Mechanized Metatheory Model-Checking

WMM 2006

James Cheney

9/21/06

Mechanized (partial) Metatheory Model-Checking

WMM 2006

James Cheney

9/21/06

A thought experiment

- Let's say, for whatever reason, you've been imprisoned in cell with an IBM PCjr connected to a candy machine and a poison machine.
- Alice, of cryptography fame, slips under the door a language reference manual together with a formal proof (in your favorite system) that the language is "safe"
- meaning; when run, no program crashes (thereby activating the poison machine).
- However, Alice also advises you that the language has never been run or tested. You can't do a "dry run".
- Your task: program the machine to produce candy so you don't starve, while also avoiding poisoning.
- What do you do? Assume you have infinite coffee, whiteboards, reference manuals, etc.

Experimental type theory — an oxymoron

- Any current verification approach introduces a “gap” between formally verified language and implemented version.
- Type systems are theories of programming language behavior.
- Testing theories against reality by attempting falsification and independent confirmation is a basic scientific principle.
- Though weaker than formal verification of “real” system, rigorous testing complements informal verification (or verification of abstract system).

Find the bug

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

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

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

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

Find the bugs

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

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

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \color{red}{\tau'}}{\Gamma \vdash e_1\ e_2 : \color{red}{\tau}} \ (*) \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$

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

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

Find the bugs, reloaded

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

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

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}\ (*) \qquad \frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}\ ({*}{*})$$

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

- Claim: Trying to verify correctness is not the fastest way to find such bugs.
- Also, it is dangerous to intentionally add errors to an example; it keeps you from looking for the unintentional ones.

Example

- Consider reduction step $\pi_2(1, ()) \to ()$
- Then we have

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

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.
- Model checking: attempt to verify finite system by searching exhaustively for counterexamples
  - Highly successful for validating hardware designs
  - More helpful in (common) case that system has bug
- Partial model checking: search for counterexamples over some finite subset of infinite search space
  - Produces a counterexample if one exists, but cannot verify system correct

Pros

- Finds shallow counterexamples quickly
- Separates concerns (researchers focus on efficiency, engineers focus on real work)
- Lifts user's brain out of inner loop
- Easy to use; theorem prover expertise/Kool-Aid^{TM} not required
- Easy to implement naive solution
- (Buzzword-compatible? Guilty as charged)

Cons

- Failure to find counterexample does not guarantee property holds
- Hard to tell what kinds of counterexamples might be missed
- "Nontrivial" bugs (e.g. $\forall/\mathsf{ref}$, $\leq/\mathsf{ref}$) currently beyond scope

Idea

- Represent object system in a suitable meta-system.
- Specify property it should have.
- System searches exhaustively for counterexamples.
- Meanwhile, you try to prove properties (or get coffee, sleep, whatever).

Realization

- Represent object system in a suitable meta-system.
  - I will use pure αProlog programs (but many other possibilities)
- Specify property it should have.
  - Universal Horn (Π_1) formulas can specify type preservation, progress, soundness, weakening, substitution lemmas, etc.
- System searches exhaustively for counterexamples.
  - Bounded DFS, negation as failure
- Meanwhile, you try to prove properties (or get coffee, sleep, whatever).
  - My office has an excellent coffee machine.

The “code” slide

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

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

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

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

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

Problem definition

- Define model M using a (pure) logic program P.
- Consider specifications of the form

$$\forall \vec{X}. G_1 \wedge \cdots \wedge G_n \supset A$$

- A *counterexample* is a ground substitution θ such that

$$M \models \theta(G_1) \wedge \cdots \wedge M \models \theta(G_n) \wedge M \not\models \theta(A)$$

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

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: 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.

Demo

- Debugging simply-typed lambda calculus spec.

Experience

- Implemented within αProlog; more or less a hack...
- Checked $\lambda^{\rightarrow\times}$ example, up to type soundness
- Checked syntactic properties (lemmas 3.2-3.5) from [Harper & Pfenning TOCL 2005]
  - NB: Found typo in preprint of HP05, but it was already corrected in journal version
- Since then, have implemented and checked Ch. 8, 9, some of Ch. 11 of TAPL too
- NB: Published, high-quality type systems are probably not the most interesting test cases...

Experience (II)

- Writing Π_1 specifications is dirt simple
  - They make great regression tests
  - I now write them as a matter of course
- Order of goals makes a big difference to efficiency; optimization principles not clear yet.
- Not enough to check “main” theorems
- Checking intermediate lemmas helps catch bugs earlier
- Bounded DFS also useful for exploration, “yes, $\neg\phi$ can happen”

Is this trivial?

- Tried a few “realistic” examples recently
- λ_{zap}: checked lemmas 2–6 up to depth 7–8; two faults break type pres at depth 10
- Naive Mini-ML with references: boredom horizon 9; smallest counterexample I can think of needs depth 18.
  - Back of envelope estimate: would need somewhere between 191 and 4.4 million years to find
  - I guess I need a faster laptop.
  - Bright side: blind search massively parallelizable...
- At this point, probably trivial; won’t catch any “real” bugs in finished products.
- But perhaps useful during development of type system

Better ideas

- There are many smarter things one could try.
- Random search?
- Random abstract interpretation $\rightarrow$ finite model checking?
- Better resource bounding?
- Modes and other optimizations?
- Negation elimination?
- Richer constraints (finite maps, substitution)?
- Same idea, different framework?

Random interpretation

- Fact: Π_1 formula ϕ valid $\iff$ true in all models $\implies \phi$ true in a finite, random model
- Hence, if ϕ fails in a random model then ϕ is invalid.
- Idea: Generate a finite interpretation A randomly
- Compute model P^A of P in A via finite lfp iteration
- Check ϕ in P^A.
- If ϕ fails, search for a “real” counterexample, hopefully using counterexample to $P^A \vDash \phi$ as guide

Negation elimination

- Using negation as finite failure is tricky
  - need to make sure all variables are instantiated properly.
  - can't delay expensive steps past negated subgoals
- Idea: Use negation elimination to avoid NFF?

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

- Have been talking to Alberto Momigliano about this...
- initial manual-negation-elimination experiments seem promising...

Conclusions

- Model checking/counterexample search techniques are useful for catching shallow bugs
- Improvement needed to improve coverage
- Many refinements possible
- Checker implemented in αProlog; will be in next release