The Problem of the Dutch National Flag

Wouter Swierstra

IFIP WG 2.1 #66

Jeremy’s Problem

The State Monad

```
State s a := s -> a * s

return : a -> State s a
(>>=) : State s a
  -> (a -> State s b)
  -> State s b
```

```
relabel : State nat (Tree nat)
relabel t = match t with
 | Leaf _ =>
    get >>= fun c =>
    put (c + 1) >>=
    return (Leaf c)
 | Node l r =>
    relabel l >>= fun l' =>
    relabel r >>= fun r' =>
    return (Node l' r')
 end
```

Idea:
Decorate the state monad with pre- and postconditions.

Pre- and postconditions

Define the following types:

```
Pre := s -> Prop

Post (a : Set) := s -> a -> s -> Prop
```

The Hoare State Type

Define the following type:

```
HoareState s P a Q :=
  {i : s | P i} ->
    {(x,f) : a * s | Q i x f}
```

Plan

- Define return and bind with a fancy HoareState type.
- Choose a suitable type for our relabelling function.

Relabelling revisited

The type of relabel becomes:

```
HoareState
  (fun i => True)
  (Tree nat)
  (fun i t f =>
     flatten t = [i .. i + size t])
```

Relabelling revisited

The type of relabel becomes:

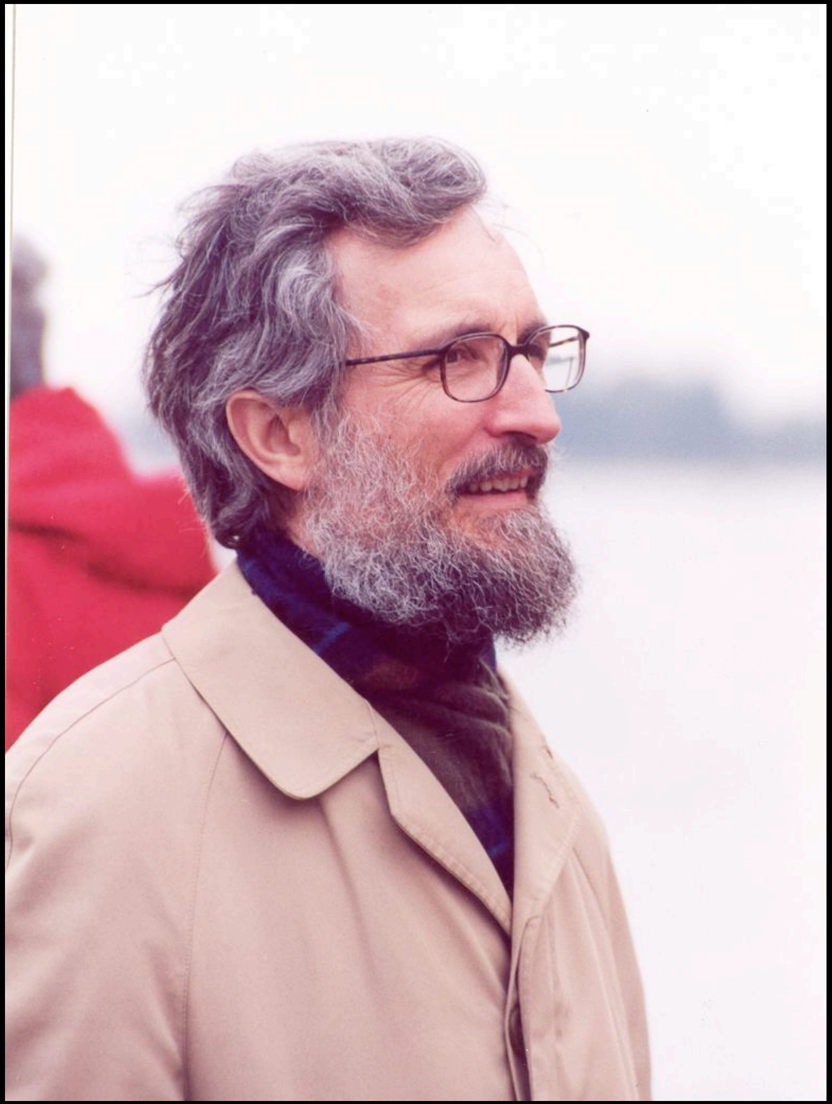
```
HoareState
  (fun i => True)
  (Tree nat)
  (fun i t f =>
     flatten t = [i .. i + size t]
     /\ f = i + size t)
```

The Problem of the Dutch National Flag

Wouter Swierstra
IFIP WG 2.1 #66

Type Theory
Per Martin-Löf

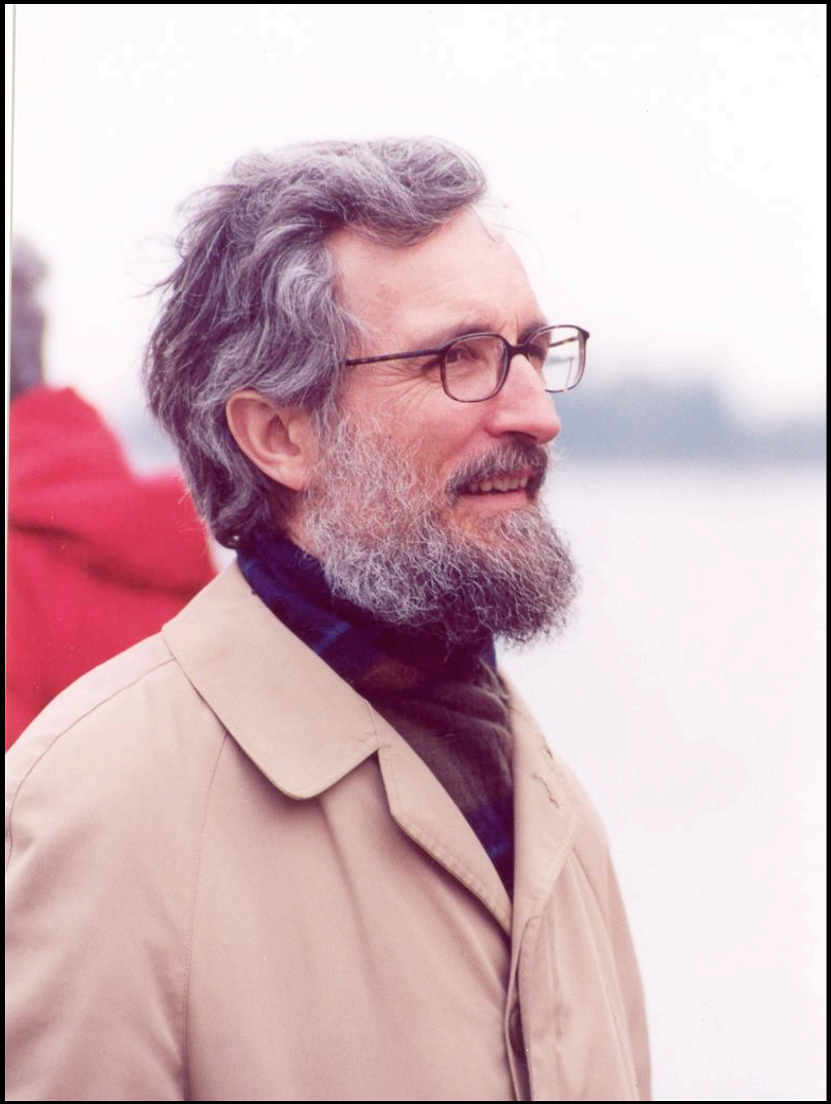
- A foundation of constructive mathematics;
- a functional programming language.



Type Theory
Per Martin-Löf

- A foundation of constructive mathematics;
- a functional programming language.

Really?



What about...

- mutable references?
- arrays?
- exceptions?
- concurrency?
- a GUI?
- a foreign function interface?
- network communication?
- a compiler?
- general recursion?
- file manipulation?
- random numbers?
- ...

There is a row of buckets numbered from 1 to n. It is given that:

- each bucket contains one pebble
- each pebble is either red, white, or blue.

A mini-computer is placed in front of this row of buckets and has to be programmed in such a way that it will rearrange (if necessary) the pebbles in the order of the Dutch national flag.

A Discipline of Programming, E.W. Dijkstra

Specification

- The mini-computer supports two commands:
  - swap (i,j) exchanges the pebbles in buckets numbered i and j for $1 \leq i,j \leq n$;
  - read (i) returns the colour of the pebble in bucket number i for $1 \leq i \leq n$.
- Solution should use one pass only and constant memory.

The Problem of the Dutch National Flag

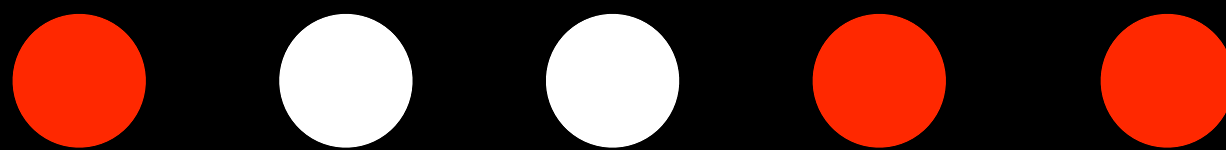
Wouter Swierstra

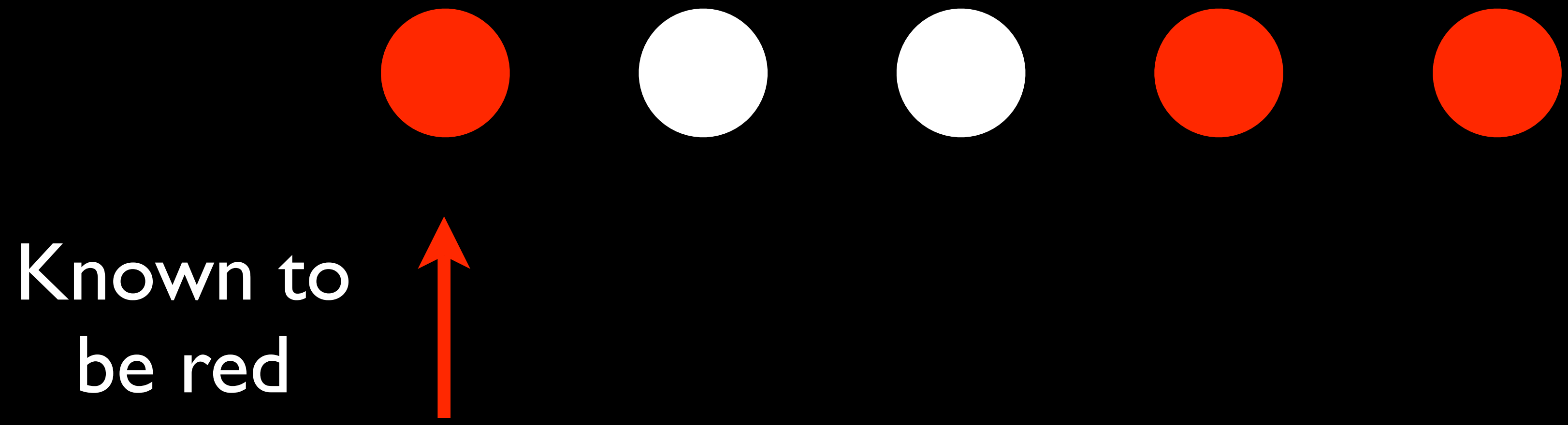
IFIP WG 2.1 #66

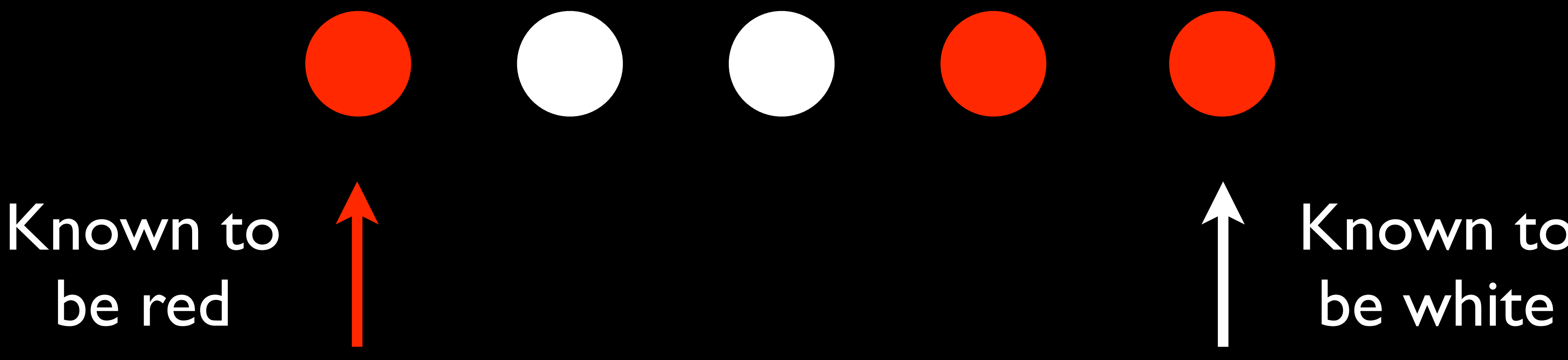
The Problem of the ~~Dutch~~ Polish National Flag

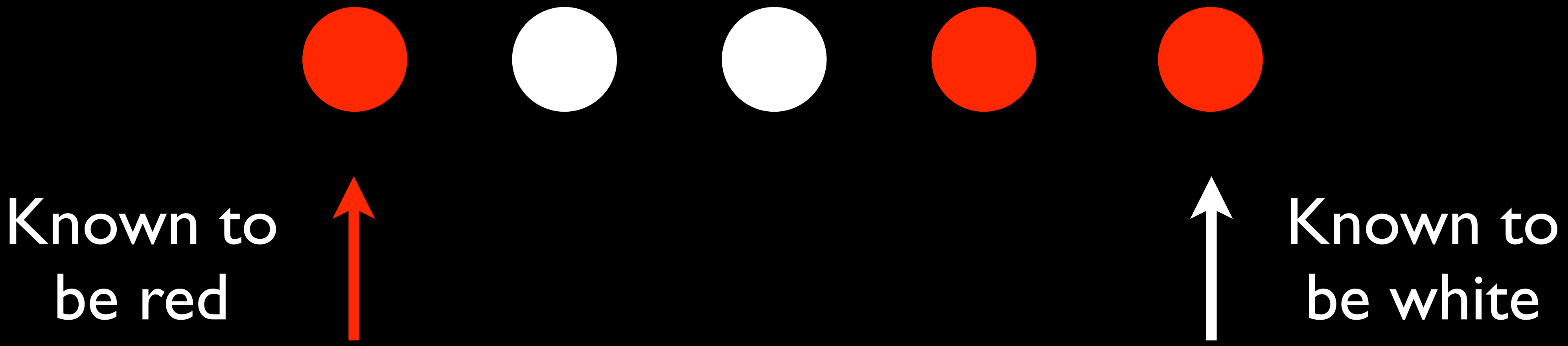
Wouter Swierstra

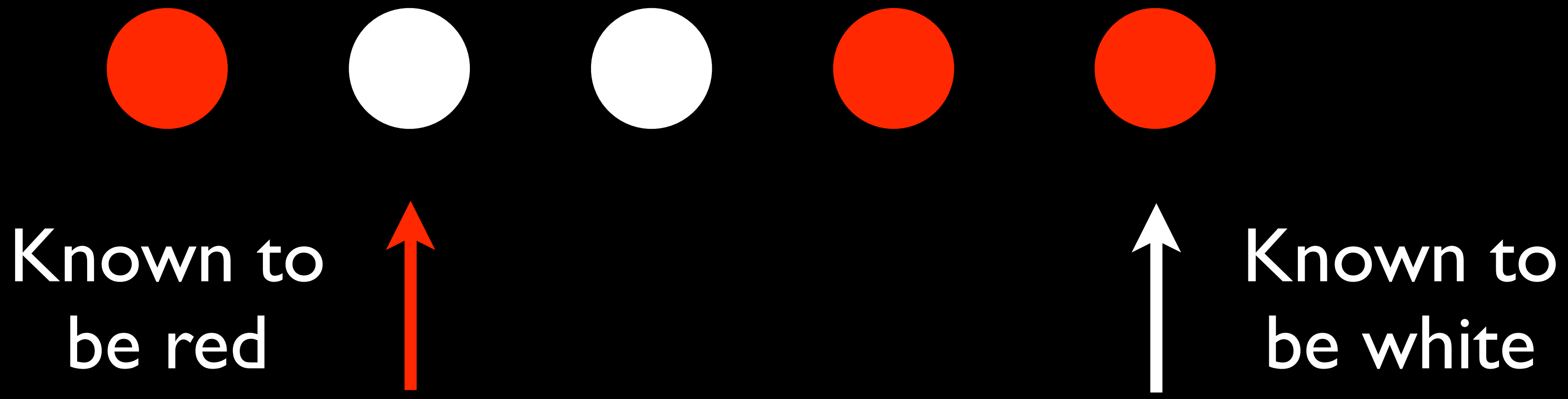
IFIP WG 2.1 #66

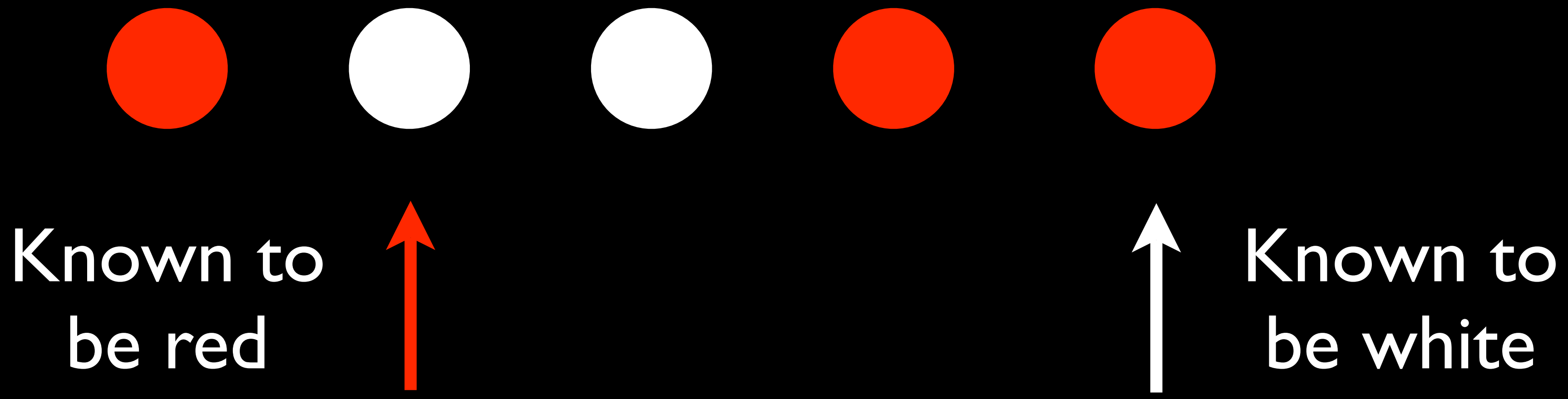


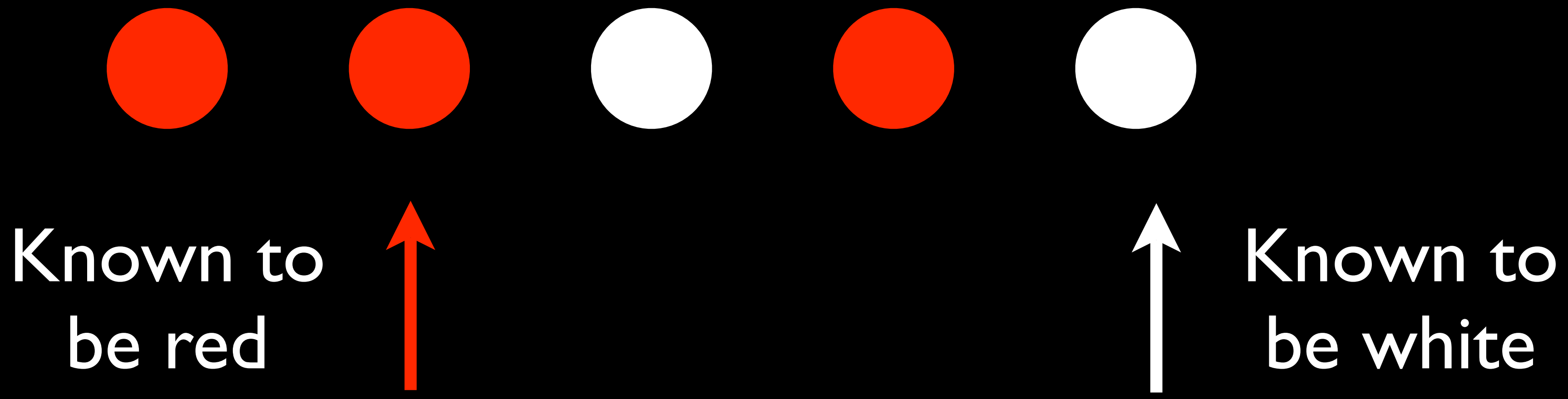

Known to
be red

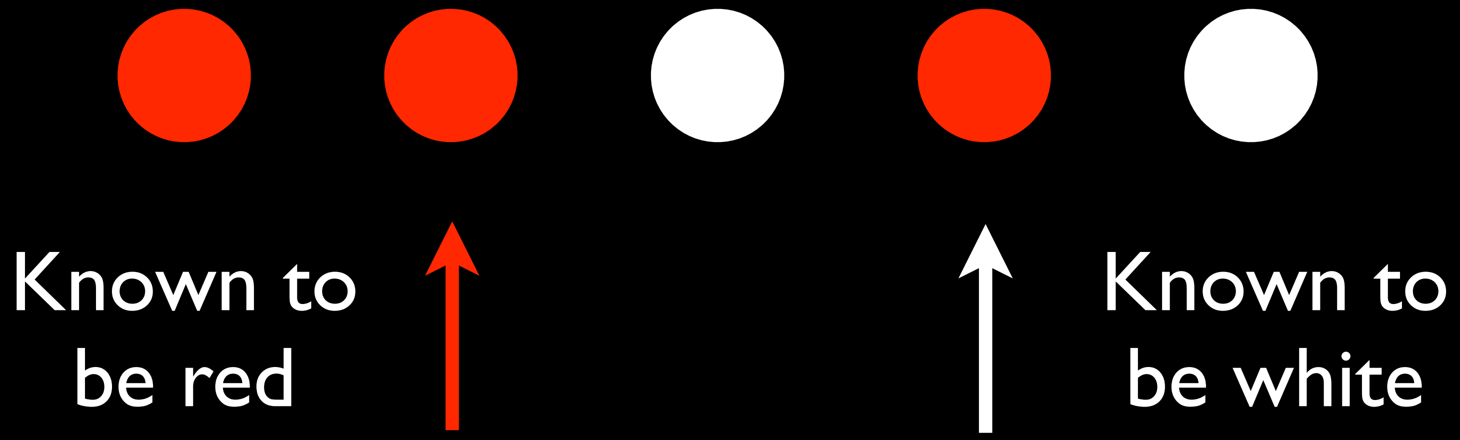

Known to
be red
Known to
be white

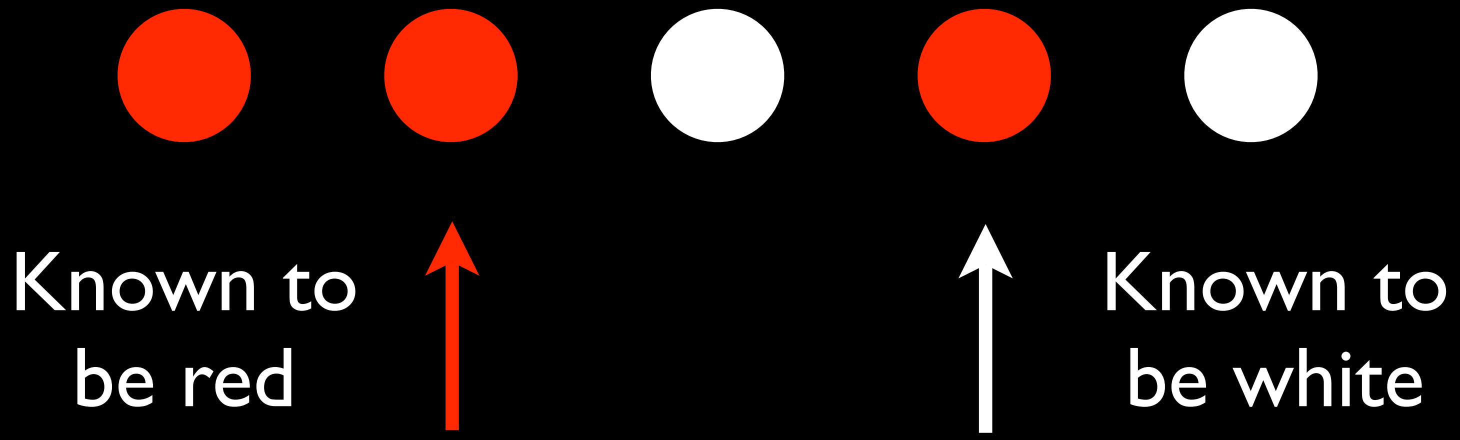

Known to be red
Known to be white

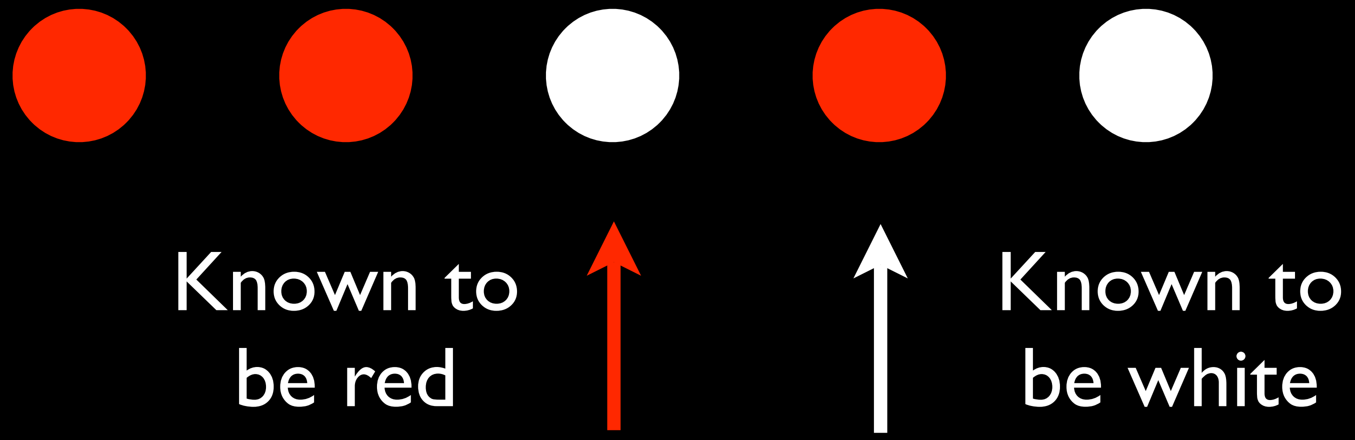

Known to be red
Known to be white

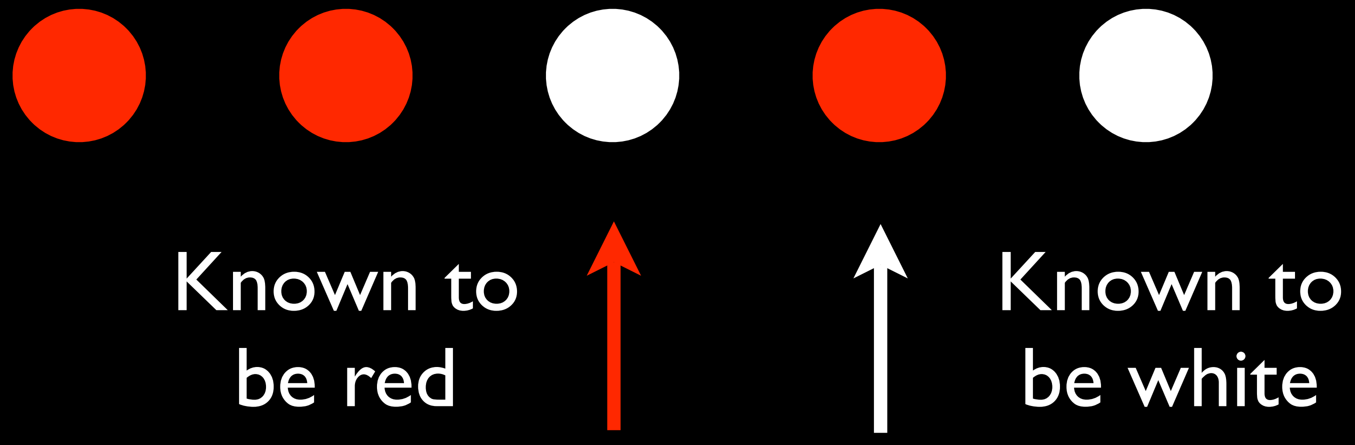

Known to be red
Known to be white

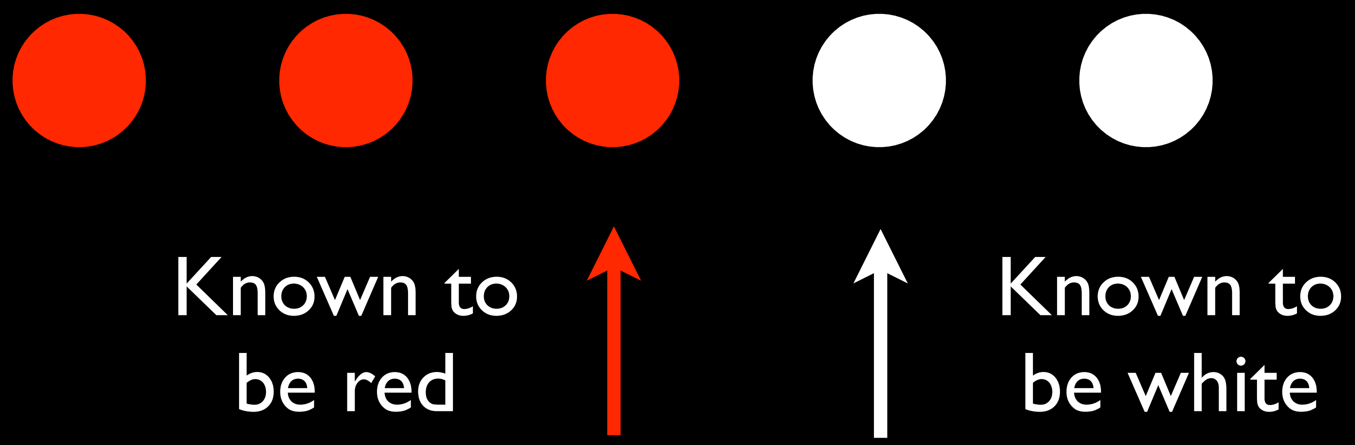

Known to be red
Known to be white

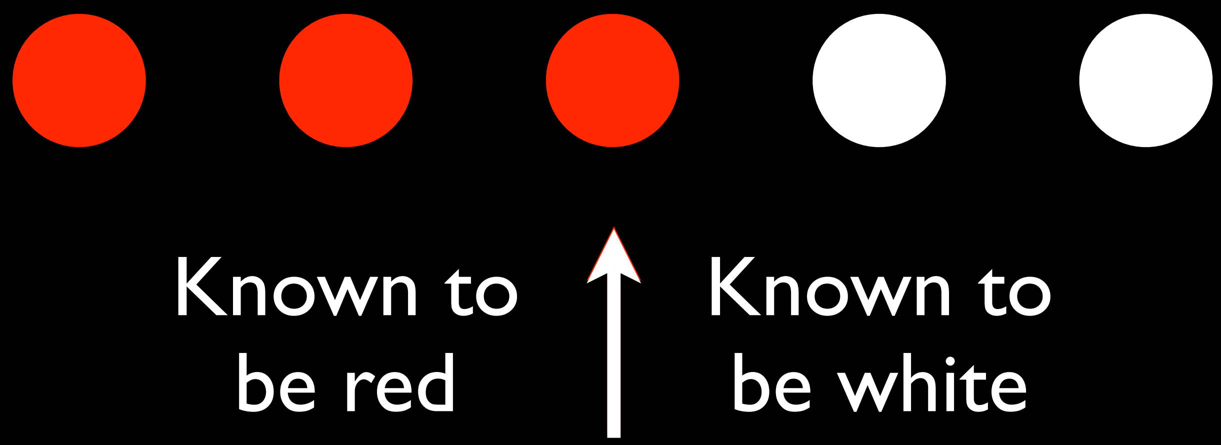

Known to be red
Known to be white


Known to
be red
Known to
be white


Known to
be red
Known to
be white


Known to be red
Known to be white


Known to be red
Known to be white


Known to be red
Known to be white

Can we find a solution:

- that terminates on all inputs;
- satisfies the specification;
- and has machine verified proofs of both these properties.

Plan of attack

- Use the dependently typed programming language Agda to:
    - implement the mini-computer;
    - write an algorithm that sorts the pebbles;
    - prove the algorithm correct.

The Mini-Computer

Pebbles

```
data Pebble : Set where
  Red : Colour
  White : Colour
```

Natural numbers

```
data Nat : Set where
  Zero : Nat
  Succ : Nat -> Nat
```

Buckets

```
data Buckets : Nat -> Set where
  Nil : Buckets Zero
  Cons : Pebble -> Buckets n ->
         Buckets (Succ n)
```

The state monad

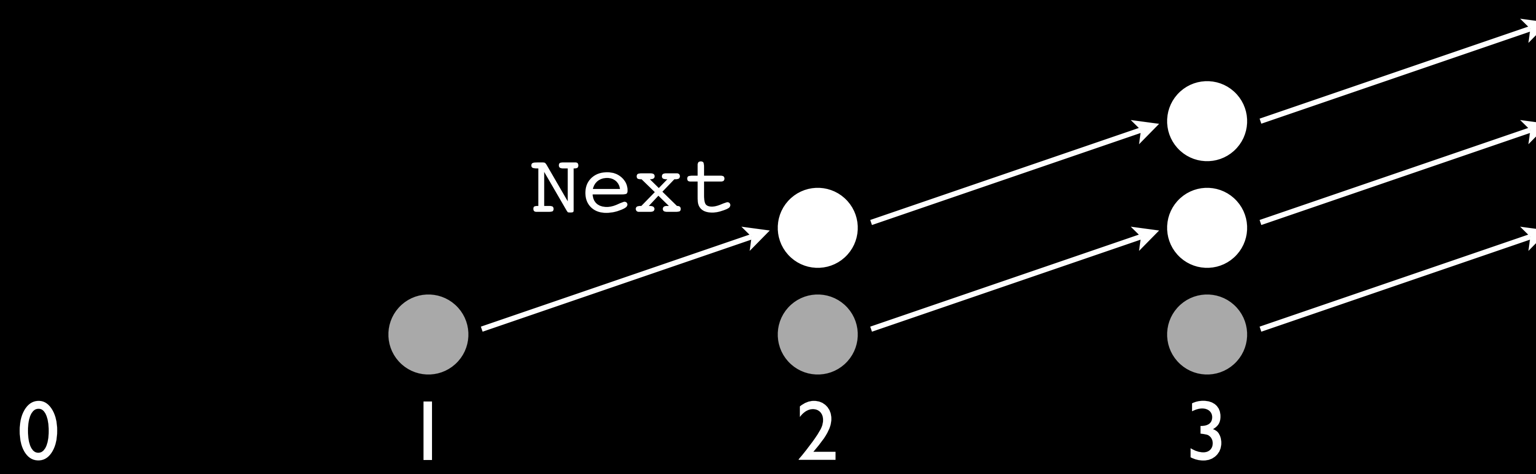
```
State : Nat -> Set -> Set
State n a =
  Buckets n -> Pair a (Buckets n)

return : a -> State n a
_>>=_ : State n a ->
  (a -> State n b) -> State n b
```

Indices

```
data Index : Nat -> Set where
  One : Index (Succ n)
  Next : Index n ->
         Index (Succ n)
```

Indices

```
data Index : Nat -> Set where
  One : Index (Succ n)
  Next : Index n ->
         Index (Succ n)
```



Reading

```
read : Index n -> State Pebble
read i bs = (bs ! i , bs)
  where
  _!_ : Buckets n -> Index n
        -> Pebble
  (Cons p _) ! One = p
  (Cons _ ps) ! (Next i) = ps ! i
```

Swap

```
swap : Index n -> Index n
       -> State n Unit
swap i j =
  read i >>= \pi ->
  read j >>= \pj ->
  write i pj >>
  write j pi
```

Back to the problem

An approximation

```
sort :: Index n -> Index n
     -> State n Unit
sort r w =
  if w == r then return unit
  else case read r of
    Red   -> sort (r + 1) w
    White -> swap r w >>
             sort r (w - 1)
```

An approximation

```
sort :: Index n -> Index n
     -> State n Unit
sort r w =
  if w == r then return unit
  else case read r of
    Red   -> sort (r + 1) w
    White -> swap r w >>
             sort r (w - 1)
```

Why does this terminate?

An approximation

```
sort :: Index n-> Index n
      -> State n Unit
sort r w =
  if r == w then return unit
  else case read r of
    White ->  sort (r + 1) w
    Red ->   swap r w >>
             sort r (w - 1)
```

An approximation

```
sort :: Index n-> Index n
    -> State n Unit
sort r w =
  if r == w then return unit
  else case read r of
    White ->  sort (r + 1) w
    Red ->  swap r w >>
              sort r (w - 1)
```

Only terminates

if $r \leq w$

Manipulating Indices

```
sort :: Index n-> Index n
      -> State n Unit
sort r w =
  if r == w then return unit
  else case read r of
    White -> sort (r + 1) w
    Red ->  swap r w >>
            sort r (w - 1)
```

Two problems

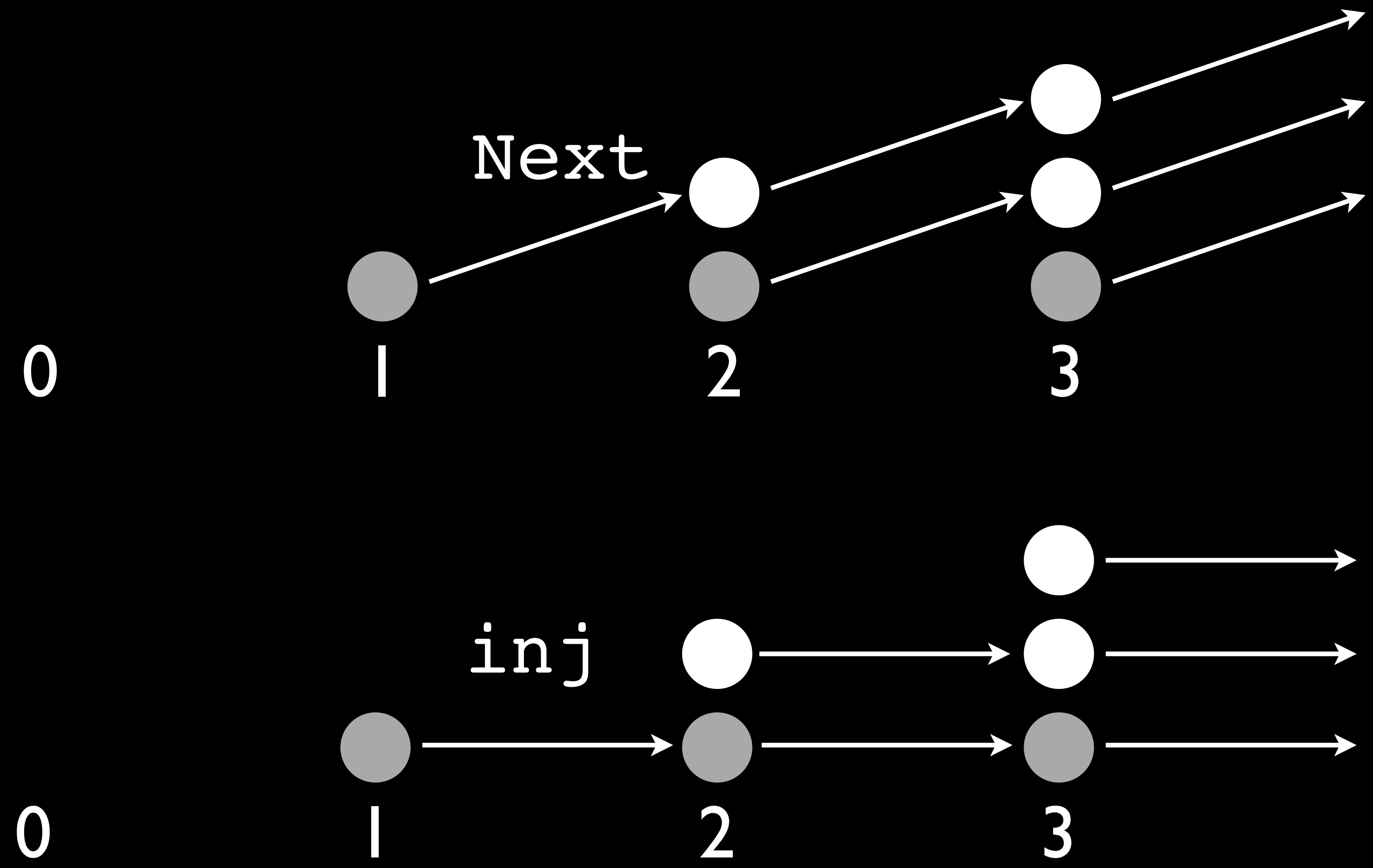
- We need to increment and decrement inhabitants of `Index n` ;
- We need to prove that our algorithm terminates.

```
Next : Index n -> Index (Succ n)
```

Injection

```
inj : Index n -> Index (Succ n)
inj One = One
inj (Next i) = Next (inj i)
```

Next or inj



Idea

- Only increment the image of `inj`;
- Only decrement the image of `Next`.

Less than or equal

```
data _<=_ : (i j : Index n) -> Set where
  Base : (i : Index (Succ n)) -> One <= i
  Step : (i j : Index n) ->
     (i <= j) -> (Next i <= Next j)
```

Difference

```
data Diff : (i j : Index n) -> Set where
  Base : (i : Index n) -> Diff i i
  Step : (i j : Index n) ->
     Diff i j -> Diff (inj i) (Next j)
```

Sort

```
sort : (r w : Index n) ->
       Diff r w ->
       State n Unit
```

Sort – Base case

```
sort : (r w : Index n) ->
       Diff r w ->
       State n Unit
sort .i .i (Base i) = return unit
```

```
sort : (r w : Index n) ->
       Diff r w ->
       State n Unit
```

```
sort : (r w : Index n) ->
       Diff r w ->
       State n Unit
sort .(inj i) .(Next j) (Step i j d) =
```

```
sort : (r w : Index n) ->
       Diff r w ->
       State n Unit
sort .(inj i) .(Next j) (Step i j d) =
    read (inj i) >>= \p ->
    case p of
      Red ->
      White ->
```

```
sort : (r w : Index n) ->
       Diff r w ->
       State n Unit
sort .(inj i) .(Next j) (Step i j d) =
    read (inj i) >>= \p ->
    case p of
      Red -> sort (Next i) (Next j) ?
      White ->
```

```
sort : (r w : Index n) ->
       Diff r w ->
       State n Unit
sort .(inj i) .(Next j) (Step i j d) =
    read (inj i) >>= \p ->
    case p of
      Red -> sort (Next i) (Next j) ?
      White ->
        swap (inj i) (Next j) >>
        sort (inj i) (inj j) ?
```

Lemmas

- We need to prove a few useful lemmas:
  - `Diff i j -> Diff (Next i) (Next j)`
  - `Diff i j -> Diff (inj i) (inj j)`

Lemmas

- We need to prove a few useful lemmas:
  - `Diff i j -> Diff (Next i) (Next j)`
  - `Diff i j -> Diff (inj i) (inj j)`

...but even then the algorithm is not *structurally* recursive.

Difference, revisited

```
data Diff : (i j : Index n) -> Set where
 Base : (i : Index n) -> Diff i i
 Step : (i j : Index n) ->
    Diff (inj i) (inj j) ->
    Diff (Next i) (Next j) ->
    Diff (inj i) (Next j)
```

Verification

Verification

the easy part

Formalizing the Invariant

```
Invariant : (r w : Index n)
  -> Buckets n -> Set
Invariant r w bs =
 (∀ i -> w < i -> bs ! i = White)
 && (∀ i -> i < r -> bs ! i = Red)
```

Correctness Theorem

```
∀ r w bs,
Invariant r w bs ->
  ∃ m : Index n,
  Invariant m m (sort r w bs)
```

Proof sketch

- Proof proceeds by induction on `Diff`
- Distinguish three cases:
  - Base case (trivial);
  - No swap happens (not too hard);
  - Swap happens (a bit trickier).
- In the latter two cases, we establish the invariant holds and make a recursive call.

The Dutch National Flag

- The *structure* of the algorithm stays the same.
  - similar invariant;
  - similar termination proof.
- Program does more case analysis...
- ... and so do the proofs.
- Messier but no harder.

Conclusions

- You need a PhD to verify a four line C program in Agda.
- ... but it is possible to verify non-structurally recursive, 'impure' functions in type theory.