Type-Level Property Based Testing

Thomas Ekström Hansen & Edwin Brady

TyDe '24 — 9th September 2024

Motivation

- Many systems exhibit Finite-State-Machine-like behaviour

Motivation

- Many systems exhibit Finite-State-Machine-like behaviour
- These can be modelled using dependent types

Motivation

- Many systems exhibit Finite-State-Machine-like behaviour
- These can be modelled using dependent types
  - Dependent types are difficult to get right

Motivation

- Many systems exhibit Finite-State-Machine-like behaviour
- These can be modelled using dependent types
  - Dependent types are difficult to get right
- How do we increase confidence in our dependent 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

Stateful Computer Systems

- Stateful systems are ubiquitous
- Embedded controllers for automatic doors, ATMs, network protocols, etc...

Stateful Computer Systems

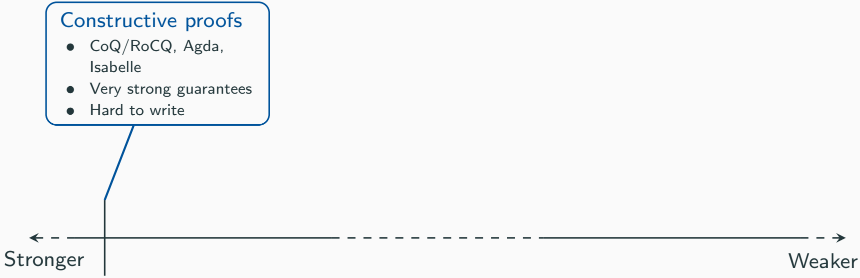
- Stateful systems are ubiquitous
- Embedded controllers for automatic doors, ATMs, network protocols, etc...
- These are all stateful

Stateful Computer Systems

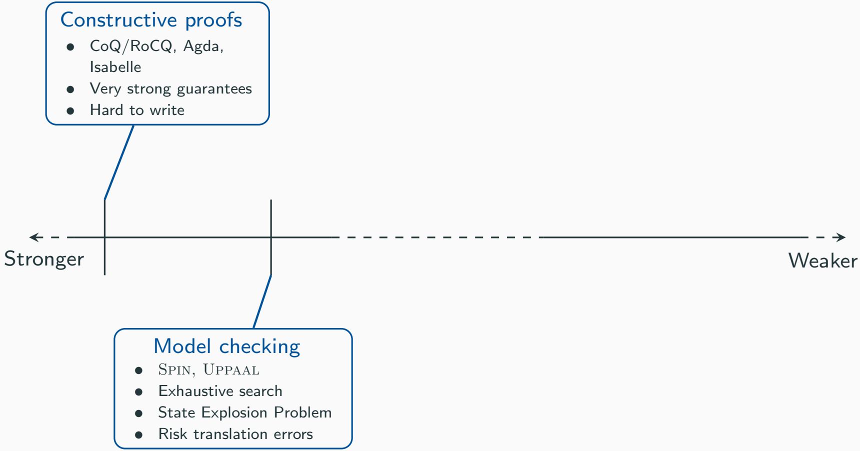
- Stateful systems are ubiquitous
- Embedded controllers for automatic doors, ATMs, network protocols, etc...
- These are all stateful
- And we would very much like them to be correct

Spectrum of Verification

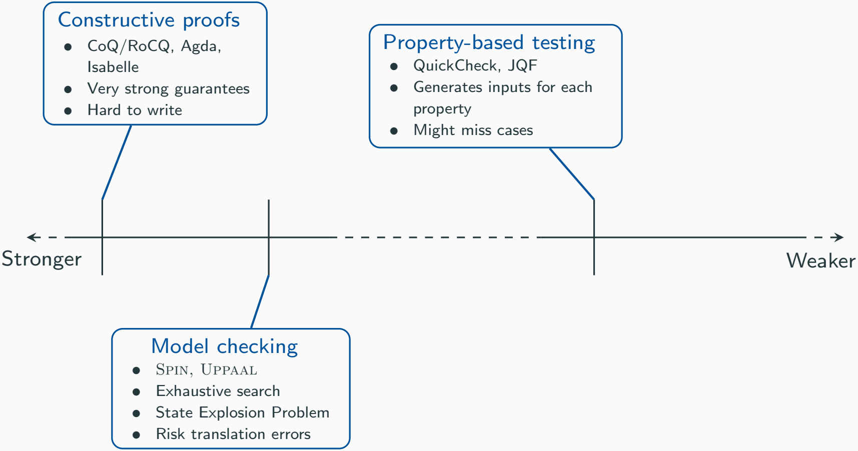
Spectrum of Verification



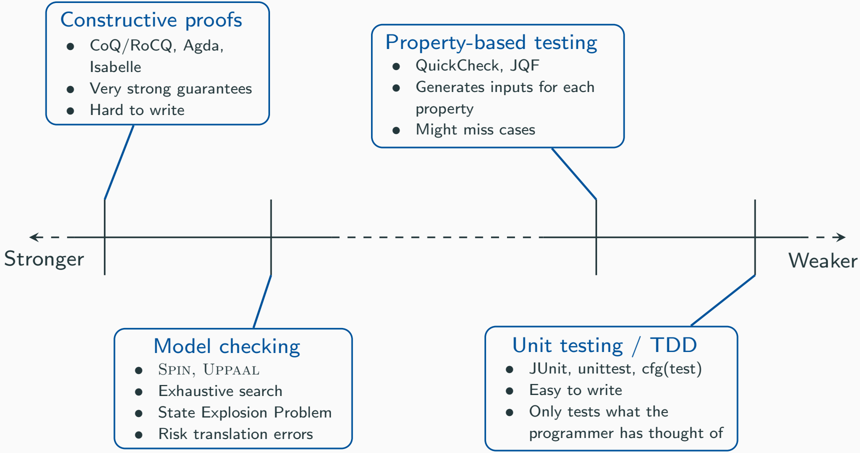
Spectrum of Verification



Spectrum of Verification



Spectrum of Verification



What about Type-Driven Development?

- Dependently typed languages
  like Agda and Idris

What about Type-Driven Development?

- Dependently typed languages like Agda and Idris
- Can construct advanced types and embedded DSLs

What about Type-Driven Development?

- Dependently typed languages like Agda and Idris
- Can construct advanced types and embedded DSLs
- Have the types and type checker guide the implementation and verify its behaviour (correct-by-construction)

What about Type-Driven Development?

- Dependently typed languages like Agda and Idris
- Can construct advanced types and embedded DSLs
- Have the types and type checker guide the implementation and verify its behaviour (correct-by-construction)

```haskell
head :: [a] -> a
-- crashes on
-- `head []`
```

What about Type-Driven Development?

- Dependently typed languages like Agda and Idris
- Can construct advanced types and embedded DSLs
- Have the types and type checker guide the implementation and verify its behaviour (correct-by-construction)

```
head :: [a] -> a
-- crashes on
-- `head []`

head :  Vect (S k) a
     -> a
-- always safe because
-- the length must be
-- at least 1
```

What about Type-Driven Development?

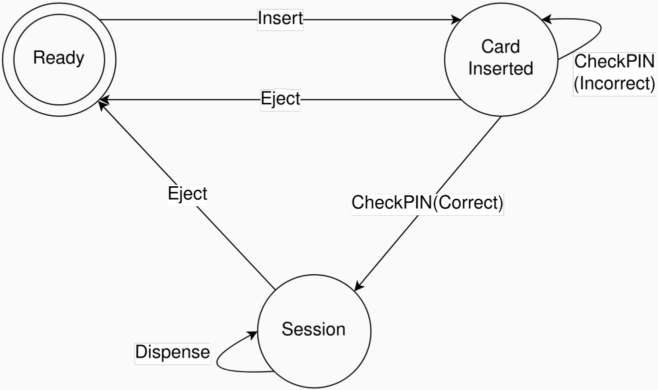
- Dependently typed languages like Agda and Idris
- Can construct advanced types and embedded DSLs
- Have the types and type checker guide the implementation and verify its behaviour (correct-by-construction)

```
head :: [a] -> a
-- crashes on
-- `head []`

head :  Vect (S k) a
     -> a
-- always safe because
-- the length must be
-- at least 1
```

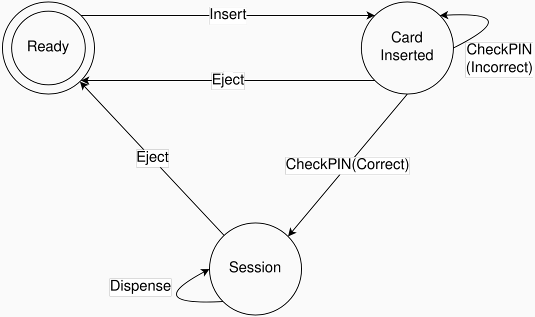
Fits somewhere in the middle

The ATM state machine



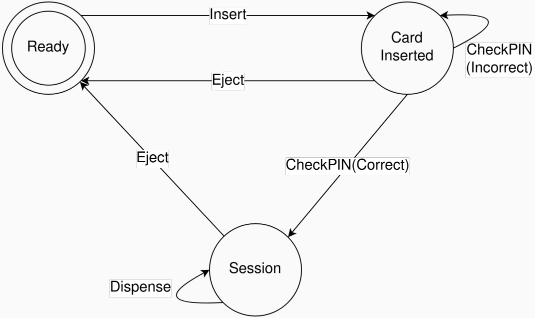
Datatype for the ATM states

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



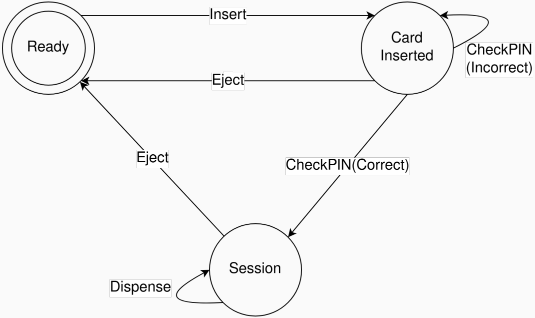
Datatype for ATM operation results

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



Datatype for ATM operation results

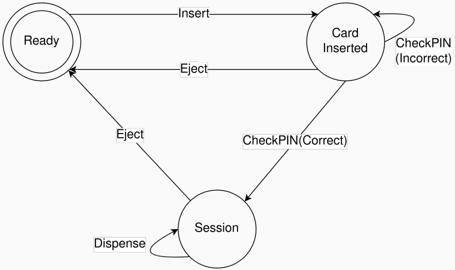
```
data PINok
  = Correct
  | Incorrect
```



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
```

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)`

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'
```

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
```

Property Based Testing

- Generate instances of types from random numbers

```
data Gen a : (Int -> a) -> Type
```

- Generate instances of types from random numbers

  ```
  data Gen a : (Int -> a) -> Type
  ```

- ```
  interface Arbitrary a where
    arbitrary : Gen a
  ```
  ```

Property Based Testing

- Generate instances of types from random numbers

  ```
  data Gen a : (Int -> a) -> Type
  ```

- ```
  interface Arbitrary a where
    arbitrary : Gen a
  ```

- We express boolean properties, which are then tested over random inputs (typically 100)

  ```
  reverse (reverse xs) == xs
  ```
  ```

What About Dependent Types?

QuickCheck works very well for regular types, what about dependent types?

Generating Dependent Types is Tricky

Consider generating arbitrary vectors:

```
Arbitrary t => Arbitrary (Vect n t) where
```

Generating Dependent Types is Tricky

Consider generating arbitrary vectors:

```
Arbitrary t => Arbitrary (Vect n t) where
  arbitrary = do
    length_ <- arbitrary
```

Generating Dependent Types is Tricky

Consider generating arbitrary vectors:

```
Arbitrary t => Arbitrary (Vect n t) where
  arbitrary = do
    length_ <- arbitrary
    arbVect <- nArbitrary length_
```

Consider generating arbitrary vectors:

```
Arbitrary t => Arbitrary (Vect n t) where
  arbitrary = do
    length_ <- arbitrary
    arbVect <- nArbitrary length_
    pure arbVect
```

Generating Dependent Types is Tricky

Consider generating arbitrary vectors:

```
Arbitrary t => Arbitrary (Vect n t) where
  arbitrary = do
    length_ <- arbitrary
    arbVect <- nArbitrary length_
    pure arbVect
    -- Error: cannot unify length_ with n
```

Generating Dependent Types is Tricky

Consider generating arbitrary vectors:

```
Arbitrary t => Arbitrary (Vect n t) where
  arbitrary = do
    length_ <- arbitrary
    arbVect <- nArbitrary length_
    pure arbVect
    -- Error: cannot unify length_ with n
```

How do we solve this?

Arbitrary Dependent Types

- The solution is more dependent types!

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
```

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)`

Arbitrary Vectors

When we know the length, we will know the type of the vector

```
Arbitrary (n : Nat ** Vect n a) where
```

Arbitrary Vectors

When we know the length, we will know the type of the vector

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

Arbitrary Vectors

When we know the length, we will know the type of the vector

```
Arbitrary (n : Nat ** Vect n a) where
  arbitrary = do
    length_ <- arbitrary
    arbVect <- nArbitrary length_
```

Arbitrary Vectors

When we know the length, we will know the type of the vector

```
Arbitrary (n : Nat ** Vect n a) where
  arbitrary = do
    length_ <- arbitrary
    arbVect <- nArbitrary length_
    pure (length_ ** arbVect)
```

Arbitrary Vectors

When we know the length, we will know the type of the vector

```
Arbitrary (n : Nat ** Vect n a) where
  arbitrary = do
    length_ <- arbitrary
    arbVect <- nArbitrary length_
    pure (_ ** arbVect)
```

Arbitrary ATMs?

Can we do a similar thing for the `ATM`?

Arbitrary ATMs?

Can we do a similar thing for the `ATM`?

With a bit of work, yes!

ATM: from CardInserted

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

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)]
```

Type-Level Property Based Testing

- In Idris2, the `So` type is inhabited iff its argument evaluates to `True`

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>)
```

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 can be removed from the compiled program

QuickCheck Spots the Error!

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

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

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

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!

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
```

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 representations
  - Spec, model, and implementation are independent

Model, Verification, and Implementation

- With most verification tools, we have to translate between representations
  - Spec, model, and implementation are independent
- This results in the risk of translation mistakes

Model, Verification, and Implementation

- With most verification tools, we have to translate between representations
  - Spec, model, and implementation are independent
- This results in the risk of translation mistakes
  - The verification tool might not support the same types as the implementation language

Model, Verification, and Implementation

- With most verification tools, we have to translate between representations
  - Spec, model, and implementation are independent
- This results in the risk of translation mistakes
  - The verification tool might not support the same types as the implementation language
  - 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)
```

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)
```

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 and operations with results — ISMs generalise

We Are Testing Types!

- We have tested the dependent types which help guide us when writing the implementation, not the implementation itself since it is being kept in check by the types

We Are Testing Types!

- We have tested the dependent types which help guide us when writing the implementation, not the implementation itself since it is being kept in check by the types
- Testing gives us confidence that our dependent types are not misleading

We Are Testing Types!

- We have tested the dependent types which help guide us when writing the implementation, not the implementation itself since it is being kept in check by the types
- Testing gives us confidence that our dependent types are not misleading
- Dependent types are type-level programs, let's test them!

Thank You

Email: teh6@st-andrews.ac.uk

Paper

doi:10.1145/3678000.3678206

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
```