DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF OXFORD

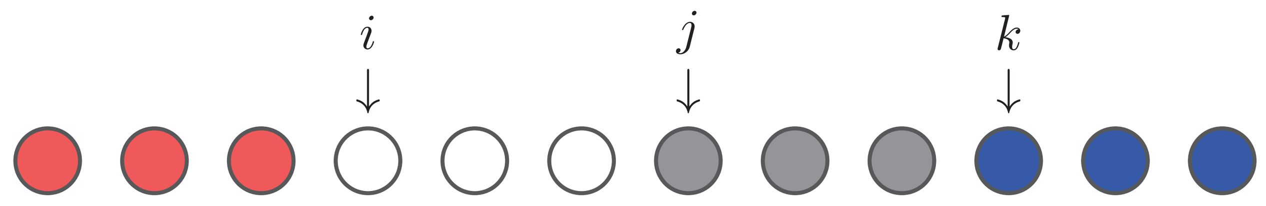
To prove that a program satisfies its specification, it is more intellectually manageable to construct the program such that its structure enables the programmer to "give convincing argument for its correctness" [2], or even serves as a proof by itself, making the program manifestly correct.

Solving the Dutch National Flag problem via datatype ornamentation

Josh Ko (University of Oxford)

The DNF problem, inductively

Dijkstra introduced the *Dutch National Flag problem*, asking for a way to rearrange an array of pebbles of colour red, white, or blue in the order of the Dutch National Flag using only swaps, and proposed an imperative solution [1]. The key invariant of the algorithm is that the array is divided into four sections containing red, white, "unknown," and blue pebbles respectively. Initially the colours of the pebbles are all regarded as unknown, and the algorithm proceeds by reducing the length of the unknown section. We formulate the invariant inductively and encode it in the list-like *Dutch vector* datatype DVec, which uses different conses for different sections and guarantees that the sections are ordered as specified. Subsequent programs are written on the Dutch vectors and thus necessarily maintain the invariant.



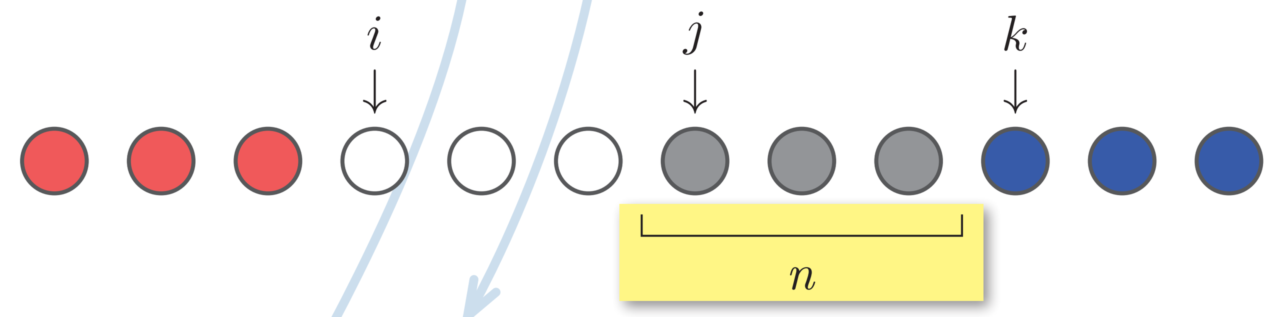
Peb *c* is the type of pebbles of colour *c*.

```
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
```

To decide how to reduce the length of the unknown section, we need to know the colour of the first unknown pebble.

```
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
```

```
Maybe : Nat → Set → Set
Maybe 0       A = ⊤
Maybe (suc _) A = A
```

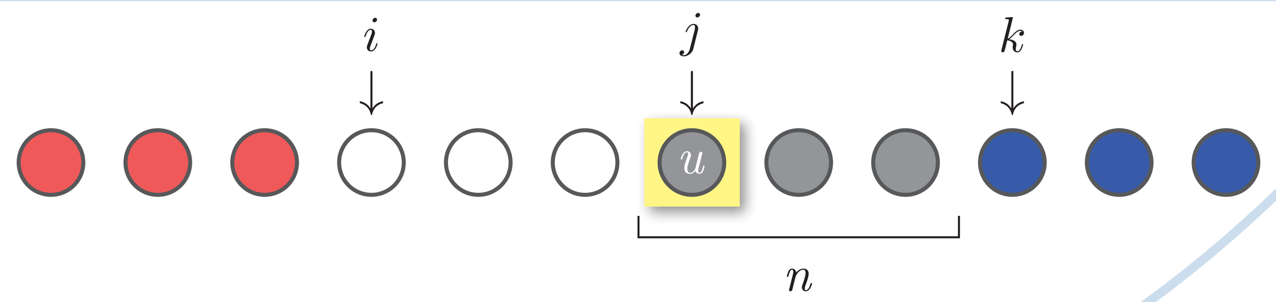



```
data DVec : (i j k n : Nat) → Set where
  []      : DVec 0 0 0 0
  _::r_   : Peb red   → ∀{i j k n} → DVec i j k n → DVec (suc i) (suc j) (suc k) n
  _::w_   : Peb white → ∀{  j k n} → DVec 0 j k n → DVec 0      (suc j) (suc k) n
  _::_    : ∀{c} → Peb c → ∀{  k n} → DVec 0 0 k n → DVec 0     0 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

```
∃³ : {A : Set} {B : A → Set} {C : (a : A) → B a → Set} →
     ((a : A) (b : B a) (c : C a b) → Set) → Set
∃³ {A} {B} {C} p = Σ (Σ A (λ a ↦ Σ (B a) (λ b ↦ C a b)))
                     (λ abc ↦ p (π1 abc) (π1 (π2 abc)) (π2 (π2 abc)))
```

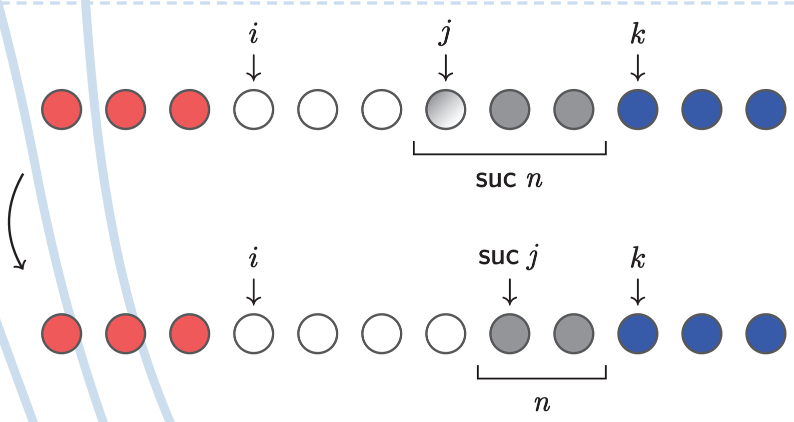
By algebraic ornamentation, the length of the unknown section is integrated into the type of the Dutch vectors. This simplifies the type of *firstUnknownColour*, and the length will serve as an explicit termination measure. We then attempt to describe how to reduce the length of the unknown section by one, which requires a case analysis on the result of *firstUnknownColour*. The case analysis does not directly reveal more information about the input Dutch vector, however — to do so, that result has to be exposed in the type by another algebraic ornamentation.

In modern dependently typed languages like Agda [4], data can be specified such that they 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.

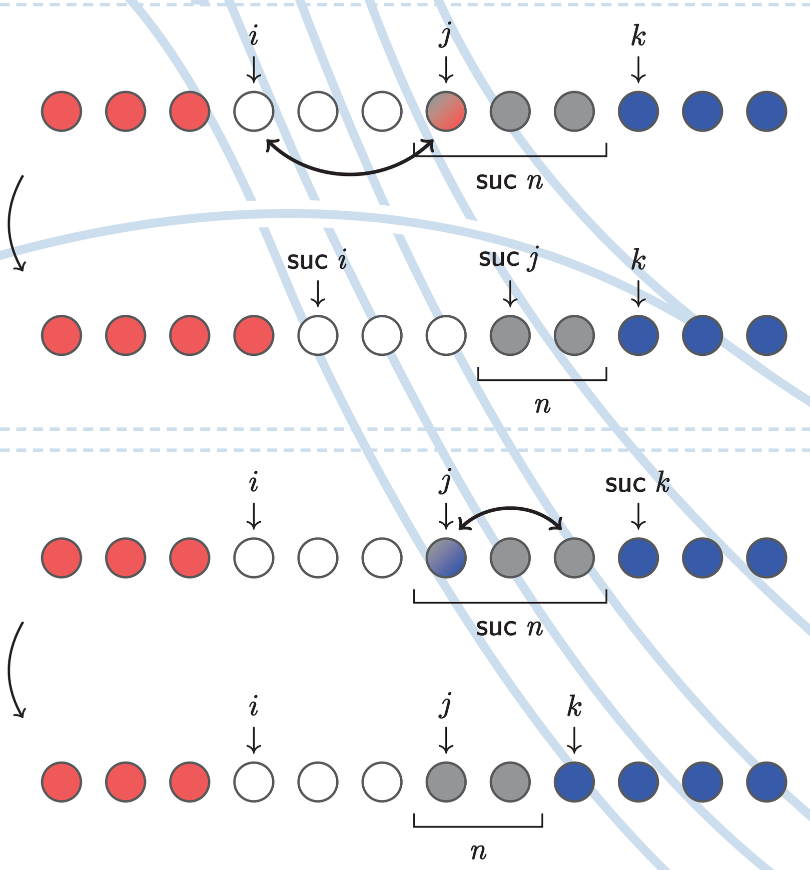



```
data DVec : (i j k n : Nat) (u : Maybe n Colour) → Set where
  []      : DVec 0 0 0 0 tt
  _::r_   : Peb red   → ∀{i j k n u} → DVec i j k n u → DVec (suc i) (suc j) (suc k) n       u
  _::w_   : Peb white → ∀{  j k n u} → DVec 0 j k n u → DVec 0      (suc j) (suc k) n       u
  _::_    : ∀{c} →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

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

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

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
```

Now we can proceed with the case analysis. The white and red cases are straightforward, but the blue case poses some problems.



At goals 3 and 4, we cannot make recursive calls because the index *k* in the type of *ys* is not a successor, and at goal 5 we cannot use the blue cons when solving the base case of the swap since *k* is not zero. These problems are superficial, however, and can be overcome if we express the connection between *j*, *k*, and *n* more precisely.




```
data _−_≈_ : (k j n : Nat) → Set where
  0   : ∀{j} → j − j ≈ 0
  suc : ∀{k j n} → k − j ≈ n → suc k − j ≈ suc n

inj : ∀{k j n} → k − j ≈ n → suc k − suc j ≈ n
inj 0       = 0
inj (suc d) = suc (inj d)

difference : ∀{i j k n u} → DVec i j k n u → k − j ≈ n
difference []         = 0
difference (x ::r xs) = inj (difference xs)
difference (x ::w xs) = inj (difference xs)
difference (x ::  xs) = suc (difference xs)
difference (x ::b xs) = 0
```

What is algebraic ornamentation?

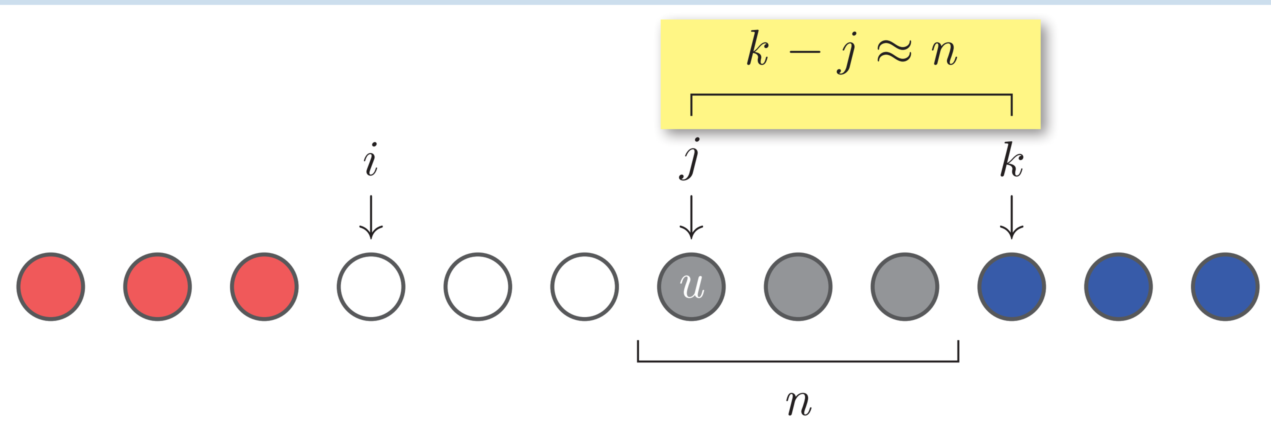
An *algebraic ornamentation* adds an extra index to a datatype such that the index in the type of an element is always the value computed by a particular fold on that element. This technique was identified and formalised by McBride in a datatype-generic framework of *ornaments* for expressing relationship between datatypes [3]. Algebraic ornamentation allows some properties that can be established by simple induction to be integrated into the data. Programs can then exploit those properties directly, rather than having to manipulate separate proofs about them.

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A → List A → List A

length : {A : Set} → List A → Nat
length []       = 0
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)
```

Poster completed on 30 Oct 2011

```
dutchFlag : List (Σ Colour Peb) → List (Σ Colour Peb)
dutchFlag = forget ∘ π2 ∘ elimUnknown ∘ π2 ∘ initialise

forget : ∀{i j k n d u} → DVec i j k n d u → List (Σ Colour Peb)
forget []         = []
forget (x ::r xs) = (red   , x) :: forget xs
forget (x ::w xs) = (white , x) :: forget xs
forget (x ::  xs) = (_     , x) :: forget xs
forget (x ::b xs) = (blue  , x) :: forget xs

initialise : (xs : List (Σ Colour Peb)) → let l = length xs in ∃ (λ u ↦ DVec 0 0 l l fuel u)
initialise []             = tt , []
initialise ((c , x) :: xs) = c , x :: π2 (initialise xs)

fuel : ∀{k} → k − 0 ≈ k
fuel {0}     = 0
fuel {suc _} = suc fuel
```

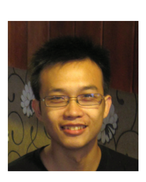
We can wrap the Dutch vector program in a list program, localising the use of the Dutch vector datatype.




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

reduce : ∀{i j k n d u} → DVec i j k (suc n) d u → ∃⁵ (λ i' j' k' d' u' ↦ DVec i' j' k' n d' u')
reduce         {u = red  } xs = _ , π2 (reduceRed xs)
reduce         {u = white} xs = _ , π2 (reduceWhite xs)
reduce {d = suc _} {u = blue } xs = _ , π2 (reduceBlue xs)

reduceWhite : ∀{i j k n d} → DVec i j k (suc n) d white → ∃² (λ d' u' ↦ DVec i (suc j) k n d' u')

reduceRed : ∀{i j k n d} → DVec i j k (suc n) d red → ∃² (λ d' u' ↦ DVec (suc i) (suc j) k n d' u')

reduceBlue : ∀{i j k n d} → DVec i j (suc k) (suc n) d blue → ∃² (λ d' u' ↦ DVec i j k n d' u')
reduceBlue (_::r_ y {d = suc _} ys) = _ , y ::r π2 (reduceBlue ys)
reduceBlue (_::w_ y {d = suc _} ys) = _ , y ::w π2 (reduceBlue ys)
reduceBlue (_::_  y {d = 0    } ys) = _ , y ::b ys
reduceBlue (_::_  y {d = suc _} ys) = _ , (focus ys) :: π2 (subst y ys)
  where focus : ∀{k n d u} → DVec 0 0 k (suc n) d u → ∃ (λ u' ↦ Peb u')
        focus {n = 0    } (z :: zs) = _ , z
        focus {n = suc _} (z :: zs) = focus zs
        subst : ∀{k n d u} → Peb blue → DVec 0 0 (suc k) (suc n) (suc d) u → ∃ (λ u' ↦ DVec 0 0 k n d u')
        subst y (_::_ z {d = 0    } zs) = _ , y ::b zs
        subst y (_::_ z {d = suc _} zs) = _ , z :: π2 (subst y zs)

elimUnknown : ∀{i j k n d u} → DVec i j k n d u → ∃⁴ (λ i' j' k' d' ↦ DVec i' j' k' 0 d' tt)
elimUnknown {n = 0    } xs = _ , xs
elimUnknown {n = suc _} xs = elimUnknown (π2 (reduce xs))
```

Now the blue case can be completed by invoking the fact that *n* is the difference between *k* and *j*.

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



About the author Hsiang-Shang 'Josh' Ko (柯向上) is a DPhil student at Oxford supervised by Professor Jeremy Gibbons. He is working on modularity and reusability issues in dependently typed programming, focusing particularly on ornamentation-based techniques. Previously he finished his undergraduate studies in Computer Science and Information Engineering at National Taiwan University, Taiwan.
Acknowledgements The author would like to thank Yen-Chen Pan (潘彥丞) for offering very helpful suggestions and feedback. This work was completed through support of the University of Oxford Clarendon Fund Scholarship and the UK Engineering and Physical Sciences Research Council project *Reusability and Dependent Types*.

References
[1] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
[2] E. W. Dijkstra. On the interplay between mathematics and programming. In *Program Construction*, LNCS 69, pp 35–46. Springer-Verlag, 1979.
[3] C. McBride. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*.
[4] U. Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, LNCS 5832, pp 230–266. Springer-Verlag, 2009.