Type-Level Property Based Testing

Thomas Ekström Hansen & Edwin Brady

TyDe '24 — 9^{th} September 2024

Overview

- Stateful systems fit nicely with dependently typed models
- How can one use property based testing with dependent types?
- A general framework for stateful, testable, and dependently typed models

Motivation

- Many systems exhibit Finite State Machine (FSM)-like behaviour
- These can be modelled using dependent types
    - Dependent types are difficult to get right
- We want to increase confidence in our types

Disclaimer: “increase confidence”

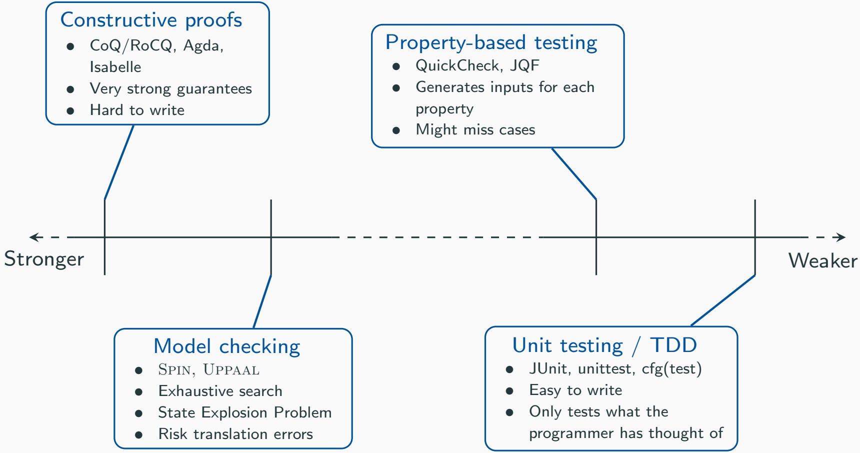
This is not a proof technique

But hopefully, it helps us catch errors faster and provides guarantees that our model behaves as intended

Stateful Computer Systems

- Stateful systems are ubiquitous
- Embedded controllers for automatic doors, ATMs, and network protocols
- These are all stateful
- And we would very much like them to be correct

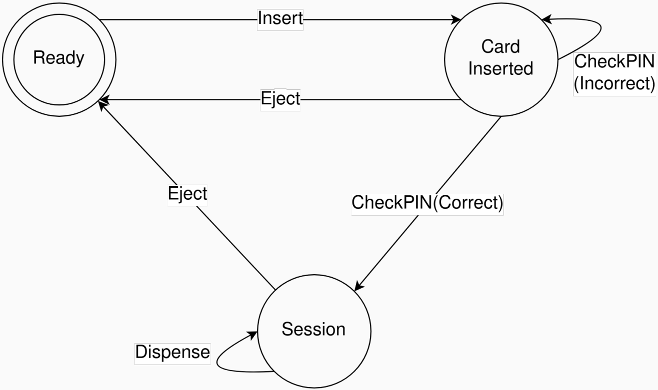
Spectrum of Verification



What about Type-Driven Development?

- Dependently Typed languages like Agda and Idris
- Can construct advanced types and embedded DSLs
- And the type checker then helps verify the program
- Fits somewhere in the middle

The ATM state machine

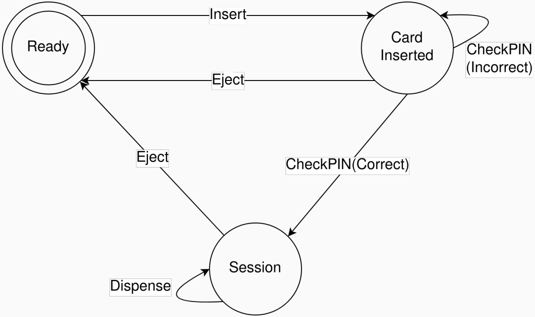


Model: Indexed State Monads (ISMs)

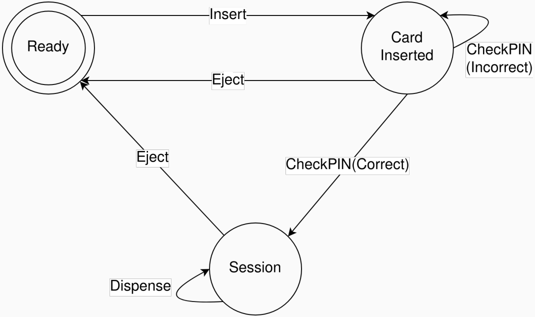
- Declare a datatype for the states
- Declare datatypes for the possible results (if any) of the operations
- Declare a datatype with constructors for each operation, such that:
  - The type checker can follow the state transitions
  - We can program with the operations

Datatype for the ATM states

```
data ATMState
  = Ready
  | CardInserted
  | Session
```



Datatype for ATM operation results

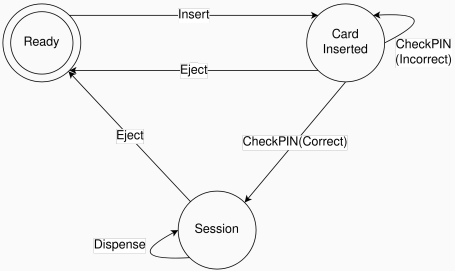



```
data PINok
  = Correct
  | Incorrect
```

Everything which does not have a result returns Unit — `()`

Dependent State Transition

```
ChkPINfn : PINok -> ATMState
ChkPINfn Correct = Session
ChkPINfn Incorrect =
↪  CardInserted
```



ATM Indexed State Monad

```
data ATM : (t : Type) -> ATMState -> (t -> ATMState) ->
↪  Type where
  Insert : ATM () Ready (const CardInserted)
  CheckPIN : (pin : Int) -> ATM PINok CardInserted
  ↪  ChkPINfn
  Dispense : (amt : Nat) -> ATM () Session (const Session)
  Eject : ATM () st (const Ready)
  Pure : (x : t) -> ATM t (stFn x) stFn
  (>>=) : ATM a s1 s2f -> ((x : a) -> ATM b (s2f x) s3f)
  ↪  -> ATM b s1 s3f
```

Programming with ISMs

- We declare our intended start and end state in the type

```
prog : ATM () Ready (const Ready)
```

- And the type-checker verifies that we don't use commands incorrectly

```
prog = do                            -- We start in Ready
  Insert  ---------------------- Ready to CardInserted
  Correct <- CheckPIN 1234  --- CI to Session
    | Incorrect => <...>  ----- (or stay in CI)
  Dispense 42  ---------------- Stay in Session
  Eject  ---------------------- Return to Ready
```

Dependent Types Only Get Some Things Right

Rejected by the type-checker:

```
badProg : ATM ()
            Ready (const
            ↪  Ready)
badProg = do
  Insert
  let pin = 1234
  Correct <- CheckPIN pin
    | Incorrect => InsertCard
  Dispense 42
  -- We never Eject, so we
  -- never come back to
  -- `Ready'
```

Accepted by the type-checker:

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

Why is this a problem?

- As-is, the PIN can be brute forced!
- 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

Suggestion: Property Based Testing

- QuickCheck is a Property Based Testing (PBT) framework initially developed for Haskell
- Define how to *generate* an instance of a type, given some pseudorandom number generator state
  - This is referred to as `Arbitrary`
- Write *properties* and *generate* their test case inputs

Generating Dependent Types is Tricky

- Consider generating arbitrary vectors:
  `Arbitrary t => Arbitrary (Vect n t)`
- `n` is bound outwith the interface
- We cannot guarantee that the generated `Vect` will have some general, unspecified length `n`
- We could generate vectors of a specific length, but this is not ideal

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

- Provided we know how to generate the elements, we generate *some* length

```
Arbitrary a => Arbitrary (n : Nat ** Vect n a) where
  arbitrary = do
    len <- arbitrary
```

- And then generate that many `arbitrary` elements

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

Arbitrary ATMs?

- Can we do a similar thing for `ATMOp` and `ATM`?
- Yes, but we need some (dependent) plumbing first

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
  {auto 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
```

Arbitrary OpRes: Prerequisites

Provided we know what state we are currently in, we can generate an operation and its result:

```
{currSt : ATMState} ->
Arbitrary (resT : _ ** nsFn : resT -> ATMState **
           OpRes resT currSt nsFn)
where
```

Arbitrary OpRes: from Ready

From `Ready`, we can insert the card:

```
arbitrary {currSt=Ready} =
  pure (_ ** _ ** MkOpRes Insert ())
```

Arbitrary OpRes: from CardInserted

Using a dummy value for the PIN, we can control the frequencies of the getting the PIN right:

```
arbitrary {currSt=CardInserted} = do
  -- we need a PIN, even though we control the result
  let arbPIN = 0
  let op1 = (_ ** _ ** MkOpRes (CheckPIN arbPIN) Correct)
  let op2 = (_ ** _ ** MkOpRes (CheckPIN arbPIN) Incorrect)
  let op3 = (_ ** _ ** MkOpRes Eject ())

  frequency $ [(1, pure op1), (4, pure op2), (1, pure op3)]
```

Arbitrary OpRes: from Session

Generate an arbitrary amount to dispense, or eject the card:

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

Properties of the ATM

- 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

- 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

All in one

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* combined into an overall system

Further work

- Running tests at the type level puts a lot of strain on the compiler, so there may be interesting optimisations to explore there
- Can we do more? ARQ with Sliding Window? Protocols with crash-stop failures?
- What kinds of properties can we test? Model Checking has been mentioned several times, so how does this compare to LTL or TLA^+?

Thank you

Links

PAPER, CODE, SLIDES

Contact: teh6@st-andrews.ac.uk

Preprint

arXiv:2407.12726

Code

GH: CodingCellist/tyde-24-code

Slides