# Validating the meta-theory of programming languages (short paper)

Guglielmo Fachini [1] Alberto Momigliano [2]

[1]INRIA Paris

[2]Università degli Studi di Milano

September 8, 2017
SEFM'17

# Introduction

- Programming languages are by now very complex artifacts
- The more advanced features they have, the more subtle are the interactions that may arise, making the language **unsound** (failure of *type safety*):
  - *Polymorphism and memory references in ML* Memory cells containing certain polymorphic values
  - *Java generics* Generics and implicit constraints on null pointers (Amin and Tate [2016])
- It's important to work with formal definitions which allow one to perform rigorous reasoning on PL properties

# Meta-theory of programming languages

- We are interested in the **meta-theory**, i.e. the study of properties that calculi underlying PL should satisfy:
  - e.g. can valid programs get stuck at run-time?
- The goal is to gain confidence in the correctness of PL design
- Machine assisted formal **verification** offers the strongest guarantees and it's what people generally aim at

# Why validation is interesting

- Machine assisted formal verification still is
  - lots of hard work
  - unhelpful when the theorem we are trying to prove is wrong
    - statement is too strong/weak
    - there are minor mistakes in the specification
- Oftentimes, a failed proof attempt is not the best way to debug those mistakes
- In a sense, verification is only worthwhile if we already "know" the system is correct, not in the design phase!
- A cheaper alternative is **validation**: instead of proving, we try to **refute** those properties

# Context

- Testing in combination with theorem proving is by now well-threaded grounds since Isabelle/HOL's adoption of *random* testing (2004)
- Several frameworks provide support in designing PL
  - e.g. Spoofax, PLT-Redex
- However, **none** of them offers adequate solutions to both
  - **testing** of the meta-theory
  - the correct treatment of **binding signatures**

# What we propose here

Set up a Haskell environment to validate PL's meta-theory:

- Using property-based testing with several strategies and tools
- Taking binders seriously and declaratively
- Limiting the efforts needed to configure and use all the relevant libraries
  - limiting the manual definition of complex generators
  - producing counterexamples in reasonable time (five minutes)
- Emphasis on catching shallow bugs during semantic engineering

# Property-based testing

- Originally introduced with a Haskell library [Claessen and Hughes, 2000], combining *executable specifications* with *test data generation*
- The goal is to use either exhaustive or random test data generation to falsify properties
- Example: insertion in an ordered list

```
insert :: Int -> [Int] -> [Int]
insert x xs = ...
```

- A property about *insert*: insertion preserves order

```
prop_insertOrdered x xs =
  ordered xs ==> ordered (insert x xs)
```

- In general, it may be hard to directly generate test data satisfying certain preconditions (e.g. well-typed programs)

# Binding signatures

- **Variables** and **scopes** are widespread concepts programming language
- Roughly, variables can be of two kinds, **free** or **bound**, depending on whether they appear or not under a **binder**: E.g. `k` is free, whereas `x` is bound in this in this C-like fragment:

```
int mulk(int x) { return k * x; }
```

- Several PL phenomena are related to binders
  - e.g. argument passing in function calls ($\beta$-reduction)
- e.g. capture-avoiding substitution, $\alpha$-equivalence, fresh-name generation

# Example: an inlining transformation (I)

- Inlining can be modeled as a transformation based on substitutions
- Consider the following code

```
int f(int k) { return mulk(k); }
```

- Naively substituting `mulk's` body in `f` yields:

```
int f(int k) { return k * k; }
```

- Problem: in `mulk`, the first `k` is **free**, but now, it has become **bound** in `f`, changing its semantics
- Where we should have:

```
int f(int k2) { return k * k2; }
```

# The approach

- Represent the object system with Haskell as the **meta-language**
- Specify properties that should hold
  - no need to invent them, they are the theorems that should hold for your calculus
- System searches (exhaustively/randomly) for counterexamples

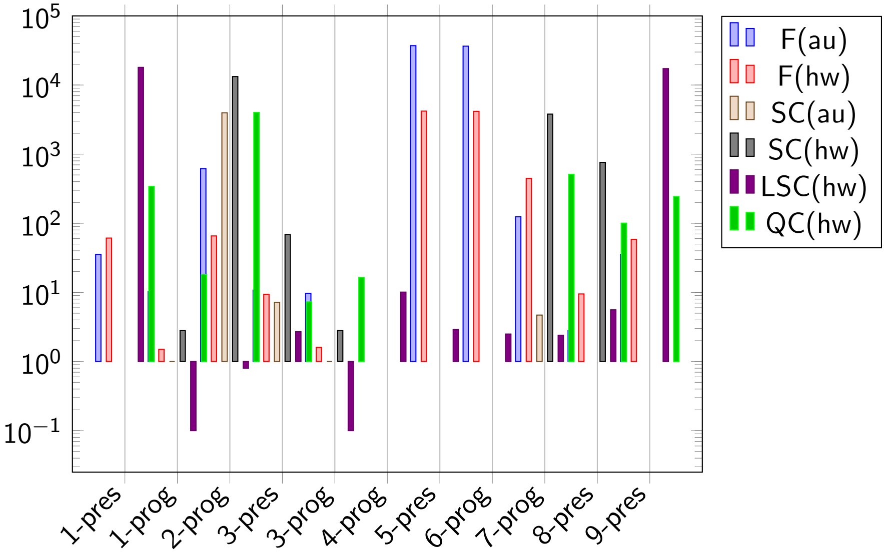
# Tools

- Testing with random generation
  - *QuickCheck* [Claessen and Hughes, 2000]
    random testing with hand-written generators
- Testing with exhaustive enumeration
  - *SmallCheck* [Runciman et al., 2008]
    exhaustive enumeration up to some depth
  - *LazySmallCheck* [Runciman et al., 2008]
    similar to SmallCheck, but leverages partially defined expressions to prune the search tree
  - *Feat* [Duregård et al., 2012]
    based on functional enumeration; exhaustive enumerations up to some size (the number of constructors)
- Binding signatures
  - *Unbound* [Weirich et al., 2011]
    based on the locally nameless approach, but offers on top of it a form of named syntax

# How we evaluated our approach

- We first analyzed a case study widely used in the literature ([Findler et al., 2015])
  - simply typed lambda calculus with constants
  - *type soundness* was the property of interest
  - manually introduced mutations [Findler et al., 2015]
  - most of them located in milliseconds, although the various tools and testing strategies performed differently
- Then, we focused on systems publicly available
  - porting of TAPL [Pierce, 2002] languages to Haskell
  - focused on a variety of properties, preliminary and including type soundness
  - the flaws that we found, although simple, broke type safety

# Experimental results on STLC



# Other interesting cases

- Manually introduced flaws in secure flow type systems
  - properties deal with data whose preconditions are fairly involded
  - only certain testing techniques are able to exhibit counterexamples
- Value restriction in a toy ML language
  - "deep" counterexamples which are generally difficult to reach with automated testing strategies
  - finding counterexamples requires some tuning of the specifications

# Conclusions

- PBT is a great choice for validating PL meta-theory
- Spec and checks make great regression tests
- Our Haskell approach offers a lot of goodies to do this conveniently and with reduced configuration effort so as to be (hopefully) usable by non-experts
- Not surprisingly, it's not clear cut to understand which testing strategy will perform better in a given domain, but having a cascade of them is a big plus

# Future Work

- Evaluate the effectiveness of stronger random generators
  - e.g. Boltzmann samplers, QuickChick's generators
- Integration with code coverage tools (i.e. hpc) when counterexamples don't show up anymore
- Import techniques from *provenance* and *declarative debugging/abduction* to locate the part of the code that is to blame for the bug
- More cases studies
  - Redex benchmarks models
  - Some *model-based testing* of existing programming languages developed in Haskell (i.e. Idris, Mini Agda)

Thanks!

# References I


Amin, N. and Tate, R. (2016). Java and Scala's type systems are unsound: the existential crisis of null pointers. In *OOPSLA 2016*, pages 838–848.

Claessen, K. and Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP 2000*, pages 268–279. ACM.

Duregård, J., Jansson, P., and Wang, M. (2012). Feat: functional enumeration of algebraic types. In Voigtländer, J., editor, *Haskell Workshop*, pages 61–72. ACM.

Findler, R. B., Klein, C., and Fetscher, B. (2015). Redex: Practical semantics engineering.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

# References II


Runciman, C., Naylor, M., and Lindblad, F. (2008). Smallcheck and lazy SmallCheck: automatic exhaustive testing for small values. In *Haskell Workshop*, pages 37–48.

Weirich, S., Yorgey, B. A., and Sheard, T. (2011). Binders unbound. In Chakravarty, M. M. T., Hu, Z., and Danvy, O., editors, *ICFP 2011*, pages 333–345. ACM.