Validating the Meta-Theory of Programming Languages

Guglielmo Fachini[1] and Alberto Momigliano[2]

[1] INRIA Prosecco, Paris, France guglielmo.fachini@inria.fr
[2] Dipartimento di Informatica, Università degli Studi di Milano, Italy
momigliano@di.unimi.it

Abstract. We report on work in progress in building an environment for the validation of the meta-theory of programming languages artifacts, for example the correctness of compiler translations; the basic idea is to couple *property-based testing* with *binders*-aware functional programming as the meta-language for specification and testing. Treating binding signatures and related notions, such as new names generation, α-equivalence and capture-avoiding substitution correctly and effectively is crucial in the verification and validation of PL (meta)theory. We use *Haskell* as our meta-language, since it offers not only a very expressive type system, but various libraries for both random and exhaustive generation of tests [7, 8, 13], as well as for binders [16]. We validate our approach on benchmarks of mutations presented in the literature [2, 10] and some examples of code "in the wild". In the former case, not only did we very quickly (re)discover all the planted bugs, but we achieved that with very little configuration effort with comparison to [10]. In the second case we located several simple bugs that had survived for years in publicly available (academic) code. We believe our tool adds to the increasing evidence of the usefulness of property-based testing for semantic engineering of programming languages, in alternative or prior to full verification.

1 Introduction

Recent years have seen major advances in what we could call the *meta-correctness* of programming, that is the (formal) verification of the trustworthiness of the *tools* with which we write programs: from static analyzers to compilers, including parsers, pretty-printers all the way down to run time systems, see projects such as *CakeML* (`cakeml.org`) and *VST* (`vst.cs.princeton.edu`).

More specifically, we are (in pretty good company [10, 15], to cite only two related projects) interested in providing support to the "working semanticist" while designing and prototyping programming languages and related artifacts. To date, very few PL are based on rigorous models, Standard ML being the shining example. On the other corner, infamous is the case of 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, see `http://itc.conversationsnetwork.org/shows/detail58.html`)

In the middle we find lengthy prose documents such as the *Java Language Specification*, whose internal consistency is but a dream, as a very recent paper shows [1]. The properties that PL artifacts should satisfy spans from type soundness as in the above case, to compiler correctness (e.g., *CompCert* `http://compcert.inria.fr/`), to more intensional guarantees related to security (*SECOMP*, `https://secure-compilation.github.io`). Further, although the average programmer is not likely to write her own programming language, she may try her hands on a *Domain Specific Language*, and this design may incorporate flaws that will trickle down and produce hard-to-find bugs in the final product. In this sense "every programmer is a language designer at some point" (Pierce, from the introduction to *Software Foundations*).

This is all good, but the formal verification of PL metatheory is still a very labor-intensive task, even (or more so) with a proof-assistant, so much that there are perhaps only a few dozen people in the world able and willing to carry out such an endeavor. A lighter alternative is *validation*, in the form of *property-based testing* (PBT) as pioneered by QuickCheck [7]: here, we try to *refute*, rather than prove, the properties of the calculus underlying our software artifacts, via *random* or *exhaustive* generation of test cases. For many classes of (typically) shallow bugs, a tool that automatically finds counterexamples can be surprisingly effective and can complement formal proof attempts by warning when the property we wish to prove has easily-found counterexamples. The beauty of this form of meta-theory model checking is that the properties that should hold are already given by means of the theorems that the calculus under study is supposed to satisfy. Of course, those need to be fine tuned for testing to be effective, but we are free of the thorny issue of specification generation.

A particular dimension in validation in this domain is the handling of *binding signatures*, by which we mean the encoding of PL constructs sensitive to naming and scoping: declarations, closures, α-equivalence of method/function arguments, capture-avoiding substitutions, generation of fresh names, nonces, etc [4]. These are ubiquitous in the specification of high-level programming languages, surprisingly easy to get wrong, often callously ignored or so awkwardly supported that they may constitute a unnecessary stumbling block for validation and verification.

This paper describes work in progress in building an environment where one can validate PL meta-theoretical properties with a combination of automated testing tools and an appropriate treatment of binders. We use *Haskell* as our meta-language, since it offers not only a very expressive type system, but various libraries for both random and exhaustive generation of tests [7, 8, 13] as well as for binders [16]. The idea is to allow the user to specify her semantic model(s) with a human-friendly notion of binders and validate them with a cascade of testing tools with the least amount of configuration effort. This is contrast with the competition where either the testing strategy is fixed [5] or binding issues are totally ignored (or both) [10]. We validate our approach both on benchmarks of mutations presented in the literature [2, 10] and some examples of code "in the wild". In the former case, not only did we very quickly (re)discover all the planted bugs, but we achieved that with very little configuration effort with comparison to [10]. In the second case we located several simple bugs that had survived for years in publicly available (academic) code. As a side effect, we have gained some new insights about which testing tool is better suited to various domains and facets in PL meta-theory.

2 Binders-aware PBT

Our tool offers a PBT environment integrating several Haskell libraries, composed, as expected with a thin layer of monadic code. There are several (non-orthogonal) dimensions around which we can arrange test data generation in a PBT setting:
- *random* vs. *exhaustive* test generation;
- *automatic* vs. *hand-written* configuration of generators;
- For exhaustive enumeration, whether we define upper bounds for generation in terms of *size* (number of constructors in a term) or *depth* (depth of the AST for a constructor).

We assume that the reader is familiar with QuickCheck [7] (random enumeration, hand-written generators, minimal counterexamples exist only if the user implements *shrinking*) and we briefly describe the others libraries we have integrated. SmallCheck [13] is essentially an exhaustive enumeration version of QuickCheck, where the upper bound is defined as term depth. LazySmallCheck [13] is a variant of SmallCheck that leverages Haskell's laziness to get around the limits of SmallCheck's brute enumeration of values: the idea is to use *partially-defined* expressions (in logical terms, non-ground ones) to stand for the set of all their instantiations; e.g. lazy evaluation will realize that a list such `(1 : 0 : undefined)` is not, say, *ordered* without generating any concrete list `(1 : 0 : xs)` of ever increasing depth. Feat [8] is built on the concept of functional enumerations, that is, efficiently computable bijections from natural numbers to values. Values are partitioned with respect to their size. This fact combined with specific implementation techniques (i.e. memoisation, memory sharing) allows the enumeration process to be performed for a specific size or, more interestingly, from an arbitrary index, without incurring in the cost of enumerating the preceding smaller values. All the above tools allows the automatic derivation of generators out of the grammar rules of the language under study; however, it is also possible to fine tune them manually, to a different degree in each tool, e.g., adjusting the weights of certain constructors.

The next ingredient is how to best represent binders, a long standing issue in the context of meta-theory verification. Once discarded using a naive named syntax, a.k.a. standard AST, for its inadequacy and prone-ness to mistakes, one choice is the locally nameless representation [3]: this couples the use of raw names for free variables with de Bruijn indexes for bound variables; the latter are basically pointers linking a bound variable to its binding site, thus collapsing all α-equivalent terms into a unique canonical representation. E.g., Java 8 anonymous functions `(int x) -> x` and `(int y) -> y` would both be mapped to the AST `(L int 1)`, for `L` a putative constructor for lambdas.

While this technique is handy and widespread, it is very hard to read for humans and furthermore it needs to be re-implemented for every binding operator in every case study one wants to validate. Unbound [16] is a Haskell library that provides a DSL for the nameless representation, while offering to the user a named surface syntax. The library ensures that we cannot encode illegal values (i.e. a bound variable without a surrounding binder) and at the same times implementing useful operations such as capture-avoiding substitutions, fresh name generations etc. Since Unbound sits on top of several other Haskell libraries, its coexistence with PBT tools is not immediate; in fact it makes the adoption of some problematic, e.g. the automation of QuickCheck generators described in `hackage.haskell.org/package/generic-random`.

3 Experiments

We validate our approach with two sets of experiments, the first showing that we easily handle some of the benchmarks of mutations presented in the literature [2, 10], the second hunting for bugs "in the wild". We give some details of the first case, while we refer to [9] for much more.

Functional programming with lists This comes from the PLT-Redex benchmark suite `http://docs.racket-lang.org/redex/benchmark.html` and concerns the type soundness of a prototypical λ-calculus with lists and related operations, whose BNF includes the following:

Types	σ	$::= int \mid ilist \mid \sigma \to \sigma'$
Terms	M	$::= x \mid \lambda x{:}\sigma.\ M \mid M_1\ M_2 \mid c$
Constants	c	$::= n \mid nil \mid cons \mid hd \mid tl \mid plus$
Values	V	$::= c \mid \lambda x{:}\sigma.\ M \mid cons\ V \mid cons\ V_1\ \ V_2 \mid plus\ V$

Given rules for typing ($\Gamma \vdash M : \tau$) and small step reduction ($M \rightsquigarrow M'$), and a judgment *error* identifying expressions that may produce run-time errors such as taking the head of an empty list, the properties we wish to validate are:

$$M : \tau \wedge M \rightsquigarrow M' \implies M' : \tau \qquad \text{(Preservation)}$$

$$M : \tau \wedge \neg(M \text{ is a value}) \wedge \neg(M\ error) \implies \exists M'. M \rightsquigarrow M' \qquad \text{(Progress)}$$

To give a feel of what using Unbound entails we report the encoding of *Terms* in which `Constant` and `Type` are the data-types for the homonymous grammar rules, whereas `Bind` come from Unbound's DSL, signaling that the constructor `Lam` has a binding occurrence of a variable `Name`; this provides for free α-equivalence of terms with binders and automatically derives functions for substitutions, free names etc.

```
data Exp = Const Constant
         | Var (Name Exp)
         | Lam Type (Bind (Name Exp) Exp)
         | App Exp Exp
```

Bug	Cl	F (au)	F (hw)	SC (au)	SC (hw)	LSC (hw)	QC (hw)	αCheck
B#1 (prog.)	S	10.2	1.5	1.0	2.8	0.1	18.0	12.2
B#1 (pres.)	S	35.4	61.1	✘	✘	18007.7	341.4	311.1
B#2 (prog.)	M	618.2	65.6	3960.7	13269.2	0.8	4010.8	271.3
B#3 (prog.)	S	9.7	1.6	1.0	2.8	0.1	16.4	7.3
B#3 (pres.)	S	10.8	9.4	7.2	68.7	2.7	7.3	39.9
B#4 (prog.)	S	✘	✘	✘	✘	10.1	✘	✘
B#5 (pres.)	S	37134.8	4191.7	✘	✘	2.9	✘	✘
B#6 (prog.)	M	36453.6	4158.8	✘	✘	2.5	✘	298234
B#7 (prog.)	S	124.1	445.7	4.7	3792.1	2.4	510.4	1042
B#8 (pres.)	U	2.8	9.5	✘	759.7	5.6	100.4	21.3
B#9 (pres.)	S	35.5	58.6	✘	✘	17297.1	243.2	18.2

Table 1. Performances on the functional programming with lists benchmark

For example, we encode the identity function on integers as `Lam TyInt (bind x) (Var x))`, where `x = s2n "x"`, using a built-in that converts strings to Unbound names; this is actually syntactic sugar for the nameless term we saw in Sect. 2.

The benchmark introduces nine mutations with Redex's difficulty classification (shallow, medium, unnatural) to be spotted as a violation of either or both properties. E.g., the first mutation introduces a bug in the typing rule for application, matching the range of the function type to the type of the argument (on the right, the correct rules):

$$\frac{\Gamma\vdash M:\sigma\rightarrow\tau \quad \Gamma\vdash N:\tau}{\Gamma\vdash M\,N:\tau}\ \mathrm{T{-}APP{-}B1} \qquad\qquad \frac{\Gamma\vdash M:\sigma\rightarrow\tau \quad \Gamma\vdash N:\sigma}{\Gamma\vdash M\,N:\tau}\ \mathrm{T{-}APP{-}OK}$$

We show in Table 1 some experimental results, taken on a machine with an Intel Core 2 Duo CPU 2.4GHz and 4GB of RAM. The measurements are reported in milliseconds and were collected by averaging the execution times of ten runs. The cells marked with '✘' indicate that no counterexamples have been produced within the time limit, which we set to 300 seconds. Cl reports Redex's classification of the hardness of bugs; hw and au stands for the hand-written and automatically derived generators, whereas F, SC, LSC and QC are respectively Feat, SmallCheck, LazySmallCheck and QuickCheck. We also add a column with results on the same benchmark of αCheck [5] (under the best available strategy, manually chosen). While comparing our tool with an experimental nominal logic programming interpreter has an orange-to-apple taste, the latter is the only tool where PBT is coupled with a declarative handling of binding signatures.

QuickCheck missed three bugs and, as usual, it required a hand-written generator whose development can be tricky, especially if one also wants shrinking. SmallCheck was the worst of the five and it found its bugs only when invoked with the exact specific depth of the bug, which of course is an unrealistic assumption. Using partially defined AST seems to really help in quickly discarding a whole classes of non-well typed terms; indeed, LazySmallCheck was the only tool that was able to find all the bugs. Feat's performances are encouraging, considering it does not use laziness as LazySmallCheck does, which, as we will see shortly it is not always a successful strategy. Feat was able to find all but one counterexamples in less than five seconds without incurring in the exponential explosion brought by enumeration by depth. The hand-written generators performed better than the automated ones, but only in two cases this would have been been discernable by the user.

By construction Feat exhibits size-minimal counter-examples, while (Lazy)SmallCheck produces depth-minimal ones. In this benchmark, however, they essentially reported very similar terms.

Information Flow Security Here we exit the comfort zone of functional programming and standard type soundness to tackle more intensional properties, related to (static) information flow security, which are notoriously hard for (random) PBT [11]. The setup, inspired by [2] is a model of a basic

bug	check	Nit	αC	F (au)	SC (au)	LSC (hw)	Description
B#1	conf	sp	30.2	12857	55.9	✘	second premise of seq rule omitted
B#1	non-inter	✘	6001	✘	9520	✘	ditto
B#2	non-inter	sp	1923	✘	83.1	4.37	var swap in $\leq$ premise of assn rule

Table 2. Performances on the info flow security benchmark

imperative language where variables are assigned a *security* level (the higher the more confidential) and where a type system $l \vdash c$, for command c and security level l, guarantees that c only contains *safe* flows to variables whose confidentiality is over l. This language has a big *step-indexed* operational semantics $\langle c,\sigma\rangle \Downarrow^n \sigma'$ relating a command and an input state to the final state, where the index is a sort of "fuel" to tame non-termination. The properties of interest relate states that agree on the value of each variable (strictly) *below* a certain security level, denoted as $\sigma_1 \approx_{<l} \sigma_2$.
Confinement If $\langle c,\sigma\rangle \Downarrow^n \tau$ and $l \vdash c$ then $\sigma \approx_{<l} \tau$;
Non-interference If $\langle c,\sigma\rangle \Downarrow^n \sigma'$, $\langle c,\tau\rangle \Downarrow^n \tau'$, $\sigma \approx_{\leq l} \tau$ and $0 \vdash c$ then $\sigma' \approx_{\leq l} \tau'$;

The challenge in validation here lies not only in the complexity of the test data that we are generating (states, security assignments, rather than simply expressions), but how constrained they are: viz. non-interference requires the generation of two execution states (σ and τ) that are indistinguishable under a certain security level. We list in Table 2 the results over two mutations in the typing rules, following [2]. Again we add a column for αCheck and one for NitPick, the counterexample generator for Isabelle/HOL, noting that those specifications are *relational* and that NitPick may produce possible false positives (sp fr "spurious"). Only SmallCheck was able to find all bugs with times comparable to αCheck. We conjecture that this is due to the possibility of setting selective upper bounds for enumeration, a feature not present in LazySmallCheck or Feat. We abandoned here QuickCheck since the complexity of the premises of our properties made coverage hopeless. Using hand-written generators did not pay off compared to the automatically derived ones. In all cases we had to manually tune the properties so that useless test data could be discarded (i.e. states with a number of variables different from the ones used in the command that we are executing). The take-home lesson is that using different testing strategies and tools is winning over a fixed one, however refined this may be.

Code "in the wild" While it is reassuring to be able to find mutations listed in the literature, the proof of the pudding is exercising code whose validity is not known, save for having stood some unit testing. This also eliminates any bias in the definition of hand-written generators, which can be skewed by the foreknowledge of the existence of a bug. Of course, we are limited to testing Haskell implementation of PL artifacts available on the net and we selected some whose soundness properties were immediate. We adopted a feedback-loop strategy by which we searched for bugs, corrected the spec and then restarted. Once we reached what seemed like a fixed point, we collected coverage statistics about the main functions and declared "victory". We set the system to use Feat first and it paid off immediately — the other strategies did not contribute any further bug.

Among several experiments (see [9]) here we mention taking on an Haskell porting (`code.google.com/archive/p/tapl-haskell/`) of the code coming with Pierce's textbook "Types and Programming Languages", in particular *fullsimple*, a model of the core of Standard ML. We found nine fairly shallow bugs falsifying the progress property, which we do not have the space to discuss. We impute them to lack of attention to the interaction of different language features, such as type ascription, variant types, etc. However trivial, they had survived some pretty extensive unit testing, at least for academic standards.

4 Conclusions and future work

Although at an early stage of development, we believe our tool adds to the increasing evidence of the usefulness of property-based testing for semantic engineering of programming languages, in alternative or prior to full verification, and should be added to the work flow of PL design and verification: spec'n'check in this context is dirt simple, quick and effective in locating shallow but irritating bugs and doubling as a compelling way to do *regression* testing.

The success of our approach can be attributed to two factors: first, the integration with a tool such as Unbound, which handles binders almost as easily as with named syntax; this without incurring in significant run-time penalties or, more importantly in this setting, false positives stemming from incorrect implementation of basic notions such as substitutions etc. Secondly, the possibility of leveraging different cascading testing strategies: in each benchmark, at least one strategy was successful in catching the required bug. Contrary to common expectations, we found random testing to be in this domain labour intensive without providing the ability to go "deep" in any meaningful way. Exhaustive enumeration, by contrast, revealed to be an excellent choice: easy to use, predictable and reasonably effective in bug finding. Recent improvements [6] in the generate-and-test approach for properties with hard to satisfy conditions will make it scale even further.

We envision our testing environment to be the target of even more declarative semantic engineering tools such as *Ott* [14], which offers the possibility of specifying PL theory as high-level texts (grammars and proof rules in ASCII) and then converting it to executable specs in a variety of proof assistants — one strong point in common with us being the use of the locally nameless style for binders. The framework would benefit from an extension with a source language for specifying and automatically deriving custom generators, possibly following [12]. We also plan to tackle bigger case studies, such as validating existing programming languages directly implemented in Haskell: Idris (`http://www.idris-lang.org/`) comes to mind, a functional programming language with dependent types for whose implementation there are several conjectured soundness properties ready to be validated.

References

1. N. Amin and R. Tate. Java and Scala's type systems are unsound: the existential crisis of null pointers. In *OOPSLA 2016*, pages 838–848, 2016.
2. J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in isabelle/hol. In *FroCoS*, volume 6989 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 2011.
3. A. Charguéraud. The locally nameless representation. *J. Autom. Reasoning*, 49(3):363–408, 2012.
4. J. Cheney. Toward a general theory of names: binding and scope. In *MERLIN*, pages 33–40. ACM, 2005.
5. J. Cheney and A. Momigliano. Mechanized metatheory model-checking. In *PPDP*, pages 75–86. ACM, 2007.
6. K. Claessen, J. Duregård, and M. H. Palka. Generating constrained random data with uniform distribution. *J. Funct. Program.*, 25, 2015.
7. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP 2000*, pages 268–279. ACM, 2000.
8. J. Duregård, P. Jansson, and M. Wang. Feat: functional enumeration of algebraic types. In J. Voigtländer, editor, *Haskell Workshop*, pages 61–72. ACM, 2012.
9. G. Fachini. Validating the meta-theory of programming languages with haskell. Technical report, Università degli Studi di Milano, 2016.
10. M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.
11. C. Hritcu and al. Testing noninterference, quickly. In *ICFP '13*, pages 455–468. ACM, 2013.
12. L. Lampropoulos, D. Gallois-Wong, C. Hritcu, J. Hughes, B. C. Pierce, and L. Xia. Beginner's luck: a language for property-based generators. In *POPL*, pages 114–129. ACM, 2017.
13. C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy SmallCheck: automatic exhaustive testing for small values. In *Haskell Workshop*, pages 37–48, 2008.
14. P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.
15. E. Visser and al. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In *Onward! 2014, SPLASH '14*, pages 95–111, 2014.
16. S. Weirich, B. A. Yorgey, and T. Sheard. Binders unbound. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *ICFP 2011*, pages 333–345. ACM, 2011.

A Appendix for the Reviewers

In this section we provide some indications on how to reproduce the discussed results. Our development lives under Linux and hereafter we assume to be in such an environment as well. Nonetheless, we believe that it should be possible to reproduce the results also on both OS X and Windows (i.e. by means of Cygwin). The code relative to the experiments can be downloaded from a dedicated github repository `https://github.com/GuglielmoS/pbtonplmt`. Inside it you will find four folders, namely:

- *stlc-redex*, which contains the simply typed lambda calculus code;
- *imp-stc*, which contains the information-flow example;
- *fullsimple* and *stlc-unbound*, which refer to the code-in-the-wild section.

The testing and binding related libraries that have been used come as Haskell packages. As a consequence, compiling and executing the experiments require their installation. To ease this phase, we adopted the Haskell tool Stack (`https://www.haskellstack.org/`). Assuming it installed on the system one can directly compile each experiment without additional effort, because the third-party libraries will be automatically downloaded and installed during the build phase. Furthermore, using *stack* implies that the versions of the Haskell compiler and of all the libraries will be consistent.

To compile the code of an experiment, enter in the experiment folder and then execute the *stack build* command. For instance, the *stlc* code can be compiled in the following way:

```
$ cd stlc-redex/
$ stack build
```

In *stlc-redex*/ and *imp-stc*/ there are Python3 and Bash scripts which allow one to automatically collect and average the timings of counterexample search for each considered mutation. The main script is called *test_all* and can be invoked as one would normally do with usual Bash scripts. As an example, reproducing the simply typed lambda calculus results can be done with `$ ./test_all` The number of runs is set to 10 and the output is printed in the CSV format. In addition to the average execution time there are also the timings for each single run of the experiment.

Regarding the code-in-the-wild counterexamples, there are no benchmarking scripts. However, one can search and obtain the counterexamples by executing the compiled code. As an example, consider the *fullsimpe* case study:

```
$ stack build && stack exec fullsimple
```

The last command will compile and then execute the code present in src/Main.hs. In particular it will search and exhibit counterexamples to the properties of interest.