Property-Based Testing PL Artifacts

An experience report

Alberto Momigliano

Università degli Studi di Milano

joint work with Guglielmo Fachini, INRIA Paris

CLA 2017

Roadmap

- Why we do this
- What we did
- What we cannot do (yet, hopefully)

Motivation

- Focus: *meta-correctness* of programming, e.g. (formal) verification of the trustworthiness of the *tools* with which we write programs:
  - from static analyzers to 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*, the *Language Workbench*, *K*...
- Let's stick to programming language design for this talk.

Motivation

- One shiny example: the definition of SML

Motivation

- 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].

Mechanized meta-theory

- We're not interested in **program** verification, but in semantics engineering, the study of the meta-theory of programming languages
- Most of it based on common syntactic proofs:
  - type soundness
  - (strong) normalization/cut elimination
  - correctness of compiler transformations
  - simulation, 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

- We're not interested in **program** verification, but in semantics engineering, the study of the meta-theory of programming languages
- Most of it based on common syntactic proofs:
  - type soundness
  - (strong) normalization/cut elimination
  - correctness of compiler transformations
  - simulation, 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
- Yeah. Right.

Mechanized meta-theory

- We're not interested in **program** verification, but in semantics engineering, the study of the meta-theory of programming languages
- Most of it based on common syntactic proofs:
  - type soundness
  - (strong) normalization/cut elimination
  - correctness of compiler transformations
  - simulation, 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
- Yeah. Right.
- mechanized meta-theory verification: using **proof assistants** to ensure with maximal confidence that those theorems hold

Not quite there yet

- Problem: Verification still is
  - lots of hard work (especially if you're no Xavier Leroy, nor Peter Sewell et co.)
  - unhelpful when the theorem I'm trying to prove is, well, wrong.

Not quite there yet

- Problem: Verification still is
  - lots of hard work (especially if you're no Xavier Leroy, nor Peter Sewell et co.)
  - 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!

Property-based testing for PL meta-theory

- A cheaper alternative is **validation**: instead of proving, we try to **refute** those properties:
- (Partial) "model-checking" approach:
  - searches for counterexamples
  - produces helpful counterexamples for incorrect systems
  - unhelpfully diverges for correct systems
  - little expertise required,
  - fully automatic, CPU-bound
- We use PBT to do mechanized meta-theory model checking;
  - Don't think I need to motivate PBT further to this audience, especially after Leonidas' talk.

The approach

- Represent the object system in a **meta-language** (could be a logical framework or an appropriate programming language).
- Specify properties that should hold – no need to invent them, they're the theorems that should hold for your calculus!
- System searches (exhaustively/randomly) for counterexamples.
- Meanwhile, try a direct proof (or go to the beer garden)

The approach

- Represent the object system in a **meta-language** (could be a logical framework or an appropriate programming language).
- Specify properties that should hold – no need to invent them, they're the theorems that should hold for your calculus!
- System searches (exhaustively/randomly) for counterexamples.
- Meanwhile, try a direct proof (or go to the beer garden)
- Testing in combination with theorem proving is by now well-threaded grounds since Isabelle/HOL's adoption of *random* testing (2004)
  - a la QuickCheck: Agda (04), PVS (06), Coq (15)
  - exhaustive/smart generators (Isabelle/HOL (12))
  - model finders (Nitpick, again in Isabelle/HOL (11))

Cons

- Failure to find counterexample does not guarantee property holds, i.e., *false* sense of security
- Hard to tell who to blame in case of failure: the theorem? The spec? If the latter, which part?
- Validation is as good as your test data, especially if you go *random*
- "Deep" bugs in published type systems may be beyond our grasp, see later in the talk

Haven't we seen this before?

- Robbie Findler and co. took on this idea and marketed as *Randomized testing for PLT Redex*

  > *PLT Redex is a domain-specific language designed for specifying and debugging operational semantics. Write down a grammar and the reduction rules, and PLT Redex allows you to interactively explore terms and to use randomized test generation to attempt to falsify properties of your semantics.*

- In other terms, it's unit tests plus *QuickCheck* for metatheory (In Racket, if you can stomach it). Few abstraction mechanisms.
- They made quite a splash at POPL12 with *Run Your Research*, where they investigated "the formalization and exploration of nine ICFP 2009 papers in Redex, an effort that uncovered mistakes in all nine papers."

What Robbie does not tell you (in his POPL talk)

- Redex offers **no** support for binding syntax:

  > *In one case (A concurrent ML library in Concurrent Haskell), managing binding in Redex constituted a significant portion of the overall time spent studying the paper. Redex should benefit from a mechanism for dealing with binding. . .*

- Test coverage can be lousy

  > *Random test case generators . . . are not as effective as they could be. The generator derived from the grammar . . . requires substantial massaging to achieve high test coverage. This deficiency is especially pressing in the case of typed object languages, where the massaging code almost duplicates the specification of the type system. . .*

- The latter point somewhat improved using CLP techniques with Fetscher's thesis, see "Making random judgment" paper [ESOP15].

Another approach: αCheck

- Some related work by James Cheney and myself: `https://github.com/aprolog-lang`
- A PBT tool on top of αProlog, a simple extension of Prolog with nominal abstract syntax
  - Equality **is** α-equivalence, facilities for fresh name generation via the Pitts-Gabbay quantifier...
- Use nominal Horn formulas to write both specs and checks
- System searches exhaustively for counterexamples, via iterative deepening.
- In a sense, not dissimilar from LazySmallCheck, but being natively based on logic programming more effective — does not need to simulate narrowing or backtracking.

What we propose here

Set up a Haskell environment as a competitor to PLT-Redex to validate PL's meta-theory:

- Taking binders seriously (no strings!) and declaratively (to me, this means no DB indexes)
- Varying the testing strategies (and the tools) from random to enumerative
- 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

Handling Binders

- Notions of binders, scope, α-equivalence, fresh name generation etc. are ubiquitous in PL theory
- De Bruijn indexes are fine for the machine, but we should offer a better service to the semantic engineer in terms of usability
- Among the many possibilities available in Haskell, we chose Binders Unbound [ICFP2011], which hides the locally nameless approach under surface named syntax:
  - Mature library
  - Easy to integrate
  - Rich API

Testing Tools

- QuickCheck
- SmallCheck and LazySmallCheck
- Feat
- We have considered both automatically derived generators (au) and manual tinkering (hw) of the latter
- Full disclosure: this work was carried out in 2016 and did not take into (full) account more recent development such as `lazy-search` and `generic-random`, nor *Luck*
- Hence our approach to generating terms under invariants has been, so far, the naive *generate-and-filter* approach

Case Studies

Three kind of experiments:

1. Benchmarks of mutations in the literature:
   - A Simply Typed Lambda Calculus with Constants
   - A Type System for Secure Flow Analysis
2. Code in the "wild":
   - Porting of TAPL implementations in Haskell
   - Examples from the Binders Unbound library
3. Search for deep bugs
   - Let-Polymorphism and references in SML-like languages

1.1 Simply Typed Lambda Calculus

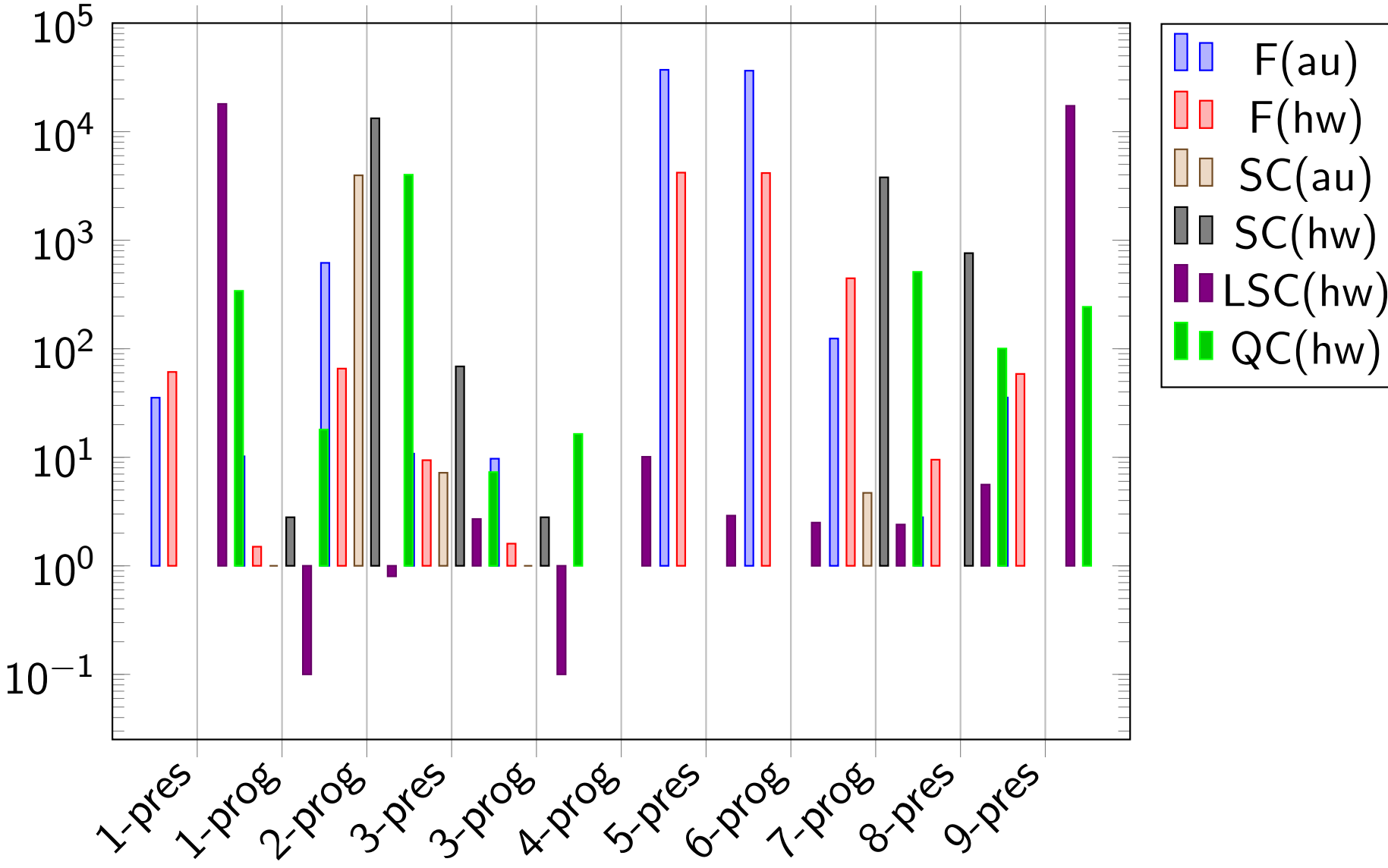
- PLT-Redex benchmark of a Simply Typed Lambda Calculus with constants for list operations
- Nine bugs were introduced in typing and reduction rules. . .
- to be spotted as violation of type soundness:
  - **Progress**
    If e is a well-typed term, then either it is a value or an error, or it can step
  - **Preservation**
    If e is a well-typed term and takes a reduction step, its type does not change

1.2 Testing Results (table)

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: Performances with STLC benchmark in milliseconds

1.2 Testing Results (without αCheck)



Comments

- LazySmallCheck was the only tool that was able to find all the bugs. Partially defined expressions really helped in generating well-typed terms
- Feat and enumeration by size missed one bug but was less volatile than LSC
- QuickCheck missed three bugs. In addition it required of course an hand-written generator
- SmallCheck was the worst: it found most of the bugs only when invoked with the *exact* specific depth – current implementation is not monotonic

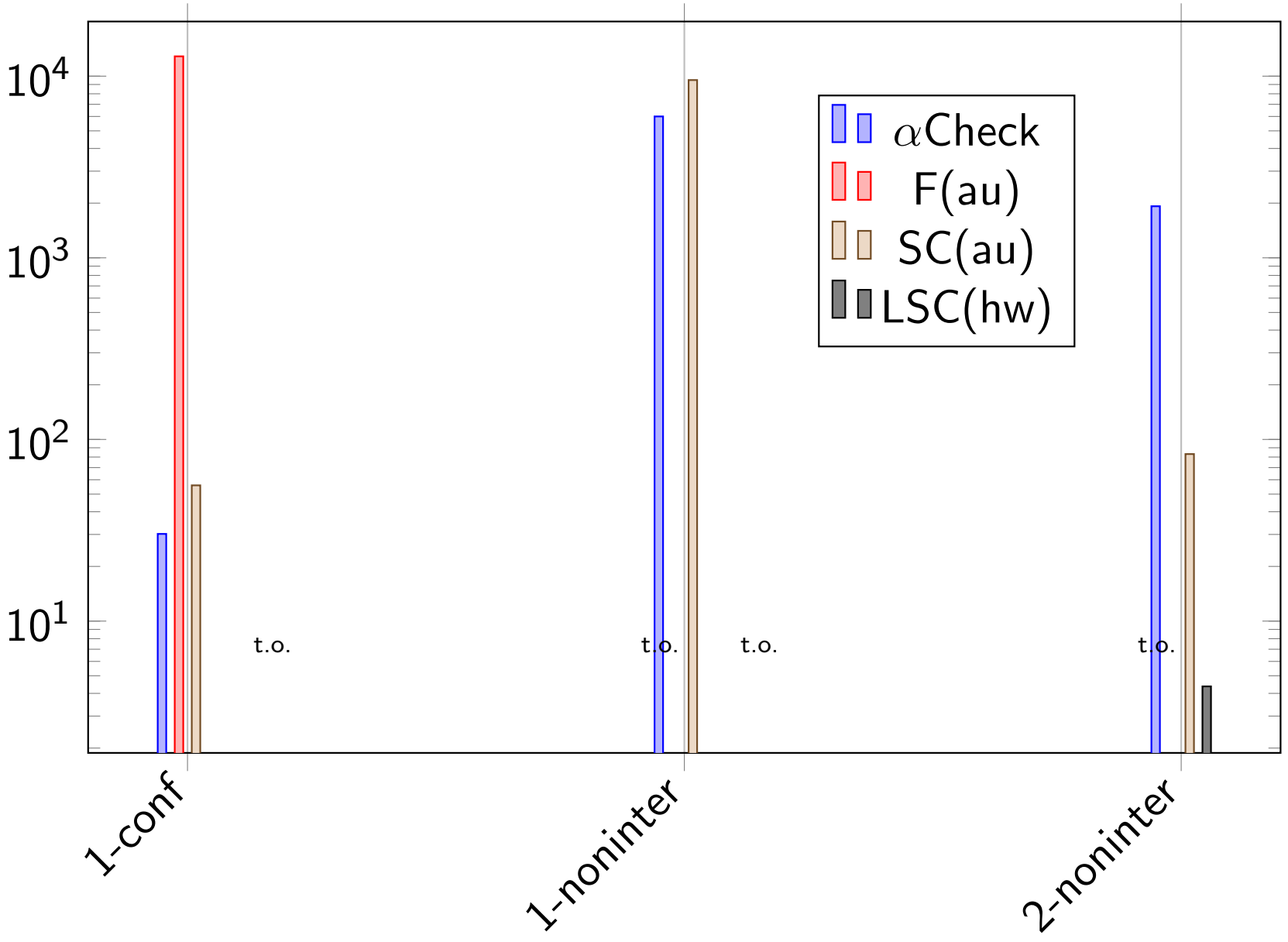
1.2 A Type System for Secure Flow Analysis

- A mild extension of Volpano et al.'s type system as formalized in Nipkow and Klein's *Concrete Semantics*
- It's the standard WHILE imperative language, where each variable has an associated security level
- The absence of information flows from private data to public observers is statically guaranteed by the type system:
  - $l \vdash c$ means that c contains only safe flows to variables with a level $\leq l$.
- We introduced two very simple bugs in the type system rules following a paper about NitPick.
- Setup is much, much simpler than *Testing non-interference, quickly*, which was dynamic IF of abstract machines. Still interesting, we argue.

Properties of Interest

- **Confinement** $[l \vdash c; (c, s) \Rightarrow t] \Longrightarrow s = t\ (< l)$
  The execution of a well-typed command in context l does not alter the variables whose level is below l.
- **Non-interference**
  $[0 \vdash c; (c, s) \Rightarrow s'; (c, t) \Rightarrow t'; s = t\ (\leq l)] \Longrightarrow s' = t'\ (\leq l)$
  Executing the same well-typed command with indistinguishable input states, brings to indistinguishable output states.
- Challenges:
  - Quantifying over complex objects, such as pairs of indistinguishable states
  - Taming non-terminating programs

1.2 Testing Results



1.2 Comments

- SmallCheck found all bugs quickly, while Feat and LazySmallCheck were able to find only one bug...
  - and we're taking fairly shallow bugs
- We didn't even try QuickCheck because of the complexity of the premises of our properties and we didn't want to try and replicate the above paper's very ingenious generators
- In all cases we had to manually tune the properties so that useless values could be discarded (i.e. environments with variables different from target program)...
  - and SC offers more tuning (i.e., selective upper bounds)
- The benchmark confirms how hard it is to test under severe constraints.

Code "in the wild"

- Previous benchmarks were *artificial* — we searched for manually introduced mutations
- We also want to exercise code whose validity is not known, save for having stood some unit testing. We choose:
  1. several models of functional languages from the porting of TAPL's code to Haskell
  2. algorithmic and definitional equality for the lambda calculus from the examples of Unbound
- We validated with Feat a variety of properties leading to type soundness
- In all systems we found flaws in static and/or dynamic semantics that, although simple, broke type safety (progress in particular)

The hunt for deeper bugs

- In the 90's it was realized that combining let-polymorphism and references in a SML-like language would lead to the unsoundness of the type system
  - One solution is the so called *value restriction*, limiting when type inference is allowed to polymorphically generalize a value.
- In 2000 Pfenning and Davies showed a similar issues wrt intersection types and computational effects
- Can we reproduce those counterexamples?

The hunt for deeper bugs

- In the 90's it was realized that combining let-polymorphism and references in a SML-like language would lead to the unsoundness of the type system
  - One solution is the so called *value restriction*, limiting when type inference is allowed to polymorphically generalize a value.
- In 2000 Pfenning and Davies showed a similar issues wrt intersection types and computational effects
- Can we reproduce those counterexamples?
- Short answer: no. And we could use some help here . . .

Our attempt

- A simple MiniML language with references: 12 constructors for expressions, 5 for types
- A naive implementation of type inference — purely functional, substitution are composed eagerly etc
- The smallest cex to preservation is big enough to make the search difficult

```
let f = \x.x in
    f := \x:unit.()
    !f 0
```

- **None** of the tools have been able to find it
  - Even using custom enumerators and expanding the time-limit to two days...
- We could build a specific QuickCheck generator and hope to get lucky, but what would that show?

Difficulties and possible countermeasures

- Memory consumption with Feat
  - Enumerating and testing terms up to size 10 required almost 2.5GB of memory
  - Naive type inference did not help on this aspect
- Unbound overhead
  - the `unbiding` operations alone represent almost 20% of execution time
  - the use of generic programming might be a possible cause
- Going beyond size 10 seems quite difficult. Possible improvements:
  - a more efficient type inference implementation (i.e. with union-find)
  - dropping Unbound

An attempt with `lazy-search`

- Naive use of `lazy-search` for filtering well-typed terms.

An attempt with `lazy-search`

- Naive use of `lazy-search` for filtering well-typed terms.
- No cigar:
  - `lazy-search` is able to successfully enumerate terms up to size 11, whereas Feat crashes due to memory usage
    - Feat has to test 434364201 terms, while for lazy-search checks only 67044859 terms (1/7th)
- At size 10: Feat is generally faster than lazy-search, but at the same time it requires more memory:
  - Feat runs in 245 seconds, while lazy-search runs in 413 (factor=1.68)
  - Feat uses 2420MB of memory, while lazy-search requires 1138MB (factor=2.1)
- Suggestions?

Conclusions

- PBT is a great choice for metatheory model checking.
- Checking specifications is dirt simple – no reason not to write & check on a regular basis
- Checking intermediate lemmas helps catch bugs earlier
- 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.
- Deeper bugs still out of grasp, but that's why I'm standing here ...

- Explore other implementation and engineering choices in our hunt for famous bugs.
- Evaluate the effectiveness of stronger random generators, see `hackage.haskell.org/package/generic-random` and recent developments in QuickChick.
- Some integration with code coverage (perhaps `hackage.haskell.org/package/hpc`), for when bugs 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.

Beyond generate-and-test, but w/o specialized generators: *intrinsically typed* and *well-scoped* terms

- expressions depending on terms `Exp t`, commands indexed by security levels `Cmd l`
- Well studied in logical frameworks (Benton et al [JAR12] in Coq, Allais et al [CPP17] in Agda):
- we could use GADT as a form of dependent types or Liquid Haskell for refinement types.

Going beyond Unbound:

- Pouillard's "Names For Free"
- Some form of HOAS? Probably not a good idea.

More case studies:

- More from the Redex benchmarks models.
- More code in the wild, e.g. LF type checking from Unbound's examples repository.
- Some *model-based testing* of existing programming languages developed in Haskell:
  - write a model of it, validate it vs. its meta-theory, then test the actual implementation against the model:
  - Idris, Andreas Abel's Mini Agda...

Thanks!