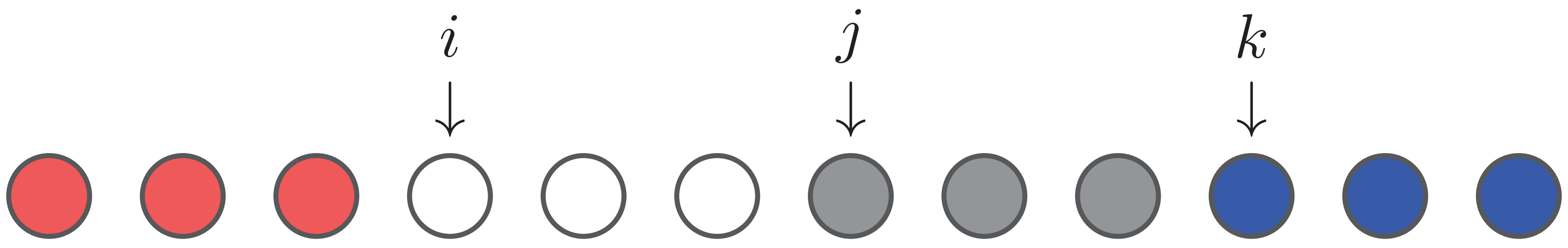
Solving the Dutch National Flag problem via datatype ornamentation

Josh Ko

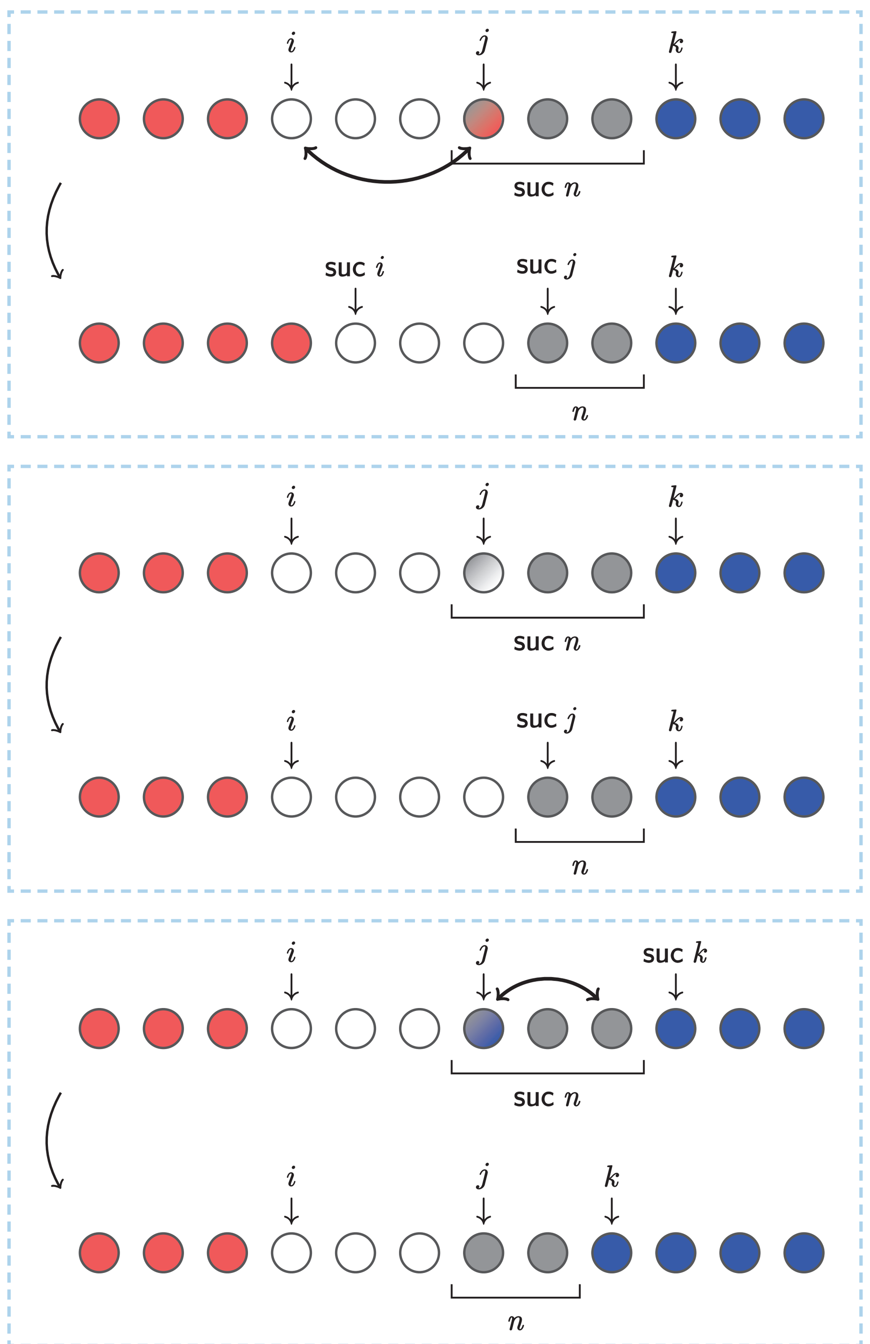
DEPARTMENT OF COMPUTER SCIENCE

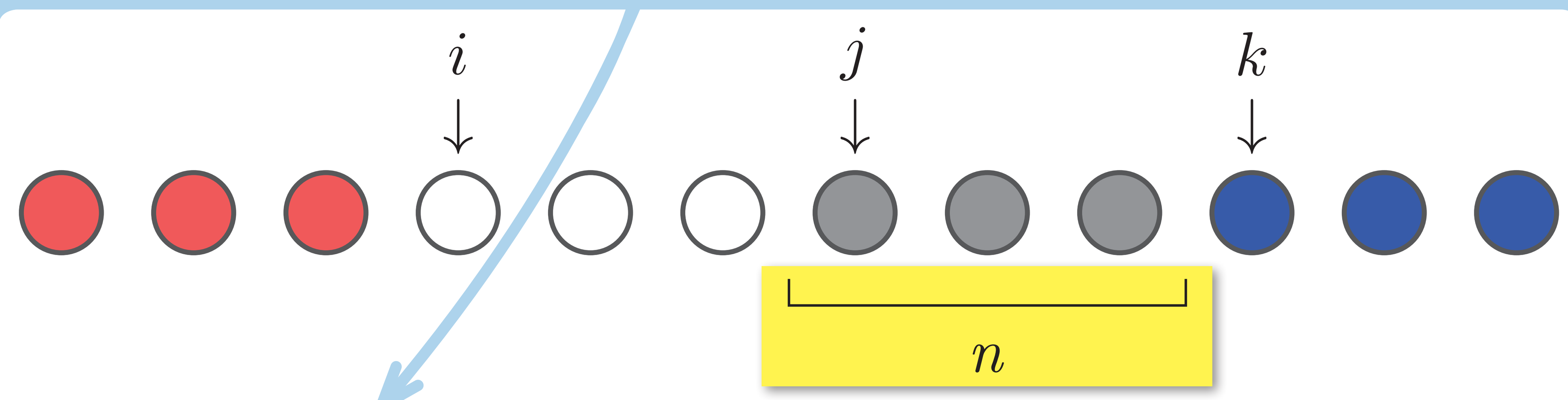
UNIVERSITY OF OXFORD

In modern dependently typed languages like Agda, datatypes can be specified such that their elements satisfy certain properties by construction. Consequently, programs constructing those data are manifestly correct without need for separate proofs, since being able to construct the data implies that the properties are indeed established.




```
data DVec : (i j k : Nat) → Set where
  []     : DVec 0 0 0
  _::r_  : Peb red        → ∀ {i j k} → DVec i j k → DVec (suc i) (suc j) (suc k)
  _::w_  : Peb white      → ∀ {  j k} → DVec 0 j k → DVec 0       (suc j) (suc k)
  _::_   : ∀ {c} → Peb c  → ∀ {    k} → DVec 0 0 k → DVec 0       0       (suc k)
  _::b_  : Peb blue       →             DVec 0 0 0 → DVec 0       0       0

lengthUnknown : ∀ {i j k} → DVec i j k → Nat
lengthUnknown []          = 0
lengthUnknown (x ::r xs)  = lengthUnknown xs
lengthUnknown (x ::w xs)  = lengthUnknown xs
lengthUnknown (x ::  xs)  = suc (lengthUnknown xs)
lengthUnknown (x ::b xs)  = 0

firstUnknownColour : ∀ {i j k} → (xs : DVec i j k) → Maybe (lengthUnknown xs) Colour
firstUnknownColour []              = tt
firstUnknownColour (x ::r xs)      = firstUnknownColour xs
firstUnknownColour (x ::w xs)      = firstUnknownColour xs
firstUnknownColour (_::_ {c} x xs) = c
firstUnknownColour (x ::b xs)      = tt
```



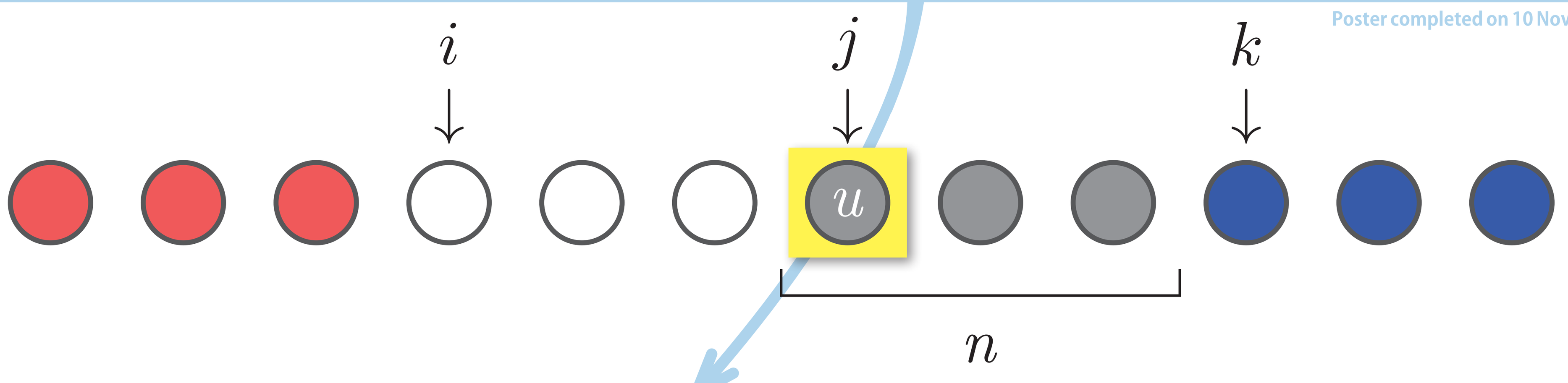



```
data DVec : (i j k n : Nat) → Set where
  []     : DVec 0 0 0 0
  _::r_  : Peb red        → ∀ {i j k n} → DVec i j k n → DVec (suc i) (suc j) (suc k) n
  _::w_  : Peb white      → ∀ {  j k n} → DVec 0 j k n → DVec 0       (suc j) (suc k) n
  _::_   : ∀ {c} → Peb c  → ∀ {    k n} → DVec 0 0 k n → DVec 0       0       (suc k) (suc n)
  _::b_  : Peb blue       → ∀ {      n} → DVec 0 0 0 n → DVec 0       0       0       0

firstUnknownColour : ∀ {i j k n} → (xs : DVec i j k n) → Maybe n Colour

reduce : ∀ {i j k n} → DVec i j k (suc n) → ∃3 (λ i' j' k' ↦ DVec i' j' k' n)
reduce xs with firstUnknownColour xs
reduce xs | red    = { }0
reduce xs | white  = { }1
reduce xs | blue   = { }2
```

About the author Hsiang-Shang 'Josh' Ko (柯向上) is a DPhil student at Oxford, supervised by Professor Jeremy Gibbons and working on modularity issues in dependently typed programming; he did his undergraduate degree at National Taiwan University, Taiwan.

Acknowledgements Thanks to suggestions from Jeremy Gibbons and Yen-Chen Pan (潘彥丞), and financial support from the University of Oxford Clarendon Fund Scholarship and the UK EPSRC project *Reusability and Dependent Types*.

Poster completed on 10 Nov 2011




```
data DVec : (i j k n : Nat) (u : Maybe n Colour) → Set where
  []     : DVec 0 0 0 0 tt
  _::r_  : Peb red        → ∀ {i j k n u} → DVec i j k n u → DVec (suc i) (suc j) (suc k) n       u
  _::w_  : Peb white      → ∀ {  j k n u} → DVec 0 j k n u → DVec 0       (suc j) (suc k) n       u
  _::_   : ∀ {c} → Peb c  → ∀ {    k n u} → DVec 0 0 k n u → DVec 0       0       (suc k) (suc n) c
  _::b_  : Peb blue       → ∀ {      n u} → DVec 0 0 0 n u → DVec 0       0       0       0       tt

reduce : ∀ {i j k n u} → DVec i j k (suc n) u → ∃4 (λ i' j' k' u' ↦ DVec i' j' k' n u')
reduce {u = red  } xs  =  _ , π2 (reduceRed   xs)
reduce {u = white} xs  =  _ , π2 (reduceWhite xs)
reduce {u = blue } xs  =  { }2

reduceRed : ∀ {i j k n} → DVec i j k (suc n) red → ∃ (λ u' ↦ DVec (suc i) (suc j) k n u')
reduceRed (y ::r ys)  =  _ , y ::r π2 (reduceRed ys)
reduceRed (y ::w ys)  =  _ , focus ys ::r π2 (subst y ys)
  where focus : ∀ {j k n} → DVec 0 j k (suc n) red → Peb red
        focus (z ::w zs)  =  focus zs
        focus (z ::  zs)  =  z
        subst : ∀ {j k n} → Peb white → DVec 0 j k (suc n) red → ∃ (λ u' ↦ DVec 0 (suc j) k n u')
        subst y (z ::w zs)  =  _ , z ::w π2 (subst y zs)
        subst y (z ::  zs)  =  _ , y ::w zs
reduceRed (y ::  ys)  =  _ , y ::r ys

reduceWhite : ∀ {i j k n} → DVec i j k (suc n) white → ∃ (λ u' ↦ DVec i (suc j) k n u')
reduceWhite (y ::r ys)  =  _ , y ::r π2 (reduceWhite ys)
reduceWhite (y ::w ys)  =  _ , y ::w π2 (reduceWhite ys)
reduceWhite (y ::  ys)  =  _ , y ::w ys

reduceBlue : ∀ {i j k n} → DVec i j (suc k) (suc n) blue → ∃ (λ u' ↦ DVec i j k n u')
reduceBlue (y ::r ys)  =  { }3
reduceBlue (y ::w ys)  =  { }4
reduceBlue (y ::  ys)  =  { }5
```

What properties to encode in datatypes are often discovered only gradually during program development. Ornamentation suggests a way of supporting incremental specification of precise datatypes to match our development patterns.

Algebraic ornamentation

```
data List (A : Set) : Set where          length : {A : Set} → List A → Nat
  []    : List A                          length []         = 0
  _::_  : A → List A → List A             length (x :: xs)  = suc (length xs)

    data Vec (A : Set) : Nat → Set where
      []    : Vec A 0
      _::_  : A → {n : Nat} → Vec A n → Vec A (suc n)

    -- Vec A n ≅ Σ (List A) (λ xs ↦ length xs ≡ n)
```