Towards Substructural Property-Based Testing

Alberto Momigliano
joint work with Marco Mantovani

DI, Università degli Studi di Milano

CILC'21, September 9th, 2021

On the meta-theory of programming languages

- We study the *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.

- What can possibly go wrong?
  Finding and Understanding Bugs in C Compilers, [Yang '11]:
  "We created Csmith, a randomized test-case generation tool [. . .] we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input."

PL theory formalized

Programming language semanticists should be the obstetricians of programming languages, not their coroners. *(John C. Reynolds)*

- Formal verification via proof assistants gives the most guarantees and its doable, though labor-intensive
- See *CompCert* (compilers), *seL4* (operative system), etc.
- Elegance is not optional: success of a formalization may depend on the representation chosen.

Formalizing typing judgments

$$\Gamma \vdash e : \tau$$

- What is Γ? A (finite) function? A list/set/multiset?
- Any implementation of a typing context must support **lookup** and lemmas such as:

  Weakening If $\Gamma \vdash e : \tau$ and $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash e : \tau$.

  Substitution If $\Gamma, x : \tau \vdash e : \tau'$ and $\Gamma \vdash m : \tau$, then $\Gamma \vdash e[x/m] : \tau'$

- Since a typing system can have hundreds of rules, these proof are conceptually easy but practically hard.

Solutions

1. The hammer: use a concrete representation and develop libraries, tactics, automation to get it done
   - The standard approach in HOL, Coq, ACL2 ...
   - Low level, labor-intensive, hard to share among systems
2. Internalize those notions in the logic: an object level context is represented by a context in an intuitionistic meta logic:
   - $x_1 : \tau_1 \ldots x_n : \tau_n$ is encoded as a set of atoms $of(x_1, \ulcorner\tau_1\urcorner) \ldots of(x_n, \ulcorner\tau_n\urcorner)$.

$$\ulcorner\Gamma \vdash e : \tau\urcorner = \ulcorner\Gamma\urcorner \vdash_{Int} of(\ulcorner e\urcorner, \ulcorner\tau\urcorner)$$

   - Now, properties such as weakening and substitution come for free, since they hold once and for all for the meta-logic
   - This idea has been successfully used in systems like *Twelf, Beluga, Abella etc.*

Encoding state-based computations

- Consider the execution of a command in a imperative language:

$$\sigma \vdash c \Downarrow \sigma'$$

- Execution updates the state: seeing the state as an intuitionistic context won't work because the logic is monotonic.
- Sure, we can **reify** the state into a data structure, but it's back to the hammer thing.
- Rather, we refine the logic linearly
- We use linear logic to represent in a logical way stateful computation (Linear Logic Programming) and to reason over such computation from first principles (Linear Logic Framework), see *Lolli, Forum, LLF*. . .

Linear Logic and the meta-theory of PL

- Linearity plays a part in many PL phenomena, recently think separation logic and session types
- The promise: if you internalize the notion of state, formalizing the meta-theory should be a breeze
- Canonical example: Type soundness of ML with references [Cervesato & Pfenning'02], proven without any technical lemmas, as opposed to dozens (e.g., in Coq's Software Foundations).
- Alas, developing a linear proof environment from the ground up takes effort: canonical forms, resource management, unification etc.
  - In fact, there are none.
- In the meantime, rather then verification, we could do some **validation**, that is trying to find bugs.

Property-based testing

- A light-weight validation approach merging two well known ideas (*QuickCheck* [Claessen & Hughes ICFP'00]):
  1. automatic generation of test data, against
  2. executable program specifications.
- The programmer specifies in a small logical DSL properties that functions should satisfy.
- PBT tries to **falsify** the properties by trying a large number of (usually) **randomly** generated cases.
- Good interaction with proof assistants (Isabelle/HOL, Coq): testing not only *in lieu of* proving, but *before* proving.

PBT: the logical view

- Specifications (think the operational semantics of a PL) are logical theories: any fragment that has a focused proof-theory.
- Trying to refute a property of the form

$$\forall x\colon \tau\ [P(x) \supset Q(x)]$$

  means searching for a focused proof of

$$\exists x[(\tau(x) \wedge P(x)) \wedge \neg Q(x)$$

- Intuition: the positive part $\exists x(\tau(x) \wedge P(x))$ (generation under preconditions) is hard, the negative one $\neg Q(x)$ is just blind computation.
- A counterexample is a t s.t. $P(t)$ holds and $Q(t)$ does not.
- Rough bottom line: PBT as Prolog-like proof search.

What we have done

- Take a fragment of linear logic that has a logic programming interpretation – we use Miller & Hodas' **Lolli**, a linear logic programming that conservatively extends (fo) λProlog.
- Instrument it with a notion of certificate that will realize various data generation strategies – we have used Miller's **Foundational Proof Certificates** architecture to implement both random and exhaustive data generation.
- Obtain PBT as focused search for counterexamples
- We have carried out several case studies to assess the feasibility of validation of the meta-theory of linear specifications of PL-models via PBT ...
- ...and run some preliminary comparisons with "vanilla" PBT.

Case study: IMP and its compilation

- IMP is a minimalist Turing-complete model of a (typed) imperative PL.
- We have encoded its static and dynamic (both small and big step) semantics as linear logic programs, heavily relying on continuation-passing style to ensure adequacy.
- We have compiled IMP language in an assembly language, which runs over a stack machine.
- We have formulated and tested the meta-theory of IMP and its compilation via PBT both on the bug-free model and via some manual mutation analysis.
  Sample property: equivalence of small and big step execution of IMP:

$$\text{if } \sigma \vdash c \Downarrow \sigma_1 \text{ and } (c, \sigma) \rightsquigarrow^* (\texttt{SKIP}, \sigma_2) \text{ then } \sigma_1 \approx \sigma_2$$

- We have compared its performances w.r.t. a traditional state-passing encoding.

Operational semantics of IMP in Lolli

- First, we encode the state as a linear context:

$$\begin{array}{rcl} \sigma & ::= & \cdot \mid \sigma,\ x \mapsto v \\ \ulcorner \sigma, x \mapsto v \urcorner & = & \ulcorner \sigma \urcorner, var(x, \ulcorner v \urcorner) \end{array}$$

- Next, consider the rule for executing an assignment:

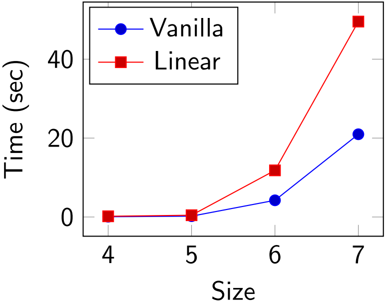
$$\frac{\sigma \vdash m \Downarrow v}{\sigma \vdash x := m \Downarrow \sigma \oplus \{x \mapsto v\}}$$

- and its encoding in Lolli via logical continuations:

```
ceval(asn(X,E),K)
    o- eval(E,V,
              (var(X,_) x (var(X,V) -o K))).
```

where `x`,`-o` are concrete syntax for lollipop and tensor.

Experimental evaluation of PBT queries



Linear vs state-passing testing of equivalence of big and small step operational semantics on a bug free model.

Conclusion

- While linear logic is heavily used to represent PL models, theorem proving technology is lacking behind.
- We propose PBT in linear logic as a way to at least validate those models.
- We have used tools from proof-theory (computations-as-deductions, focused proof search, certificates) to give such a road map.
- We have taken the first step with PBT'ing Lolli and validated the approach with a mid-sized case study.
- The empirical evaluation is not a washout, considering our setup.

Future work

- A less naive implementation
  - w.r.t. resource management and/or better data structures
  - Embed it into mainstream linear logic PL (which one?)
- Adapt some of the sophisticated generation strategies in the literature to search for deeper bugs.
- Get out of the '90 and deal with richer substructural logics (order, bunches, subexponentials, forward chaining . . .)
- Apply PBT to more challenging case studies (with binders).

Thanks for the attention