* and let it off the hook. Builtins and trusted predicates are *not* meta-interpreted, that is we simply `call` them. Abducibles must be non-trusted *pred* definitions.

The abduction/explanation mechanism can be formulated in terms of a calculus $AE(P)$, based on the notion of *proof-tree* (see [14], for example), whose nodes are constructed with the following inference rules:

$$\frac{}{\mathit{true}}\ \top \qquad \frac{G\sigma}{A\sigma}\ r \qquad \frac{G_1 \quad G_2}{G_1 \wedge G_2}\ \wedge \qquad \frac{G_i}{G_1 \vee G_2}\ \vee_i \quad \text{i = 1 or 2}$$

$$\frac{}{A\sigma}\ \mathit{asm} \qquad \frac{}{\overline{A\sigma}}\ \mathit{builtin} \qquad \frac{}{\overline{A\sigma}}\ \mathit{trusted}$$

Given a fixed program P, rule r corresponds to a single SLD step with clause `A :- rule(r),G` in P. Rules $\top, \wedge, \vee$ are self-explanatory. Rule *asm* closes a proof whenever A is declared as abducible. The last two rules are somewhat different (and hence the double bar): *builtin* and *trusted* both end the proof returning an answer substitution: in the first case if there is an external computation of the builtin predicates. In the *trusted* case, we apply the rule only if there is a closed proof-tree, i.e. *without* assumptions returning σ. In fact, this is not technically a finitary rule, in the same sense that NAF is not; nevertheless, the way in which we use the calculus, namely by replaying already found counterexamples, will ensure that the rule is effective. For example, the second clause for `bb` asks for a computation of `append(V,W,VW)` where `VW` is ground. This is simply executed as a Prolog query outside the calculus, returning the ground substitution σ.

The meta-interpreter computes over those proof-trees, simultaneously collecting abducible assumptions and Prolog terms encoding the proof-tree itself, to be later verbalized. The meta-interpreter enjoys, we claim, the following correctness property: a proof-tree computed by the meta-interpreter is a proof in the calculus above. Conversely, for all goals G, if there is proof in $AE(P)$ of $G\sigma$, the interpreter will produce a more general proof-tree for $G\delta$ with the same sequence of rules.

Proof verbalization Clearly, proof-trees of height 2 as the ones occurring in Section 2 are not that difficult to interpret. However, we address the general case by offering various levels of explanation, from printing out the whole tree to instead presenting a skeleton containing only the involved rule names. We currently do this with different forms of `printf`, while it would be more flexible to use a form of *proof distillation* [2], by which the proof-tree itself is pruned at the desired level of explanation and only then verbalized.

Heuristics Suppose you have a counterexample for property $\forall \boldsymbol{X}.\ G \supset A$: how do you go about trying to decide whether it is a MA or a WA? We suggest the following steps with the blame tool:

1. if A is a builtin (including constraints) or a trusted predicate, it is clear that it is a case of WA and we directly verbalize the proof tree of the premise(s).
2. Otherwise, we use abduction to investigate the failure of A; start by adding A and its dependency graph as abducibles.
   - 2.1 If the failure of A points to a case of MA, use the explanation mechanism to see if there is a clause missing in the program or which clause is otherwise involved.
   - 2.2 If the set of assumptions found by abduction is "unreasonable", that is we have atoms that should not hold in the intended model, it is likely a case of WA and we turn to verbalizing the proof-tree for G.

4 Mutation testing for αProlog

Mutation analysis [16] aims to evaluate software testing techniques with a form of white box testing, whereby a source program is changed in a localized way by introducing a single (syntactic) *fault*. The resulting program is called a "mutant" and hopefully is also semantically different from its ancestor. A testing suite should recognize the faulted code, which is known as "killing" the mutant. The higher the number of killed mutants, the better the testing suite.

What is the connection with our endeavor? A killed mutant is a good candidate for blame assignment, since it should simulate reasonable bugs that occur in a program without being planted by ourselves. This is justified by the "competent programmer assumption" [16], according to which programmers tend to develop programs close to the correct version and thus the difference between current and correct code for each fault is small. A mutator provides us (automatically) with a multitude of faulty programs to try and explain. At the same time, such a tool helps to gauge the effectiveness of αCheck in locating bugs, under its various strategies. In previous work we either had to use faulty models from the literature [6], or to resort to *manual* generation of mutants [19] to compare αCheck with other PBT tools — but this is obviously biased, labor-intensive and very hard to scale.

We have thus developed a mutator for αProlog programs, with an emphasis on trying to devise *mutation operators* for our intended domain, namely models of programming languages artifacts. Those operators specify the mutations that the tool will inject and therefore a mutator is as effective as operators are relevant. Mutation testing comes from imperative languages and the operators thereby are pretty useless. Even operators proposed for declarative programs [9, 20] make only partial sense. In particular those in [9] not only are completely oblivious of αProlog's type discipline, but mostly address the operational semantics of Prolog: clause reordering, permutation of cuts etc. Those are not relevant to us, since the checker uses a complete search strategy and does not understand cuts.

Keeping in mind that we strive to produce mutants that cannot be detected already by static analysis such as type checking in αProlog or singleton variable analysis in standard Prolog, we have converged on this initial set of operators:

Clause mutations: deletion of a predicate in the body of a clause, deleting the whole clause if a fact. Replacement of disjunction by conjunction and vice versa.

Operator mutations: arithmetic and relational operator mutation, say $<$ in $\leq$.

Variable mutations: replacing a variable with an (anonymous) variable and vice versa.

Constant mutations: replacing a constant by a constant (of the same type), or by an (anonymous) variable and vice versa.

α**Prolog-specific** Removing freshness annotations (#), changing their scope or confusing them with equality.

In our implementation, a simple driver produces a given number of randomly generated mutants from an αProlog source P and a list LM of mutation operators. At every generation step we make a random choice of operator in LM and of clause in P such that the operator is applicable, in the spirit of [27]. If the mutant is new, i.e., it has not been produced before, it is passed to the next phase: here, we try to weed out semantically equivalent mutants following the proposal in [23] of comparing them with their ancestor on a finite portion of input. We achieve this by asking αCheck to try to refute their equivalence up to a given bound. Keeping the bound small makes this a fairly quick, if imperfect, filter. If the mutant survives, we pass it again to αCheck, this time under the property that the original model was supposed to satisfy. If it fails, we go back to the flow in Fig. 1.

We have been experimenting with the mutation and explanation of three main case studies:

1. The preservation and progress property for the typed arithmetic language in Chapter 8 of [25].
2. Non-interference for static flow information systems, which we had started to address in [6].
3. Type preservation for low-level abstract machines, as in [19].

While the results are encouraging, we are still in the process of collecting evidence.

5 Conclusions and future work

We have given an "appetizer" of our approach to blame assignment for counterexamples found by PBT. We have used old-fashioned ideas in the literature, such as abduction and explanation trees, trying to strike a reasonable balance with declarative debugging: we only ask for the user's input when we let her decide if the answer given by abduction is sensible; this is quite different, we argue, from the the constant prodding that declarative debugging requires. Our approach is also much simpler (perhaps simple-minded) than previous work aiming to explain why a query is true or false in a model, typically under the answer set semantics, as in e.g. [26].

There is so much work that lies in front of us: we need to gather more experience using the tool, especially with a user who is different from the authors. This is particularly crucial for tuning the amount of details in proof verbalization.

Mutation testing for αProlog is of independent interest, besides the way we have used it here. It allows us to evaluate how well αCheck performs in spotting bugged models, similarly to the interaction of *MuCheck* [20] with QuickCheck. However, designing mutation operators that make sense in our intended domain is an open question, see [24] for the more general issue of mutations vs. real faults. The operators that we have proposed are a starting point, but it would be nice to provide the user with a basic DSL to encode her own.

Finally, the main drawback of the current setup is that we can play the blame game only for αProlog programs that are indeed Prolog code, i.e. that do not use the nominal features, since they cannot be faithfully replayed in standard Prolog. The new version, already under development, will instead handle the whole of

αProlog, by porting the blame tool to the latter, where αProlog's programs are reified in object-level clauses following the two-level approach [12].

References

1. J. Blachette. *Picking Nits: A User's Guide to Nitpick for Isabelle/HOL*. TUM, 2018. `http://isabelle.in.tum.de/dist/doc/nitpick.pdf`.
2. R. Blanco, Z. Chihani, and D. Miller. Translating between implicit and explicit versions of proof. In *CADE*, volume 10395 of *LNCS*, pages 255–273. Springer, 2017.
3. R. Caballero, A. Riesco, and J. Silva. A survey of algorithmic debugging. *ACM Comput. Surv.*, 50(4):60:1–60:35, Aug. 2017.
4. J. Cheney and A. Momigliano. Mechanized metatheory model-checking. In *PPDP*, pages 75–86. ACM, 2007.
5. J. Cheney and A. Momigliano. αCheck: A mechanized metatheory model checker. *TPLP*, 17(3):311–352, 2017.
6. J. Cheney, A. Momigliano, and M. Pessina. Advances in property-based testing for αProlog. In B. K. Aichernig and C. A. Furia, editors, *TAP*, volume 9762 of *LNCS*, pages 37–56. Springer, 2016.
7. J. Cheney and C. Urban. Nominal logic programming. *ACM Trans. Program. Lang. Syst.*, 30(5):26:1–26:47, 2008.
8. N. Dershowitz and Y. Lee. Logical debugging. *J. Symb. Comput.*, 15(5/6):745–773, 1993.
9. A. Efremidis, J. Schmidt, S. Krings, and P. Körner. Measuring coverage of prolog programs using mutation testing. *CoRR*, abs/1808.07725, 2018.
10. G. Fachini and A. Momigliano. Validating the meta-theory of programming languages. In *SEFM*, volume 10469 of *LNCS*, pages 367–374. Springer, 2017.
11. A. P. Felty and A. Momigliano. Reasoning with hypothetical judgments and open terms in Hybrid. In *PPDP*, pages 83–92. ACM, 2009.
12. A. P. Felty and A. Momigliano. Hybrid - A definitional two-level approach to reasoning with higher-order abstract syntax. *J. Autom. Reasoning*, 48(1):43–105, 2012.
13. G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, pages 74–80, July 1997.
14. W. Hodges. Logical features of Horn clauses. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming (Vol. 1)*, pages 449–503. Oxford University Press, Inc., 1993.
15. J. Hughes. Quickcheck testing for fun and profit. In *PADL'07*, LNCS, pages 1–32, Berlin, Heidelberg, 2007. Springer-Verlag.
16. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, Sept. 2011.
17. A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1992.
18. C. Klein and et al. Run your research: on the effectiveness of lightweight mechanization. In *POPL '12*, pages 285–296, New York, NY, USA, 2012. ACM.
19. F. Komauli and A. Momigliano. Property-based testing of the meta-theory of abstract machines: an experience report. In *CILC*, volume 2214 of *CEUR*, pages 22–39, 2018.

20. D. Le, M. A. Alipour, R. Gopinath, and A. Groce. Mucheck: An extensible tool for mutation testing of haskell programs. In *ISSTA 2014*, pages 429–432. ACM, 2014.
21. X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
22. L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3):3–29, 1997.
23. A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.
24. M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae. Are mutation scores correlated with real fault detection? ICSE '18, pages 537–548. ACM, 2018.
25. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
26. C. Viegas Damásio, A. Analyti, and G. Antoniou. Justifications for logic programming. In P. Cabalar and T. C. Son, editors, *LPNMR*, pages 530–542. Springer, 2013.
27. L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *ASE*, pages 92–102. IEEE, 2013.