Stateful Systems

John Hughes

QuickCheck Generators 101

- newtype Gen a = ...

A random generator for values of type a

- sample :: Show a => Gen a -> IO ()

A function that generates and displays random samples

choose

```
> sample $ choose (1,100)
23
25
40
51
39
17
45
98
88
90
68
```

elements :: [a] -> Gen a

```
> sample $ elements ["a","b","c"]
"b"
"b"
"a"
"c"
"c"
"a"
"c"
"b"
"c"
"b"
"b"
```

oneof :: [Gen a] -> Gen a

```
> sample $ oneof [choose (1,10),choose(100,1000)]
859
978
7
9
336
3
8
390
936
3
947
```

class Arbitrary a **where**
arbitrary :: Gen a

```
> sample (arbitrary :: Gen [Int])
[]
[-1]
[-4,-2]
[]
[1,3,-1,6,6]
[-4,2,-7,-10,-6,-8,-9,10,-8,-9]
[-3,9,-8,5,9]
[-7,0,14,7,-9,-11,-5,9,-12,9]
[15,11]
[-15,-12,-17,-12,-5,14,-12,16,16,4,-14]
[2,-6,10,7,11]
```

Default generators for a bunch of built-in types; extensible for each new type

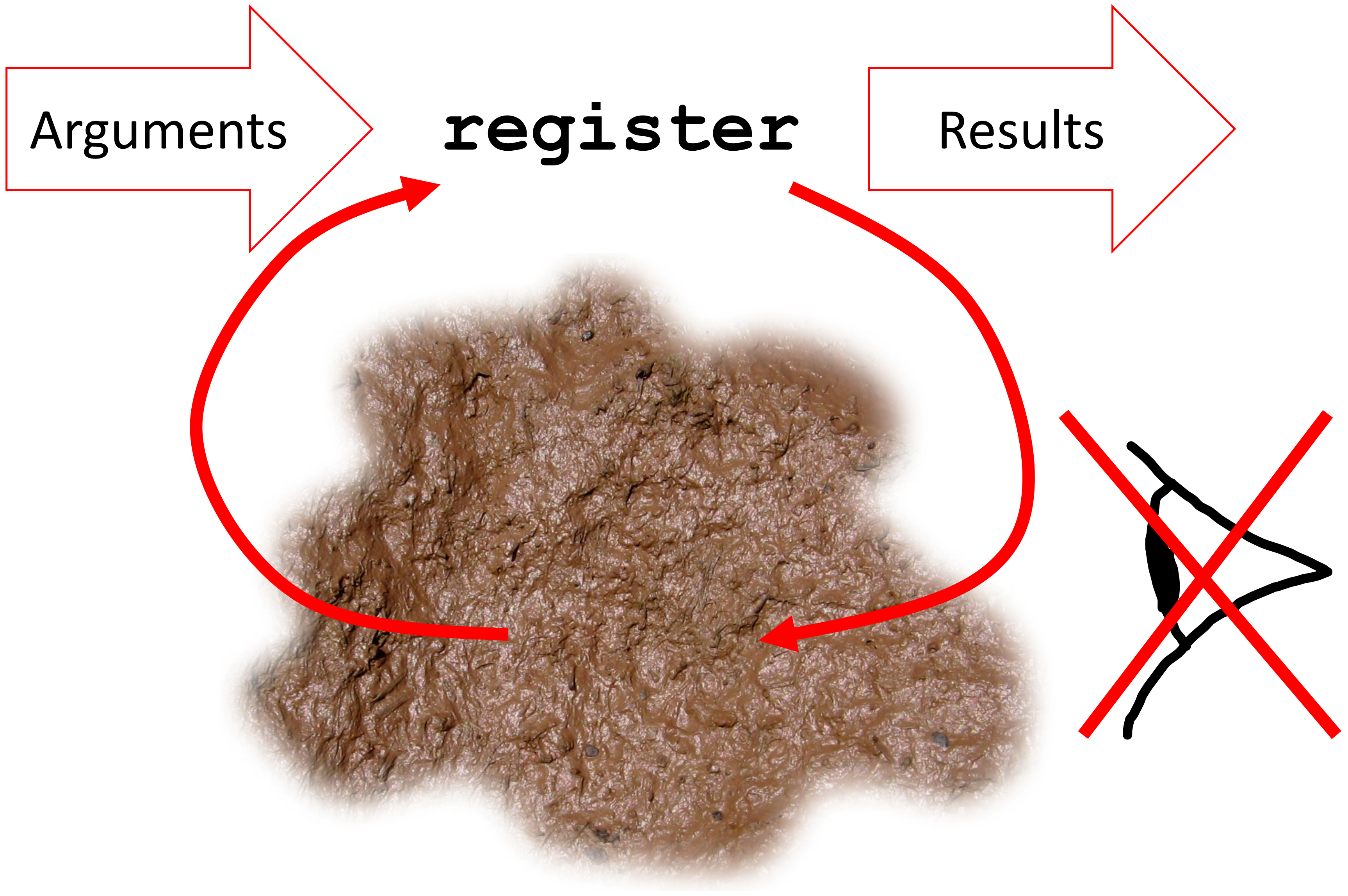
Example: a Process Registry

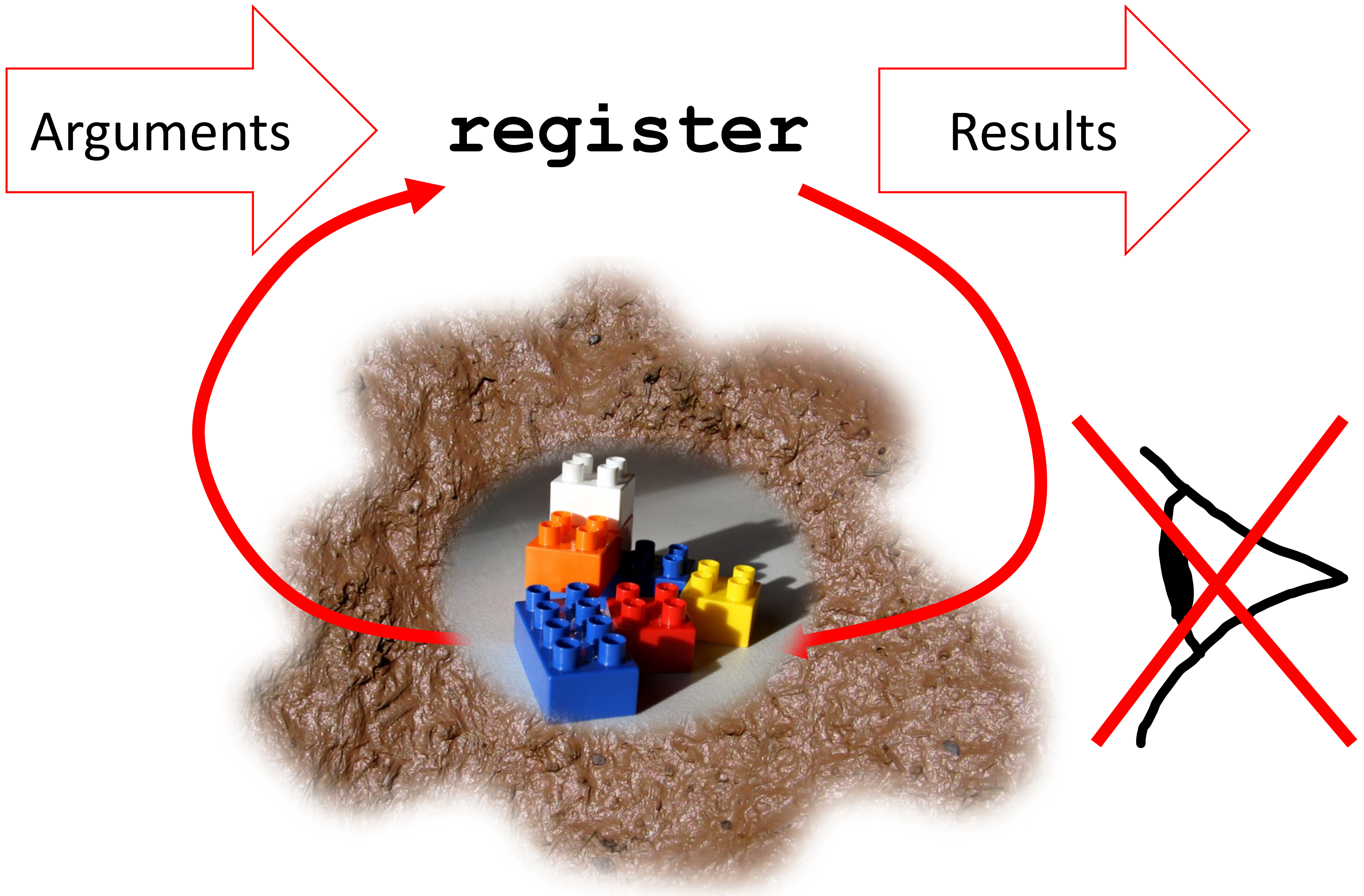
- Three operations [inspired by the Erlang process registry]

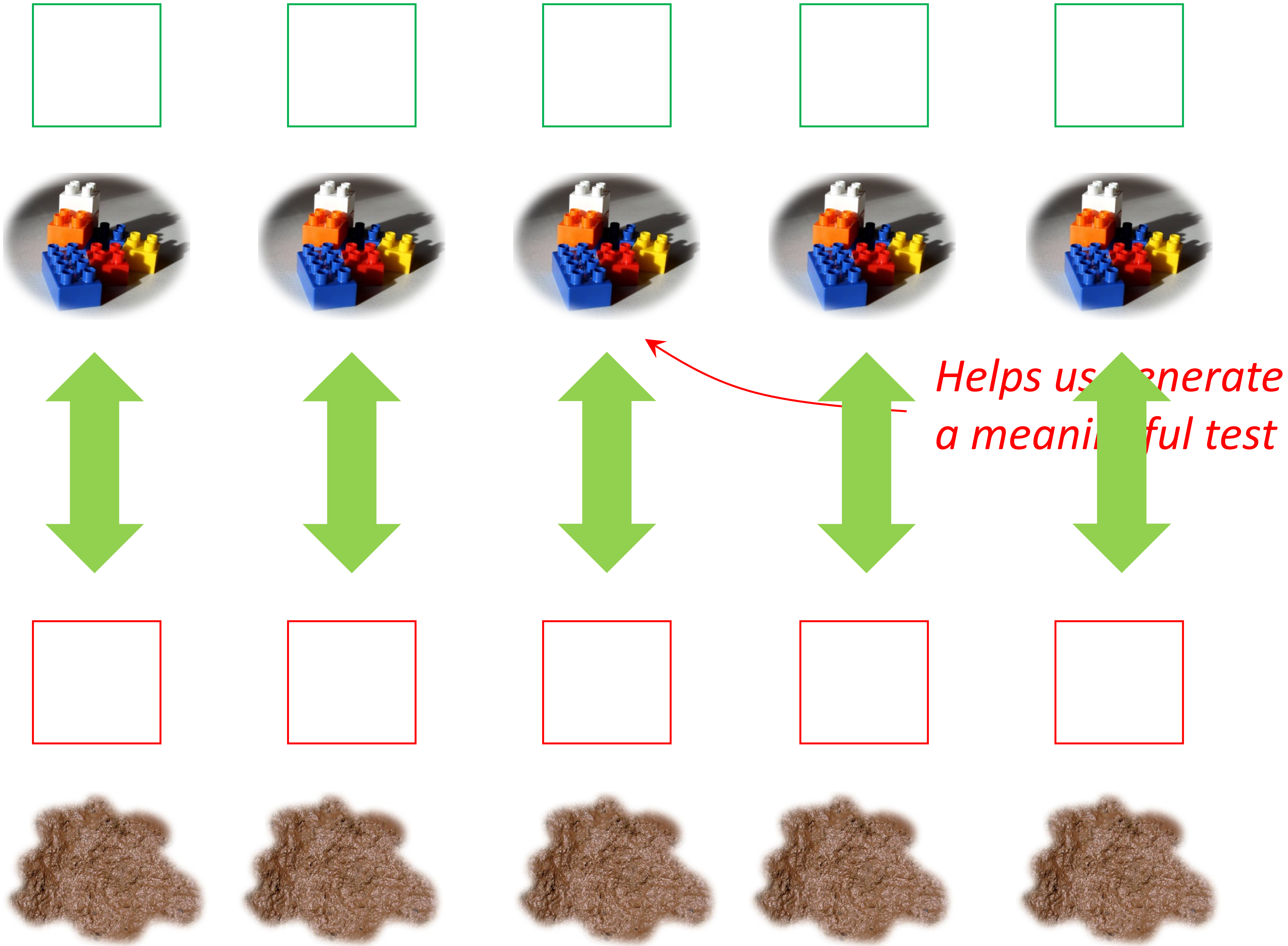
```
register   :: String -> ThreadId -> IO ()
unregister :: String -> IO ()
whereis    :: String -> IO (Maybe ThreadId)
```

- A simple example

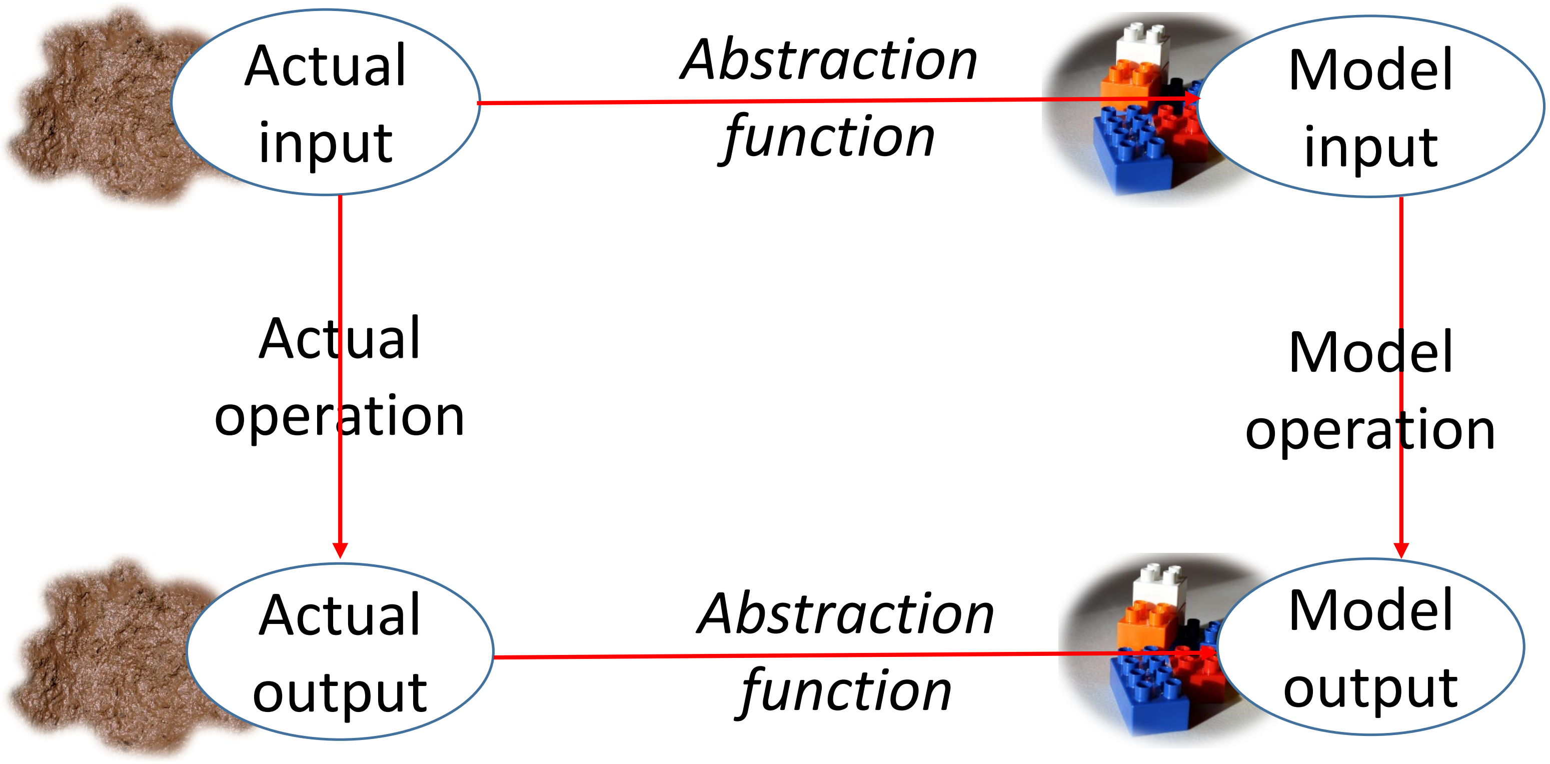
```
> tid <- forkIO (threadDelay 100000000)
> tid
ThreadId 252
> register "me" tid
> whereis "me"
Just ThreadId 252
> unregister "me"
> whereis "me"
Nothing
```


Arguments
register
Results

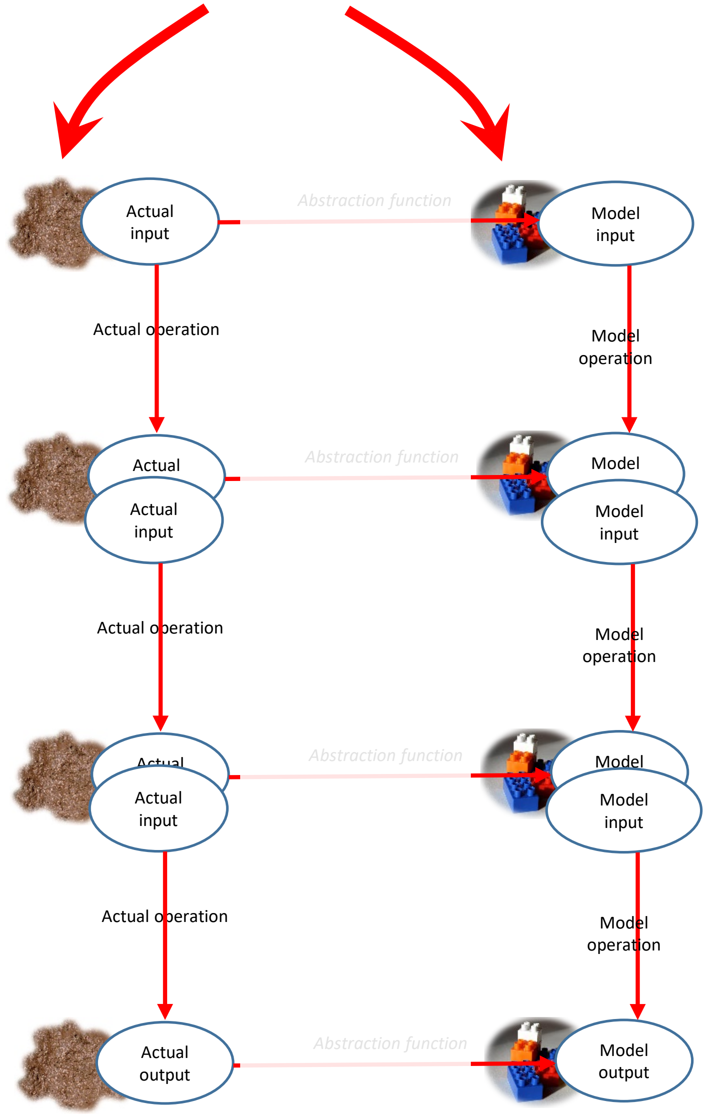

Arguments
register
Results


Helps us generate
a meaningful test

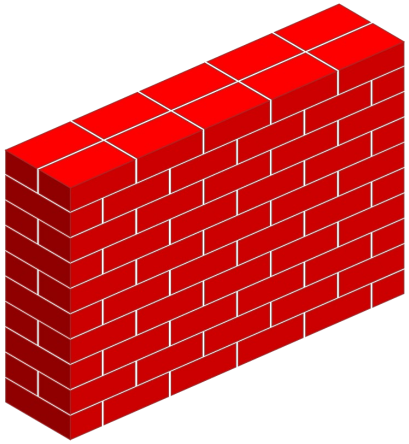
Day 2: Model based tests



Known state


Actual input
Abstraction function
Model input
Actual operation
Model operation
Actual
Actual input
Abstraction function
Model
Model input
Actual operation
Model operation
Actual
Actual input
Abstraction function
Model
Model input
Actual operation
Model operation
Actual output
Abstraction function
Model output

Generate, the



State Modelling Libraries

- Concept of a *state* and an *action*
- The *library* generates, shrinks, and executes the action sequences…
- …given that the user does the same for the *actions*

State Modelling Libraries

- Quviq QuickCheck **`eqc_statem`**
- …
- **`Test.StateMachine`** in quickcheck-state-machine

- A simple one: **`StateModel.hs`**

Back to the Registry

We'll fill this is in as we discover what's needed

- What is the state model?

```
data RegState = RegState{ … }
```

- What are the actions?

```
register
unregister
whereis
spawn
```

Is there anything else I need to do in test sequences?

I need to create threads dynamically in each test run!

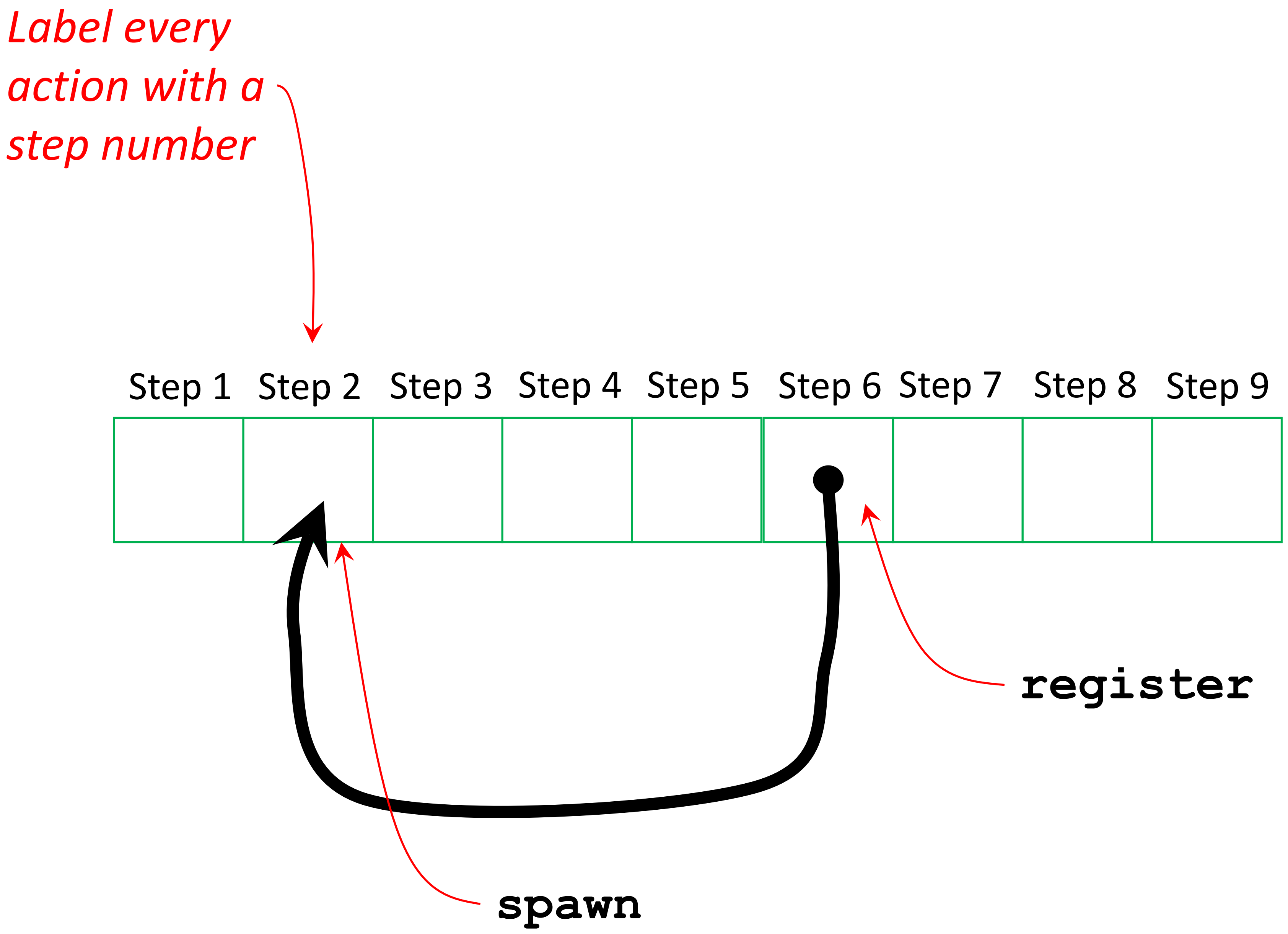
Models are instances of this class

```
instance StateModel RegState where

  data Action RegState =
      Spawn
    | WhereIs String
    | Register String ThreadId
    | Unregister String
```

Models are instances of this class

```
instance StateModel RegState where

  data Action RegState =
      Spawn
    | WhereIs String
    | Register String ThreadId
    | Unregister String
```

This isn't available until test execution time!


Label every action with a step number
Step 1
Step 2
Step 3
Step 4
Step 5
Step 6
Step 7
Step 8
Step 9
`register`
`spawn`

```
instance StateModel RegState where

  data Action RegState =
      Spawn
    | WhereIs String
    | Register String Step
    | Unregister String
```

```
instance StateModel RegState where

  data Action RegState =
      Spawn
    | WhereIs String
    | Register String Step
    | Unregister String
```

```
instance StateModel RegState where

  …

 arbitraryAction s =
    oneof [return Spawn,
           Register
             <$>   ...a name...
             <*>   ...a step...
          ]
```

How should names be chosen?

- We *want* the same name to appear repeatedly in the same test case
- Probably the actual strings used is not important

```
allNames = ["a", "b", "c", "d", "e"]

arbitraryName = elements allNames
```

```
instance StateModel RegState where

  …

 arbitraryAction s =
    oneof [return Spawn,
           Register
             <$> arbitraryName
             <*>  ...a step...
          ]
```

How should a step be chosen?

- Random step number?
- One of the steps of a previous **`Spawn`**!
- How can we know *which* steps were Spawn?
- We keep track of it in the model state!

Thread ids

Usually we use a record so we can easily extend it

```
data RegState = RegState{
    tids :: [Step]
  }

instance StateModel RegState where

  …
  initialState = RegState []
```

State

Action

Current step

```
  nextState s Spawn step =
    s{tids = step:tids s}
  nextState s _ _      = s
```

Default case specifies no effect for other actions… so far

```
instance StateModel RegState where

  …

 arbitraryAction s =
    oneof [return Spawn,
           Register
             <$> arbitraryName
             <*> elements (tids s)
          ]
```

Action generation can depend on the state

Just choose a result from a previous **`Spawn`***!*

Now we can generate tests!

White lie:

The code won’t compile without

```
data Ret RegState = Ret
type ActionMonad RegState = IO
```

Now we can generate tests!

```
> sample (arbitrary :: Gen (Script RegState))
Script
 [(Step 1,Spawn),
  (Step 2,Register "d" (Step 1))]
Script
 [(Step 1,Spawn),
  (Step 2,Register "b" (Step 1)),
  (Step 3,Register "c" (Step 1)),
  (Step 4,Spawn)]
Script
 [(Step 1,Spa   ),
  (Step 2,Spa   )]
Script
 [(Step 1,Register "c" (Step *RegistryModel> ***
Exception: QuickCheck.elements used with empty list
```

A test case is called a **`Script`**

Steps paired with Actions

```
instance StateModel RegState where

  …

  precondition s (Register name step)
    = step `elem` tids s
  precondition _ _ = True
```

`StateModel` ***does not generate*** *(or shrink to) sequences with a* `False` `precondition`

This is `False` *if* `tids s` *is empty*

Now we *really* can generate tests!

```
*RegistryModel> sample (arbitrary :: Gen (Script RegState))
Script []
Script []
Script
 [(Step 1,Spawn),
  (Step 2,Spawn),
  (Step 3,Spawn),
  (Step 4,Register "e" (Step 3)),
  (Step 5,Spawn),
  (Step 6,Register "c" (Step 3)),
  (Step 7,Spawn)]
Script
 [(Step 1,Spawn)]
Script
 [(Step 1,Spawn),
  (Step 2,Spawn),
  (Step 3,Spawn),
  (Step 4,Spawn),
  (Step 5,Register "e" (Step 1))]
Script
 [(Step 1,Spawn),
  (Step 2,Spawn),
  (Step 3,Spawn)]
Script
 [(Step 1,Spawn),
  (Step 2,Spawn)]
Script
 [(Step 1,Spawn),
  (Step 2,Register "c" (Step 1)),
  (Step 3,Register "d" (Step 1)),
  (Step 4,Spawn),
  (Step 5,Register "a" (Step 4)),
  (Step 6,Register "a" (Step 1)),
  (Step 7,Register "b" (Step 4)),
  (Step 8,Spawn),
  (Step 9,Spawn),
  (Step 10,Spawn),
  (Step 11,Register "e" (Step 8)),
  (Step 12,Spawn),
  (Step 13,Register "e" (Step 12)),
  (Step 14,Register "a" (Step 12)),
  (Step 15,Register "b" (Step 9)),
  (Step 16,Register "b" (Step 8)),
  (Step 17,Register "a" (Step 9)),
  (Step 18,Register "d" (Step 8)),
  (Step 19,Register "e" (Step 4)),
  (Step 20,Register "b" (Step 12)),
  (Step 21,Register "d" (Step 4)),
  (Step 22,Spawn),
  (Step 23,Spawn),
  (Step 24,Spawn),
  (Step 25,Register "c" (Step 1)),
  (Step 26,Spawn),
  (Step 27,Spawn),
  (Step 28,Register "a" (Step 12)),
  (Step 29,Register "c" (Step 9)),
  (Step 30,Spawn),
  (Step 31,Spawn),
  (Step 32,Spawn),
  (Step 33,Spawn),
  (Step 34,Register "e" (Step 32)),
  (Step 35,Spawn),
  (Step 36,Register "d" (Step 9)),
  (Step 37,Register "e" (Step 24)),
  (Step 38,Register "a" (Step 27)),
  (Step 39,Spawn),
  (Step 40,Register "a" (Step 26))]
Script
 [(Step 1,Spawn),
  (Step 2,Spawn),
  (Step 3,Register "d" (Step 2)),
  (Step 4,Register "a" (Step 2)),
  (Step 5,Spawn),
  (Step 6,Spawn),
  (Step 7,Spawn),
  (Step 8,Register "b" (Step 6)),
  (Step 9,Register "e" (Step 6)),
  (Step 10,Spawn)]
Script
 [(Step 1,Spawn),
  (Step 2,Spawn),
  (Step 3,Register "a" (Step 2)),
  (Step 4,Spawn),
  (Step 5,Register "c" (Step 2)),
  (Step 6,Register "d" (Step 4)),
  (Step 7,Spawn),
  (Step 8,Register "c" (Step 1)),
  (Step 9,Spawn),
  (Step 10,Register "e" (Step 1)),
  (Step 11,Spawn),
  (Step 12,Register "b" (Step 1)),
  (Step 13,Spawn),
  (Step 14,Spawn)]
Script
 [(Step 1,Spawn),
  (Step 2,Register "c" (Step 1)),
  (Step 3,Spawn),
  (Step 4,Register "e" (Step 3)),
  (Step 5,Spawn)]
```

How do we perform Actions?

```
perform Spawn
   = forkIO (threadDelay 10000000)
perform (Register name step)
   = register name step
```

Ten second wait time—enough

This is a **`Step`**, *not a* **`ThreadId`**

Different types

Return Values

```
instance StateModel RegState where

  …

  data Ret RegState
    = Tid  ThreadId
    | None ()
```

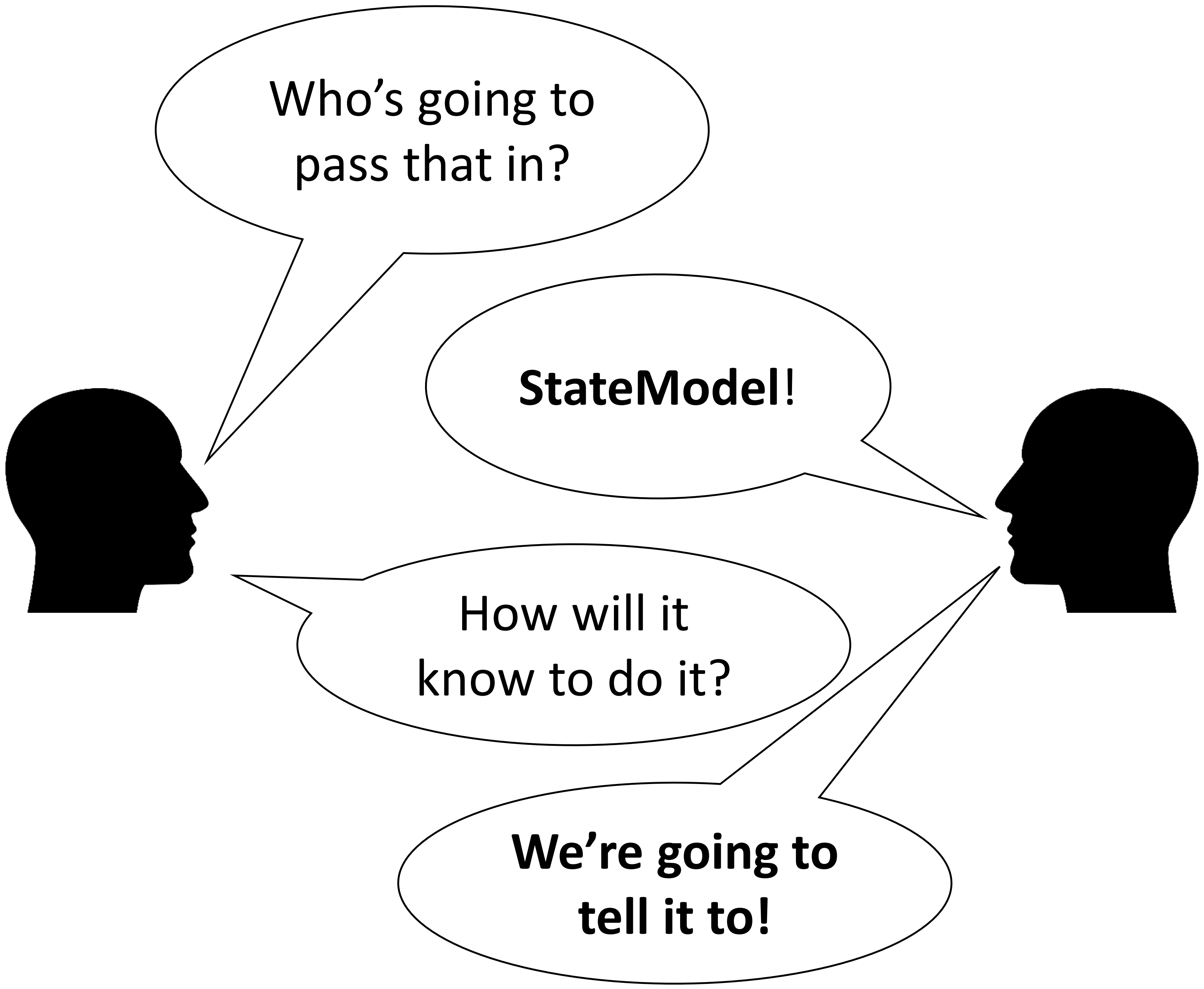
Return type from **Spawn**

Return type from **Register**

```
perform Spawn
  = Tid  <$> forkIO (threadDelay 10000000)
perform (Register name step)
  = None <$> register name step
```

We need a **ThreadId**

Let's just pass one in


Who’s going to pass that in?
StateModel!
How will it know to do it?
We’re going to tell it to!

While performing a test,
`StateModel` *determines*
what each action **`needs`***…*

```
needs (Register _ step) = [step]
needs    _              = []
```

…and passes it
to **`perform`**

```
perform Spawn                     []
  = Tid  <$> forkIO (threadDelay 10000000)
perform (Register name step)  [Tid tid]
  = None <$> register name tid
```

```
perform :: Action state -> [Ret state] ->
                           IO (Ret state)
```

```
type ActionMonad RegState = IO
```

The property

```
prop_Registry :: Script RegState -> Property
prop_Registry s = monadicIO $ do
  runScript s
  assert True
```

Model State

```
data RegState = RegState{ tids :: [Step] }
  deriving Show
```

Associated Types

```
instance StateModel RegState where

  data Action RegState = Spawn
                       | WhereIs String
                       | Register String Step
                       | Unregister String
    deriving Show

  data Ret RegState = Tid ThreadId
                    | None ()
    deriving (Eq,Show)

  type ActionMonad RegState = IO
```

Action Generator

```
  arbitraryAction s =
    oneof [return Spawn,
           Register
             <$> arbitraryName
             <*> elements (tids s)
          ]
```

State Transitions

```
  initialState = RegState []

  nextState s Spawn step =
    s{tids = step:tids s}
  nextState s _     _    = s
```

Preconditions

```
  precondition s (Register name step) =
    step `elem` tids s
  precondition _ _ = True
```

```
  needs (Register _ step) = [step]
  needs _                 = []
```

Performing Actions

```
  perform Spawn _
    = Tid <$> forkIO (threadDelay 10000000)
  perform (Register name step) [Tid tid]
    = None <$> register name tid
```

Extra Generators

```
arbitraryName = elements allNames

allNames = ["a", "b", "c", "d", "e"]
```

Overall Property

```
prop_Registry :: Script RegState -> Property
prop_Registry s = monadicIO $ do
  runScript s
  assert True
```

<50 LOC

We can run tests!

```
*RegistryModel> quickCheck prop_Registry
*** Failed! (after 4 tests and 1 shrink):
Exception:
  bad argument
  CallStack (from HasCallStack):
    error, called at .\Registry.hs:50:10 in main:Registry
Script
 [(Step 1,Spawn),
  (Step 2,Register "d" (Step 1))]
```

We can run tests!

```
*RegistryModel> quickCheck prop_Registry
*** Failed! (after 4 tests and 1 shrink):
Exception:
  bad argument
  CallStack (from HasCallStack):
    error, called at .\Registry.hs:50:10 in main:Registry
Script
 [(Step 1,Spawn),
  (Step 2,Register "d" (Step 1))]
```

The script

We can run tests!

```
*RegistryModel> quickCheck prop_Registry
*** Failed! (after 4 tests and 1 shrink):
Exception:
  bad argument
  CallStack (from HasCallStack):
    error, called at .\Registry.hs:50:10 in main:Registry
Script
 [(Step 1,Spawn),
  (Step 2,Register "d" (Step 1))]
```

The script

Let me run it again…

```
*RegistryModel> quickCheck . withMaxSuccess 1 $
prop_Registry $ Script
 [(Step 1,Spawn),
  (Step 2,Register "d" (Step 1))]
+++ OK, passed 1 test:
100% Register
100% Spawn

Actions (2 in total):
50% Register
50% Spawn
```

Copied and pasted the test case

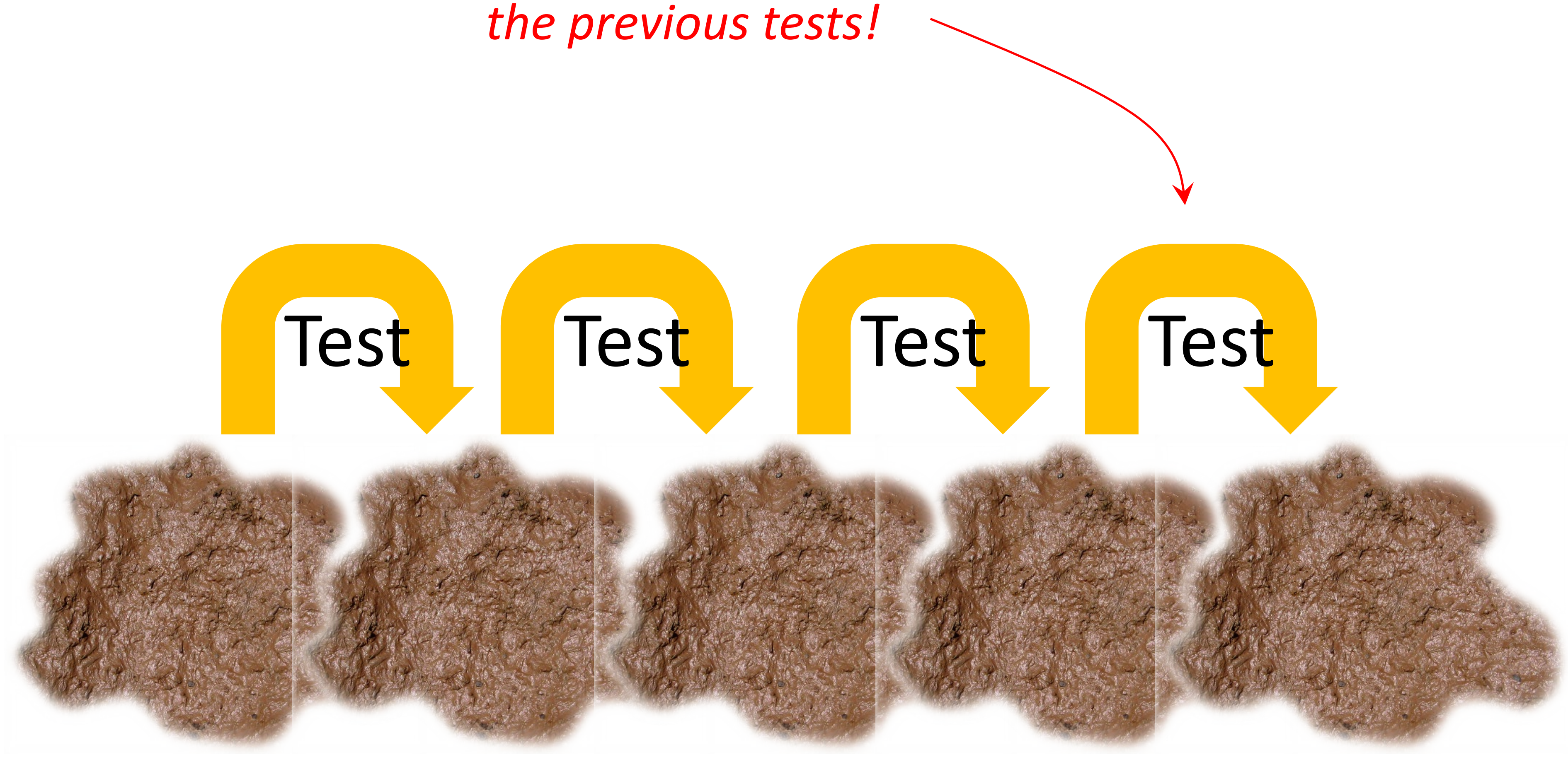
It passes!

Let’s run it again!

```
*RegistryModel> quickCheck . withMaxSuccess 1 $
prop_Registry $ Script
 [(Step 1,Spawn),
  (Step 2,Register "d" (Step 1))]
*** Failed! (after 1 test):
Exception:
  bad argument
  CallStack (from HasCallStack):
    error, called at .\Registry.hs:50:10 in main:Registry
```

Fails when ***less than ten seconds*** *passed since the last test*

Test outcomes depend on the previous tests!



Tests that succeed or fail at random strongly suggest *interference* between tests

DON’T TRY TO DEBUG THIS!!!

Always start in a known state!

```
prop_Registry :: Script RegState -> Property
prop_Registry s = monadicIO $ do
  run cleanUp
  runScript s
  assert True

cleanUp = sequence
  [try (unregister name)
     :: IO (Either ErrorCall ())
   | name <- allNames]
```

At the beginning of the test case

```
*RegistryModel> quickCheck prop_Registry
*** Failed! (after 5 tests and 2 shrinks):
Exception:
  bad argument
  CallStack (from HasCallStack):
    error, called at .\Registry.hs:50:10 in main:Registry
Script
 [(Step 3,Spawn),
  (Step 4,Spawn),
  (Step 5,Register "d" (Step 3)),
  (Step 11,Register "d" (Step 4))]
```

We tried to **`register`** *the same name twice!*

We get a shrunk test case with ***all*** *the relevant info*

Positive testing

- We test the cases that should work
  - Our tests *should not include* calls that will fail!
  - **Advantage:** we test the *interesting* intended behaviour

Negative testing

- We *include* failing calls in our tests
  - We catch exceptions and check that the error behaviour is as it should be
  - **Advantage:** can expose all kinds of dangerous behaviours and vulnerabilities in cases many forget to test

Positive testing

Strengthen the **`precondition`**

- We *should not* call register twice with the same name

- We need to know *which names have been registered*

Enrich the model state

Enriching the model state

```
data RegState = RegState{
    tids   :: [Step],
    regs :: [(String,Step)]
  }
```

The registered name

The registered **`ThreadId`** *(represented by the* **`Step`** *when it was created)*

Updating the model state

```
initialState = RegState [] []

nextState s Spawn step =
  s{tids = step:tids s}
nextState s (Register name tid) step =
  s{regs = (name,tid):regs s}
nextState s _ _ = s
```

The new precondition

```
precondition s (Register name step) =
     step `elem` tids s
  && name `notElem` map fst (regs s)
precondition _ _ = True
```

Repeating the same test

```
*RegistryModel> quickCheck . withMaxSuccess 1 $
prop_Registry $ Script
 [(Step 3,Spawn),
  (Step 4,Spawn),
  (Step 5,Register "d" (Step 3)),
  (Step 11,Register "d" (Step 4))]
*** Gave up! Passed only 0 tests; 10 discarded tests:
```

The **`precondition`** *caused every test to be discarded*

```
*RegistryModel> quickCheck . withMaxSuccess
10000 $ prop_Registry
+++ OK, passed 10000 tests:
92.97% Spawn
82.05% Register

Actions (253566 in total):
88.3257% Spawn
11.6743% Register
```

The proportion of tests that performed a **Spawn** *or a* **Register** ***at all***

Spawn *and* **Register** *as a proportion of all actions performed*

Positive testing of `unregister`

Exercise for the reader!

Adding whereis

```
whereis :: String -> IO (Maybe ThreadId)
```

```
arbitraryAction s =
    oneof […,
           WhereIs
                <$> arbitraryName
          ]
```

Performing WhereIs

```
perform (Whereis name) []
   = MaybeTid <$> whereis name
```

A new type of result

```
data Ret RegState =
     Tid ThreadId
   | None ()
   | MaybeTid (Maybe ThreadId)
```

Tests pass, but…

```
*RegistryModel> quickCheck . withMaxSuccess 1000 $
prop_Registry
+++ OK, passed 1000 tests:
87.0% Spawn
85.8% WhereIs
73.9% Register
54.1% Unregister

Actions (25755 in total):
36.199% Spawn
35.733% WhereIs
16.622% Register
11.446% Unregister
```

We're not checking the result!

Checking whereis

Do we get **`Just tid`** *when* **`name`** *is in the registry?*

```
whereis :: String -> IO (Maybe ThreadId)
```

Do we get the correct **`ThreadId`***?*

Action

`WhereIs String`

Return Value

`MaybeTid (Maybe ThreadId)`

Model State

`regs :: [(String,Step)]`

We need to know the **value** at each **`Step`**

```
postcondition    ::
   state                 ->
   Action state          ->
   (Step -> Ret state)   ->
   Ret state             -> Bool

postcondition s
                (WhereIs name)
 Maybe          stepValue
  (Ret RegState) (MaybeTid mtid) =

 (stepValue <$> lookup name (regs s))
    ==
 (Tid <$> mtid)                  Maybe Step
                Maybe ThreadId
postcondition _ _ _ _ = True
postcondition    ::
   state                     ->
   Action state              ->
   (Step -> Ret state)       ->
   Ret state                 -> Bool

postcondition s
              (WhereIs name)
              stepValue
              (MaybeTid mtid) =
  (stepValue <$> lookup name (regs s))
     ==
  (Tid <$> mtid)

postcondition _ _ _ _ = True
```

If this pattern doesn't match, the error is undetected!

Rather than a catch-all…

Check the return type for each **`Action`**

```
postcondition s Spawn                      _ (Tid _)   = True
postcondition s (Register name step)       _ (None _)  = True
postcondition s (Unregister name)          _ (None _)  = True

postcondition _ _ _ _                                  = False
```

Fail if any call returns a wrongly-tagged result (defends against mistakes in **`perform`***)*

Negative testing

Weaken the **`precondition`**

- We *should* include calls that might fail in test cases—e.g. call register twice with the same name
- We should test *whether or not* an exception was correctly raised

Catch exceptions and write a **`postcondition`** *to check them*

```
precondition s (Register name step) =
     step `elem` tids s
  && name `notElem` map fst (regs s)
```

Of course we still can't **`register`** *a non-existent tid*

```
precondition s (Unregister name) =
     ...
precondition _ _ = True
```

We will ***still*** *need to know whether a call* ***ought*** *to succeed*

```
positive s (Register name step) =

positive s _ = True
```

```
*RegistryModel> quickCheck prop_Registry
*** Failed! (after 9 tests and 4 shrinks):
Exception:
  bad argument
  CallStack (from HasCallStack):
    error, called at .\Registry.hs:54:10 in
main:Registry
Script
 [(Step 2,Spawn),
  (Step 3,Spawn),
  (Step 4,Register "a" (Step 2)),
  (Step 7,Register "a" (Step 3))]
```

Catching the exception

```
perform (Register name step) [Tid tid]
  = None <$> register name tid
  = Caught <$> try (register name tid)

data Ret RegState =
     Tid ThreadId
   | None ()
   | Caught (Either ErrorCall ())
```

```
*RegistryModel> quickCheck prop_Registry
*** Failed! Assertion failed (after 6 tests and 3
shrinks):
Script
 [(Step 10,Spawn),
  (Step 12,Register "a" (Step 10))]
Step 10: Spawn [] --> Tid ThreadId 194198
Step 12: Register "a" (Step 10) [Tid ThreadId 194198] -->
  Caught (Right ())
```

postcondition *failed because the tag was wrong*

When there's no exception, we see the arguments and return values

```
postcondition
   s (Register name step) _ (None _)
   = True
```

↓

```
postcondition
   s (Register name step) _ (Caught (Right()))
   = True
```

```
*RegistryModel> quickCheck prop_Registry
*** Failed! Assertion failed (after 13 tests and 4 shrinks):
Script
 [(Step 2,Spawn),
  (Step 4,Spawn),
  (Step 9,Register "e" (Step 2)),
  (Step 10,Register "e" (Step 4))]
Step 2: Spawn [] --> Tid ThreadId 194312
Step 4: Spawn [] --> Tid ThreadId 194313
Step 9: Register "e" (Step 2) [Tid ThreadId 194312] -->
Caught (Right ())
Step 10: Register "e" (Step 4) [Tid ThreadId 194313] -->
Caught (Left bad argument
CallStack (from HasCallStack):
  error, called at .\Registry.hs:54:10 in main:Registry)
```

A postcondition for +/-ve cases

```
postcondition s (Register name step) _ (Caught res) =
  positive s (Register name step)
    ==
   (res == Right ())
```

```
*RegistryModel> quickCheck . prop_Registry $ Script
 [(Step 2,Spawn),
  (Step 4,Spawn),
  (Step 9,Register "e" (Step 2)),
  (Step 10,Register "e" (Step 4))]
+++ OK, passed 100 tests:
…
```

```
class (...) => StateModel state where

  data Action state
  data Ret state
  type ActionMonad state :: * -> *

  arbitraryAction :: state -> Gen (Action state)
  perform         :: Action state -> [Ret state] ->
                       ActionMonad state (Ret state)
  needs           :: Action state -> [Step]

  initialState    :: state
  nextState       :: state -> Action state -> Step -> state

  precondition    :: state -> Action state -> Bool
  postcondition   :: state -> Action state ->
                     (Step -> Ret state) -> Ret state ->
                       Bool
```

Key takeaways

- Stateful software is harder to test than pure functions, but state-machine models offer an *effective way* to do so.
- Random generation and shrinking is still highly effective, but intricate enough that a good library is essential.
- Stateful software is widespread: most tests used by Quviq customers are of this form.