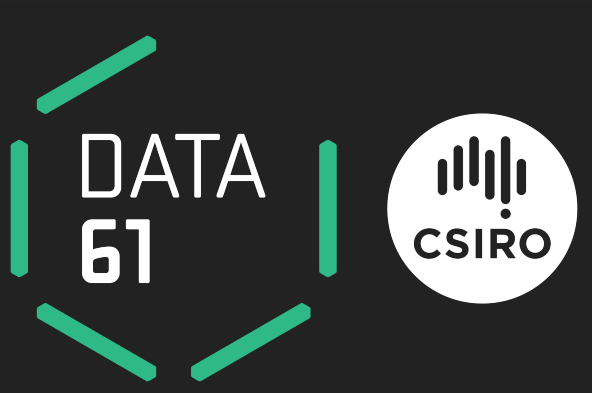
PROPERTY BASED STATE MACHINE TESTING

Andrew McCluskey

2018-05-22



WHAT AND WHY?

```haskell
-- Reverse is involutive
propReverse :: Property
propReverse =
  property $ do
    xs <- forAll $ Gen.list (Range.linear 0 100) Gen.alpha
    reverse (reverse xs) === xs
```

What about testing the properties of a whole system?

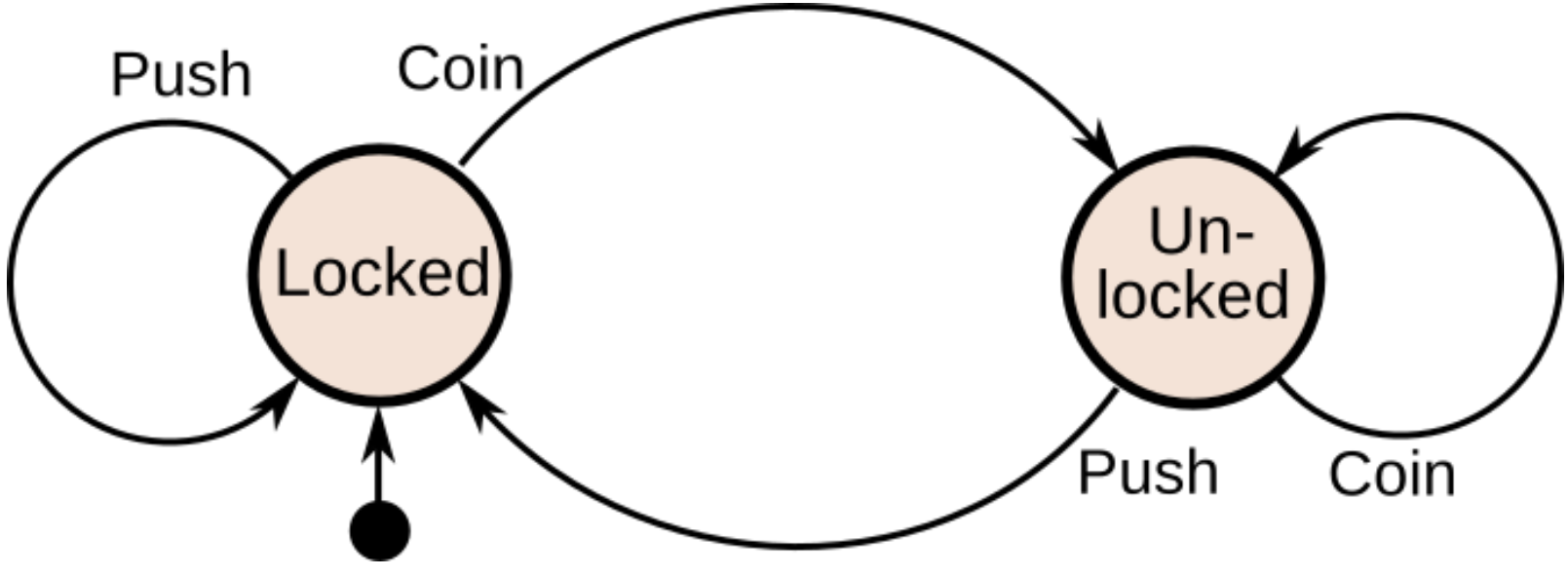
- Retrieved data matches submitted data.
- Uniqueness constraints.
- Users only see what they’re supposed to.

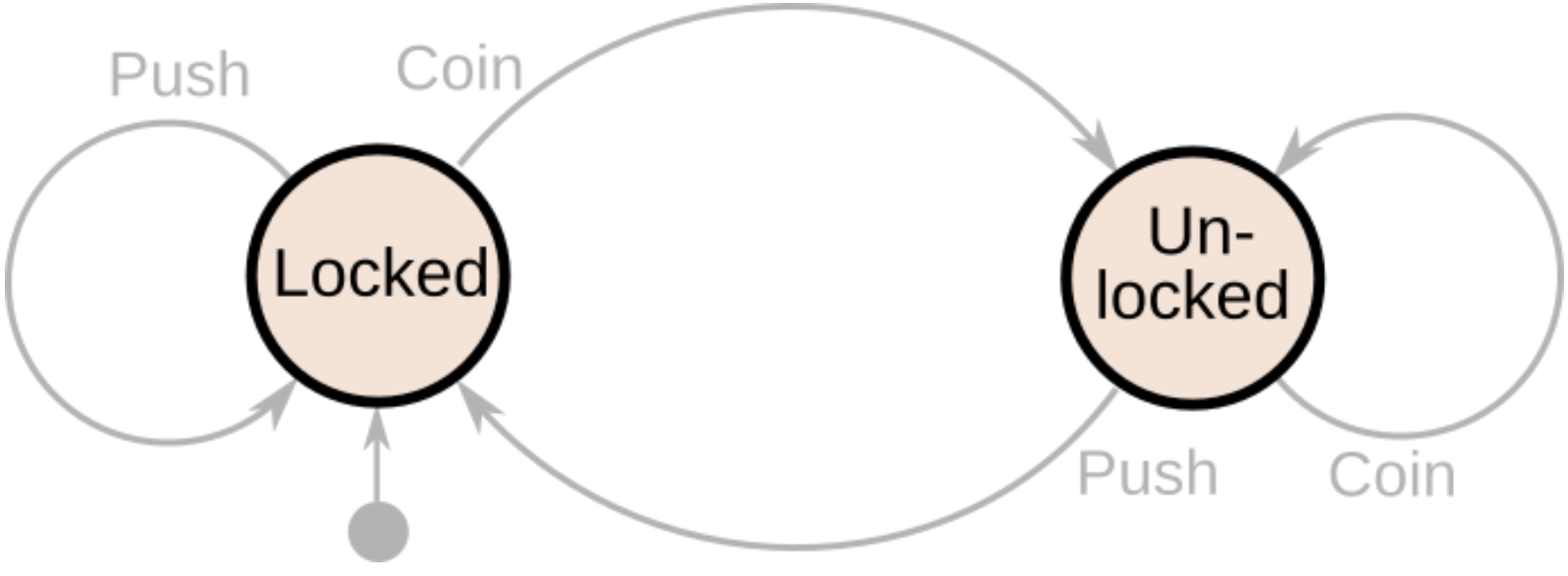
STATE MACHINE TESTING!

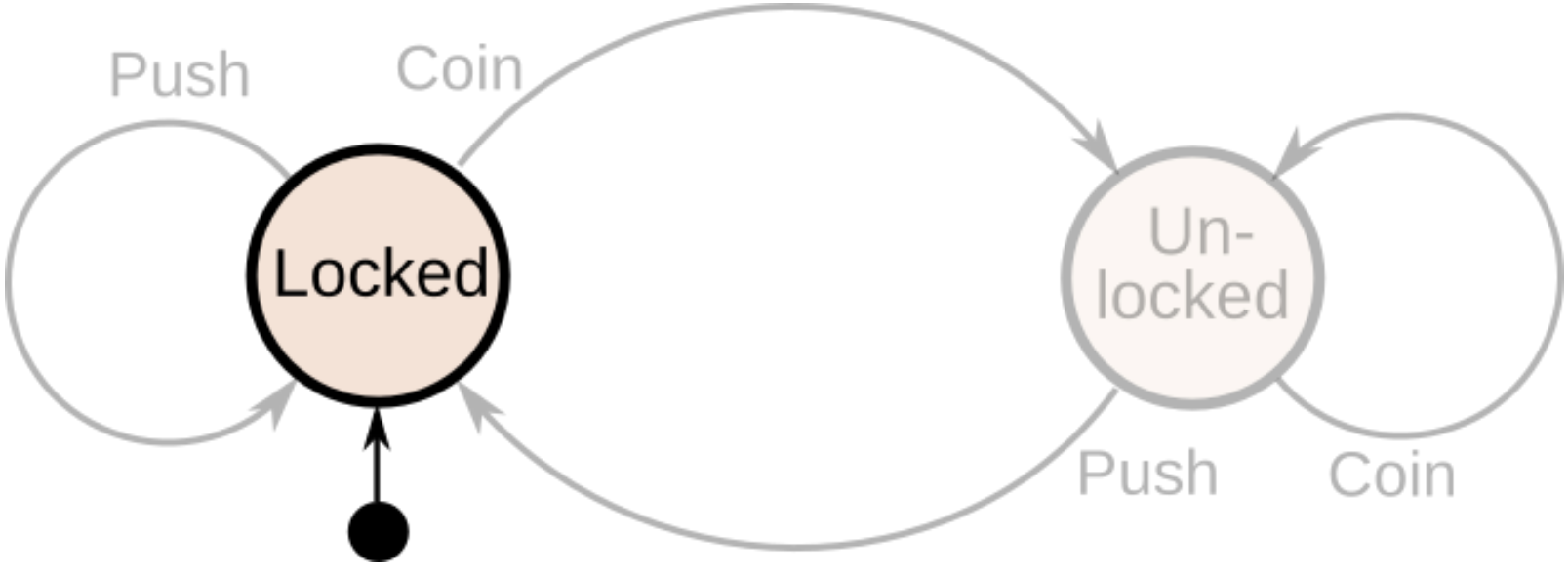
THE PLAN

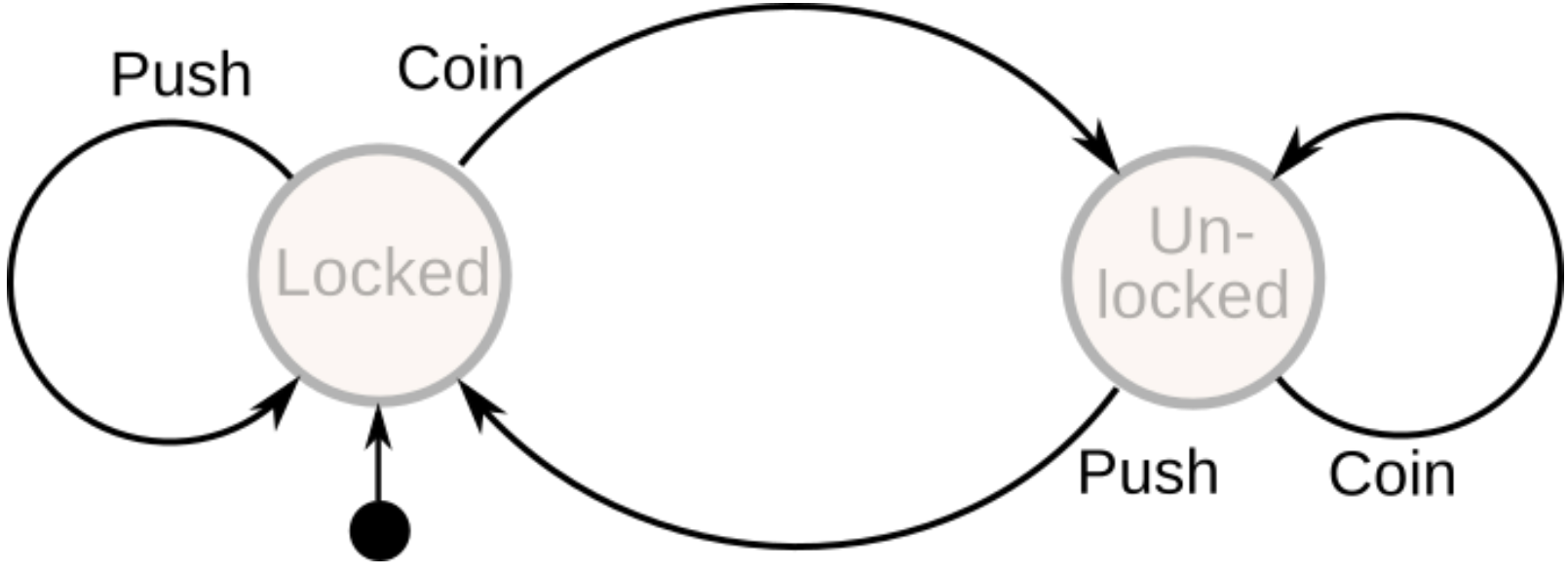
- State machines
- Property based testing *for* state machines
- Examples

STATE MACHINES


Push
Coin
Locked
Un-
locked
Push
Coin


Push
Coin
Locked
Un-
locked
Push
Coin


Push
Coin
Locked
Un-
locked
Push
Coin


Push
Coin
Locked
Un-
locked
Push
Coin

What else can be modelled as a state machine?

- Games
- Routing
- Web applications

WEB APP STATE MACHINE

States	Database + in-memory
Inputs	HTTP requests
Initial state	App before any requests

LEADERBOARD

Keeps track of game scores and player ratings.

API

/player/register-first
/player/register
/player/me
/player/player-count

PROPERTIES

- Can only `register-first` once
- Player count matches number of successful registrations
- Retrieving a player matches what was POSTed

STATE MACHINE TESTING

BIG PICTURE

- Use a state machine to model an app.
- Randomly generate inputs.
- Execute inputs against the app.
- Update the model.
- Check that the model agrees with reality.

STATE

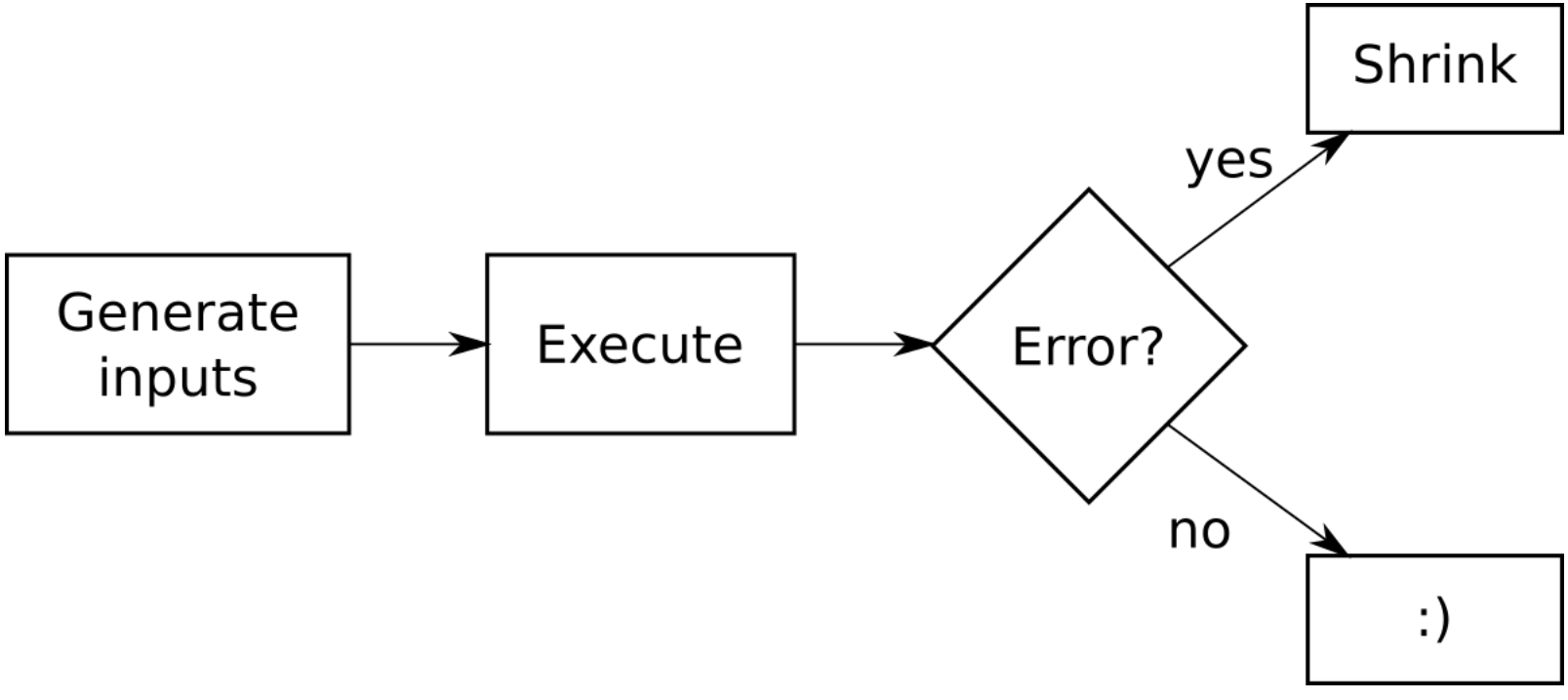
```
data LeaderboardState (v :: * -> *) =
  LeaderboardState Bool
```

INPUTS

```
data PlayerCount (v :: * -> *) =
  PlayerCount

data RegisterFirst (v :: * -> *) =
  RegisterFirst Registration

data Register (v :: * -> *) =
  Register AdminToken Registration
```

INITIAL STATE

```
initialState :: LeaderboardState v
initialState = LeaderboardState False
```


Generate inputs
Execute
Error?
yes
Shrink
no
:)

SOMETHING TO CONSIDER

All inputs are generated before execution.

All inputs are generated before execution.

```
token1 <- RegisterFirst registration1
```

All inputs are generated before execution.

```
token1 <- RegisterFirst registration1
token2 <- Register token1 registration2
```

SOLUTION

```
data Var a v
```

SOLUTION

```
data Var a v

Var a Symbolic
Var a Concrete
```

SOLUTION

```
data Var a v

Var a Symbolic
Var a Concrete

Symbolic :: * -> *
Concrete :: * -> *
```

COMMANDS

Bundle together:

- Generation of an input.
- Execution of an input.
- Preconditions.
- State updates.
- Postconditions / assertions.

```
data Command n m state =
  forall input output.
  Command {




  }
```

```
data Command n m state =
  forall input output.
  Command {
    commandGen ::
      state Symbolic -> Maybe (n (input Symbolic))




  }
```

```
data Command n m state =
  forall input output.
  Command {
    commandGen ::
      state Symbolic -> Maybe (n (input Symbolic))

  , commandExecute ::
      input Concrete -> m output



  }
```

```
data Command n m state =
  forall input output.
  Command {
    commandGen ::
      state Symbolic -> Maybe (n (input Symbolic))

  , commandExecute ::
      input Concrete -> m output

  , commandCallbacks ::
      [Callback input output state]
  }
```

```
data Callback input output state =
    Require ( state Symbolic
              -> input Symbolic
              -> Bool)
```

```
data Callback input output state =
    Require ( state Symbolic
              -> input Symbolic
              -> Bool)
  | Update ( forall v. Ord1 v
             => state v
             -> input v
             -> Var output v
             -> state v)
```

```
data Callback input output state =
    Require ( state Symbolic
              -> input Symbolic
              -> Bool)
  | Update ( forall v. Ord1 v
             => state v
             -> input v
             -> Var output v
             -> state v)
  | Ensure ( state Concrete
             -> state Concrete
             -> input Concrete
             -> output
             -> Test ())
```

HTRAVERSABLE

```haskell
class HTraversable t where
  htraverse
    :: Applicative f
    => (forall a. g a -> f (h a)) -> t g -> f (t h)
```

```
(             a -> f b    ) -> t a -> f (t b)

(forall a. g a -> f (h a)) -> t g -> f (t h)
```

```
htraverse
  :: (forall a. Symbolic a -> Either e (Concrete a))
  -> t Symbolic
  -> Either e (t Concrete)
```

```
data PlayerCount (v :: * -> *) =
  PlayerCount

data RegisterFirst (v :: * -> *) =
  RegisterFirst Registration

data Register (v :: * -> *) =
  Register AdminToken Registration
```

EXAMPLE1: REGISTER-FIRST

PROPERTY:

Registering our first player only succeeds once

STATE

```
newtype SimpleState (v :: * -> *) =
  SimpleState Bool
```

INPUTS

```
newtype RegFirst (v :: * -> *) =
    RegFirst RegisterPlayer
  deriving (Eq, Show)
```

INPUTS

```
newtype RegFirst (v :: * -> *) =
    RegFirst RegisterPlayer
  deriving (Eq, Show)

newtype RegFirstForbidden (v :: * -> *) =
  RegFirstForbidden RegisterPlayer
  deriving (Eq, Show)
```

cRegisterFirst

```
cRegisterFirst
  :: ( MonadGen n
     , MonadIO m
     , MonadTest m
     )
  => ClientEnv
  -> Command n m SimpleState
cRegisterFirst env =
  Command cRegisterFirstGen
          (cRegisterFirstExe env)
          cRegisterFirstCallbacks
```

```
cRegisterFirstGen
  :: MonadGen n
  => SimpleState Symbolic
  -> Maybe (n (RegFirst Symbolic))
```

```haskell
cRegisterFirstGen
  :: MonadGen n
  => SimpleState Symbolic
  -> Maybe (n (RegFirst Symbolic))
cRegisterFirstGen (SimpleState registeredFirst) =
  if registeredFirst
  then Nothing
  else Just (RegFirst <$> genRegPlayerRandomAdmin)
```

```
cRegisterFirstExe
  :: ( MonadIO m
     , MonadTest m
     )
  => ClientEnv
  -> RegFirst Concrete
  -> m ResponsePlayer
```

```haskell
cRegisterFirstExe
  :: ( MonadIO m
     , MonadTest m
     )
  => ClientEnv
  -> RegFirst Concrete
  -> m ResponsePlayer
cRegisterFirstExe env (RegFirst rp) =
  evalEither =<< successClient env (registerFirst rp)
```

```
successClient
  :: MonadIO m
  => ClientEnv -> ClientM a -> m (Either ServantError a)

evalEither
  :: (MonadTest m, Show x, HasCallStack)
  => Either x a -> m a

  evalEither =<< successClient env (registerFirst rp)
```

```
cRegisterFirstCallbacks
  :: [Callback RegFirst ResponsePlayer SimpleState]
```

```
cRegisterFirstCallbacks
  :: [Callback RegFirst ResponsePlayer SimpleState]
cRegisterFirstCallbacks =
  [ Require $ \(SimpleState registeredFirst) _i ->
      not registeredFirst









  ]
```

```haskell
cRegisterFirstCallbacks
  :: [Callback RegFirst ResponsePlayer SimpleState]
cRegisterFirstCallbacks =
  [ Require $ \(SimpleState registeredFirst) _i ->
      not registeredFirst
  , Update $ \_sOld _i _o -> SimpleState True









  ]
```

```
cRegisterFirstCallbacks
  :: [Callback RegFirst ResponsePlayer SimpleState]
cRegisterFirstCallbacks =
  [ Require $ \(SimpleState registeredFirst) _i ->
      not registeredFirst
  , Update $ \_sOld _i _o -> SimpleState True
  , Ensure $ \_sOld _sNew _input out ->
      case out of
        (ResponsePlayer (LS.PlayerId (Auto mId))
                        (Token token)) -> do
          assert $ not (BS.null t)
          assert $ maybe False (>= 0) mId
  ]
```

cRegisterFirstForbidden

```haskell
cRegisterFirstForbiddenGen
  :: MonadGen n
  => SimpleState Symbolic
  -> Maybe (n (RegFirstForbidden Symbolic))
```

```haskell
cRegisterFirstForbiddenGen
  :: MonadGen n
  => SimpleState Symbolic
  -> Maybe (n (RegFirstForbidden Symbolic))
cRegisterFirstForbiddenGen (SimpleState registeredFirst) =
  if registeredFirst
  then Just (RegFirstForbidden <$> genRegPlayerRandomAdmin)
  else Nothing
```

```
cRegisterFirstForbiddenExe
  :: ( MonadIO m
     , MonadTest m
     )
  => ClientEnv
  -> RegFirstForbidden Concrete
  -> m ServantError
```

```haskell
cRegisterFirstForbiddenExe
  :: ( MonadIO m
     , MonadTest m
     )
  => ClientEnv
  -> RegFirstForbidden Concrete
  -> m ServantError
cRegisterFirstForbiddenExe env (RegFirstForbidden rp) =
  evalEither =<< failureClient env (registerFirst rp)
```

```
cRegisterFirstForbiddenCallbacks
  :: [Callback RegFirstForbidden ServantError SimpleState]
```

```haskell
cRegisterFirstForbiddenCallbacks
  :: [Callback RegFirstForbidden ServantError SimpleState]
cRegisterFirstForbiddenCallbacks = [
    Require $ \(SimpleState registeredFirst) _input ->
      registeredFirst




  ]
```

```haskell
cRegisterFirstForbiddenCallbacks
  :: [Callback RegFirstForbidden ServantError SimpleState]
cRegisterFirstForbiddenCallbacks = [
    Require $ \(SimpleState registeredFirst) _input ->
      registeredFirst
  , Ensure $ \_sOld _sNew _input se ->
      case se of
        FailureResponse{..} ->
          responseStatus === forbidden403
        _ -> failure
  ]
```

propRegisterFirst

```haskell
propRegisterFirst :: ClientEnv -> IO () -> TestTree
propRegisterFirst env reset =
```

```haskell
propRegisterFirst :: ClientEnv -> IO () -> TestTree
propRegisterFirst env reset =
  testProperty "register-first" . property $ do
    let
      initialState = SimpleState False
```

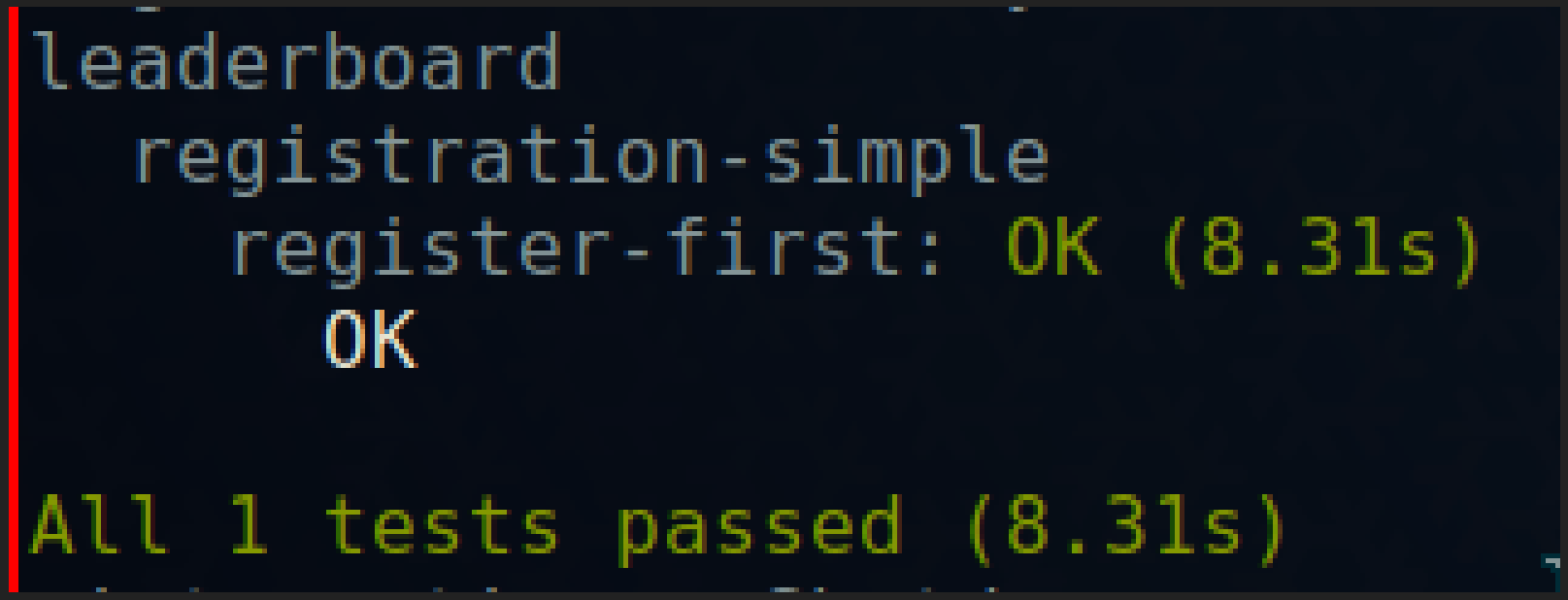
```haskell
propRegisterFirst :: ClientEnv -> IO () -> TestTree
propRegisterFirst env reset =
  testProperty "register-first" . property $ do
    let
      initialState = SimpleState False
      cs = [ cRegisterFirst env
           , cRegisterFirstForbidden env]
```

```haskell
propRegisterFirst :: ClientEnv -> IO () -> TestTree
propRegisterFirst env reset =
  testProperty "register-first" . property $ do
    let
      initialState = SimpleState False
      cs = [ cRegisterFirst env
           , cRegisterFirstForbidden env]
    actions <- forAll $
      Gen.sequential (Range.linear 1 100) initialState cs
```

```haskell
propRegisterFirst :: ClientEnv -> IO () -> TestTree
propRegisterFirst env reset =
  testProperty "register-first" . property $ do
    let
      initialState = SimpleState False
      cs = [ cRegisterFirst env
           , cRegisterFirstForbidden env]
    actions <- forAll $
      Gen.sequential (Range.linear 1 100) initialState cs

    liftIO reset
    executeSequential initialState actions
```

TEST SETUP

- Start a temporary database instance
- Fork a thread to run our server (starting with DB migration)
- Run properties with `ClientEnv`

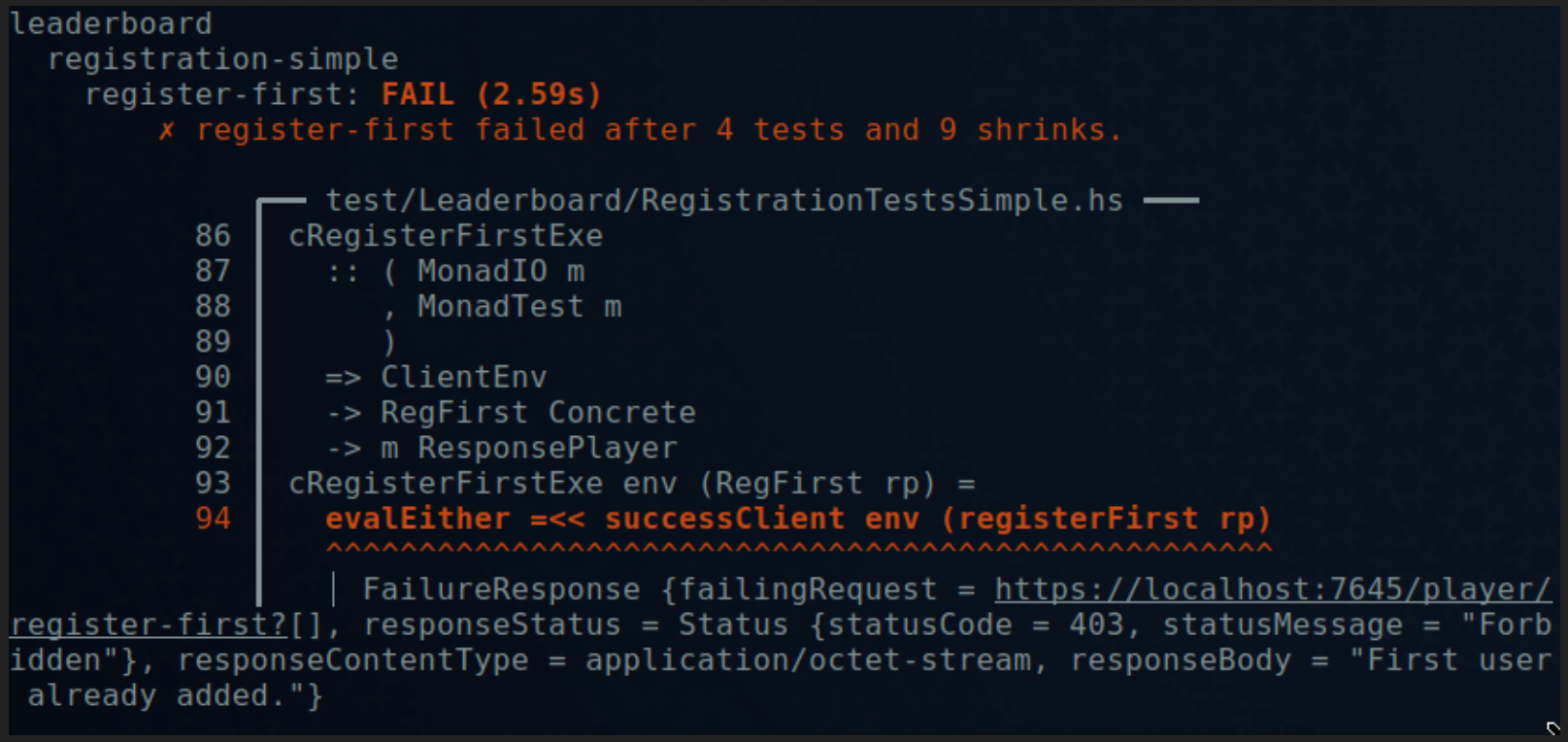

leaderboard
registration-simple
register-first: OK (8.31s)
OK
All 1 tests passed (8.31s)

```haskell
cRegisterFirstGen (SimpleState registeredFirst) =
  if registeredFirst
  then Nothing
  else Just (RegFirst <$> genRegPlayerRandomAdmin)
```

```
cRegisterFirstGen (SimpleState registeredFirst) =
  Just (RegFirst <$> genRegPlayerRandomAdmin)
```

```haskell
cRegisterFirstCallbacks =
  [ Require $ \(SimpleState registeredFirst) _i ->
      not registeredFirst
  , Update $ \_sOld _i _o -> SimpleState True
  , Ensure $ \_sOld _sNew _input out ->
      case out of
        (ResponsePlayer (LS.PlayerId (Auto mId))
                        (Token token)) -> do
          assert $ not (BS.null t)
          assert $ maybe False (>= 0) mId
  ]
```

```
cRegisterFirstCallbacks =
  [

    Update $ \_sOld _i _o -> SimpleState True
  , Ensure $ \_sOld _sNew _input out ->
      case out of
        (ResponsePlayer (LS.PlayerId (Auto mId))
                        (Token token)) -> do
          assert $ not (BS.null t)
          assert $ maybe False (>= 0) mId
  ]
```

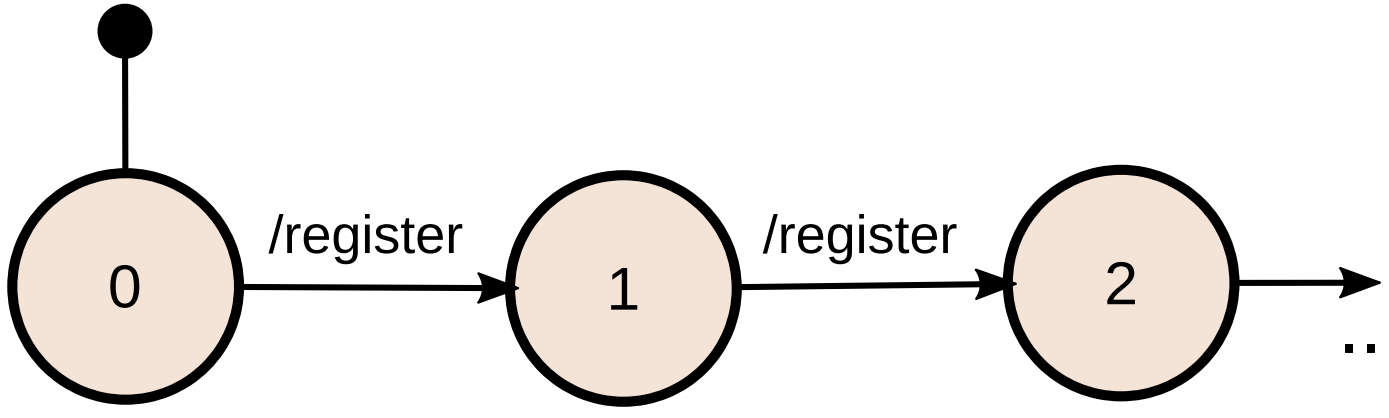

leaderboard
registration-simple
register-first: FAIL (2.59s)
✗ register-first failed after 4 tests and 9 shrinks.
test/Leaderboard/RegistrationTestsSimple.hs
86 cRegisterFirstExe
87 :: (MonadIO m
88 , MonadTest m
89)
90 => ClientEnv
91 -> RegFirst Concrete
92 -> m ResponsePlayer
93 cRegisterFirstExe env (RegFirst rp) =
94 evalEither =<< successClient env (registerFirst rp)
| FailureResponse {failingRequest = https://localhost:7645/player/register-first?[], responseStatus = Status {statusCode = 403, statusMessage = "Forbidden"}, responseContentType = application/octet-stream, responseBody = "First user already added."}

```
156   actions <- forAll $
157     Gen.sequential (Range.linear 1 100) initialState commands
        | Var 0 = RegFirst
        |             LeaderboardRegistration
        |               { _lbrEmail = "a"
        |               , _lbrUsername = "a"
        |               , _lbrPassword = "a"
        |               , _lbrIsAdmin = Nothing
        |               }
        | Var 1 = RegFirst
        |             LeaderboardRegistration
        |               { _lbrEmail = "a"
        |               , _lbrUsername = "a"
        |               , _lbrPassword = "a"
        |               , _lbrIsAdmin = Nothing
        |               }
```

EXAMPLE 2: PLAYER COUNT

PROPERTY

Successfully registering a player increments the player count.


0
/register
1
/register
2
...

STATE

```haskell
data LeaderboardState (v :: * -> *) =
  LeaderboardState
  { _players :: M.Map Text (PlayerWithRsp v)
  , _admins  :: S.Set Text
  }
deriving instance Show1 v => Show (LeaderboardState v)
deriving instance Eq1 v => Eq (LeaderboardState v)
```

```haskell
data PlayerWithRsp v =
  PlayerWithRsp
  { _pwrRsp       :: Var ResponsePlayer v



  }
  deriving (Eq, Show)
```

```haskell
data PlayerWithRsp v =
  PlayerWithRsp
  { _pwrRsp      :: Var ResponsePlayer v
  , _pwrEmail    :: Text
  , _pwrUsername :: Text
  , _pwrPassword :: Text
  , _pwrIsAdmin  :: Maybe Bool
  }
  deriving (Eq, Show)
```

COMMANDS

cRegFirst

```
Update $
  \_ (RegFirst lbr@LeaderboardRegistration{..}) rsp ->
```

```
Update $
  \_ (RegFirst lbr@LeaderboardRegistration{..}) rsp ->
    let
      player = mkPlayerWithRsp lbr rsp
```

```
Update $
  \_ (RegFirst lbr@LeaderboardRegistration{..}) rsp ->
     let
       player = mkPlayerWithRsp lbr rsp
       players = M.singleton _lbrEmail player
```

```
Update $
  \_ (RegFirst lbr@LeaderboardRegistration{..}) rsp ->
     let
       player = mkPlayerWithRsp lbr rsp
       players = M.singleton _lbrEmail player
       admins = S.singleton _lbrEmail
```

```
Update $
  \_ (RegFirst lbr@LeaderboardRegistration{..}) rsp ->
    let
      player = mkPlayerWithRsp lbr rsp
      players = M.singleton _lbrEmail player
      admins = S.singleton _lbrEmail
    in
      LeaderboardState players admins
```

cGetPlayerCount

```haskell
data GetPlayerCount (v :: * -> *) =
    GetPlayerCount
  deriving (Eq, Show)
```

```
Ensure $ \(LeaderboardState ps as) _sNew _i c -> do
  length ps === fromIntegral c
  assert $ length ps >= length as
```

cRegister

```haskell
data Register (v :: * -> *) =
  Register RegisterPlayer (PlayerWithRsp v)
  deriving (Eq, Show)
```

```
cRegisterGen rs@(LeaderboardState ps as) =
  if null as
  then Nothing
```

```
cRegisterGen rs@(LeaderboardState ps as) =
  if null as
  then Nothing
  else
    let
      maybeGenAdmin :: Maybe (n (PlayerWithRsp v))
      maybeGenAdmin =
        pure $ (M.!) ps <$> (Gen.element . S.toList $ as)
```

```haskell
cRegisterGen rs@(LeaderboardState ps as) =
  if null as
  then Nothing
  else
    let
      maybeGenAdmin :: Maybe (n (PlayerWithRsp v))
      maybeGenAdmin =
        pure $ (M.!) ps <$> (Gen.element . S.toList $ as)
    in
      (Register <$> genRegPlayerRandomAdmin ps <*>)
        <$> maybeGenAdmin
```

```haskell
cRegisterExecute
  :: ( MonadIO m
     , MonadTest m
     )
  => ClientEnv
  -> Register Concrete
  -> m ResponsePlayer
cRegisterExecute env (Register rp p) =
  let
    t = clientToken p
  in
    evalEither =<< successClient env (register t rp)
```

```
Require $ \(LeaderboardState _ as) (Register _ p) ->
  S.member (_pwrEmail p) as
```

```
Update $
  \(LeaderboardState ps as)
   (Register rp@LeaderboardRegistration{..} _)
   rsp ->
```

```
Update $
  \(LeaderboardState ps as)
   (Register rp@LeaderboardRegistration{..} _)
   rsp ->
     let
       newPlayers =
         M.insert _lbrEmail (mkPlayerWithRsp rp rsp) ps
```

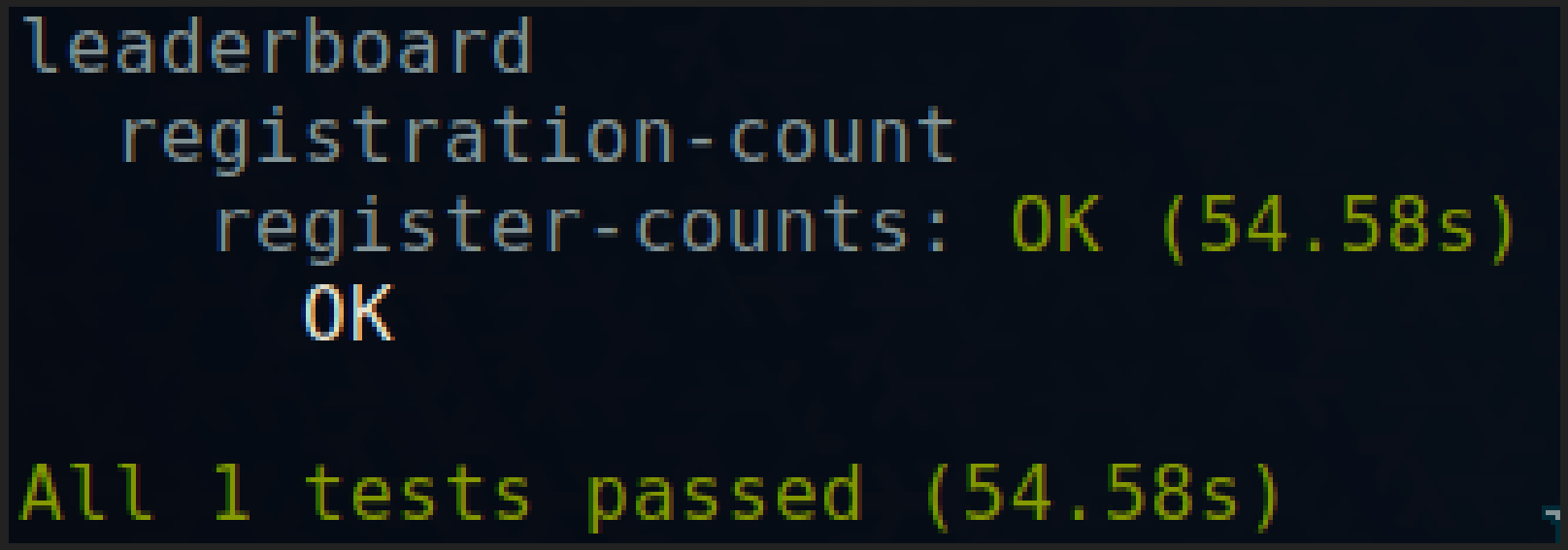
```
Update $
  \(LeaderboardState ps as)
   (Register rp@LeaderboardRegistration{..} _)
   rsp ->
     let
       newPlayers =
         M.insert _lbrEmail (mkPlayerWithRsp rp rsp) ps
       newAdmins =
         case _lbrIsAdmin of
           Just True -> S.insert _lbrEmail as
           _         -> as
```

```haskell
Update $
  \(LeaderboardState ps as)
   (Register rp@LeaderboardRegistration{..} _)
   rsp ->
     let
       newPlayers =
         M.insert _lbrEmail (mkPlayerWithRsp rp rsp) ps
       newAdmins =
         case _lbrIsAdmin of
           Just True -> S.insert _lbrEmail as
           _         -> as
     in
       LeaderboardState newPlayers newAdmins
```

```
propRegisterCount env reset =
  testProperty "register-counts" . property $ do
```

```haskell
propRegisterCount env reset =
  testProperty "register-counts" . property $ do
  let
    initialState = LeaderboardState M.empty S.empty
```

```haskell
propRegisterCount env reset =
  testProperty "register-counts" . property $ do
  let
    initialState = LeaderboardState M.empty S.empty
    commands = [
        cRegisterFirst env
      , cRegisterFirstForbidden env
      , cRegister env
      , cGetPlayerCount env
      ]
  -- rest is the same as register-first
```


leaderboard
registration-count
register-counts: OK (54.58s)
OK
All 1 tests passed (54.58s)

EXAMPLE 3: ROUND TRIP

PROPERTY

Data retrieved for a player matches data entered

cMe

```
cMe
  :: ( MonadGen n
     , MonadTest m
     , MonadIO m
     )
  => ClientEnv
  -> Command n m LeaderboardState
cMe env =
    Command cMeGen (cMeExecute env) cMeCallbacks
```

```haskell
cMeGen
  :: MonadGen n
  => LeaderboardState Symbolic
  -> Maybe (n (Me Symbolic))
cMeGen (LeaderboardState ps _) =
  (fmap . fmap) Me (genPlayerWithRsp ps)
```

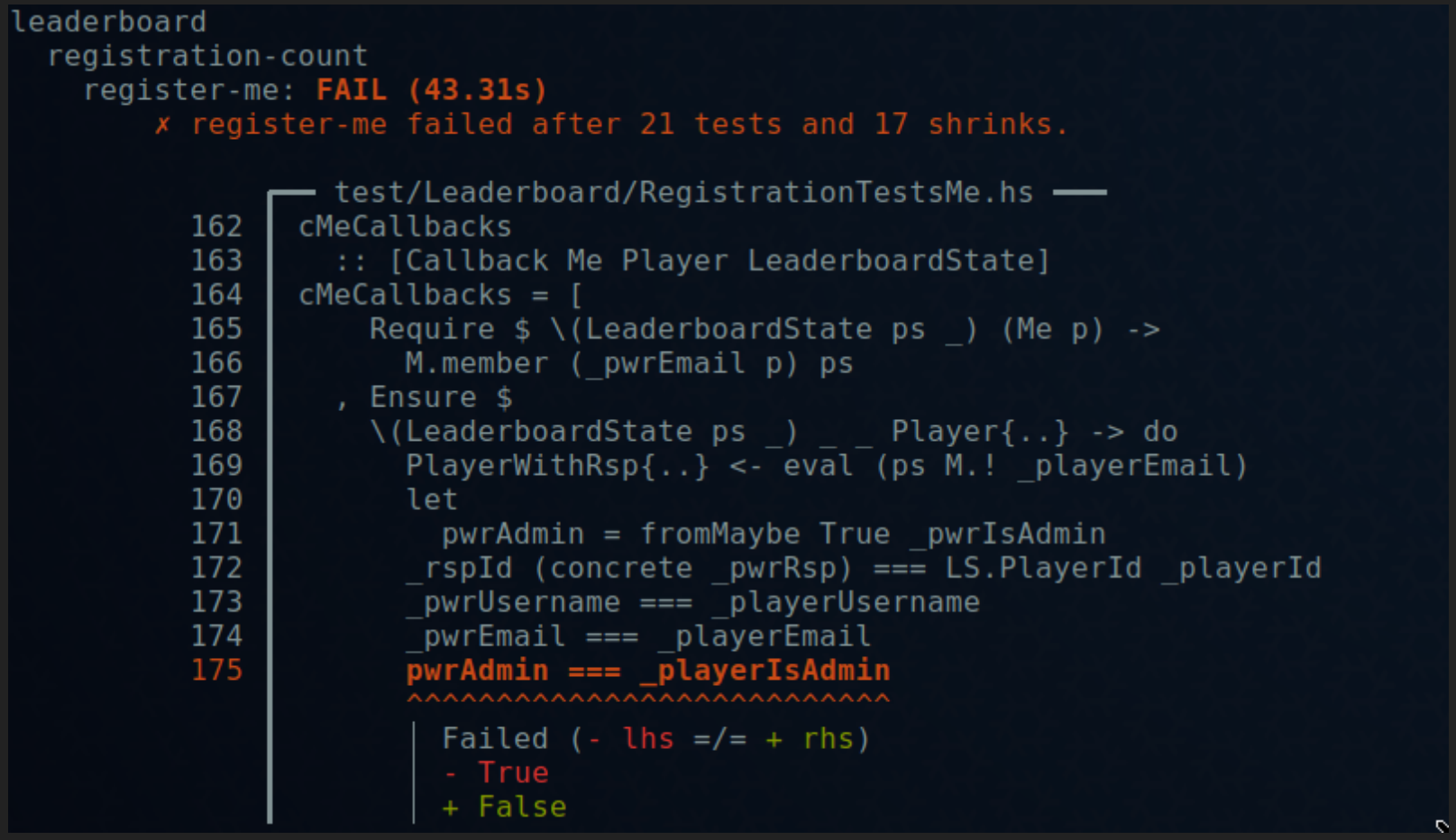
```haskell
cMeExecute
  :: ( MonadIO m
     , MonadTest m
     )
  => ClientEnv
  -> Me Concrete
  -> m Player
cMeExecute env (Me pwr) =
  evalEither =<< successClient env (me (clientToken pwr))
```

```
cMeCallbacks =
  [ Require $ \(LeaderboardState ps _) (Me p) ->
      M.member (_pwrEmail p) ps









  ]
```

```
cMeCallbacks =
  [ Require $ \(LeaderboardState ps _) (Me p) ->
      M.member (_pwrEmail p) ps
  , Ensure $
    \(LeaderboardState ps _) _ _ Player{..} -> do
      PlayerWithRsp{..} <- eval (ps M.! _playerEmail)









  ]
```

```
cMeCallbacks =
  [ Require $ \(LeaderboardState ps _) (Me p) ->
      M.member (_pwrEmail p) ps
  , Ensure $
    \(LeaderboardState ps _) _ _ Player{..} -> do
      PlayerWithRsp{..} <- eval (ps M.! _playerEmail)
      let
        pwrAdmin = fromMaybe True _pwrIsAdmin



  ]
```

```
cMeCallbacks =
  [ Require $ \(LeaderboardState ps _) (Me p) ->
      M.member (_pwrEmail p) ps
  , Ensure $
    \(LeaderboardState ps _) _ _ Player{..} -> do
      PlayerWithRsp{..} <- eval (ps M.! _playerEmail)
      let
        pwrAdmin = fromMaybe True _pwrIsAdmin
      _rspId (concrete _pwrRsp) === LS.PlayerId _playerId
      _pwrUsername === _playerUsername
      _pwrEmail === _playerEmail
      pwrAdmin === _playerIsAdmin
  ]
```

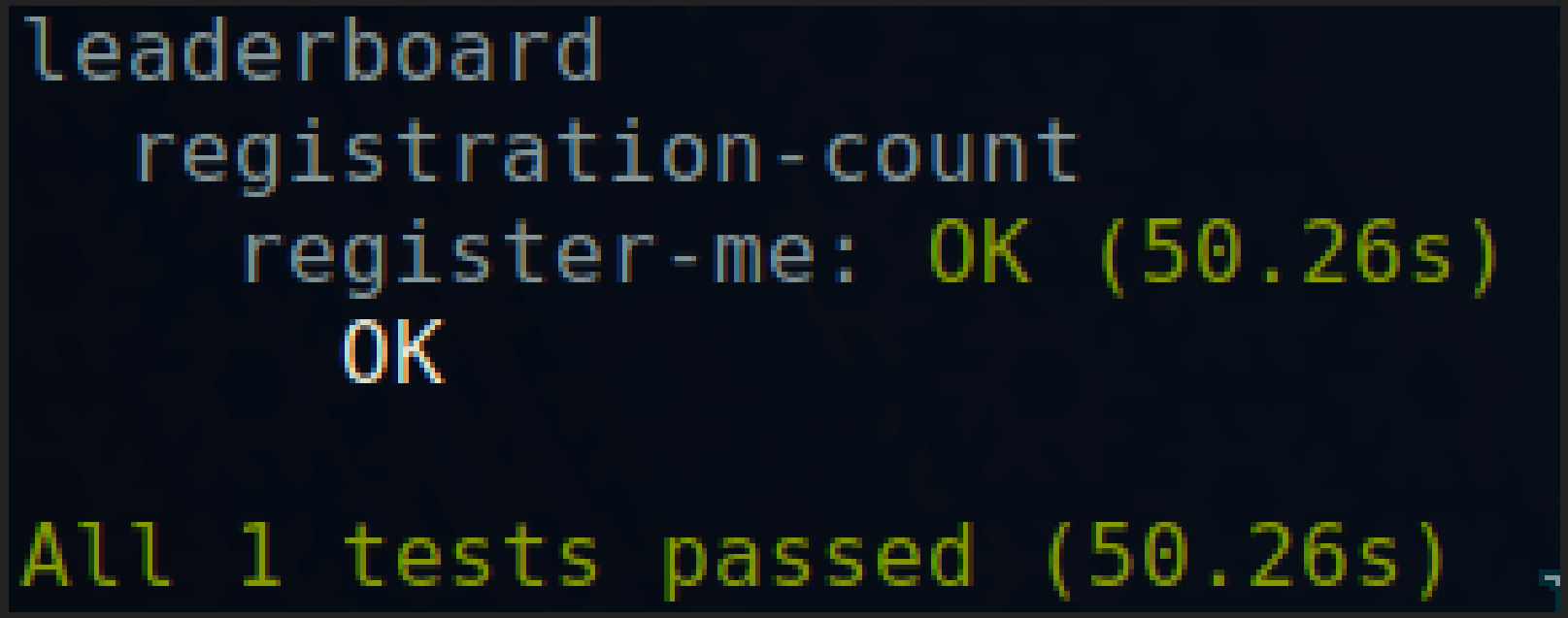

leaderboard
registration-count
register-me: FAIL (43.31s)
✗ register-me failed after 21 tests and 17 shrinks.
test/Leaderboard/RegistrationTestsMe.hs
162 cMeCallbacks
163 :: [Callback Me Player LeaderboardState]
164 cMeCallbacks = [
165 Require $ \(LeaderboardState ps _) (Me p) ->
166 M.member (_pwrEmail p) ps
167 , Ensure $
168 \(LeaderboardState ps _) _ _ Player{..} -> do
169 PlayerWithRsp{..} <- eval (ps M.! _playerEmail)
170 let
171 pwrAdmin = fromMaybe True _pwrIsAdmin
172 _rspId (concrete _pwrRsp) === LS.PlayerId _playerId
173 _pwrUsername === _playerUsername
174 _pwrEmail === _playerEmail
175 pwrAdmin === _playerIsAdmin
^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Failed (- lhs =/= + rhs)
- True
+ False

```
Var 0 = RegFirst
          LeaderboardRegistration
            { _lbrEmail = "SI"
            , _lbrUsername = "a"
            , _lbrPassword = "a"
            , _lbrIsAdmin = Nothing
            }
Var 6 = Register
          LeaderboardRegistration
            { _lbrEmail = "oumaM"
            , _lbrUsername = "a"
            , _lbrPassword = "a"
            , _lbrIsAdmin = Nothing
            }
          PlayerWithRsp
            { _pwrRsp = Var 0
            , _pwrEmail = "SI"
            , _pwrUsername = "pxmll"
            , _pwrPassword = "pyMxO"
            , _pwrIsAdmin = Just True
            }
Var 14 = Me
           PlayerWithRsp
             { _pwrRsp = Var 6
             , _pwrEmail = "oumaM"
             , _pwrUsername = "hXA0f"
             , _pwrPassword = "bp2L"
             , _pwrIsAdmin = Nothing
             }
```

```
cMeCallbacks =
  [ Require $ \(LeaderboardState ps _) (Me p) ->
      M.member (_pwrEmail p) ps
  , Ensure $
    \(LeaderboardState ps _) _ _ p@Player{..} -> do
      pwr@PlayerWithRsp{..} <- eval (ps M.! _playerEmail)
      let
        pwrAdmin = fromMaybe True _pwrIsAdmin
      _rspId (concrete _pwrRsp) === LS.PlayerId _playerId
      _pwrUsername === _playerUsername
      _pwrEmail === _playerEmail
      pwrAdmin === _playerIsAdmin
  ]
```

```
cMeCallbacks =
  [


        pwrAdmin = fromMaybe True _pwrIsAdmin



  ]
```

```
cMeCallbacks =
  [


        pwrAdmin = fromMaybe False _pwrIsAdmin



  ]
```


leaderboard
registration-count
register-me: OK (50.26s)
OK
All 1 tests passed (50.26s)

REFERENCES

THIS TALK

Slides
https://github.com/qfpl/state-machine-testing

Leaderboard:
https://github.com/qfpl/leaderboard

SOURCES

Hedgehog on GitHub
https://github.com/hedgehogqa/haskell-hedgehog

Tim Humphries' blog post
https://teh.id.au/posts/2017/07/15/state-machine-testing/index.html

IMAGES

Turnstile state machine
https://en.wikipedia.org/wiki/Finite-state_machine

Running hedgehog
https://www.buzzfeed.com/tomphillips/this-running-hedgehog-is-basically-a-real-life-sonic

Failing hedgehog
https://gifrific.com/hedgehog-running-on-wheel-fail/

Enter the hedgehog__
https://deanoinamerica.wordpress.com/2011/06/26/fro
basketball-to-the-way-of-the-dragon-nevermind/