

Mechanized Metatheory Model-Checking

Fun in the Afternoon

James Cheney

November 16, 2006

Mechanized **(partial)** Metatheory Model-Checking

Fun in the Afternoon

James Cheney

November 16, 2006

Background

- *Type systems* are a powerful techniques for verifying properties of programs (e.g. memory safety)
 - Provides guarantee that all programs in language have property
 - To stay decidable, some “safe” programs must be disallowed
- Much research in PL of the form “design a type system/program analysis to enforce P ” (recently, P often a security property)
- Problem: How to **specify and verify** type systems?

Example

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

$$\frac{}{\Gamma \vdash () : \text{unit}} \quad \frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau}$$
$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$
$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \tau_2}$$

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

- Claim: This version is full of bugs.

Metatheory verification

- Current state of practice:
 1. write down typing rules, operational semantics
 2. try to prove syntactic properties, culminating in soundness
 3. if proof fails, goto 1.
- Step 2 tedious & sensitive to changes, so tempting to “handwave”
 - Especially hours before paper deadline (I’m certainly guilty of this)
- But this is dangerous (ML \forall + ref bug, Java array subtyping bug, Cyclone \exists + ref bug)

Mechanized metatheory verification

- Computers should be doing most of the work of verification.
- Recent interest in making metatheory verification tools “ready for prime-time” (POPLMark Challenge)
- Long-term research program on metatheory verification at CMU using higher order abstract syntax & LF
 - Starting to bear fruit (e.g. POPL 2007)
 - Probably the most practical approach
 - But still a lot of work to learn
- Several other syntax encodings (de Bruijn, nominal) and theorem provers also considered (Coq, HOL, Isabelle/HOL)

Find the bug

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

$$\frac{}{\Gamma \vdash () : \text{unit}} \quad \frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau}$$
$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$
$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \tau_2}$$

Find the bugs

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

$$\frac{}{\Gamma \vdash () : \text{unit}} \quad \frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau}$$
$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \underline{\tau'}}{\Gamma \vdash e_1 e_2 : \underline{\tau}} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$
$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \underline{\tau_1}}$$

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

Find the bugs, reloaded

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

$$\frac{}{\Gamma \vdash () : \text{unit}} \quad \frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau}$$
$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \underline{\tau'}}{\Gamma \vdash e_1 e_2 : \underline{\tau}} \quad \frac{\Gamma, \underline{x:\tau} \vdash e : \underline{\tau}}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$
$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \underline{\tau_1}}$$

- 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 $\pi_2(1, ()) \rightarrow ()$
- Then we have

$$\frac{\frac{\cdot \vdash 1 : \text{int} \quad \cdot \vdash () : \text{unit}}{\cdot \vdash (1, ()) : \text{int} \times \text{unit}}}{\cdot \vdash \pi_2(1, ()) : \text{int}} \quad (*)$$

But no derivation of

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

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

Experimental metatheory?!

- 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).

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™ 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 /ref, \leq /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 α Prolog 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).

The “code” slide

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

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

$tc(G, var(X), T) \quad :- \quad List.mem((X, T), G).$

$tc(G, app(M, N), U) \quad :- \quad \exists T. tc(G, M, arr(T, U)), tc(G, N, T).$

$tc(G, lam(\langle x \rangle M), arr(T, U)) \quad :- \quad x \# G, tc([\langle x, T \rangle | G], M, U).$

$sub(var(X), X, N) \quad = \quad N.$

$sub(var(X), Y, N) \quad = \quad var(X) :- X \# Y.$

$sub(app(M_1, M_2), Y, N) \quad = \quad app(sub(M_1, Y, N), sub(M_2, Y, N)).$

$sub(lam(\langle x \rangle M), Y, N) \quad = \quad lam(\langle x \rangle sub(M, Y, N)) :- x \# (Y, N).$

- Equality coincides with \equiv_α , $\#$ means “not free in”, $\langle x \rangle M$ is an M with x bound.

Problem definition

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

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

(note: disjunctive, existential A, B_i possible by adding clauses)

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

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

- 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.

$$?- \text{gen}(X_1) \wedge \dots \wedge \text{gen}(X_n) \wedge G_1 \wedge \dots \wedge G_n \wedge \text{not}(A)$$

- 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.

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

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

Demo

- Debugging simply-typed lambda calculus spec.

Experience

- Implemented within α Prolog; more or less a hack...
- Checked $\lambda^{\rightarrow \times}$ example, up to type soundness
- Checked some syntactic properties of an LF typechecking algorithm
- 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 just check “main” theorems
 - System could be “trivially” sound
 - 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
- 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, 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 → finite model checking?
- Better resource bounding?
- Modes and other optimizations?
- Negation elimination?
- Richer constraints (finite maps, substitution)?
- Analysis to identify rules which “cause” bug?
- Same idea, different framework?

Conclusions

- Simplistic model checking/counterexample search techniques are useful for catching shallow bugs
- Improvement needed to increase coverage
- Many refinements possible
- Checker implemented in α Prolog; will be in next release