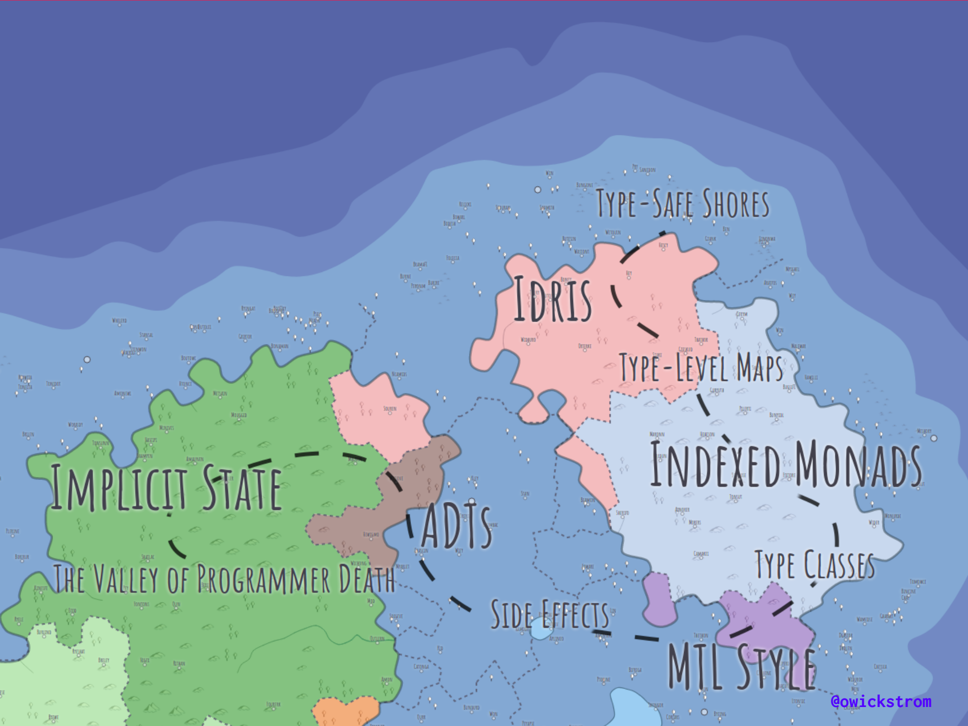
Finite State Machines?

Your compiler wants in!

Oskar Wickström

lambda DAYS
22-23 FEBRUARY 2018
KRAKÓW | POLAND


Type-Safe Shores
Idris
Type-Level Maps
Indexed Monads
Implicit State
ADTs
The Valley of Programmer Death
Side Effects
Type Classes
MTL Style
@owickstrom

State

Stateful Programs

- The program *remembers* previous events
- It may transition to another state based on its current state

Implicit State

- The program does not explicitly define the set of legal states
- State is scattered across many mutable variables
- Hard to follow and to ensure the integrity of state transitions
- Runtime checks “just to be sure”

Making State Explicit

- Instead, we can make states *explicit*
- It is clearer how we transition between states
- Make stateful programming less error-prone

Finite-State Machines

Finite-State Machines

- We model a program as an abstract *machine*
- The machine has a finite set of *states*
- The machine is in one state at a time
- *Events* trigger state transitions
- From each state, there's a set of legal transitions, expressed as associations from events to other states

Our Definition

$$State(S) \times Event(E) \rightarrow Actions\ (A),\ State(S\prime)$$

If we are in state S and the event E occurs, we should perform the actions A and make a transition to the state S′.

— Erlang FSM Design Principles [1]

[1] http://erlang.org/documentation/doc-4.8.2/doc/design_principles/fsm.html

Excluded

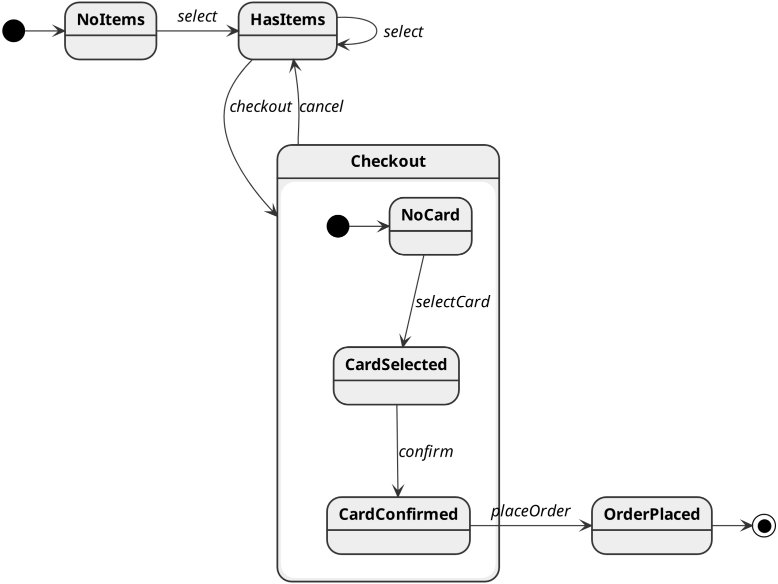
- Not strictly Mealy or Moore machines
- No hierarchical machines
- No guards in our models
- No UML statecharts

States as Data Types

- We model the set of legal states as a data type
- Each state has its own value constructor
- You can do this in most programming languages
- We'll use Haskell to start with

Encoding with Algebraic Data Types

Example: Checkout Flow



@owickstrom

States as an ADT

```haskell
data CheckoutState
  = NoItems
  | HasItems (NonEmpty CartItem)
  | NoCard (NonEmpty CartItem)
  | CardSelected (NonEmpty CartItem)
                 Card
  | CardConfirmed (NonEmpty CartItem)
                  Card
  | OrderPlaced
  deriving (Show, Eq)
```

Events as an ADT

```haskell
data CheckoutEvent
  = Select CartItem
  | Checkout
  | SelectCard Card
  | Confirm
  | PlaceOrder
  | Cancel
  deriving (Show, Eq)
```

FSM Type

```
type FSM s e =
  s -> e -> s

checkout :: FSM CheckoutState CheckoutEvent
```

State Machine with IO

```
type ImpureFSM s e =
  s -> e -> IO s
```

Checkout using ImpureFSM

```
checkoutImpure :: ImpureFSM CheckoutState CheckoutEvent
```

Checkout using ImpureFSM (cont.)

```
checkoutImpure NoItems (Select item) =
  return (HasItems (item :| []))

checkoutImpure (HasItems items) (Select item) =
  return (HasItems (item <| items))

...
```

Checkout using ImpureFSM (cont.)

```
...

checkoutImpure (CardConfirmed items card) PlaceOrder = do
  PaymentProvider.chargeCard card (calculatePrice items)
  return OrderPlaced
```

Impure Runner

```haskell
runImpure :: ImpureFSM s e -> s -> [e] -> IO s
runImpure = foldM
```

Logging FSM

```haskell
withLogging ::
     (Show s, Show e)
  => ImpureFSM s e
  -> ImpureFSM s e
withLogging fsm s e = do
  s' <- fsm s e
  liftIO $
    printf "- %s × %s → %s\n" (show s) (show e) (show s')
  return s'
```

Impure Runner Example

```
runImpure
  (withLogging checkoutImpure)
  NoItems
  [ Select "food"
  , Select "fish"
  , Checkout
  , SelectCard "visa"
  , Confirm
  , PlaceOrder
  ]
```

Impure Runner Example Output

```
- NoItems × Select "food" → HasItems ("food" :| [])
- HasItems ("food" :| []) × Select "fish" → HasItems ("fish" :| ["food"])
- HasItems ("fish" :| ["food"]) × Checkout → NoCard ("fish" :| ["food"])
- NoCard ("fish" :| ["food"]) × SelectCard "visa" → CardSelected ("fish" :| ["food"]) "visa"
- CardSelected ("fish" :| ["food"]) "visa" × Confirm → CardConfirmed ("fish" :| ["food"]) "visa"
Charging $666
- CardConfirmed ("fish" :| ["food"]) "visa" × PlaceOrder → OrderPlaced
```

ADT Summary

- We have explicit states using data types
- Standardized way of running state machine programs
  - It's simple to add logging, metrics
  - Instead of a list of events, we could use conduit[2] or pipes[3]
- We still have IO coupled with transitions (harder to test)
- Legal state transitions are not enforced

[2] https://hackage.haskell.org/package/conduit

[3] https://hackage.haskell.org/package/pipes

MTL Style and Associated Types

MTL Style with an Associated Type

- We will write our state machines in “MTL style”
- Some extra conventions for state machines
- With MTL style, we can:
    - combine with monad transformers (error handling, logging, etc)
    - build higher-level machines out of lower-level machines

Typeclass and Abstract Program

- A typeclass encodes the state machine transitions
- Events are represented as typeclass methods
- The current state is passed as a *value*
- The state transitioned to is returned as a value
- The state type is *abstract* using an associated type alias
- We write a program depending on the typeclass
- The typeclass and the program together form the state machine

Instances

- An instance is required to run the state machine program
- The instance performs the state transition side-effects
- The instance chooses the concrete data type
- We can write test instances without side-effects

States as Empty Types

```haskell
data NoItems

data HasItems

data NoCard

data CardSelected

data CardConfirmed

data OrderPlaced
```

State Machine with Class

```
class Checkout m where
  type State m :: * -> *

  ...
```

State Machine with Class (cont.)

The `initial` method gives us our starting state:

```
initial :: m (State m NoItems)
```

State Machine with Class (cont.)

Some events transition from exactly one state to another:

```
confirm ::
     State m CardSelected -> m (State m CardConfirmed)
```

The Select Event

- Some events are accepted from many states
- Both `NoItems` and `HasItems` accept the `select` event
- We could use `Either`

Selection States

```haskell
data SelectState m
  = NoItemsSelect (State m NoItems)
  | HasItemsSelect (State m HasItems)
```

Signature of `select`

```
select ::
     SelectState m
  -> CartItem
  -> m (State m HasItems)
```

The Cancel Event

- There are *three* states accepting `cancel`
- `Either` would not work, only handles *two*
- Again, we create a datatype:

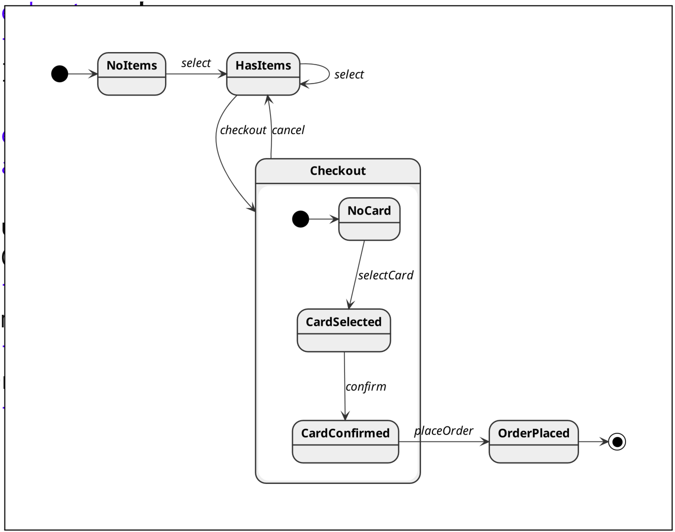
```haskell
data CancelState m
  = NoCardCancel (State m NoCard)
  | CardSelectedCancel (State m CardSelected)
  | CardConfirmedCancel (State m CardConfirmed)
```

- And the signature of `cancel` is:

```haskell
cancel :: CancelState m -> m (State m HasItems)
```

The Complete Typeclass

```
class Ch
  type S
  initia
  select
      S
    -> C
    -> m
  checko
  select
      S
  confir
      S
  placeO
      S
  cancel
  end ::
```



A State Machine Program

```haskell
fillCart ::
     (Checkout m, MonadIO m)
  => State m NoItems
  -> m (State m HasItems)
fillCart noItems = do
  first <- prompt "First item:"
  select (NoItemsSelect noItems) first >>= selectMoreItems
```

A State Machine Program (cont.)

```haskell
selectMoreItems ::
     (Checkout m, MonadIO m)
  => State m HasItems
  -> m (State m HasItems)
selectMoreItems s = do
  more <- confirmPrompt "More items?"
  if more
    then prompt "Next item:" >>=
         select (HasItemsSelect s) >>=
         selectMoreItems
    else return s
```

A State Machine Program (cont.)

```haskell
startCheckout ::
     (Checkout m, MonadIO m)
  => State m HasItems
  -> m (State m OrderPlaced)
startCheckout hasItems = do
  noCard <- checkout hasItems
  card <- prompt "Card:"
  cardSelected <- selectCard noCard card
  useCard <-
    confirmPrompt ("Confirm use of '" <> card <> "'?")
  if useCard
    then confirm cardSelected >>= placeOrder
    else cancel (CardSelectedCancel cardSelected) >>=
         selectMoreItems >>=
         startCheckout
```

A State Machine Program (cont.)

```haskell
checkoutProgram ::
     (Checkout m, MonadIO m)
  => m OrderId
checkoutProgram =
  initial >>= fillCart >>= startCheckout >>= end
```

The Abstract Part

- We only depend on the `Checkout` typeclass[4]
- Together with the typeclass, `checkoutProgram` forms the state machine

[4] We do use `MonadIO` to drive the program, but that could be extracted.

A Checkout Instance

- We need an instance of the `Checkout` class
- It will decide the concrete `State` type
- The instance will perform the effects at state transitions
- We'll use it to run our `checkoutProgram`

Concrete State Data Type

```haskell
data CheckoutState s where
  NoItems :: CheckoutState NoItems

  HasItems :: NonEmpty CartItem -> CheckoutState HasItems

  NoCard :: NonEmpty CartItem -> CheckoutState NoCard

  CardSelected
    :: NonEmpty CartItem
    -> Card
    -> CheckoutState CardSelected

  CardConfirmed
    :: NonEmpty CartItem
    -> Card
    -> CheckoutState CardConfirmed

  OrderPlaced :: OrderId -> CheckoutState OrderPlaced
```

CheckoutT

```haskell
newtype CheckoutT m a = CheckoutT
  { runCheckoutT :: m a
  } deriving ( Monad
             , Functor
             , Applicative
             , MonadIO
             )
```

Checkout Instance

```haskell
instance (MonadIO m) => Checkout (CheckoutT m) where
  type State (CheckoutT m) = CheckoutState

  ...
```

Initial State

```
...

initial = return NoItems

...
```

Select

```haskell
...

select state item =
  case state of
    NoItemsSelect NoItems ->
      return (HasItems (item :| []))
    HasItemsSelect (HasItems items) ->
      return (HasItems (item <| items))

...
```

Select

```
...

placeOrder (CardConfirmed items card) = do
  orderId <- newOrderId
  let price = calculatePrice items
  PaymentProvider.chargeCard card price
  return (OrderPlaced orderId)
```

Putting it all together

```
example :: IO ()
example = do
  orderId <- runCheckoutT checkoutProgram
  T.putStrLn ("Completed with order ID: " <> orderId)
```

Summary

- We've modeled state machines using:
  - Type classes/MTL style
  - Associated types for states
  - Explicit state values
  - "Abstract" program
  - Instances for side-effects
- Stricter than ADT-based version
- Not necessarily safe
  - State values can be *reused* and *discarded*
  - Side-effects can be reperformed *illegally*
  - Nothing enforcing transition to a terminal state

Reusing State Values

```
placeOrderTwice cardConfirmed = do
  _ <- placeOrder cardConfirmed

  orderPlaced <- placeOrder cardConfirmed
  log "You have to pay twice, LOL."

  end orderPlaced
```

Monad, Carry Thy State!

- One solution would be linear types
- Another is to carry the state *inside* the monad
- No need for explicit state values:

```
placeOrderTwice = do
    placeOrder
    placeOrder -- BOOM, type error!
    end
```

- We parameterize the monad, or *index* it, by the state type

Indexed Monads

Indexed Monad Type Class

- A monad with two extra type parameters:
  - Input
  - Output
- Can be seen as type before and after the computation
- Type class:

```
class IxApplicative m => IxMonad (m :: k -> k -> * -> *) where
    ...
```

Monad bind

```
bind  :: m     a -> (a -> m     b) -> m     b
```

ibind (simplified)

```
bind  :: m     a -> (a -> m     b) -> m     b

ibind :: m i j a -> (a -> m j k b) -> m i k b
```

@owickstrom

ibind (simplified)

```
bind  :: m     a -> (a -> m     b) -> m     b

ibind :: m i j a -> (a -> m j k b) -> m i k b
```

Specializing ibind

```
ibind
  ::        m i     j     a
  -> (a  -> m j     k     b )
  ->        m i     k     b
```

Specializing `ibind` (cont.)

```
ibind
  ::         m State1 State2 ()
  -> (() -> m State2 State3 ())
  ->         m State1 State3 ()
```

Indexed Bind Example

```
checkout :: m HasItems NoCard ()

selectCard :: m NoCard CardSelected ()

(checkout `ibind` const selectCard) :: m HasItems CardSelected ()
```

Indexed State Monad

- We hide the state *value*
- Only the state type is visible
- We cannot use a computation twice *unless the type permits it*

Composability

- The indexed monad describe *one* state machine
- Hard to compose
- We want *multiple* state machines in a single computation
    - Opening two files, copying from one to another
    - Ticket machine using a card reader and a ticket printer
    - A web server and a database connection
- One solution:
    - A type, mapping from names to states, as the index
    - Named state machines are independent
    - Apply events *by name*

Row Types in PureScript

- PureScript has a *row kind* (think type-level record):

```
(out :: File, in :: Socket)
```

- Can be polymorphic:

```
forall r. (out :: File, in :: Socket | r)
```

- Used as indices for record and effect *types*:

```
Record (out :: File, in :: Socket)
-- is the same as:
{ out :: File, in :: Socket }
```

Row Types for State Machines

```
-- Creating `myMachine` in its initial state:
initial
  :: forall r
   . m r (myMachine :: InitialState | r) Unit

-- Transitioning the state of `myMachine`.
someTransition
  :: forall r
   . m (myMachine :: State1 | r) (myMachine :: State2 | r) Unit

-- Deleting `myMachine` when in its terminal state:
end
  :: forall r
   . m (myMachine :: TerminalState | r) r Unit
```

Running Row Type State Machines

```
runIxMachines
    :: forall m
    . Monad m
    => IxMachines m () () a      -- empty rows!
    -> m a
```

Related Libraries

- `Control.ST` in Idris contrib library[5]
- "purescript-leffe" (The **L**abeled **Eff**ects **E**xtension)[6]
- "Motor" for Haskell[7]

[5] http://docs.idris-lang.org/en/latest/st/state.html

[6] https://github.com/owickstrom/purescript-leffe

[7] http://hackage.haskell.org/package/motor

More on Indexed Monads

- Read the introduction on "Kwang's Haskell Blog"[8]
- Haskell package `indexed`[9]
- Also, see `RebindableSyntax` language extension
- Can be combined with session types[10]

---

[8] https://kseo.github.io/posts/2017-01-12-indexed-monads.html

[9] https://hackage.haskell.org/package/indexed

[10] Riccardo Pucella and Jesse A. Tov, Haskell session types with (almost) no class, Haskell '08.

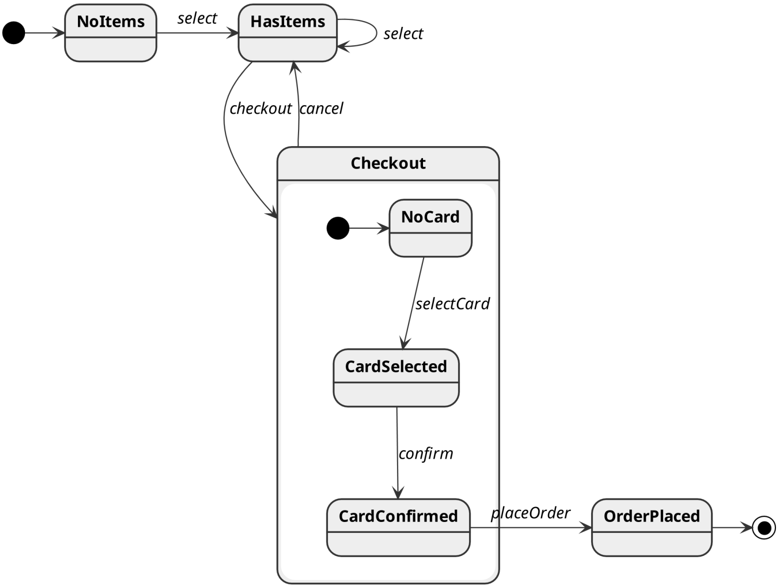
Dependent Types in Idris

Idris and Control.ST

- Dependent types makes some aspects more concise
  - Multiple states accepting an event
  - Error handling
  - Dependent state types
- The `Control.ST` library in Idris supports multiple "named" resources
- "Implementing State-aware Systems in Idris: The ST Tutorial"[11]

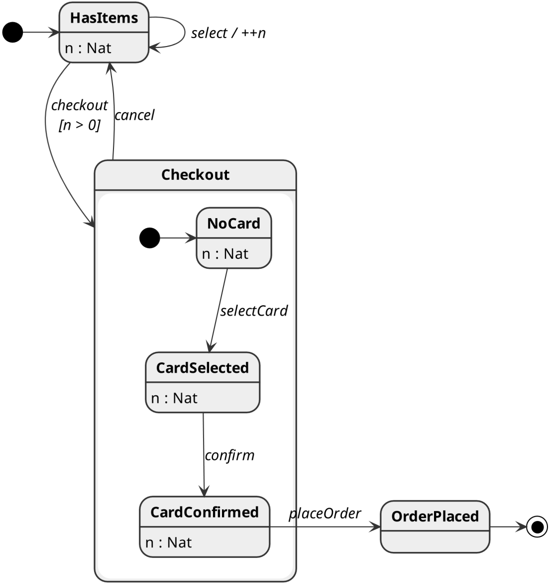
[11] http://docs.idris-lang.org/en/latest/st/index.html

Revisiting Checkout



@owickstrom

Extended State HasItems



Protocol Namespace

```
namespace Protocol

  Item : Type
  Item = String

  Items : Nat -> Type
  Items n = Vect n Item

  Card : Type
  Card = String

  OrderId : Type
  OrderId = String

  ...
```

Checkout States

```haskell
data CheckoutState
  = HasItems Nat
  | NoCard Nat
  | CardEntered Nat
  | CardConfirmed Nat
  | OrderPlaced
```

Checkout Interface

```
interface Checkout (m : Type -> Type) where
  State : CheckoutState -> Type

  ...
```

Initial State

```
initial
  : ST m Var [add (State (HasItems 0))]
```

One More Item

```
select
  : (c : Var)
  -> Item
  -> ST m () [c ::: State (HasItems n)
                    :-> State (HasItems (S n))]
```

Checking Out Requires Items

```
checkout
  : (c : Var)
  -> ST m () [c ::: State (HasItems (S n))
                    :-> State (NoCard (S n))]
```

States Accepting Cancel

- Again, we have *three* states accepting cancel
- In Idris we can express this using a predicate over states
- “Give me proof that your current state accepts cancel”

Cancellable State Predicate

```
data CancelState : CheckoutState -> (n : Nat) -> Type where
  NoCardCancel : CancelState (NoCard n) n
  CardEnteredCancel : CancelState (CardEntered n) n
  CardConfirmedCancel : CancelState (CardConfirmed n) n
```

Cancelling

```
cancel
  : (c : Var)
  -> { auto prf : CancelState s n }
  -> ST m () [c ::: State s
                    :-> State (HasItems n)]
```

Console Checkout Program

```
total
selectMore
  : (c : Var)
  -> ST m () [c ::: State {m} (HasItems n)
                    :-> State {m} (HasItems (S n))]
selectMore c {n} = do
  if n == 0
    then putStrLn "What do you want to add?"
    else putStrLn "What more do you want to add?"
  item <- getStr
  select c item
```

Console Checkout Program (cont.)

```
total
checkoutWithItems
  : (c : Var)
  -> ST m Bool [c ::: State {m} (HasItems (S n))
                     :-> (State {m} OrderPlaced
                          `orElse`
                          State {m} (HasItems (S n)))]
checkoutWithItems c = do
  checkout c
  True <- continueOrCancel c | False => pure False
  putStrLn "Enter your card:"
  selectCard c !getStr
  True <- continueOrCancel c | False => pure False
  confirm c
  True <- continueOrCancel c | False => pure False
  placeOrder c
  pure True
```

Console Checkout Program (cont.)

```
total
checkoutOrShop
  : (c : Var)
  -> STLoop m () [remove c (State {m} (HasItems (S n)))]
checkoutOrShop c = do
  True <- checkoutWithItems c | False => goShopping c
  orderId <- end c
  putStrLn ("Checkout complete with order ID: " ++ orderId)
  pure ()
```

Console Checkout Program (cont.)

```
total
goShopping
  : (c : Var)
  -> STLoop m () [remove c (State {m} (HasItems n))]
goShopping c = do
  selectMore c
  putStrLn "Checkout? (y/n)"
  case !getStr of
    "y" => checkoutOrShop c
    _ => goShopping c
```

Console Checkout Program (cont.)

```
total
program : STransLoop m () [] (const [])
program = do
  c <- initial
  goShopping c
```

Console Checkout Program (cont.)

```
runCheckout : IO ()
runCheckout =
  runLoop forever program (putStrLn "Oops.")
```

Summary

Summary

- Implicit state is hard and unsafe when it grows
  - Very unclear, no documentation of states and transitions
  - “Better safe than sorry” checks all over the place
- Just making the states explicit is a win
  - You probably have “hidden” state machines in your code
  - Use data types for states and events (ADTs)
  - This can be done in most mainstream languages!

Summary (cont.)

- By lifting more information to types, we can get more safety
    - You can do *a lot* in Haskell and PureScript
    - Protect side-effects with checked state transitions
    - Even better documentation
    - Make critical code testable
- Steal ideas from other languages
    - Dependent types, linear types
- Start simple!

Takeaway

Reify your design in code.

Questions?

Links

- Slides and code:
  github.com/owickstrom/fsm-your-compiler-wants-in
- Website: https://wickstrom.tech
- Twitter: @owickstrom