An Overview of Property-Based Testing for the Working Semanticist

Alberto Momigliano
based on joint work with James Cheney, Matteo Pessina, Guglielmo Fachini, Francesco Komauli, Rob Blanco, and Dale Miller

DI, University of Milan

KCL
June 18, 2018

Motivation

- We're not interested in program verification in general, but in semantics engineering, the study of the meta-theory of programming languages
- This means *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*. . .

Why bother?

- One shiny example: the definition of SML.

Why bother?

- 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 (MMT)

- 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 (MMT)

- 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 (MMT)

- 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

- After almost 20 years of formal verification with Twelf, Isabelle/HOL, Coq, Abella, I'm a bit worn out
- I still find it a very demanding, often frustrating, day job.
  - It 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

- After almost 20 years of formal verification with Twelf, Isabelle/HOL, Coq, Abella, I'm a bit worn out
- I still find it a very demanding, often frustrating, day job.
  - It 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!
  - That's why I'm inclined to give *testing* a try (and I'm in good company!), in particular property-based testing.

PBT

- 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.

QuickCheck's Hello World! (FsCheck, actually)

```
let rec rev ls =
    match ls with
    | [] -> []
    | x :: xs -> append (rev xs, [x])

let prop_revRevIsOrig (xs:int list) =
    rev (rev xs) = xs;;

do Check.Quick prop_revRevIsOrig ;;
>> Ok, passed 100 tests.

let prop_revIsOrig (xs:int list) =
    rev xs = xs
do Check.Quick prop_revIsOrig ;;

>>  Falsifiable, after 3 tests (5 shrinks) (StdGen (518275965,..
[1; 0]
```

Not so fast… 1/2

- Sparse pre-conditions:

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

- Random lists not likely to be ordered … Obvious issue of coverage. QC's answer:
- monitor the distribution
- write your own generator (here for ordered lists)
  - Writing generators may overwhelm SUT and become a research project in itself — IFC's generator consists 1500 lines of "tricky" Haskell [JFP15]
  - When the property in an invariant, you have to duplicate it as a generator and as a predicate and keep them in sync.
  - Do you trust your generators? In Coq's QC, you can *prove* your generators sound and even complete. Not exactly painless.
- We need to shrink random cex to understand them (δ-debugging). So, you need to implement (and trust) shrinkers as wells.

Not so fast... 2/2

Lots of current work on supporting coding or automatic derivation of (random) generators:

- Needed Narrowing: incremental generate-and-test: traverse the pre-condition instantiating locally the unknown and backtracking if that does not work:
  - Classen [JFP15], Fetscher [ESOP15] with applications to generation of well-typed terms
  - (Note: narrowing also used in exhaustive generation/symbolic execution)
- General constraint solving: **Focaltest** [2010], **Target** [ESOP 15]
- A combination of the two in the **Luck** [POPL17], a DSL to write Haskell-like generator keeping control of the distribution and of the amount of constraints solving.

Exhaustive data generation

An alternative, based on the small scope hypothesis: enumerate systematically all elements up to a certain bound:

- The granddaddy: **Alloy** [Jackson 06];
- **(Lazy)SmallCheck** [Runciman 08]: exhaustive version of QC, plus symbolic execution via laziness. Bound is # of constructors
- **EasyCheck** [Fischer 07]: automatic enumeration test data satisfying a condition in *Curry*
- α**Check**
- Most of the testing techniques in Isabelle/HOL

Back to mechanized meta-theory

- PBT is a form of partial “model-checking”:
  - tries to refute specs of the SUT
  - produces helpful counterexamples for incorrect systems
  - unhelpfully diverges for correct systems
  - little expertise required
  - fully automatic, CPU-bound

Back to mechanized meta-theory

- PBT is a form of partial "model-checking":
  - tries to refute specs of the SUT
  - produces helpful counterexamples for incorrect systems
  - unhelpfully diverges for correct systems
  - little expertise required
  - fully automatic, CPU-bound
- **PBT** for MMT means:
  - Represent object system in a logical framework.
  - Specify properties it should have.
  - System searches (exhaustively/randomly) for counterexamples.
  - Meanwhile, user can try a direct proof (or go to the pub).

Testing and proofs: friends or foes?

- Isn't Dijkstra going to be very, very mad?

  "None of the program in this monograph, **needless to say**, *has been tested on a machine" [Introduction to A Discipline of Programming, 1980]*

- Isn't testing the very thing theorem proving want to replace?
- Oh, no: test a conjecture before attempting to prove it and/or test a subgoal (a lemma) inside a proof
- The beauty (wrt general testing) is: you don't have to invent the specs, they're exactly what you want to prove anyway.
- In fact, PBT is everywhere:

PBT and Proof Assistants

- Random-based:
  - Isabelle/HOL Isabelle/HOL broke the ice some 15 years ago; then Agda (2004) [probably not there anymore], PVS (2006), ACL2 (2009)
  - Coq's foundational **QuickChick** (2015-17) is the most advanced, soon to be featured in the *Software Foundations* curricula, featuring now generator automation (but not for dependent types yet)
- Enumeration-based: this is all Isabelle/HOL, AFAIK
  - Bulwhahn's **smart** generators automatically creates enumerators satisfying a precondition via compilation to logic programs. Exhaustive generation is now the default in Isabelle for executable specs.
  - Blanchette's **Nitpick**, a model finder for higher-order logic by translation to SAT using Alloy's backend.

My work around PBT (not in chronological order)

1. Haskell harness for PBT tools (with Guglielmo).
2. αCheck (with James, Matteo etc.).
3. A reconstruction of PBT in terms of focusing and foundational proof certificates (with Dale and Rob).

Common theme is doing PBT at a good level of abstraction and in a logically justified way.

- I'm looking at you, PLT-Redex . . .

Haven't we seen this before?

- Robbie Findler and co. took on the PBT-for-MMT idea 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 use randomized test generation to attempt to falsify properties of your semantics.*

- 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."
- The authors, you ask? Hudak, Peyton Jones, Henrik Nilsson, Avik Chaudhuri, Jay McCarthy, Oderski. . . The bugs? Nothing major: typesetting rules, some omitted rules, some unexpected behaviour of a model, one false theorem (but fixable)

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

1. The Haskell thing

Aim: set up a Haskell environment as a competitor to PLT-Redex

- Taking binders seriously (no strings!) and declaratively: we used Binders Unbound [ICFP2011]
  - hides the locally nameless approach under named syntax:
  - Mature library, Easy to integrate, Rich API

1. The Haskell thing

Aim: set up a Haskell environment as a competitor to PLT-Redex

- Taking binders seriously (no strings!) and declaratively: we used Binders Unbound [ICFP2011]
  - hides the locally nameless approach under named syntax:
  - Mature library, Easy to integrate, Rich API
- Varying the testing strategies (and the tools) from random to enumerative (QC,SmallCheck, LazySmallCheck,Feat);

1. The Haskell thing

Aim: set up a Haskell environment as a competitor to PLT-Redex

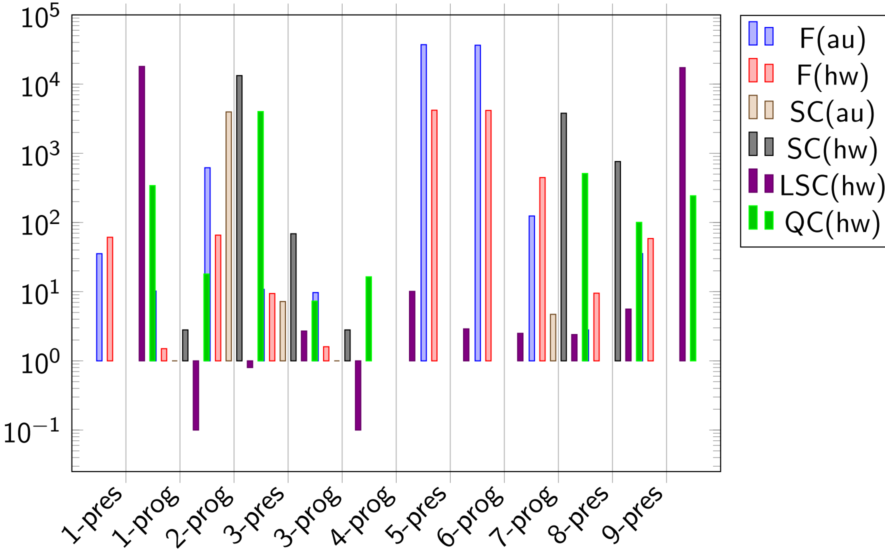
- Taking binders seriously (no strings!) and declaratively: we used Binders Unbound [ICFP2011]
  - hides the locally nameless approach under named syntax:
  - Mature library, Easy to integrate, Rich API
- Varying the testing strategies (and the tools) from random to enumerative (QC,SmallCheck, LazySmallCheck,Feat);
- Limiting the efforts needed to configure and use all the relevant libraries;
  - avoid the manual definition of complex generators, if possible
  - producing counterexamples in reasonable time or bust
- Full disclosure: work carried out in 2016 and did not take into account developments such as `lazy-search` 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 Lists
     - Mutations in typing and reduction rules from the PLT-Redex model, to be killed by preservation/progress:
   - 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

1.2 Testing Results for STLCI



Comments

- LazySmallCheck only tool that killed all mutants. Partially defined expressions really helped in generating well-typed terms
- Feat and enumeration by size missed one mutant but was less volatile than LSC
- QuickCheck missed three mutants. 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

Different outcomes in other case studies — difficult to draw hard conclusions

The hunt for deeper bugs

- In the 90's it was realized that combining let-polymorphism and references in a SML-like makes the the type system unsound

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

- In 2000 Pfenning and Davies showed a similar issues w.r.t. 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 makes the the type system unsound

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

- In 2000 Pfenning and Davies showed a similar issues w.r.t. intersection types and computational effects
- Can we reproduce those counterexamples?
- Kinda. No, not really.

MLR

- 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 lies at depth 13, too deep!
- **None** of the tools found it, even using custom enumerators and expanding the time-limit to two days. . .
- We got it with **Feat** by changing the statement to make store typing explicit:

```
term          App (Deref (StoreLoc 0)) Unit
store typing  {0 -> forall [a]. Ref (a -> a)}
store         {0 -> Lam (bind x (Const 0))}
```

- Here it's simple to reconstruct the classical cex, but how general is this?

2. αCheck

- Our recently (re)released tool: `https://github.com/aprolog-lang`
- On top of αProlog, a simple extension of Prolog with nominal abstract syntax.
- Equality coincides with $\equiv_\alpha$, $\#$ means "not free in", $\langle \mathsf{x} \rangle M$ is an M with x bound, И is the Pitts-Gabbay quantifier.
- Allows functional notation, but it is flattened to relations
- Use nominal Horn formulas to write specs and checks
- αCheck searches exhaustively for counterexamples.
- Iterative deepening search strategy

Problem definition

- Consider a (pure) nominal logic program P and a model $\mathcal{M}$.
- Consider specifications of the form

$$\text{И}\vec{a}\forall\vec{X}.A_1 \wedge \cdots \wedge A_n \supset A$$

  where A can be equality/freshness constraints – encode disjunctions/ existentials as usual in Prolog
- A *counterexample* is a ground substitution θ such that

$$\mathcal{M} \vDash \theta(A_1) \wedge \cdots \wedge \mathcal{M} \vDash \theta(A_n) \wedge \mathcal{M} \nvDash \theta(A)$$

- The *partial model checking problem*: Does a counterexample exist? If so, construct one. Obviously undecidable.
- What do we mean by $\nvDash \theta(A)$? It's logic programming, so careful here: Two forms of negation: negation as failure and negation elimination

An example

- Consider an encoding of a λ-calculus with constants, with predicate `tc` for typing and `step` for the operational semantics.
- Let's insert a mutation (in the typing rule for application) and run the tool

```
tc(G,app(M,N),T):-
  tc(G,M,funTy(T,U)),tc(G,N,U). % should be funTy(U,T)

#check "prsrv" 7: tc([],E,T), step(E,E') => tc([],E',T).
Checking depth 1 2 3 4
Total: 0.008 s:
E = app(lam(x\u),u)
E1 = u
T = pairTy(unitTy,unitTy)
```

Implementation with *NAF*

- A `check` $\text{И}\vec{a}\forall\vec{X}.A_1 \wedge \cdots \wedge A_n \supset A$ is basically a bounded query:

$$?\!- \text{И}\vec{a}.\ A_1 \wedge \cdots \wedge A_n \wedge gen(X_1) \wedge \cdots \wedge gen(X_n) \wedge not(A)$$

- Search for complete (up to the bound) proof trees of all hypotheses
- Instantiate all remaining variables $X_1 \ldots X_n$ occurring in A with exhaustive generator predicates for all base types, automatically provided by the tool.
- Then, see if conclusion fails using NF.
- Easy peasy. Still, NF is messy (semantically and computationally); can we do better?

Implementation with NE

- Idea: Use *negation elimination* instead
  - AKA "intensional negation", similar to "constructive negation",
- For each predicate A, define predicate not_A denoting the "complement" of A
- Avoids need to instantiate variables unless needed in derivation; can reorder goals past negation:

$$?\!- \ A_1 \wedge \cdots \wedge A_n \wedge not\_A \wedge A_{n+1} \wedge \cdots$$

- New implementation (NEs) makes it competitive with NF, though search spaces are quite different.
- Proven sound.

Case studies carried out with αCheck

- Several λ-calculi (from the Redex benchmark suite), MiniML with references, λ-zap, where the properties of interests are related to *type preservation*; up to LF equivalence algorithms and their structural properties.
- The *listmachine* benchmark by Appel & Leroy in the area of compiler correctness.
- 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*

Rough appraisal:

- Immediate to setup, and pretty useful for shallow bugs, typos etc — I really miss it when using **Abella** for example.
- Deeper ones, well, not so much.

3. PBT via Foundational proof Certificates

- Functional approaches to PBT are rediscovering logic (programming): narrowing/mode analysis in Isabelle and Coq's QC, Randomized CLP in PLT-Redex.
- If we take a proof-theoretic view of LP, good things start to happen, and this now means focusing in a sequent calculus.
- Generate-and-test approach to PBT can be seen in terms of focused sequent calculus proof where the positive phase corresponds to generation and a single negative one to testing.
- Searching for a cex is searching for a proof of a polarized formula like $\exists x\, [\tau(x) \wedge^{+} P(x) \wedge^{+} \neg Q(x)]$. This is a single bipole — a positive phase followed by a negative one.
- Intuition: generation is hard, testing is but a deterministic computation.

μMALL

- As the plan is to have a PBT tool for **Abella**, we have in mind specs and checks in multiplicative additive linear logic with (for the time being) least fixed points (Baelde & Miller)
- E.g. , the `append` predicate is:

$$\begin{aligned} app \equiv & \mu\lambda A \lambda xs \lambda ys \lambda zs\ (xs = \mathbf{nl} \wedge^{+} ys = zs) \vee \\ & \exists x' \exists xs' \exists zs' (xs = \mathbf{cns}\ x'\ xs' \wedge^{+} zs = \mathbf{cns}\ x'\ zs' \wedge^{+} A\ xs'\ ys\ zs' \end{aligned}$$

- Usual polarization for LP: everything is positive — note, there are no atoms.

A further step: FPC

- A flexible and general way to look at those proofs is as a proof reconstruction problem in Miller's Foundational Proof Certificate framework
- FPC proposed as a means of defining proof structures used in a range of different theorem provers
- If you're not familiar with it, think a focused sequent calculus augmented with predicates (**clerks** for the negative phase and **experts** for the positive one) that produce and process information to drive the checking/reconstruction of a proof.
- For PBT, we suggest a lightweight use of FPC as a way to describe generators by fairly simple-minded experts.

FPC for the common man

- We defined certificates for families of proofs (the generation phase) limited either by the number of inference rules that they contain, by their size, or by both.
- They essentially translate into meta-interpreters that perform bounded generation of derivations.
- As a proof of concept, we implement this in λProlog and we use *NAF* to implement negation — it's a shortcut, but theoretically, think fixed point and negation as $A \rightarrow \perp$.
- We use the two-level approach: OL specs are encoded as `prog` clauses and `check` will meta-interpret them using the (size/height) certificate to guide the generation.
- Checking $\forall x{:}elt, \forall xs, ys{:}eltlist.\ rev\ xs\ ys \rightarrow xs = ys$ is

```
cexrev Xs Ys :-
   check (qgen (qheight 3)) (is_eltlist Xs), % generate
   solve (rev Xs Ys), not (Xs = Ys).         % test
```

From algebraic to binding signatures

- The proof-theoretic view allows us to move seamlessly from standard first-order terms to higher-order LP with λ-tree syntax, which was the whole selling point.
  - No current tool supports proofs **and** disproofs with binders
- This means accommodating the ∇-quantifier
- Here we take another shortcut and restrict to Horn specs (no hypothetical encodings).
  - ... but we have experimented with kernels for logics such LG as well
- It's well known that in this setting nabla can be soundly encoded by λProlog's universal quantification

Measurements for STLCI

bug	check	αC	λP	Description/Rating
1	preservation	0.3	0.05	range of function in app rule
	progress	0.1	0.02	matched to the arg. (S)
2	progress	0.27	0.06	value *(cons v) v* omitted (M)
3	preservation	0.04	0.01	order of types swapped
	progress	0.1	0.04	in function pos of app (S)
4	progress	t.o.	207.3	the type of cons return *int* (S)
5	preservation	t.o.	0.67	tail reduction returns the head (S)
6	progress	24.8	0.4	hd reduction on part. applied cons (
7	progress	1.04	0.1	no eval for argument of app (M)
8	preservation	0.02	0.01	lookup always returns int (U)
9	preservation	0.1	0.02	vars do not match in lookup (S)

- Mostly it's λProlog compiler kicking αCheck's ass
- Perhaps also the use *size* vs. α*Check*'s fixed *height*

Conclusions

- PBT is a great choice for meta-theory model checking.
- Checking specifications with αCheck is immediate – no reason not to spec'n'check on a regular basis
- Checking intermediate lemmas helps catch bugs earlier
- Spec and checks make great regression tests
- Our Haskell things suggests it's not clear cut which testing strategy performs better in which domain, but having a cascade of them is a big plus — see also Isabelle/HOL's plurality of tools
- Deeper bugs still in general out of grasp-

Future work: αCheck

- αCheck works surprisingly well, given the naivete of its implementation: basically an iterative deepening modification of the original OCaml interpreter for αProlog
- But recent experiments with encoding abstract machines (CICMark and IFC) reminds us of how powerless we are w.r.t. the combinatorial explosion
- Change the hard-wired notion of bound (# of clauses used) and how it is distributed over subgoals:
  - Take ideas from **Tor**
- Bring in some random-ness by doing random backchaining: flip a coin instead of doing chronological backtracking
- Prune the search space by not generating terms that exercise "equivalent" part of the spec

Future work: the blame game

- Suppose your PBT tool reports a cex. Now what? You're not getting payed just for finding faults. . .
- Staring at a potentially huge spec even with a cex in hand not the best way to go. Two issues:
  1. Soundness: your spec is plain wrong and returns an answer that should not hold
  2. Completeness: you've forgotten to encode some info and some answers are not produced.
- Proof-theory to the rescue (possibly):
  1. Use certificate distillation to restrict to a more manageable set of suspects – or just inspect the proof-term
  2. Use abduction (perhaps as an expert) to collect sets of assumptions that should hold but don't.

Future work: going sub-structural

- If you think FPC as glorified meta-interpreters in the logic programming sense, then why stick to *vanilla*?
- It's folklore that linear logical framework are well suited to encode object logic with imperative features, e.g. Pfenning and Cervesato's encoding of MLR in **LLF**;
- Data structures for heaps, stores. . . are replaced by **linear, affine, etc** predicates
  - This seems promising for **exhaustive** PBT, where every constructor counts
  - Work in progress: linear version of the list-machine benchmark
- Sub-structural PBT can bring some form of validation to frameworks such as **Celf**, whose meta-theory is not there yet
- Meta-interpreters are not viable in the long run:
  - give the αCheck treatment to languages such as **LolliMon**
  - use program specialization to do amalgamation
    - Initial experiments with **VeriMAP** to unfold **Tor's** disjunction

Future work: Meta Mutation Testing

- Mutation analysis seems a good way to evaluate your PBT tool:
  - generate mutants of your OL
  - compute the killing ratio
  - use model based PBT to estimate the equivalent mutant issue
- However, mutation testing should be **automatic**, rather than manual as in the literature (still looking at you PLT-Redex)
- Exception is **Mutabelle** in Isabelle/HOL, which weirdly mutates theorems not OL
- For Haskell, one can use **MuCheck**, yet:
  - What are sensible mutations operators for PL artifacts?
  - How to avoid re-writing a mutation tester for each PBT tool?
- Just write one for a meta-language such as **Ott/Lem** and extract mutations for target languages (Coq, Isabelle, OCaml)

Thanks!