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

\[ \lambda \rightarrow : \text{typing} \]

\[
\begin{align*}
\Gamma \vdash () : \text{unit} & \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \\
\Gamma \vdash e_1 : \tau \rightarrow \tau' & \quad \Gamma \vdash e_2 : \tau' \\
& \quad \frac{\Gamma \vdash e_1 \ e_2 : \tau}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'} \\
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
& \quad \frac{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1} \\
& \quad \frac{\Gamma \vdash \pi_2(e) : \tau_1}{\Gamma \vdash e : \tau_1 \times \tau_2}
\end{align*}
\]
Find the bugs

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

\[
\begin{align*}
& \frac{}{\Gamma \vdash () : \text{unit}} \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \\
& \frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \; e_2 : \tau} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'} \\
& \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \tau_1}
\end{align*}
\]

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

\[ \lambda \rightarrow \times \text{ typing} \]

\[
\begin{array}{c}
\Gamma \vdash () : \text{unit} \\
\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau' \\
\Gamma \vdash e_1 \, e_2 : \tau \\
\hline
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}
\end{array}
\]

(\frac{x : \tau \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'} (**) )

\[
\Gamma \vdash e_1 : \tau_1 \\
\Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash e : \tau_1 \times \tau_2 \\
\Gamma \vdash \pi_1(e) : \tau_1 \\
\Gamma \vdash \pi_2(e) : \tau_1
\]

(\frac{\ast}{\ast} )

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

\[
\begin{align*}
\cdot &\vdash 1 : \text{int} & \cdot &\vdash () : \text{unit} \\
\cdot &\vdash (1, ()) : \text{int} \times \text{unit} \\
\cdot &\vdash \pi_2(1, ()) : \text{int} \quad (\star)
\end{align*}
\]

But no derivation of

\[
\cdot \vdash () : \text{int}
\]

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™ 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. ∀/ref, ≤ /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 $\alpha$Prolog programs (but many other possibilities)
- Specify property it should have.
  - Universal Horn ($\Pi_1$) 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.
α-Prolog: a simple extension of Prolog with nominal abstract syntax.

\[
\begin{align*}
\text{var} & : \text{name} \to \text{exp}. \quad \text{app} : (\text{exp}, \text{exp}) \to \text{exp}. \quad \text{lam} : \langle \text{name}\rangle \text{exp} \to \text{exp}.
\end{align*}
\]

\[
\begin{align*}
tc(G, \text{var}X, T) & : = \text{List.mem}((X, T), G).
\text{tc}(G, \text{app}(M, N), U) & : = \exists T. \text{tc}(G, M, \text{arr}(T, U)), \text{tc}(G, N, T).
\text{tc}(G, \text{lam}(\langle x \rangle M), \text{arr}(T, U)) & : = x \# T, \text{tc}([\langle x, T \rangle | G], M, U).
\end{align*}
\]

\[
\begin{align*}
\text{sub}(\text{var}(X), X, N) & = N.
\text{sub}(\text{var}(X), Y, N) & = \text{var}(Y) : = X \# Y.
\text{sub}(\text{app}(M_1, M_2), Y, N) & = \text{app}(\text{sub}(M_1, Y, N), \text{sub}(M_2, Y, N)).
\text{sub}(\text{lam}(\langle x \rangle M), Y, N) & = \text{lam}(\langle x \rangle \text{sub}(M, Y, N)) : = x \# (Y, N).
\end{align*}
\]

Equality coincides with \(\equiv_\alpha\), \(\#\) 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

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

- A counterexample is a ground substitution $\theta$ such that

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

- 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.

\[
\text{\texttt{?- \ gen(X_1) \land \cdots \land \ gen(X_n) \land G_1 \land \cdots \land G_n \land \neg (A)}}
\]

- 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.

?– \( G_1 \land \cdots \land G_n \land gen(X_1) \land \cdots \land gen(X_n) \land not(A) \)

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

Debugging simply-typed lambda calculus spec.
Experience

- Implemented within $\alpha$Prolog; 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 $\Pi_1$ 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
- $\lambda_{\text{zap}}$: 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: $\Pi_1$ formula $\phi$ valid $\iff$ true in all models $\implies \phi$ true in a finite, random model

Hence, if $\phi$ fails in a random model then $\phi$ is invalid.

Idea: Generate a finite interpretation $A$ randomly

Compute model $P^A$ of $P$ in $A$ via finite lfp iteration

Check $\phi$ in $P^A$.

If $\phi$ fails, search for a “real” counterexample, hopefully using counterexample to $P^A \models \phi$ 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?

\[ \neg A \land \text{gen}(X_1) \land \cdots \land \text{gen}(X_n) \]

- 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 $\alpha$Prolog; will be in next release