Describing State with Dependently-Typed FSMs

RSConf 2019

Types

```
const f = (x: number): number => x + 37
```

Types

```
const f = (x: number): number => x + 37

f('abc')
```

Types

```
const f = (x: number): number => x + 37
```

```
f('abc' as any)
```

Type Systems

are programming languages
interpreted by compilers

Type-Level Programming

Idris

PureScript

Haskell

TypeScript

Generic Types as Type Functions

```
interface List<T> {
    items: T[]
}
```

```
List<string>

List<{ x: number, y: number }>

List<List<string>>
```

Generic Types as Type Functions

```ts
interface List<T> {
    items: T[]
}
```

```idris
data List : Type -> Type
```

Hindley-Milner Notation

first argument **return value**

```
add : Int -> Int -> Int
```

second argument

Peano Numbers

```
data Nat = Z | S Nat
```

Peano Numbers

```
data Nat = Z | S Nat

zero : Nat
zero = Z

one : Nat
one = S Z

two : Nat
two = S (S Z)
```

n-Vectors

```
data Vect : Nat -> Type -> Type

append : Vect n elem
      -> Vect m elem
      -> Vect (n + m) elem
```

n-Vectors

```
data Vect : Nat -> Type -> Type

append : Vect n elem
         -> Vect m elem
         -> Vect (n + m) elem

add : Vect n a -> Vect n a -> Vect n a

zip : Vect n a -> Vect n b -> Vect n (a, b)
```

printf

```
printf (
   "Hi %s! Your favourite number is %d”,
   "Bill", 16
)
```

printf


BEWARE
printf()
of the Dog.
The Cat is not
trustworthy
either.
memeblender.com

printf 

```
format
    (SProxy :: SProxy
        "Hi %s! Your favourite number is %d")
    "Bill" 16
```

```
"Hi Bill! Your favourite number is 16"
```

printf

```
format
   (SProxy :: SProxy
      "Hi %s! Your favourite number is %d")
   "Bill" 16
```

```
:t format (SProxy :: SProxy "Hi %s! Your
favourite number is %d")
```

printf

```
format
   (SProxy :: SProxy
      "Hi %s! Your favourite number is %d")
   "Bill" 16
```

```
:t format (SProxy :: SProxy "Hi %s! Your
favourite number is %d")

String -> Int -> String
```

n-Queens

https://aphyr.com/posts/342-typing-the-technical-interview

```
*Main> :type solution (nil :: N6)
solution (nil :: N6)
  :: Cons (Queen (S (S (S (S (S Z))))) (S Z))
       (Cons (Queen (S (S (S (S Z)))) (S (S (S Z))))
         (Cons (Queen (S (S (S Z))) (S (S (S (S (S (S Z)))))))
           (Cons (Queen (S (S Z)) Z)
             (Cons (Queen (S Z) (S (S Z)))
               (Cons (Queen Z (S (S (S (S (S Z))))))
                 Nil)))))
```

Snooker

Snooker

Table with Pockets

Red balls and Colors

Players take shots and pot balls

Snooker

Phase One: red -> any color -> red

Phase Two: all colors in a specific order

Snooker

Phase One: red -> any color -> red

Phase Two: all colors in a specific order

1

2

3

4

5

6

7

Snooker Types

```
data ColorBall
    = Yellow
    | Green
    | Brown
    | Blue
    | Pink
    | Black
```

Snooker Types

```
colorScore : ColorBall -> Nat
colorScore Yellow = 2
colorScore Green  = 3
colorScore Brown  = 4
colorScore Blue   = 5
colorScore Pink   = 6
colorScore Black  = 7
```

Snooker Types

```
data Shot : Type -> Type
```

Snooker Types

```
data Shot : Type -> Type where
    Pure  : a -> Shot a
    (>>=) : Shot a -> (a -> Shot b) -> Shot b
```

Snooker Types

```
data Shot : Type -> Type where
    Pure  : a -> Shot a
    (>>=) : Shot a -> (a -> Shot b) -> Shot b
```

```
red : Shot Nat
red = Pure 1
```

Snooker Types

```
data Shot : Type -> Type where
    Pure  : a -> Shot a
    (>>=) : Shot a -> (a -> Shot b) -> Shot b
```

```
red : Shot Nat
red = Pure 1
```

```
color : ColorBall -> Shot Nat
color = Pure . colorScore
```

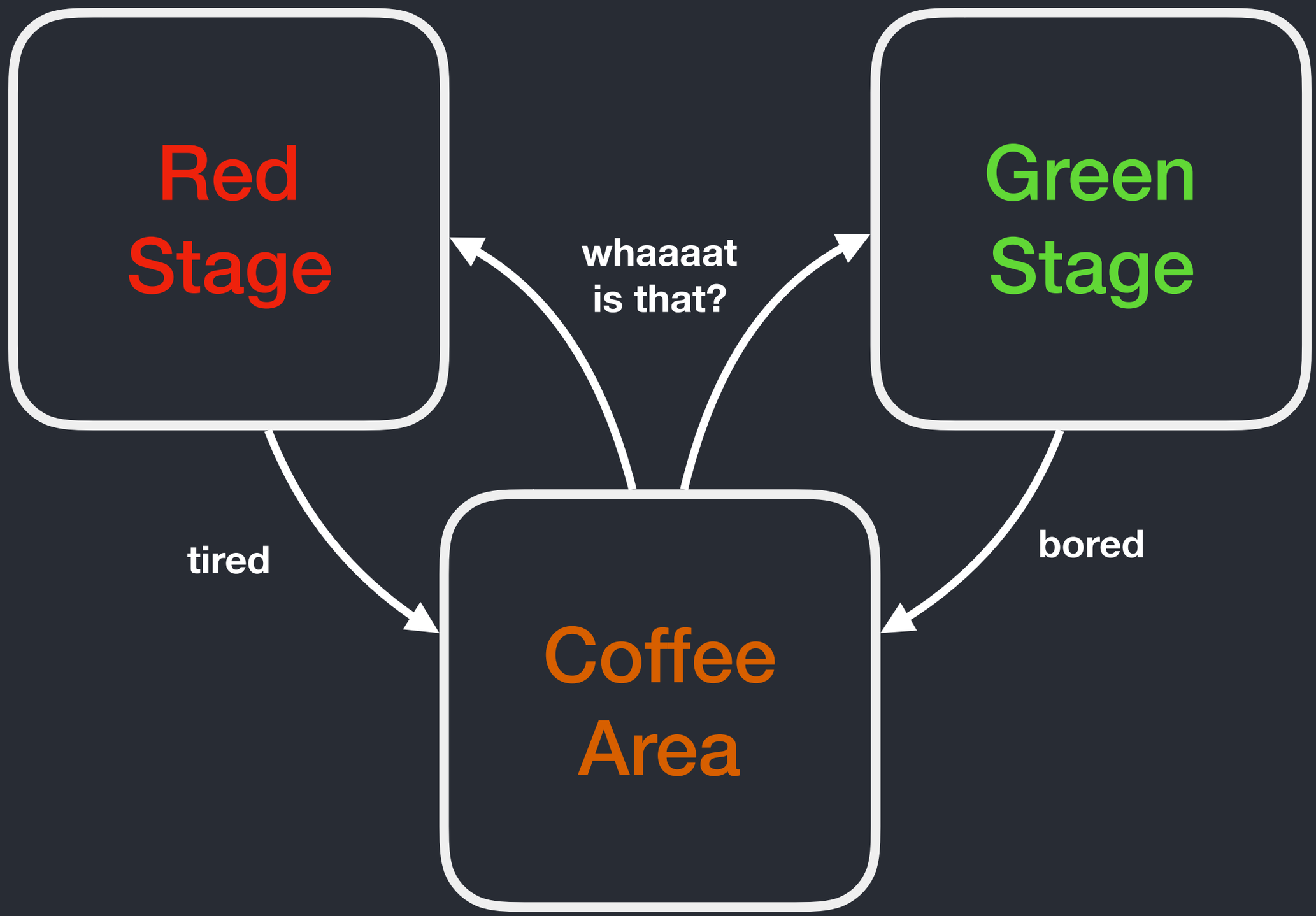
Break Sequence

```
break : Shot Nat
break = do
    red
    color Blue
    red
    color Pink
    red
    color Black
```

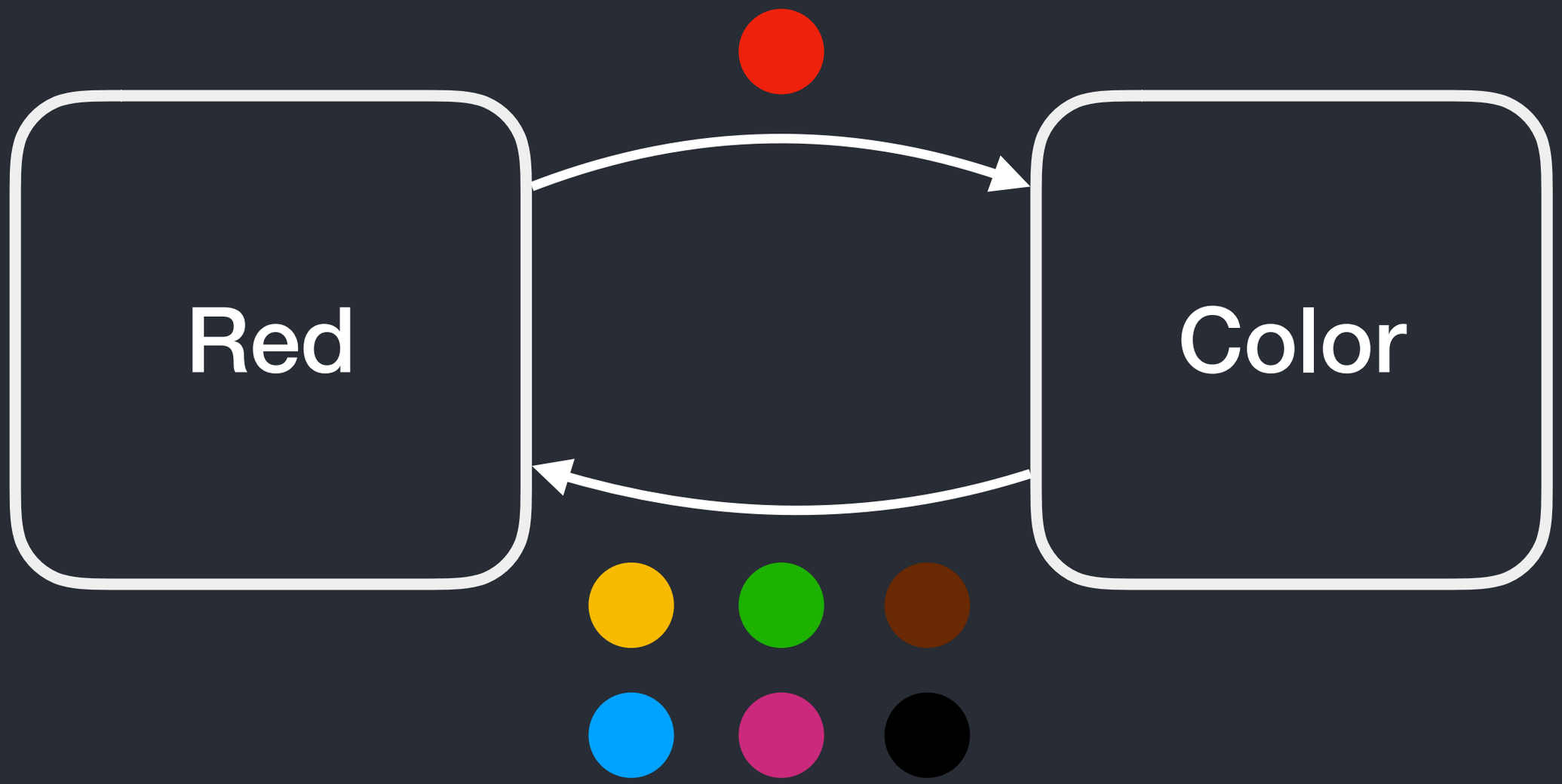
Break Sequence

```
break : Shot Nat
break = do
    red
    color Blue
    color Green
    red
    color Pink
    red
    color Black
```

Finite State Machines



Red-Color-Red



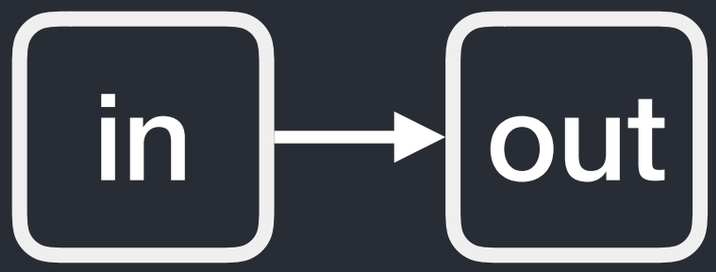
Snooker Types

```
data Ball = Red | Color
```

Snooker Types

```
data Ball = Red | Color
```

```
data Shot : Type -> Ball -> Ball -> Type
```



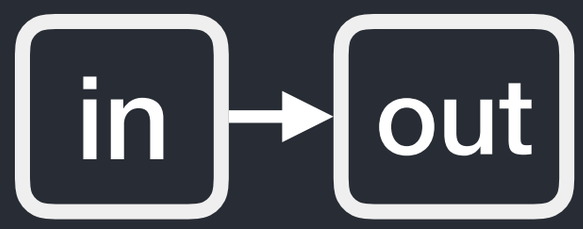
Snooker Types

```
data Ball = Red | Color
```

```
data Shot : Type -> Ball -> Ball -> Type where
  Pure : a -> Shot a b c

  (>>=) : Shot a s1 s2
          -> (a -> Shot b s2 s3)
          -> Shot b s1 s3
```

Snooker Types

```
data Ball = Red | Color

data Shot : Type -> Ball -> Ball -> Type where

  red : Shot Nat Red Color
  red = Pure 1

  color : ColorBall -> Shot Nat Color Red
  color = Pure . colorScore
```



Break Sequence

```
break : Shot Nat Red Red
break = do
    red
    color Blue
    red
    color Pink
    red
    color Black
```

Break Sequence

```
break : Shot Nat Red Red
break = do
    red
    color Blue
    color Green
    red
    color Pink
    red
    color Black
```

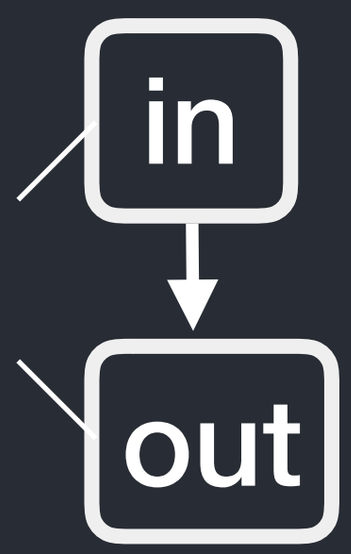
Break Sequence

```
    |
58  |      color Green
    |      ~~~~~~~~~~~~
When checking right hand side of Main.break with expected type
        Shot Nat Red Red

When checking an application of constructor Main.>>=:
Type mismatch between
        Shot Nat Color Red (Type of color _)
and
        Shot a Red s2 (Expected type)

Specifically:
        Type mismatch between
                Color
        and
                Red
```

Counting the Reds

```
data Ball = Red | Color

data GameState = State Nat Ball
```

```
data Shot : Type
          -> GameState
          -> GameState
          -> Type
```



Counting the Reds

```
break : Shot Nat (State 15 Red) (State 12 Red)
break = do
    red
    color Blue
    red
    color Pink
    red
    color Black
```

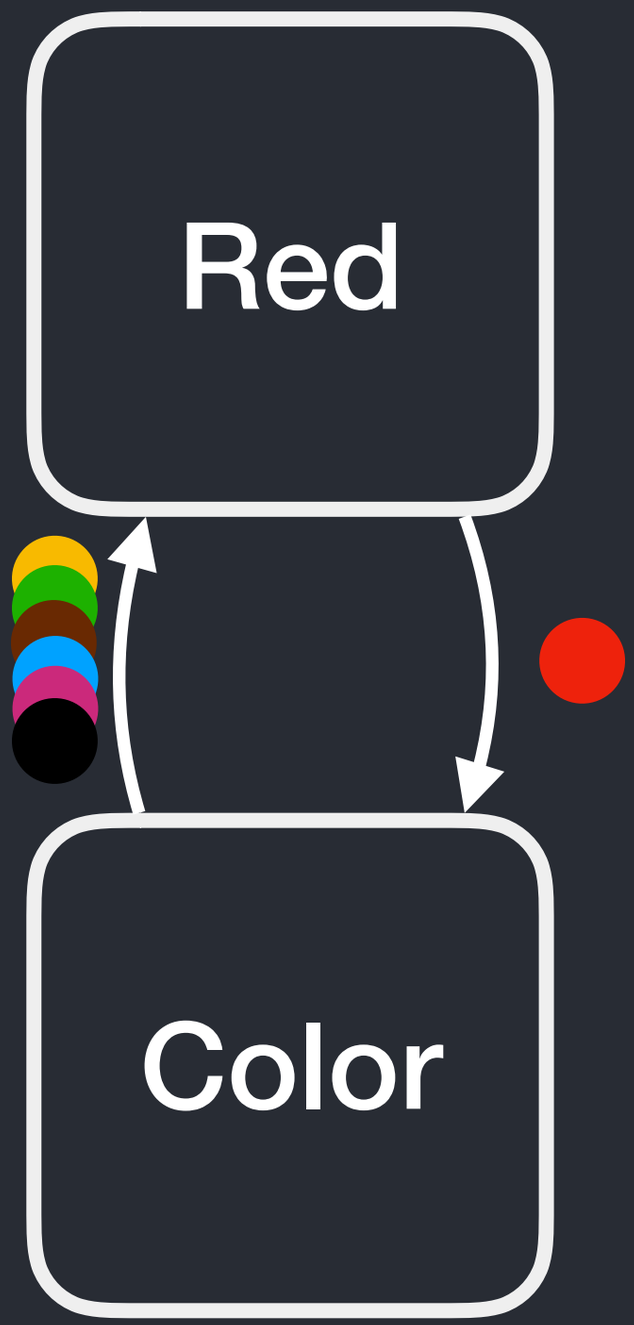
Counting the Reds

```
break : Shot Nat (State 15 Red) (State 12 Red)
break = do
    red
    color Blue
    red
    color Pink
    red
    color Black
    red
    color Black
```

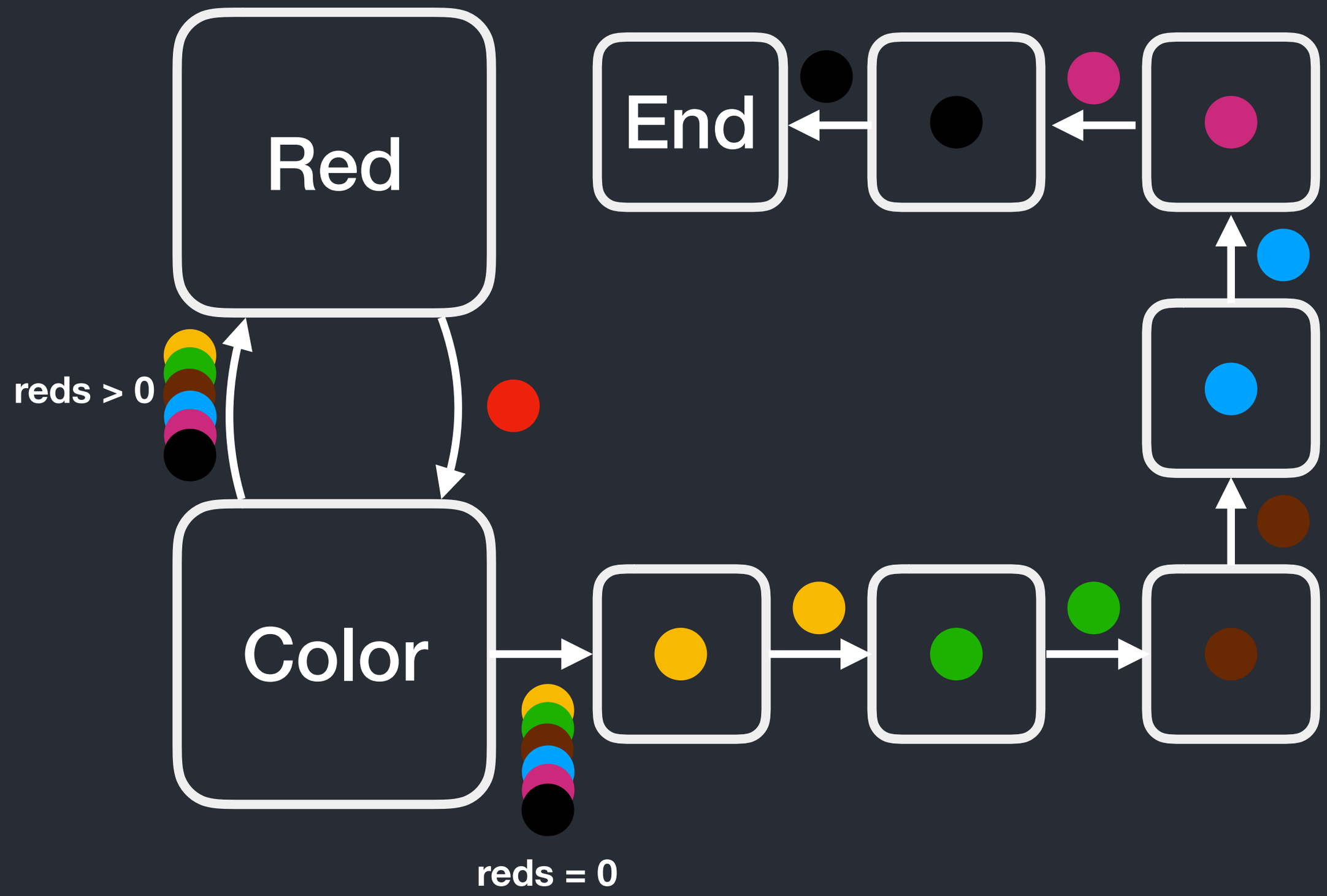
Counting the Reds

```
   |
59 |      red
   |      ~~~
When checking right hand side of Main.break with expected type
        Shot Nat (State 15 Red) (State 12 Red)

When checking an application of constructor Main.>>=:
Type mismatch between
        Shot Nat (State reds Color) (State reds Red) (Type of color _)
and
        Shot Nat (State 11 Color) (State 12 Red) (Expected type)

Specifically:
        Type mismatch between
                0
        and
                1
```

The Two Phases



The Two Phases



The Two Phases

```
data State
     = Reds Nat Ball
     | Colors ColorBall
     | End
```

The Two Phases

```
data State
     = Reds Nat Ball
     | Colors ColorBall
     | End

red : Shot Nat
        (Reds (S reds) Red)
        (Reds reds Color)
red = Pure 1
```

The Two Phases

```
data State
     = Reds Nat Ball
     | Colors ColorBall
     | End

color : ColorBall -> Shot Nat
                     (Reds (S reds) Color)
                     (Reds (S reds) Red)
color = Pure . colorScore
```

The Two Phases

```
data State
     = Reds Nat Ball
     | Colors ColorBall
     | End

last_red_color : ColorBall
                 -> Shot Nat
                    (Reds Z Color)
                    (Colors Yellow)
last_red_color = Pure . colorScore
```

The Two Phases

```
data State
     = Reds Nat Ball
     | Colors ColorBall
     | End
```

```
yellow : Shot Nat (Colors Yellow) (Colors Green)
yellow = Pure 2
```

The Two Phases

```
data State
     = Reds Nat Ball
     | Colors ColorBall
     | End
```

```
yellow : Shot Nat (Colors Yellow) (Colors Green)
yellow = Pure 2

 green : Shot Nat (Colors Green) (Colors Brown)
 green = Pure 3
```

The Two Phases

```
data State
     = Reds Nat Ball
     | Colors ColorBall
     | End
```

```
green : Shot Nat (Colors Green) (Colors Brown)
green = Pure 3

black : Shot Nat (Colors Black) End
black = Pure 7
```

The Two Phases

```
break : Shot Nat (Reds 3 Red) End
break = do
    red
    color Black
    red
    color Black
    red
    last_red_color Black
```

```
yellow
green
brown
```

The Two Phases

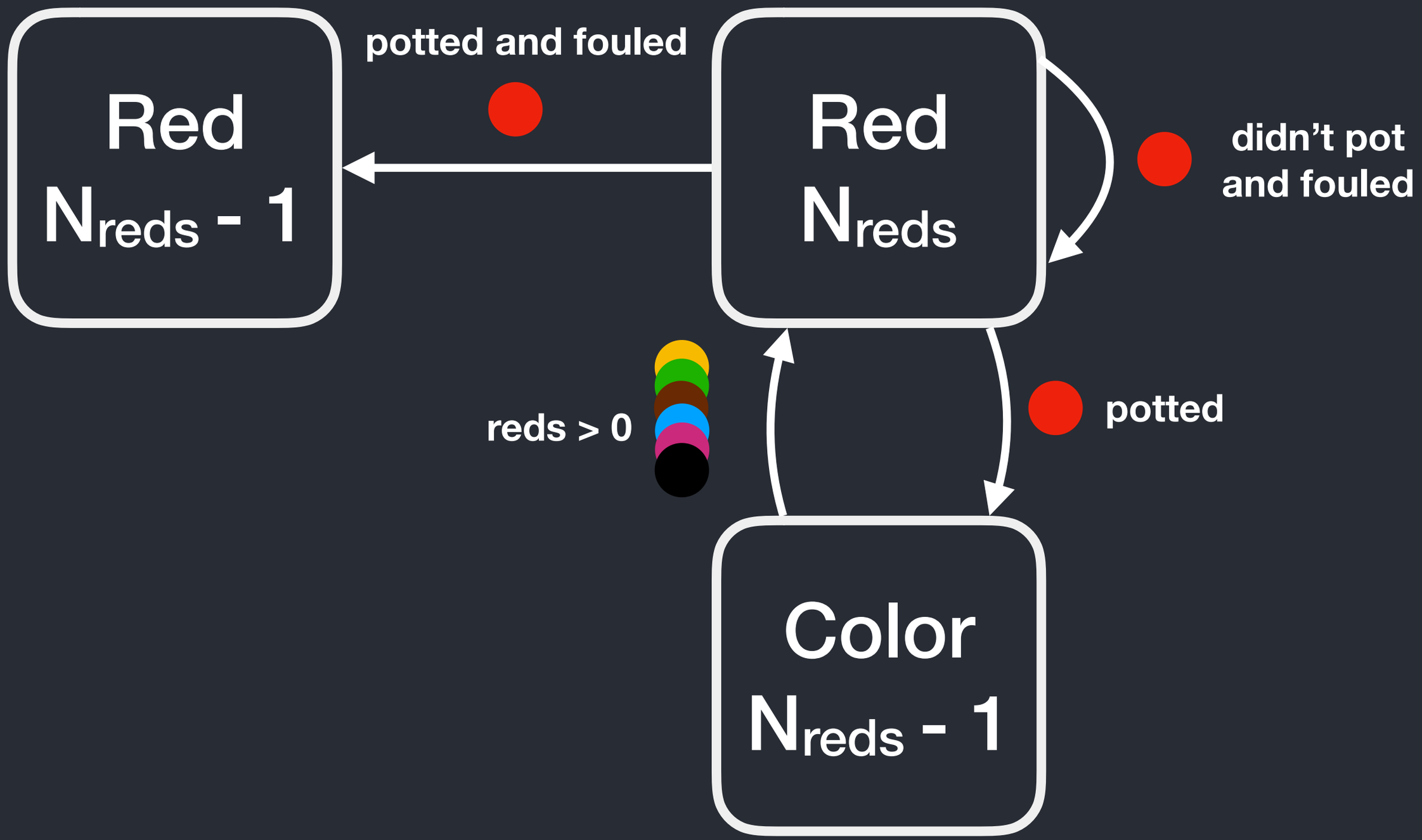
```
break : Shot Nat (Reds 3 Red) End
break = do
    ...

    yellow
    green
    brown
    blue
    pink
    black
```

The Two Phases

```
   |
69 |     pink
   |     ~~~~
When checking right hand side of Main.break with expected type
        Shot Nat (Reds 3 Red) End

When checking an application of constructor Main.>>=:
        Type mismatch between
                Shot Nat (Colors Pink) (Colors Black) (Type of pink)
        and
                Shot Nat (Colors Blue) s2 (Expected type)

        Specifically:
                Type mismatch between
                        Pink
                and
                        Blue
```

Foul Handling

Foul Handling



Foul Handling

```
data ShotResult
    = Pot
    | Foul Bool
```

potted or not

Foul Handling

```
data Shot : (a : Type)
         -> GameState
         -> (a -> GameState)
         -> Type
```

decision function

Foul Handling

```
data Shot : (a : Type)
         -> GameState
         -> (a -> GameState)
         -> Type
```

decision function

```
color : ColorBall
     -> Shot ShotResult
        (Reds (S reds) Color)
        (const (Reds (S reds) Red))
```

const

```
const : a -> b -> a
const a b = a

const cnst = a => b => a
```

```
color : ColorBall
        -> Shot ShotResult
          (Reds (S reds) Color)
          (const (Reds (S reds) Red))
```

Foul Handling

```
data Shot : (a : Type)
         -> GameState
         -> (a -> GameState)
         -> Type
```

decision function

```
color : ColorBall
     -> Shot ShotResult
        (Reds (S reds) Color)
        (const (Reds (S reds) Red))
```

Foul Handling

```
red : Shot ShotResult
        (Reds (S reds) Red)
        (\res =>
          case res of
            Pot => Reds reds Color
            Foul True => Reds reds Red
            Foul False => Reds (S reds) Red)
```

Foul Handling

```
last_red_color : ColorBall
                 -> Shot ShotResult
                 (Reds Z Color)
                 (const (Colors Yellow))
```

Foul Handling

```
second_phase : Shot ShotResult
                    (Colors Yellow)
                    (const End)
second_phase = do
    yellow
    green
    brown
    blue
    pink
    black
```

Foul Handling

```
break : Shot ShotResult (Reds 1 Red) (const End)
break = do
    red
    last_red_color Pink
    second_phase
```

```
   |
61 |     last_red_color Pink
   |     ~~~~~~~~~~~~~~~~~~~
When checking right hand side of Main.break with expected type
        Shot ShotResult (Reds 1 Red) (const End)

When checking an application of constructor Main.>>=:
        Type mismatch between
                Shot ShotResult (Reds 0 Color) (const (Colors Yellow))
        and
                Shot ShotResult
                     (case _ of
                        Pot _ => Reds reds Color
                        Foul True => Reds reds Red
                        Foul False => Reds (S reds) Red)
                    (\value => Colors Yellow) (Expected type)

        Specifically:
                Type mismatch between
                        Reds 0 Color
                and
                        case _ of
                          Pot _ => Reds reds Color
                          Foul True => Reds reds Red
                          Foul False => Reds (S reds) Red
```

Foul Handling

```
Specifically:
        Type mismatch between
                Reds 0 Color
    and
            case _ of
              Pot _ => Reds reds Color
              Foul True => Reds reds Red
              Foul False => Reds (S reds) Red
```

```
break : Shot ShotResult (Reds 1 Red) (const End)
break = do
    res <- red
    case res of
        Pot -> do
            last_red_color Pink
            second_phase
    Foul True ->
        second_phase
    Foul False ->
        Pot <- red
        last_red_color Blue
        second_phase
```

Foul Handling

```
red : Shot ShotResult
        (Reds (S reds) Red)
        (\res => case res of
                   Pot => Reds reds Color
                   Foul True => case reds of
                                  Z => Colors Yellow
                                  n => Reds n Red
                   Foul False => Reds (S reds) Red)
```

Analogy Mapping

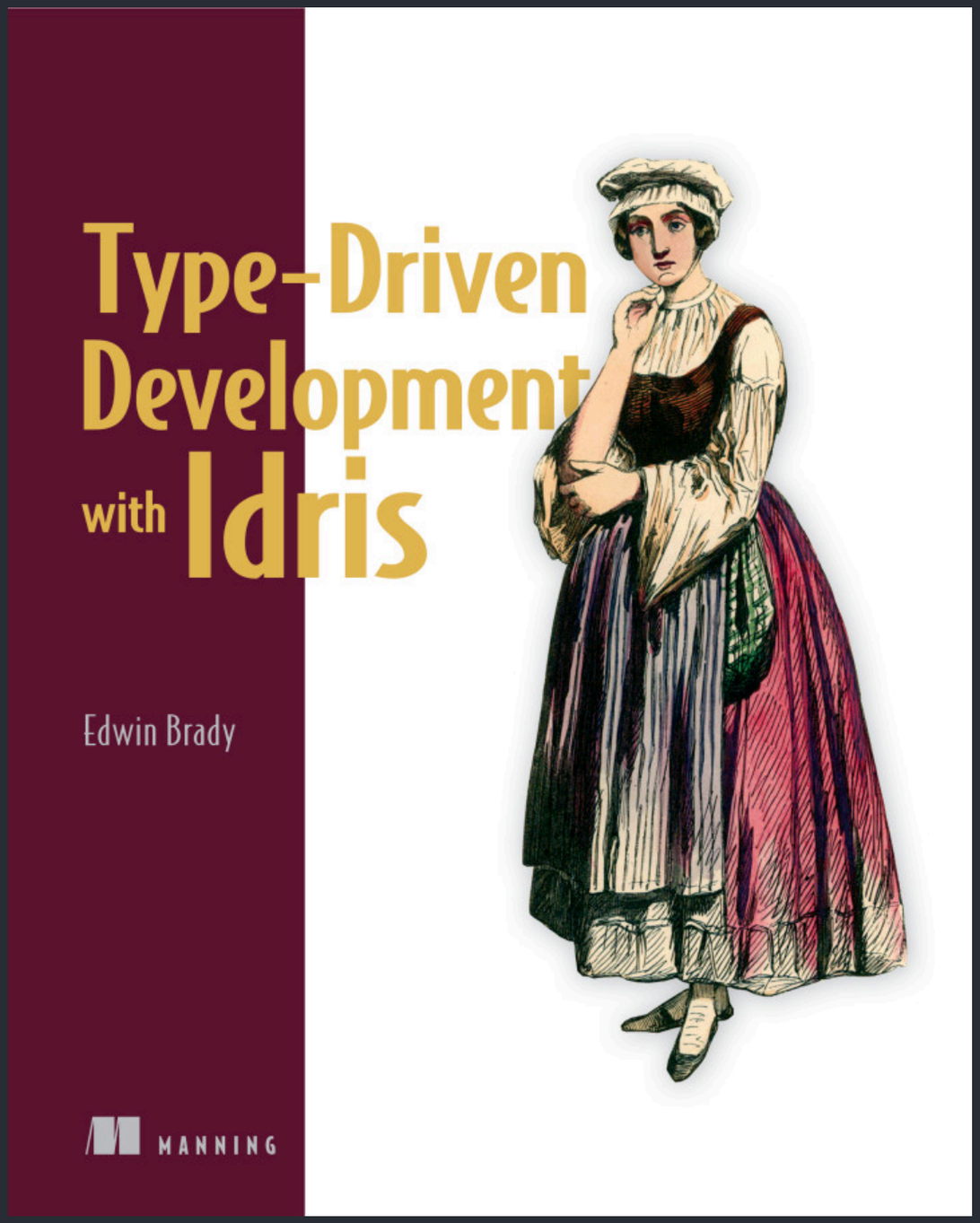
Snooker Rules = Domain Logic

Snooker Game = Application

Break = State Modification

Foul = Error / Exception

What's next?



What's next?

Thanks!