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

<{ ... }> →
parse these as Imp code.

<{ ... }> ≅ <{ ... }>   program equivalence

Tactics :
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 ← has trade offs
Depends on
Imp can be run.

Memory mem: string → nat

(_ !→ 0)   all-zeros map

Fixpoint : recursive fns.

Booleans and number exprs are <u>pure</u> in Imp.

Problem : eval for commands does not necessarily terminate.
(bc of while)
It's not total!

→ specify using relations instead.

st =[cmd]=> st'   ~~Small~~ Large step semantics.

In Coq:

Inductive ceval : com → mem → mem → Prop

<u>Q</u> : Diff betwn Inductive and Fixpoint. ?

Inductive: defines <u>datatype</u>

Fixpoint: inductively defined <u>function</u> must be total!

Drawbacks of large-step : need to do induction on derivations instead of ~~by~~ on syntax.
~~defined inductively~~
~~by inductio~~

<u>Q</u> : Can we do small-step?
can be made to work. but need to deal with indicies everywhere. diff design space.

MONADS

Large-step : we

Fixpoint :

let
let
st

mem → mem
init state   final sta

S → S

state_bind

A generalizati

let x := m

<u>Q</u> Is there

Yes. not

<u>Q</u> monad laws

Unbundle proof

pure . in Imp.

does not necessarily terminate.
(bc of while)
It's not total!

ead.

step semantics.

mem → Prop

and Fixpoint. ?

Inductively defined ~~function~~ must be total!

do induction on derivations
of ~~by~~ on syntax.

need to deal with indicies

..

MONADS : "well-behaved sequential composition"

Large-step : need explicit plumbing for intermediate ~~step~~ states.

$$st =[c_1]\Rightarrow st' =[c_2]\Rightarrow st''$$

Fixpoint :

```
let st' := ... st in
let st'' := ... st' in
st''
```

plumbing!

$$\underset{\text{init state}}{mem} \to \underset{\text{final state}}{mem} \times \underset{\text{aux. answer}}{nat}$$

$$S \to S \times B$$

$$\text{state\_bind}\ \underset{\text{some computation}}{(m : \text{state } S\ A)}\ \ \underset{\text{continuation}}{(k : A \to \text{state } S\ B)}$$

A generalization of let

$$\text{let } x := m \text{ in } \underbrace{\text{rest}}_{k}\ x.$$

Q Is there any reason to implement ret as a notation instead of a function?

Yes, not all "ret" are functions.

Q monad laws?

Unbundle proof components is a best 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.

~~x : type~~
In order to prove this law, also need to prove that capture avoidance
~~variable capture~~ 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.