Type-Level Property Based Testing

Thomas Ekström Hansen & Edwin Brady

TyDe '24 — 9th September 2024

Overview

- Stateful systems fit nicely with dependently typed models
- Dependent types are difficult to get right, how do we test them?
- 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”

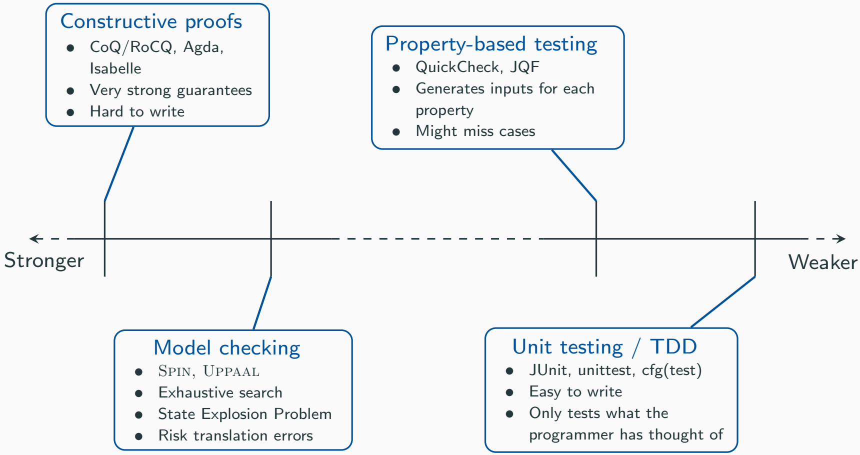
This is not a proof technique

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

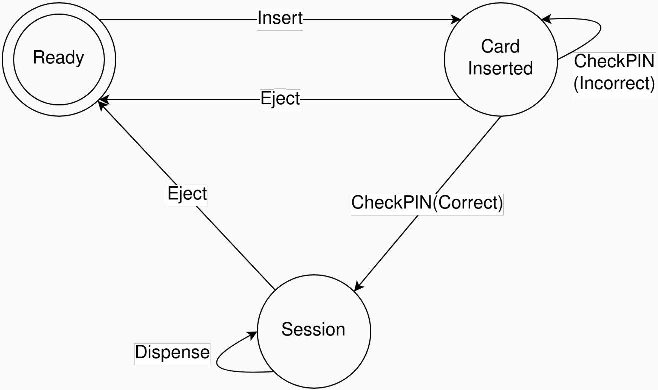
Spectrum of Verification



What about Type-Driven Development?

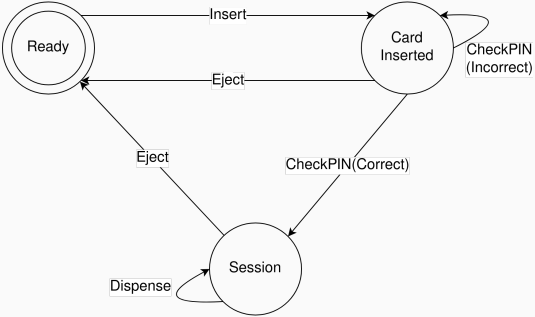
- Dependently Typed languages like Agda and Idris
- Can construct advanced types and embedded DSLs
- And the type checker then helps verify the program
- Fits somewhere in the middle

The ATM state machine



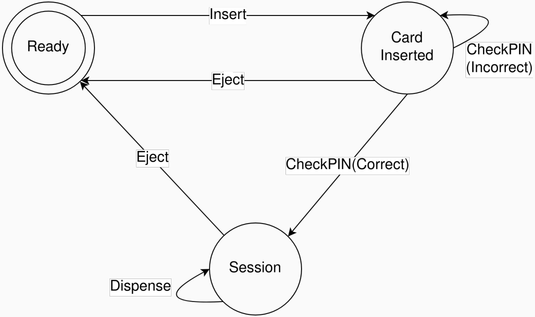
Datatype for the ATM states

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



Datatype for ATM operation results

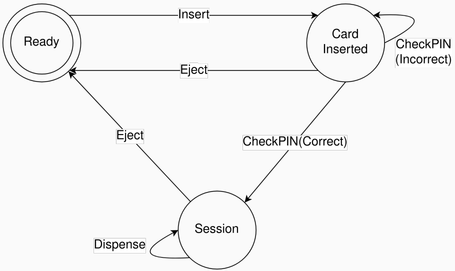
```
data PINok
  = Correct
  | Incorrect
```



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

State Transition Function

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



Dependent State Transition

```
data ATM : (t : Type) -> ATMState -> (t -> ATMState) ->
↪  Type where
  CheckPIN :  (pin : Int)
          -> ATM PINok CardInserted ChkPINfn
      ⋮

  (>>=) : ATM a s1 s2f -> ((x : a) -> ATM b (s2f x) s3f)
  ↪  -> ATM b s1 s3f
```

ATM Indexed State Monad

```
data ATM : (t : Type) -> ATMState -> (t -> ATMState) ->
↪  Type where
  CheckPIN :  (pin : Int)
           -> ATM PINok CardInserted ChkPINfn
  Insert : ATM () Ready (const CardInserted)
  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
```

Why Is This Neat?

- 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 operations 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

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

Type-Level Property Based Testing

- In Idris2, the `So` type is inhabited iff its argument evaluates to `True`
- In other words, we can run property based testing as part of the type checking process!
  `So (quickCheck <property>)`
- Idris2 is built on Quantitative Type Theory, which has erasure, meaning the tests are removed from the compiled program

Generating Dependent Types is Tricky

- Consider generating arbitrary vectors:
  `Arbitrary t => Arbitrary (Vect n t)`
- The type index `n` is implicit
- 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 — an instance for each length

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

- We know how to generate the elements, so start by generating *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`?
- With a bit of work, yes!

A Bit of Work

- Store the operation and its result instance — whether `PINok` was successful, for example
- Store an operation-result pair, and the state this moved us to
- A chain of these make up a *trace*

Generating Traces

- Given the current state, pattern matching allows the type checker to reduce the state function
- So we know which operations are possible
- Pick an arbitrary one, apply it, log its results, and repeat the process

ATM: from CardInserted

This is still QuickCheck, we can control the frequency of generated instances:

```
options 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)]
```

QuickCheck Spots the Error!

```
0 PROP_eventuallyReady : Fn (ATMTrace Ready 10) Bool
PROP_eventuallyReady = MkFn
  (\case (MkATMTrace _ trace) => elem Ready (map (.resSt) trace))
```

With a property to eventually return to `Ready`, the file no longer type checks:

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

QuickCheck Gives a Trace

- Investigating by running QuickCheck at the REPL, the error is exactly the fault in the model:

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

Fixing Things

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

```
ChkPINfn : (retries : Nat) -> PINok -> ATMState
ChkPINfn 0     Correct   = Session
ChkPINfn 0     Incorrect = Ready
ChkPINfn (S k) Correct   = Session
ChkPINfn (S k) Incorrect = CardInserted k
```

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

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

Model, Verification, and Implementation

- With most verification tools, we have to translate between models
  - Spec, model, and implementation are independent
- This results in the risk of 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) -> (st : stT) -> Gen (Vect steps
↪  (TraceStep opT))
trace 0 _ = pure []
trace (S j) st = do
  (_ ** stFn ** opR@(MkOpRes op res)) <- arbitrary
  let nextSt = stFn res
  pure $ (MkTS opR nextSt) :: !(trace j nextSt)
```

And this works for anything expressed in terms of states, results, and operations — ISMs generalise

We Are Testing Types!

- We have not tested the implementation
- We have tested the dependent types which help guide us when writing the implementation
- Which gives us confidence that our types capture the right behaviours
- Dependent types are type-level programs, let's test them!

Links

Email: teh6@st-andrews.ac.uk

Preprint

arXiv:2407.12726

Code

GitHub: CodingCellist/tyde-24-code

Slides

GitHub: CodingCellist/talks/2024-09-06-tyde-24-milan

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? There are parallels to Model Checking, so how does this compare to LTL or TLA$^+$?

Generic ISM Datatype

```
op : forall st . (t' : Type) -> st -> (t' -> st) -> Type

data Prog : {0 stT : _} -> (opT : (t' : _) -> stT -> (t'
↪  -> stT) -> Type) -> (t : Type) -> (from : stT) -> (to
↪  : t -> stT) -> Type where
  Pure : (x : t) -> Prog opT t (stFn x) stFn
  Op : {0 opT : (t' : _) -> stT -> (t' -> stT) -> Type} ->
  ↪  opT t st stFn -> Prog opT t st stFn
  (>>=) : Prog opT resT1 st1 stFn1 -> ((x : resT1) -> Prog
  ↪  opT resT2 (stFn1 x) stFn2) -> Prog opT resT2 st1
  ↪  stFn2
```

Operation-Result Pairs

```
record OpRes {0 stT : _} (opT : (t' : _) -> stT -> (t' ->
↪  stT) -> Type) (resT : Type) (currSt : stT) (0 nsFn :
↪  resT -> stT) where
  constructor MkOpRes
  op : opT resT currSt nsFn
  res : resT
  {auto opShow : Show (opT resT currSt nsFn)}
  {auto rShow : Show resT}
```

TraceStep

```
record TraceStep (opT : (t' : _) -> stT -> (t' -> stT) ->
↪  Type) where
  constructor MkTS
  {0 stepRT : _}
  {0 stepSt : stT}
  {0 stepFn : stepRT -> stT}

  opRes : OpRes opT stepRT stepSt stepFn
  resSt : stT

  {auto showStT : Show stT}
```

Trace

```
data Trace : (opT : (t' : _) -> stT -> (t' -> stT) ->
↪  Type) -> stT -> Nat -> Type where
  MkTrace : Show stT => (initSt : stT) -> {bound : Nat}
          -> (trace : Vect bound (TraceStep opT))
          -> Trace opT initSt bound
```