Run your Research, Mind the Binders

Alberto Momigliano
joint work with James Cheney, Edinburgh

DI, MIlano

March 18, 2014

The problem

- Syntactic proofs underlie much modern PL/logic...
  - type soundness
  - (strong) normalization/cut elimination
  - type preserving translations
- But, such proofs are notoriously fragile, one may say boring, and thus often "write-only", when not left to the reader

The problem

- Syntactic proofs underlie much modern PL/logic...
  - type soundness
  - (strong) normalization/cut elimination
  - type preserving translations
- But, such proofs are notoriously fragile, one may say boring, and thus often “write-only”, when not left to the reader
- Yeah. Right.

The problem

- Syntactic proofs underlie much modern PL/logic...
  - type soundness
  - (strong) normalization/cut elimination
  - type preserving translations
- But, such proofs are notoriously fragile, one may say boring, and thus often "write-only", when not left to the reader
- Yeah. Right.
- That's way we're in the business of *mechanized metatheorety*

Not quite there yet

- Problem: Verification is
  - lots of work!
  - unhelpful if system has a bug
  - only worthwhile if we already “know” the system is correct

Not quite there yet

- Problem: Verification is
  - lots of work!
  - unhelpful if system has a bug
  - only worthwhile if we already “know” the system is correct
- “model-checking” approach:
  - searches for counterexamples
  - produces helpful counterexamples for incorrect systems
  - unhelpfully diverges for correct systems
  - little expertise required
  - fully automatic, CPU-bound

Find the bug 1

- A “published” formalization in Abella of Milner & Tofte: *Co-Induction in Relational Semantics*. Look at the environment-based operational semantics

Find the bug

- A "published" formalization in Abella of Milner & Tofte: *Co-Induction in Relational Semantics*. Look at the environment-based operational semantics

```
Define eeval: venv -> tm -> val -> prop by
eeval (cons W K) one ;
eeval (cons W' K) (shift M) W :=  eeval K M W;
eeval K (abs M) (closure K (abs M));
eeval K (fix (abs M)) (clo c\ (closure (cons c K) (abs M)));
eeval K (app M N) W :=
          exists M' W',eeval K M (closure K (abs M')) /\
            eeval K N W' /\ eeval (cons W' K) M' W.
```

- There are (at least) two bugs here, but this did not stop the formalizer to go on and prove subject reduction.

Find the bug 2

```
...
eeval K (abs M) closure K (abs M) );
eeval K (fix (abs M)) (clo c\ (closure (cons c K) (abs M)));
eeval K (app M N) W :=
              exists M' W', eeval K M (closure K (abs M')) /\
              eeval K N W' /\ eeval (cons W' K) M' W.
```

- The app case is wrong. The first premise should be

  ```
  eeval K (app M N) W :=
          exists M' W' K', eeval K M (closure K' (abs M'))...
  ```

- More seriously, evaluation gets stuck if `M` is a fix point.
- And who is this jerk anyway?

Find the bug 2

```
...
eeval K (abs M) closure K (abs M) );
eeval K (fix (abs M)) (clo c\ (closure (cons c K) (abs M)));
eeval K (app M N) W :=
              exists M' W', eeval K M (closure K (abs M')) /\
              eeval K N W' /\ eeval (cons W' K) M' W.
```

- The app case is wrong. The first premise should be

  ```
  eeval K (app M N) W :=
          exists M' W' K', eeval K M (closure K' (abs M'))...
  ```
- More seriously, evaluation gets stuck if `M` is a fix point.
- And who is this jerk anyway? Oops. It's me.

Find the bug 2

```
...
eeval K (abs M) closure K (abs M) );
eeval K (fix (abs M)) (clo c\ (closure (cons c K) (abs M)));
eeval K (app M N) W :=
              exists M' W', eeval K M (closure K (abs M')) /\
              eeval K N W' /\ eeval (cons W' K) M' W.
```

- The app case is wrong. The first premise should be

  ```
  eeval K (app M N) W :=
          exists M' W' K', eeval K M (closure K' (abs M'))...
  ```
- More seriously, evaluation gets stuck if `M` is a fix point.
- And who is this jerk anyway? Oops. It's me.
- Brigitte found that out (the specs being bugged, clearly) while redoing the proof in Beluga
- But some kind of testing/counterexample search would have saved us the trouble

Find more bugs:

- $\lambda^{\rightarrow\times}$ typing

$$\frac{}{\Gamma \vdash () : \mathsf{unit}} \quad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \tau_1}$$

$$\begin{array}{rcl} (\lambda x.e)\ e' & \rightarrow & e[e'/x] \\ \pi_i(e_1, e_2) & \rightarrow & e_i \end{array}$$

- This version is intentionally full of bugs.

Find more bugss

- $\lambda^{\to\times}$ typing

$$\frac{}{\Gamma \vdash () : \mathsf{unit}} \quad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau} \quad \frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \tau_1}$$

- Claim: Trying to verify correctness is not the fastest way to find such bugs.

Example

- What is a "bug"?
- Here, I mean "counterexample to type soundness"
- Consider reduction step $\pi_2(1, ()) \to ()$
- Then we have

$$\dfrac{\dfrac{\overline{\cdot \vdash 1 : \mathsf{int}} \quad \overline{\cdot \vdash () : \mathsf{unit}}}{\cdot \vdash (1, ()) : \mathsf{int} \times \mathsf{unit}}}{\cdot \vdash \pi_2(1, ()) : \mathsf{int}}\ (*)$$

  But no derivation of

$$\cdot \vdash () : \mathsf{int}$$

- If only we had a way of systematically searching for such counterexamples...

Metatheory model-checking

- Goal: Catch "shallow" bugs in type systems, operational semantics, etc.
- (Finite) Model checking: attempt to verify finite system by searching exhaustively for counterexamples
  - Highly successful for validating hardware designs
  - Helpful in the (common) case that system has bug
- Partial model checking: search for counterexamples over some finite subset of infinite search space
  - Produce a counterexample if one exists
  - Diverges if system is correct

Idea

- Represent object system in a suitable meta-system (logical framework).
- Specify properties it should have.
- System searches (exhaustively/randomly) for counterexamples.
- Meanwhile, user can try a direct proof (or do something more fun)

Pros

- Finds shallow counterexamples quickly
- Separates concerns (metatheory researchers focus on efficiency, type system designers focus on semantics)
- Easy to use; don't need to buy into/learn theorem prover
- Easy to implement (the naive solution)
  - Many not-so-naive refinements possible

Cons

- Failure to find counterexample does not guarantee property holds
- i.e., not doing *automatic verification*
- Hard to tell what kinds of counterexamples might be missed
- “Deep” bugs that were discovered in published type systems (e.g. polymorphism vs. references) currently beyond scope

Redex

- Robbie Findler and co. took on this idea and marketed as *Randomized testing for PLT Redex* (`http://redex.racket-lang.org/`)

  > *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 animation plus *QuickCheck* for operational semantics, but in Scheme!!!! No abstraction mechanisms.

Redex

- Robbie Findler and co. took on this idea and marketed as *Randomized testing for PLT Redex* (`http://redex.racket-lang.org/`)

  > *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 animation plus *QuickCheck* for operational semantics, but in Scheme!!!! No abstraction mechanisms.
- **Proofs** you say? Forget about it

- Handles suites of unit tests (via animation)
- Nice facilities for OTT-style paper-to-code translation of specs
- Latex from specs, which is handy.
- Random testing of specs.
- 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? type setting rules, some rules missing, some unexpected behaviour of the model, one false theorem (but fixable)

What Robbie does not tell you (in his talk)

- Redex offers **no** support for what we care about: binders

  > *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,*

- Judgments

  > *Redex lacks direct support for non-algorithmic relations such as declarative typing rules . . . we were forced to escape to Redexs host language or to adopt an elaborate encoding, which we would not expect a casual user to be comfortable with. Extending Redex with support for logic programming. . .*

- Test coverage

  > *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*

Other related work

- Programming
  - QuickCheck for Haskell [Claessen and Hughes 2000] provides type class libraries for generator functions that randomly construct test data, and a logical specification language to describe the properties the program should satisfy.
  - Smallcheck [Christiansen and Sebastian Fischer 2008] the same but exhaustive (up to *n*) counterexample search
- Proving
  - IsaQuickcheck combines Isabelles code generation infrastructure with random testing. It analyses the definitions of inductively defined predicates to generate values that satisfies them by construction
  - Nitpick: systematic model enumeration using a SAT solver.

Our approach

- Represent object system in a suitable meta-system.
  - James has implemented it for αProlog programs, and so it also applicable to Nominal Isabelle
  - The idea applies as well to λProlog/Twelf.
- Specify property it should have.
  - Universal Horn formulas can specify type preservation, progress, soundness, weakening, substitution lemmas, etc.
- System searches exhaustively for counterexamples.
  - Bounded depth-first search, negation as failure/negation elimination

Example

- αProlog: a simple extension of Prolog with nominal abstract syntax.

$$var : name \to exp. \quad app : (exp, exp) \to exp. \quad lam : \langle name \rangle exp \to exp.$$

$$\begin{array}{lcl}
tc(G, var(X), T) & :- & List.mem((X, T), G). \\
tc(G, app(M, N), U) & :- & \exists T.tc(G, M, arr(T, U)), tc(G, N, T). \\
tc(G, lam(\langle \mathrm{x} \rangle M), arr(T, U)) & :- & \mathrm{x} \# G, tc([(\mathrm{x}, T)|G], M, U).
\end{array}$$

$$\begin{array}{lcl}
sub(var(X), X, N) & = & N. \\
sub(var(X), Y, N) & = & var(X) :- X \# Y. \\
sub(app(M_1, M_2), Y, N) & = & app(sub(M_1, Y, N), sub(M_2, Y, N)). \\
sub(lam(\langle \mathrm{x} \rangle M), Y, N) & = & lam(\langle \mathrm{x} \rangle sub(M, Y, N)) :- \mathrm{x} \# (Y, N).
\end{array}$$

- Equality coincides with $\equiv_\alpha$, $\#$ means “not free in”, $\langle \mathrm{x} \rangle M$ is an M with x bound.

Problem definition

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

$$\forall \vec{X}. B_1 \wedge \cdots \wedge B_n \supset A$$

  (note: disjunctive, existential A, B_i possible by adding clauses)
- A *counterexample* is a ground substitution θ such that

$$\mathcal{M} \vDash \theta(G_1) \wedge \cdots \wedge \mathcal{M} \vDash \theta(G_n) \wedge \mathcal{M} \nvDash \theta(A)$$

- The *partial model checking problem*: Does a counterexample exist? If so, construct one.
- Obviously undecidable

Implementation

- Naive idea: generate substitutions and test; iterative deepening.
- Write "generator" predicates for all base types.
- For all combinations, see if hypotheses succeed while conclusion fails.

$$?\!-\; gen(X_1) \wedge \cdots \wedge gen(X_n) \wedge G_1 \wedge \cdots \wedge G_n \wedge not(A)$$

- Problem: extremely high branching factor
  - even if we abstract away infinite base types
- Can only check up to max depth 1-3 before boredom sets in.

Implementation (II)

- Fact: Searching for instantiations of variables first is wasteful.
- Want to delay this expensive step as long as possible.
- Less naive idea: generate *derivations* and test.
- Search for complete proof trees of all hypotheses
- Instantiate all remaining variables
- Then, see if conclusion fails.

$$?\!-\ G_1 \wedge \cdots \wedge G_n \wedge gen(X_1) \wedge \cdots \wedge gen(X_n) \wedge not(A)$$

- Raises boredom horizon to depths 5-10 or so.

Implementation (III)

- Negation-as-failure is messy; can we do better?
- Idea: Use *negation elimination* instead
  - AKA/similar to "constructive negation", "intensional 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:

$$?\!-\ G_1 \wedge \cdots \wedge G_n \wedge not\_A \wedge G_{n+1} \wedge \cdots$$

- Implemented this also; finds some counterexamples faster
- Details in paper.

Negation elimination example

- Negation-eliminated versions of *tc*, *subst*: (after manual simplification):

$$\begin{array}{lcl}
not\_tc(G, var(X), T) & :- & not\_mem(G, X, T). \\
not\_tc(G, app(M, N), U) & :- & \forall^* T.(not\_tc(G, M, arr(T, U)); \\
 & & \qquad not\_tc(G, N, T)). \\
not\_tc(G, lam(\langle \mathrm{x} \rangle M), arr(T, U)) & :- & \mathrm{x} \# G, not\_tc([(\mathrm{x}, T)|G], M, U). \\
 & & \\
not\_subst(M, X, N, M') & :- & neq_{exp}(subst(M, X, N), M').
\end{array}$$

- Note: Need a "case-unfolding" universal quantifier $\forall^*$
- Current implementation of $\forall^*$ examines type info at runtime
  - Many unexploited optimization opportunities
- Note: Need *neq* at each type; we defined generically

Demo

- Debugging simply-typed lambda calculus spec.

Experience

- Implemented within αProlog
- Checked a bunch of examples from TAPL, LF type-checking algorithm, the “faulty” λ-calculus
- We’re in the process of redoing some of Findler et co. *Run your research* case studies
- Ideally such a facility should work atomatically in the background when once you’re written up a model and its properties (as it happens with IsaQuickCheck)

Experience (II)

- Writing specifications is dirt simple
  - They make great regression tests
  - No reason not to write & check on a regular basis
- Order of goals makes a big difference to efficiency; optimization principles not clear yet.
- Not enough to just check "main" theorems
  - System could be "trivially" sound
  - Checking intermediate lemmas helps catch bugs earlier
- Bounded DFS and negation elimination have other applications, so good to have anyway
  - "show me all derivations of height ≤ 2"
  - coverage analysis

Reality check

- Can we find known, deep type system bugs? Not yet...
- Naive Mini-ML with ref/$\forall$ bug:
  - boredom horizon 9
  - smallest counterexample I can think of needs depth 18
  - usual counterexample needs depth 30.
- So at this point, won't catch "deep" bugs
- But useful for eliminating "shallow" bugs during development of type system

Conclusions

- Model checking/counterexample search techniques are useful for catching shallow bugs
- Complements, but doesn't replace proof/verification
- *Negation elimination* improves efficiency/coverage; hope to make more efficient too
- Many other refinements (heuristics?) possible
- Checker implemented in αProlog 1 1.43 (not the on-line version).