Increasing Confidence in Types

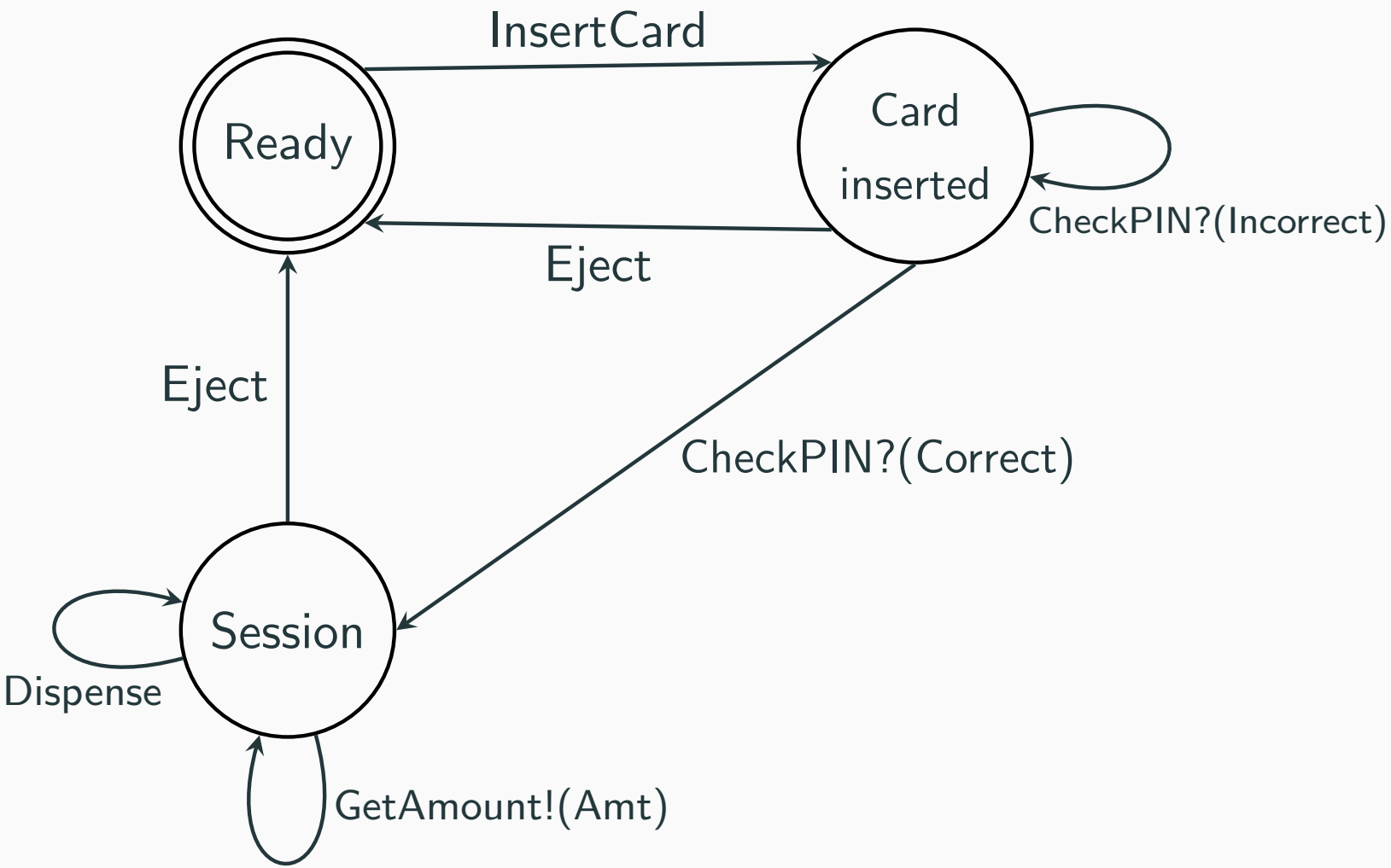
Thomas Ekström Hansen

6th March 2024

Overview

- Many systems exhibit Finite State Machine (FSM)-like behaviour
  - These can be neatly modelled with Dependent Types
- ATM example of this from Edwin Brady's "Type Driven Development with Idris" book
  - Turns out this example is subtly incorrect
- How can we spot this? Can we increase confidence in our type-level modelling without having to "just get it right"?
- Spoilers: How does QuickCheck fit in with dependent types?
- Not a proof technique, but hopefully catches errors faster and provides guarantees that our model behaves as intended

The ATM state machine



State machine of an ATM

Indexed State Monads (ISMs)

```
data ATMSt =  Ready | CardInserted | Session

data CheckPINRes  =  Incorrect | Correct

data ATMOp : (ty : Type) -> ATMSt -> (ty -> ATMSt) -> Type
  where
  Insert : ATMOp () Ready (const CardInserted)

  CheckPIN : (pin : Nat)
          -> ATMOp CheckPINRes CardInserted
                   (\case Incorrect => CardInserted
                          Correct => Session)

  GetAmount  : ATMOp Nat Session (const Session)
  Dispense :  (amt : Nat)
          -> ATMOp () Session (const Session)
  Eject  : ATMOp () state (const Ready)
```

Using the operations

- Using the ISM operations requires another ISM, defining `pure`, `op`, `bind`, and `seq`

```
data ATM : (t : Type) -> ATMSt -> (t -> ATMSt) -> Type
  where
  Pure : (x : t) -> ATM t (stFn x) stFn
  Op : ATMOp t st st' -> ATM t st st'
  (>>=) :  ATM a s1 s2f -> ((x : a) -> ATM b (s2f x) s3f)
        -> ATM b s1 s3f
  (>>) :  ATM () s1 s2f -> (ATM b (s2f ()) s3f)
       -> ATM b s1 s3f
```

Programming with ISMs

- We declare our intended start and end state in the type
- And the type-checker verifies that we don't use commands incorrectly

```
prog : ATM () Ready (const Ready)
prog = do
  Op Insert
  Correct <- Op $ CheckPIN 1234
    | Incorrect => <...>
  amount <- Op GetAmount
  Op $ Dispense amount
  Op Eject
```

Using types only gets you part of the way there

Rejected by the type-checker:

```
badProg : ATM ()
badProg = do
  Op Insert
  let pin = 1234
  Correct <- Op $ CheckPIN pin
    | Incorrect => InsertCard
  amt <- Op GetAmount
  Op $ Dispense amt
  -- We never Eject, so we
  -- never come back to
  -- `Ready'
```

Accepted by the type-checker:

```
loopProg : ATM ()
loopProg = do
    Op InsertCard
    let pin = 4321
    loopIncorrect pin
  where
    loopIncorrect : Nat -> ATM ()
    loopIncorrect p = do
      Incorrect <- Op $ CheckPIN p
        | Correct => -- <...>
      loopIncorrect p
```

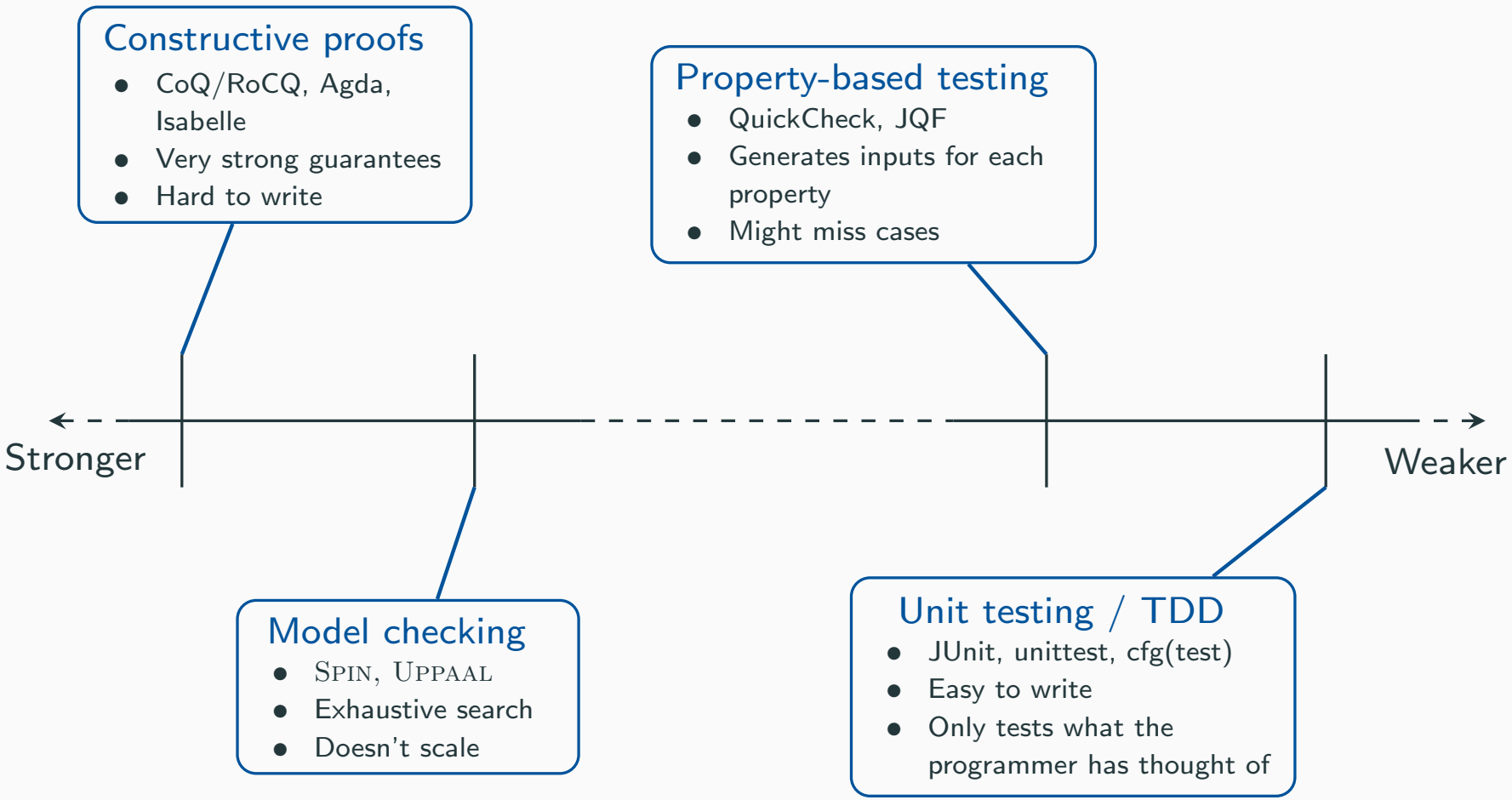
Why is this a problem?

- We expect an ATM to reject the card after 3 PIN attempts
  - Not to be permanently unavailable if we retry forever
- However, the programmer is unlikely to catch this
- The model looks correct and rigorous, after all
- Programming with it will catch most errors
- And the type-checker is happy with our sequence of operations

How do we solve this?

- We could spot the issue when it happens
  - Someone will (hopefully) spot the issue during development
  - Or, worst case, spot it when it happens after deployment
- And then we update our model and everything is good
- Why not try to spot it *automatically* before either of those?
- Modelling can clearly go wrong, so how do we increase our confidence in the models? Who type-checks the types?

Spectrum of Verification



The eternal problem with verification systems

- All verification systems face the same problem: ergonomics
- If the system is obstructive, or even just perceived as such, people are unlikely to use it
  - This is especially true for complex systems
  - "Fighting with the Rust borrow-checker"
  - "I'm experienced enough to write safe C/C++"
- Where does Idris fit in here?
  - General-purpose with dependent types, allowing us to program to different areas of the spectrum
  - Compiler and type-system assist you rather than hinder
  - Verify as you go along, tuning the strictness as necessary
  - Unit tests are not thorough enough, so QuickCheck seems like a good middle ground
  - Dependent types allow us to run the tests at compile time, and quantities to erase their results at runtime!

How do you generate a dependent type?

- QuickCheck's bread and butter is `Arbitrary`
  - Define how to generate an instance of a type, given some pseudorandom number generator state
  - Reasonably straightforward for random numbers, picking an element, or structures where the type of the constructors are known at generation-time
- However, our types are *dependent*
- So we cannot know the exact type at generation time
  - We can know *a* type, but not all. For example, `Vect 3 Nat` is trivial: `[!arbitrary, !arbitrary, !arbitrary]`
  - The problem is `Vect n Nat`
  - Or even `Vect n t`

Arbitrary dependent types

- The solution is more dependent types!
- Specifically: dependent pairs

```
record DPair a (p : a -> Type) where
  constructor MkDPair
  fst : a
  snd : p fst
```

- As long as we know how to generate an `` `Arbitrary a` ``, we can generate an `` `Arbitrary (x : a ** p x)` ``
  - (The `**` syntax is sugar for `DPair` / `MkDPair` depending on the context)

Arbitrary vectors

- To generate an arbitrary vector, we generate *some* length...
- ... and then generate that many `arbitrary` elements

```
Arbitrary a => Arbitrary (n : Nat ** Vect n a) where
  arbitrary = do
    len <- arbitrary
    vect <- nArbitrary len
    pure (len ** vect)
  where
    nArbitrary : (n : Nat) -> Gen (Vect n a)
    nArbitrary 0 = []
    nArbitrary (S k) = !arbitrary :: nArbitrary k
```

Plumbing for operations

```
record OpRes (resT : Type) (currSt : ATMState)
             (nsFn : resT -> ATMState) where
  constructor MkOpRes

  -- The operation
  op : ATMOp resT currSt nsFn

  -- The result of the operation
  res : resT

  -- Results must be `Show`-able for QC to work
  rShow : Show resT
```

Tracing ATMs

```
record TraceStep where
  constructor MkTS

  -- The `ATMOp`, along with some result,
  -- which took us to the traced state
  opRes : OpRes rT aSt aStFn

  -- The `ATMState` we ended up in
  resSt : ATMState

-- A bounded sequence of trace steps
data ATMTrace : ATMState -> Nat -> Type where
  MkATMTrace :  (initSt : ATMState)
             -> {bound : Nat}
             -> (trace : Vect bound TraceStep)
             -> ATMTrace initSt bound
```

Generating arbitrary OpRes

```
{currSt : ATMState} ->
Arbitrary (resT : _ ** nsFn : resT -> ATMState ** OpRes resT currSt nsFn)
where
  arbitrary {currSt=Ready} =
    pure (_ ** _ ** MkOpRes Insert () %search)

  arbitrary {currSt=CardInserted} = do
    -- we need _a_ PIN, even though we control the result
    let arbPIN = 0
    let op1 = (_ ** _ ** MkOpRes (CheckPIN arbPIN) Correct %search)
    let op2 = (_ ** _ ** MkOpRes (CheckPIN arbPIN) Incorrect %search)
    let op3 = (_ ** _ ** MkOpRes Eject () %search)
    -- can adjust the frequencies of getting the PIN wrong
    frequency $ [(1, pure op1), (4, pure op2), (1, pure op3)]

  arbitrary {currSt=Session} = do
    arbAmount <- arbitrary
    let op1 = (_ ** _ ** MkOpRes (Dispense arbAmount) () %search)
    let op2 = (_ ** _ ** MkOpRes Eject () %search)
    oneof [pure op1, pure op2]
```

Properties of the ATM

- As you can see, generating arbitrary ATM steps is a bit more involved than non-dependent types, but it is doable
- And now that we have that, we can specify properties like "Within 5 state-transitions, we should be back in `Ready`"

```
public export 0
PROP_eventuallyReady : Fn (ATMTrace Ready 5) Bool
PROP_eventuallyReady =
  MkFn (\case (MkATMTrace _ trace) =>
              elem Ready (map (.resSt) trace))
```

- *And* we can test it at compile-time

```
public export 0
EventuallyReady_OK : So (QuickCheck PROP_eventuallyReady)
EventuallyReady_OK = Oh
```

Model, verification, and implementation all-in-one

- With most verification tools, we have to translate between models
  - Spec, model, and implementation are independent
- This facilitates translation mistakes
  - Might think we're verifying the same thing, when in actual fact the semantics have changed between representations
- In our case, the specification *is* the model; *everywhere*

```
trace :  (steps : Nat) -> (currSt : ATMState)
      -> Gen (Vect steps TraceStep)
trace 0 _ = pure []
trace (S k) currSt =
  the (Gen (x ** y ** OpRes x currSt y)) arbitrary
  >>=
  \case (_ ** nsFn ** opR@(MkOpRes _ res _)) =>
     do let nextState = nsFn res  -- ! nsFn from ISM
        pure $ (MkTS opR nextState) :: !(trace k nextState)
```

QuickCheck spots the error!

- If we try to type-check the file we get:

```
-- Error: While processing right hand side of
--        EventuallyReady_OK. When unifying:
    So True
-- and:
    So (QuickCheck PROP_eventuallyReady)
-- Mismatch between: True and False.
```

- And if we investigate by running QC at the REPL, the error is exactly the fault in the model:

```
MkQCRes (Just False) <log> """
Falsifiable, after 4 tests:
Starting @ Ready:
[ (<ATMOp 'Insert ~ ()'>, CardInserted)
, (<ATMOp 'CheckPIN 0 ~ Incorrect'>, CardInserted)
, (<ATMOp 'CheckPIN 0 ~ Incorrect'>, CardInserted)
, (<ATMOp 'CheckPIN 0 ~ Incorrect'>, CardInserted)
, (<ATMOp 'CheckPIN 0 ~ Incorrect'>, CardInserted)
]\n"""
```

Fixing things

- Now that we know there's an error, we can fix things!

```
data ATMState
  = Ready
  | CardInserted Nat   -- track #retries
  | Session
<...>
CheckPIN : (pin : Nat)
  -> ATMOp PINok (CardInserted (S tries))
        (\case Correct => Session
               Incorrect => ifThenElse (isZero tries)
                                       Ready
                                       (CardInserted tries))
```

- Carrying this through to the generators, our QC passes: file reloads successfully, the REPL reports

```
> QuickCheck PROP_eventuallyReady
MkQCRes (Just True) <log> "OK, passed 100 tests"
```

Benefits of a multifaceted approach

1. Adaptability — being able to use different tools
2. Speed — can trade speed for level of verification
    - This isn't about proving things, it is about increasing confidence in our typed models
3. **Coherence** — all done in one system
    - No need to translate to model-checking tool
    - Specification lives alongside model lives alongside implementation
    - The implementation is just there; it *is* runnable code
    - Parts can be verified independently *while* combining into an overall system

Further work

- As neat as this is, it is still convoluted to write the types, generators, etc.
- There are an abundance of FSM-like systems — the ARQ protocol, pick-and-place machines — which we plan to model
- This should hopefully reveal common patterns, which we can then factor out and automate large parts of this

Thank you

Slides

github.com/CodingCellist/talks