Formal Verification of Monadic Computations

Steve Zdancewic

Reasoning about monadic programs in Coq

Imp

Syntax + inductive defs in Coq

* Can step through the defs step by step in literate Coq

\( \{ \ldots \} \Rightarrow \{ \ldots 3 \} \) program equivalence

Tactics:

\texttt{cbn.} : call by name

Q: How much can be automated by using advanced tactics?

Yes

Q: Can we run the program in Coq?

How we represent the program \( \Rightarrow \) has trade offs

Records on

Imp can be run.
Monov mem: String → mem

(_ !→ 0) all- erot map

Fixpoint: recursive fns.

Booleans and number exprs are pure in Imp.

Problem: eval for commands does not necessarily terminate (bc of while).

It's not total!

→ specify using relations instead.

st = [cmd] → st' step semantics.

In Coq:

Inductive eval : com → mem → mem → Prop

Q: Diff between Inductive and Fixpoint?

defines datatype inductively defined function must be total!

Drawbacks of large-step: need to do induction on derivations instead of syntax.

Q: Can we do small-step?

can be made to work, but need to deal with indices everywhere, diff design space.
MONADS: "well-behaved sequential composition"

Large-step: need explicit plumbing for intermediate states.

\[ st = [s_0] \Rightarrow st' = [s_1] \Rightarrow st'' \]

Fixpoint:
\[
\begin{align*}
  \text{let } \delta' &= \ \cdots \ st' \in \\
  \text{let } \delta &= \ \cdots \ st \in \\
  \text{plumbing!}
\end{align*}
\]

pure in Imp.

does not necessarily terminate.
(\because \text{ of while})
It's not total!

read.

step semantics.

mem \rightarrow \text{Prop}
and Fixpoint. ?

\text{inductively defined function}
\uparrow
\text{must be total!}

do induction on derivations of by syntax.

Q: Is there any reason to implement \text{ret} as a notation instead of a function?

Yes, not all "ret" are \text{ret} functions.

Q: Monad laws?

Unbundle proof components request practice in Coq.
Q. Is monad bad for modelling concurrency?
over-serialized computation, so yes
might want to relax
but still useful in the sequential part.

let is associative.

"let-hoisting", "let-lifting" & also called.

In order to prove this law, also need to prove that capture avoidance
is handled correctly.

With Monad laws, can prove program equivalence.

Tomorrow: what does this "=" mean?
functional extensionality & not necessarily the right way.

Define equivalence over behaviors.