Type Theory meets Effects

Greg Morrisett
A Famous Phrase:

“Well typed programs won’t go wrong.”

1. Describe abstract machine: \( M ::= \langle \sigma, c \rangle \)

2. Give transition relation: \( M_1 \Rightarrow M_2 \)
   \( \langle \sigma, x:=42; c \rangle \Rightarrow \langle \sigma \{ x \rightarrow 42 \}, c \rangle \)
   \( \langle \sigma, \text{if true then } c_1 \text{ else } c_2 \rangle \Rightarrow \langle \sigma, c_1 \rangle \)

3. Classify all terminal states as “bad” or “good”
   good: \( \langle \sigma, 42 + 10 \rangle, \langle \sigma, \text{if true then } 43 \text{ else } 21 \rangle \)
   bad: \( \langle \sigma, \text{if } 42 \text{ then } e_1 \text{ else } e_2 \rangle, \langle \sigma, \text{“Bob” / true} \rangle \)

What’s “good” and “bad”?

- I could say \(<\sigma, "Bob" / true> \Rightarrow <\sigma, 42>\).

- I could say \(<\sigma, \text{exit}(0)>\) is “bad”.

- It’s up to you! (Or rather, it should be...) 

- But of course, for even simple safety policies, \textit{statically} proving a program (much less a language) won’t “go wrong” is pretty challenging.
Thus, we cheat:

• For languages (Java, C#, Scheme...):
  – We add some artificial transitions:
    
    \[
    <\sigma, 42 / 0> \Rightarrow <\sigma, \text{throw(DivByZero)}>
    \]
  – and then label some bad states as good:
    
    \[
    <\sigma, \text{throw(v)}>
    \]

• Other examples:
  – Null pointer dereference, array index out of bounds, bad
downcast, stack inspection error, file already closed,
deadlock, ...

• So the reality is that today, well-typed
  programs don’t \textit{continue} to go wrong.
  – Better than a code injection attack.
  – But little comfort when your airplane crashes.
Exceptions

- The escape hatch for typing:
  \[
  \text{throw} : \forall \alpha. \text{exn} \rightarrow \alpha
  \]

- In languages such as ML & Haskell, they don’t appear in interfaces:
  - \text{div} : \text{int} \rightarrow \text{int} \rightarrow \text{int}
  - \text{sub} : \forall \alpha. \text{array } \alpha \rightarrow \text{int} \rightarrow \alpha

- In Java & C# we have throws clauses:
  - \text{div} : \text{int} \rightarrow \text{int} \rightarrow \text{int}
    \[
    \text{throws DivByZero}
    \]
Problems with Throws:

- Need effect polymorphism:
  - `map : ∀α,β.(α→β) → list α → list β`
  - `map div vs. map sub`
  - `map : ∀α,β,σ.(α→β throws σ) → list α → list β throws σ`

- Need flow/path sensitivity:
  ```plaintext
  if (n != 0) avg := div(sum,n);
  else avg := 0;
  ```
What We Really Want:

• Refinements:
  - $\text{div} : \text{int} \to (\text{y: int}) \to \text{int}$ requires $\text{y} \neq 0$
  - $\text{sub} : \forall \alpha. (\text{x: array } \alpha) \to (\text{i: int}) \to \alpha$
    requires $\text{i} \geq 0 \&\& \text{i} < \text{size(x)}$
  - $\text{csub} : \forall \alpha. (\text{x: array } \alpha) \to (\text{i: int}) \to \alpha$
    throws $\text{BoundsError}$ when
    $\text{i} < 0 \mid \mid \text{i} \geq \text{size(x)}$

• And even:
  - $\text{printf} : (\text{x: string}) \to (\text{vs: list obj}) \to \text{unit}$
    requires $(\exists \text{ts, parses(x,ts) \&\& have_types(vs,ts))}$
  - $\text{prove} : (\text{p: prop}) \to (\text{b: bool})$
    ensures $(\text{b} = \text{true} \Rightarrow \text{p})$
  - $\text{compile} : (\text{x: ast}) \to (\text{y: x86})$
    ensures $(\text{bisimilar(x,y)})$
Static EXtended Checking

ESC/Java, Spec#, Cyclone, Deputy, Sage, ...  
• Take existing languages (Java, C#, C).  
• Aimed at eliminating language bugs:    
  – null pointers, array bounds, downcasts, ...  
• Augment types with pre/post-conditions.  
• Calculate refinements at each program point.  
  – use weakest-pre or strongest-post-conditions  
  – in conjunction with some abstract interpretation techniques to generate loop invariants  
• Use SMT prover to check pre/post-conditions.
Tremendous Progress

• Some key abstraction patterns
  – e.g., object invariants, ownership/confinement

• Much improvement in provers:
  – SMT provers integrate decision procedures
  – Advances with SAT, BDDs, ILPs, ...

• Improved invariant finders:
  – e.g., polyhedral domains
  – counter-example guided refinement

For 70 Kloc in the Cyclone compiler, discharge 95% of the null & array bounds checks.
Reality: Static EXtended Checking

- Still too many false positives:
  - Still have 1000 checks left in Cyclone compiler
  - And this is for shallow verification conditions
  - Programmers will dismiss false positives

- Many Culprits:
  - Language of specifications is too weak
  - Calculated invariants are too weak
  - Theorem provers are too weak
  - Memory, aliasing, framing (more on this later)

- Seems hopeless, no?
Why not give programmers the ability to work around short-comings of automation?

- Magic is good as long as it doesn’t prevent you from getting real work done...
- Languages shouldn’t be designed around what we can automate today, but rather, based on what we want to say tomorrow.

So give programmers a way to build explicit proofs within the language.

- if automation can’t find proof, at least programmer can try to construct one.

Not a new idea: this is the essence of type theory!
How Does All This Scale?

X. Leroy [PoPL ’06]: correct, optimizing compiler from C to PowerPC:

- Build interpreter for C code.
- Build interpreter for PowerPC code.
- compile: \( S \rightarrow (T, \text{Cinterp}(S) \approx \text{PPCinterp}(T)) \)
  - compiler comparable to good ugrad class
    - CSE, constant prop, register allocation, trace scheduling ...
  - decomposed into series of intermediate stages
  - as much certifying compiler as certified compiler

- Coq extracts Ocaml code by erasing proofs
  - not just modeling code and proving model correct.

Bottom line: it’s feasible to build *mechanically* verified software using this kind of approach.
Great Progress, but…

• 4,000 line compiler:
  – 7,000 lines of lemmas and theorems
    • includes interpreters/models of C and PPC code
    • much is re-usable in other contexts
  – 17,000 lines of proof scripts

• Many research opportunities here:
  – Advances in SMT provers not yet adopted.
  – Can we maintain proofs when code changes?
    • Proof scripts (a la Coq) are unreadable though smaller & less sensitive to change than explicit proofs.
    • Explicit proofs (a la Twelf) are bigger, but perhaps force better abstraction, readability, & maintainability.
Another Big Problem:

Systems like Coq (and ACL2, Isabelle/HOL, etc.) are limited to pure, total functions:

– no hash tables, union-find, splay trees, ...
  • So Xavier is forced to use functional data structures
  • Not a bad thing per se, but we should be able to get good algorithmic complexity where needed (e.g., unification.)

– no I/O, no exceptions, no diverging computations, no concurrency, ...
  • So building a server in Coq is out of the question.

Note: you can *model* these things in Coq.

– but then you have the model/code disconnect.
Why Only Total Functions?

At all costs, there should be no (closed) term of type False.

- i.e., there should be no proof of False.
- In ML: `fun bot()=bot() : ∀α.unit→α`
- If we can code `bot` in Coq:
  `bot() : False`
- Note that other things, including state, concurrency, continuations, can lead to the same sort of problems.
A Solution: Monads

As in Haskell, distinguish purity with types:

- **e : int**
  - e is equivalent to an integer *value*
- **e : ♦ int**
  - e is a *delayed computation* which when run in a world w either diverges, or yields an int and some new world w'.
  - Because computations are delayed, they are pure.
  - So we can safely manipulate them within types and proofs.
- **e : ♦ False**
  - possible, but means e must diverge when run!
Reasoning with ♦:

By *refining* ♦ with predicates, we can capture the effects of an imperative computation within its type.

\[ e : \diamond \{P\} x:\text{int} \{Q\} \]

When run in a world satisfying \( P \), \( e \) either

– diverges, or else

– terminates with an integer \( x \) and world satisfying \( Q \).

*i.e.*, Hoare-logic meets Type Theory
The Rest of My Bit...

- Building a (functional) type-\textit{inference} procedure for simply-typed lambda calculus.
  - uses dependent and refinement types in an interesting way
  - emphasize the “Chlipala-style” for proof development in Coq

- Hoare Type Theory
  - the basic ST monad in Coq
  - separation logic and the STsep monad
  - building and verifying (mutable) ADTs
  - concurrency and separation (time permitting)