ZU064-05-FPR DNFP 18 October 2010 13:1

FUNCTIONAL PEARLS
Sorted
Verifying the Problem of the Dutch National Flag in Type Theory

Wouter Swierstra
Radboud University Nijmegen
(*e-mail:* `w.swierstra@cs.ru.nl`)

1 Introduction

The problem of the Dutch national flag was formulated by Dijkstra (1976) as follows:

> *There is a row of buckets numbered from* 1 *to n. It is given that:*
> *P1*: *each bucket contains one pebble;*
> *P2*: *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.*

The mini-computer in question should perform this rearrangement using two commands:

- *swap* (i,j) for $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant n$ exchanges the pebbles stored in the buckets numbered i and j;
- *read* (i) for $1 \leqslant i \leqslant n$ returns the colour of the pebble currently lying in bucket number i. Dijkstra originally named this operation *buck*.

Finally, a solution should also satisify the following two non-functional requirements:

- the mini-computer may only allocate a constant amount of memory;
- every pebble may be inspected at most once.

This pearl describes how to solve and verify the problem of the Dutch national flag in type theory. For the sake of presentation, most of this paper considers the problem of the Polish national flag, where the pebbles are either red or white. Initially, we will only be concerned with finding a satisfactory solution to the problem (Sections 3–5). Although I will not cover the proof of correctness in detail, I will sketch the key lemmas and definitions (Section 6) and discuss how it may be extended to handle the case for three colours (Section 7).

I will use the dependently typed programming language Agda (Norell, 2007) throughout this paper and assume that you have had some previous exposure to programming with dependent types. If you are unfamiliar with either of these topics, you may want to consult one of the many articles that are currently available on these topics (Bove & Dybjer, 2008; McBride, 2004; Norell, 2008; Oury & Swierstra, 2008).

2 A functional specification of the mini-computer

Before we can tackle the problem of the Dutch national flag, we need to give a type theoretic account of the mini-computer and its commands.

The primitive commands with which we can program the mini-computer take numbers between 1 and *n* as their arguments. One way to represent these numbers is as follows:

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

The type *Index n* has *n* canonical inhabitants. You may already be familiar with some examples of such finite types: *Index* 0 is isomorphic to the empty type; *Index* 1 is isomorphic to the unit type; *Index* 2 is isomorphic to the Boolean type.

Note that in the typeset code, I will adopt Haskell and Epigram's convention of implicitly universally quantifying any unbound variables in type signatures. For example, the variable *n* used in both constructors of the *Index* type is implicitly quantified at the start of both type declarations. To make such implicit arguments explicit, either in a type signature or in a function definition, they will be enclosed in a pair of curly brackets.

The *Buckets n* data type below describes the pebbles that are currently in each of the *n* buckets:

```
data Pebble : Set where
  Red : Pebble
  White : Pebble
data Buckets : Nat → Set where
  Nil : Buckets Zero
  Cons : Pebble → Buckets n → Buckets (Succ n)
```

The solution we present here will be structured using a state monad:

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

I will use Haskell's notation for the unit (*return*) and bind (>>=) operations.

A function in the state monad can be executed to observe its effects:

```
exec : State n a → Buckets n → Buckets n
exec f bs = snd (f bs)
```

We can now define the *read* function, that returns the pebble stored at the bucket with its argument index. We do so using an auxiliary dereferencing operator that looks up the pebble stored at a particular index:

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

Note that the dereferencing operator is *total*. In the *Nil* branch, we know that there is no possible inhabitant of *Index Zero* and we supply the 'impossible' pattern () and omit the right-hand side of the definition accordingly.

Before defining the *swap* operation, it is convenient to define the following functions:

```
write : Index n → Pebble → State n Unit
write i p bs = (unit, update i p bs)
  where
  update : Index n → Pebble → Buckets n → Buckets n
  update One x (Cons p ps) = Cons x ps
  update (Next i) x (Cons p ps) = Cons p (update i x ps)
```

Calling *write i p* replaces the pebble stored in bucket number *i* with its argument pebble. Although the interface of the mini-computer does not support this operation, we can use it to define *swap* as follows:

```
swap : Index n → Index n → State n Unit
swap i j = read i >>= λp_i →
           read j >>= λp_j →
           write i p_j >>
           write j p_i
```

Providing definitions for *swap* and *read* completes the functional specification of the mini-computer. This specification is in fact a degenerate case of the functional specification of mutable state in my thesis (Swierstra, 2008). As the mini-computer cannot allocate new buckets, it is considerably simpler.

3 A first attempt

It's now time to sketch a solution to the simplified version of the problem with only two colours. In the coming sections, we will refine this solution to a valid Agda program.

Dijkstra's key insight is that during the execution of any solution, the row of buckets must be divided into separate zones of consecutively numbered buckets. In the simple case with only two colours, we will need three disjoint zones: the zone of buckets storing pebbles known to be red; the zone of buckets storing pebbles known to be white; and the zone of buckets storing pebbles of indeterminate colour.

To delineate these zones, we need to keep track of two numbers r and w. Throughout the execution of our solution, we will maintain the following invariant on r and w:

- for all k, where $1 \leqslant k < r$, the pebble in bucket number k is known to be red;
- and for all k, where $w < k \leqslant n$, the pebble in bucket number k is known to be white.

Note that this invariant does not say anything about the pebbles stored in buckets numbered k for $r \leqslant k \leqslant w$. In particular, if we initialize r and w to 1 and n respectively the invariant is trivially true.

With this invariant in mind, we might arrive at the (pseudocode) solution for the problem of the Polish national flag in Figure 1.

```
$sort : Index\ n \rightarrow Index\ n \rightarrow State\ n\ Unit$
$sort\ r\ w = \mathbf{if}\ r \equiv w\ \mathbf{then}\ return\ unit$
              $\mathbf{else}\ read\ r \gg\!\!= \lambda c \rightarrow$
                $\mathbf{case}\ c\ \mathbf{of}$
                  $Red \rightarrow sort\ (r+1)\ w$
                  $White \rightarrow swap\ r\ w \gg sort\ r\ (w-1)$
```

Fig. 1. A pseudocode definition of the *sort* function

If $r \equiv w$, there is no further sorting necessary as a consequence of our invariant. Otherwise we inspect the pebble stored in bucket number r. If this pebble is red, we have established the invariant holds for $r+1$ and w. We can therefore increment r and make a recursive call without having to reorder any pebbles.

If we encounter a white pebble in bucket number r, there is more work to do. The call *swap* (r, w) ensures that all the pebbles in buckets with a number k, for $w \leqslant k \leqslant n$, are white. Put differently, after this *swap* we can establish that our invariant holds for r and $w-1$. In contrast to the previous case, the recursive call decrements w instead of incrementing r.

There are several problems with this definition. Firstly, it is not structurally recursive and therefore it is rejected by Agda's termination checker. This should come as no surprise: the function call *sort r w* only terminates provided $r \leqslant w$, as the difference between w and r decreases in every recursive step. The Agda solution must make this informal argument precise.

Furthermore, we have not defined how to increment or decrement inhabitants of *Index n*. Before we try to implement the *sort* function in Agda, we will have a closer look at the structure of such finite types.

4 Finite types

How shall we define the increment and decrement operations on inhabitants of *Index n*?

You might be tempted to use the constructor *Next* as our increment operation. Recall that *Next* has type $Index\ n \rightarrow Index\ (Succ\ n)$, whereas we would like to have a function of type $Index\ n \rightarrow Index\ n$. Similarly, "peeling off" a *Next* constructor does not give yield to a decrement operation of the desired type.

The *Next* constructor, however, is not the only way to embed an inhabitant of *Index n* into $Index\ (Succ\ n)$. Another choice of embedding is the *inj* function, given by:

```
$inj : Index\ n \rightarrow Index\ (Succ\ n)$
$inj\ One = One$
$inj\ (Next\ i) = Next\ (inj\ i)$
```

Despite appearances, *inj* is not the identity function. The *inj* function maps $One : Index\ n$ to $One : Index\ (Succ\ n)$ – thereby changing its type. We can visualise the difference between *inj* and *Next*, mapping *Index* 3 to *Index* 4, in Figure 2.

Figure 2(a) contains the graph of the *inj* function. I have numbered the elements of *Index* 3 and *Index* 4 on the left and right respectively. The *inj* function maps the *One*

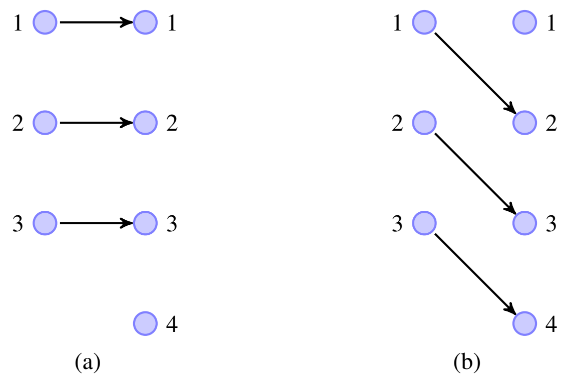

Fig. 2. The graph of the *inj* function (a) and the *Next* constructor (b) on *Index* 3

element of *Index* 3 to the *One* element of *Index* 4; similarly, *Next i* is mapped to *Next* (*inj i*). As the Figure 2(b) illustrates, the *Next* constructor behaves quite differently. It increments all the indices in *Index* 3, freeing space for a new index, *One* : *Index* 4.

From this picture, we can make the central observation: we should only increment indices in the image of *inj* and only decrement indices in the image of *Next*.

The question remains: how do we know when an index is in the image of *inj* or *Next*? Surprisingly, we will acquire this information as a consequence of making the algorithm structurally recursive.

5 A structurally recursive solution

To revise our definition of sorting, we need to make the structure of the recursion explicit. Informally, we have previously established that the function call *sort r w* function will terminate provided $r \leqslant w$. The usual choice of order on inhabitants of *Index n* is given by the following data type:

```
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
```

The base case states that *One* is the least inhabitant of any non-empty finite type. Provided we have already established that $i \leqslant j$, the *Step* constructor proves that *Next* $i \leqslant$ *Next* j.

This definition, however, does not reflect the structure of our algorithm. A better choice is to define the following data type, representing the difference between two inhabitants of *Index n*:

```
data Difference : (i j : Index n) → Set where
  Same : (i : Index n) → Difference i i
  Step : (i j : Index n) → Difference i j → Difference (inj i) (Next j)
```

The base case, *Same*, captures the situation when the two indices are the same; the *Step* constructor increases the difference between the two indices by incrementing the greater of the two.

```
sort : (r : Index n) → (w : Index n) → Difference r w → State n Unit
sort ⌊i⌋ ⌊i⌋ (Same i) = return unit
sort ⌊inj i⌋ ⌊Next j⌋ (Step i j p) =
  read (inj i) >>= λc →
  case c of
    Red → sort (Next i) (Next j) (nextDiff i j p)
    White → swap (inj i) (Next j) >>
               sort (inj i) (inj j) (injDiff i j p)
```

Fig. 3. The definition of the *sort* function

Using this definition of *Difference*, we define our sorting function by induction on the difference between *r* and *w* in Figure 3. Note that Agda does not provide local **case** statements – the accompanying code defines *sort* using a fold over the *Pebble* type. I have chosen to typeset this fold as a case statement for the sake of clarity.

The pattern matching in this definition deserves some attention. In the first branch, we match on the *Same* constructor. As a result of this pattern match, we learn that *r* and *w* can only be equal to the argument *i* of the *Same* constructor. This information is reflected by the forced pattern $\lfloor i \rfloor$ that we see in place of the arguments *r* and *w*.

By pattern matching on the *Step* constructor, we also learn something about *r* and *w*: as they are not equal, *r* and *w* must be in the images of *inj* and *Next* respectively. The definition of this branch closely follows the pseudocode solution we have seen previously. It reads the pebble in bucket number *inj i*. If it is red, we continue sorting with *Next i* and *Next j*, thereby incrementing *inj i*. If it is white, we perform a swap and continue sorting with *inj i* and *inj j*, thereby decrementing *Next j*. This is where we apply our observation on incrementing and decrementing inhabitants of *Index n* from the previous section.

To perform the recursive calls, we need to define two lemmas with the following types:

```
nextDiff : (i j : Index n) → Difference i j → Difference (Next i) (Next j)
injDiff : (i j : Index n) → Difference i j → Difference (inj i) (inj j)
```

Both these lemmas are easy to prove by induction on the *Difference* between *i* and *j*.

Unfortunately, this definition of *sort* is still not structurally recursive. The *sort* function is defined by induction on the difference between *r* and *w*, but the recursive calls are not to structurally smaller subterms, but rather require the application of an additional lemma. Therefore it is still not accepted by Agda's termination checker.

The solution is to revise our *Difference* data type once again. Instead of requiring the application of the above two lemmas, we bake the proofs required for the two recursive calls into the data type over which we recurse:

```
data SortT : (i j : Index n) → Set where
  Base : (i : Index n) → SortT i i
  Step : (i j : Index n) →
           SortT (Next i) (Next j) → SortT (inj i) (inj j) → SortT (inj i) (Next j)
```

```
sort : (r : Index n) → (w : Index n) → SortT r w → State n Unit
sort ⌊i⌋ ⌊i⌋ (Base i) = return unit
sort ⌊inj i⌋ ⌊Next j⌋ (Step i j pDiff pInj) =
   read (inj i) ≫= λc →
   case c of
      Red → sort (Next i) (Next j) pDiff
      White → swap (inj i) (Next j) ≫
                 sort (inj i) (inj j) pInj
```

Fig. 4. The final definition of the *sort* function

Of course, we can show this definition to be equivalent to the original *Difference* type using the *nextDiff* and *injDiff* lemmas. I have chosen the name *SortT* for this data type as it encodes a the conditions under which the *sort* function will terminate. More generally, this is an instance of the Bove-Capretta method (Bove & Capretta, 2005), that calculates such a type from any non-structurally recursive definition.

The final definition of the *sort* function, using the *SortT* predicate, is given in Figure 4.

All that remains to be done to solve the problem of the Polish national flag is to call *sort* with suitable initial arguments. We initialise r to *One* and w to *maxIndex k*, the largest inhabitant of *Index* (*Succ k*). To kick off the sorting function, we must still provide a proof that *SortT One* (*maxIndex k*) is inhabited, calculated by the *terminates* function.

```
polish : (n : Nat) → State n Unit
polish Zero = return unit
polish (Succ k) = sort One (maxIndex k) terminates
   where
      maxIndex : (n : Nat) → Index (Succ n)
      maxIndex Zero = One
      maxIndex (Succ k) = Next (maxIndex k)
      terminates : SortT One (maxIndex k)
      terminates = toSortT One (maxIndex k) (Base (maxIndex k))
```

The easiest way to prove termination is by exploiting the equivalence between $i \leqslant j$ and *SortT i j*, witnessed by the function *toSortT*. Clearly *One* $\leqslant$ *maxIndex k*, by the *Base* constructor. Passing this proof as an argument to *toSortT* then gives the required proof of termination. The definition of *toSortT* proceeds by recursion over the two *Index* arguments.

This completes our solution to the problem of the Polish national flag. Now all we need to do, is prove it correct.

6 Verification

With this relatively simple definition, the verification turns out to be straightforward. Stepping through large proof terms written in type theory can be rather tedious and hence I will refrain from doing so. To give you some idea of how the verification proceeds, I shall present the key definitions, formulate the necessary lemmas, and sketch their proofs.

Many of the proofs use the following property of the *swap* function:

$$\begin{array}{l}
\mathit{swapProp} : \mathit{Buckets}\ n \to (i\ j\ k : \mathit{Index}\ n) \to \mathit{Set} \\
\mathit{swapProp}\ \mathit{bs}\ i\ j\ k = \\
\quad \textbf{let}\ \mathit{bs}' = \mathit{exec}\ (\mathit{swap}\ i\ j)\ \mathit{bs}\ \textbf{in} \\
\quad (k \not\equiv i) \wedge (k \not\equiv j)\ \to\ (\mathit{bs}'\,!\,k \equiv \mathit{bs}\,!\,k)\ \wedge\ (\mathit{bs}'\,!\,j \equiv \mathit{bs}\,!\,i)\ \wedge\ (\mathit{bs}'\,!\,i \equiv \mathit{bs}\,!\,j)
\end{array}$$

It states that *swap i j* exchanges the pebbles in buckets *i* and *j*, but leaves all other buckets unchanged. Recall that the (!) operator looks up the pebble stored at a particular index.

We continue by formalizing the invariant stated at the beginning of Section 3. We define the following property on two indices and an array of buckets:

$$\begin{array}{l}
\mathit{Invariant} : (r\ w : \mathit{Index}\ n) \to \mathit{Buckets}\ n \to \mathit{Set} \\
\mathit{Invariant}\ r\ w\ \mathit{bs} = (\forall\ i \to i < r \to (\mathit{bs}\,!\,i) \equiv \mathit{Red}) \wedge (\forall\ i \to w < i \to (\mathit{bs}\,!\,i) \equiv \mathit{White})
\end{array}$$

As you would expect, this property states that all buckets to the left of the index *r* contain red pebbles and all buckets to the right of the index *w* contain white pebbles.

The key statement we prove is the following:

$$\mathit{sortInv} : \mathit{Invariant}\ r\ w\ \mathit{bs} \to \exists m : \mathit{Index}\ n, \mathit{Invariant}\ m\ m\ (\mathit{exec}\ (\mathit{sort}\ r\ w\ d)\ \mathit{bs})$$

In words, it says that if the above invariant holds initially for some array of buckets *bs*, the invariant still holds after executing our *sort* function.

To prove this statement, we need to identify three separate cases.

Base case In the base case, *r* and *w* are equal. The *sort* function does not perform any further computation and we can trivially re-establish that the invariant holds.

No swap If the pebble in bucket *r* is red, the algorithm increments *r* and recurses. To re-establish the invariant, we need to prove that for every index *i* such that $i < r + 1$ the pebble in bucket number *i* is red. After defining a suitable view on the *Index* data type (McBride & McKinna, 2004), we can distinguish two cases:

- if $i \equiv r$, we have just established that the pebble in this bucket is red.;
- otherwise $i < r$ and we can apply our assumption.

Swap If the pebble in bucket *r* in white, the algorithm swaps two pebbles, decrements *w*, and recurses. This is the only tricky case. To re-establish our invariant we need to show that:

- the pebbles in the buckets numbered from *One* to *r* are all red after the swap. This follows from our assumption, together with the first conjunct of *swapProp*.
- the pebbles in buckets numbered from $w - 1$ onwards are all white. This case closely mimics the branch in which no swap occurred. The only difference is that we need to use the second conjunct of the *swapProp* to show that the pebble in bucket number *w* is now indeed white.

Finally, we use this lemma to establish our main result:

$$\mathit{correctness} : \exists m : \mathit{Index}\ n, \mathit{Invariant}\ m\ m\ (\mathit{exec}\ \mathit{polish}\ \mathit{bs})$$

To complete this proof, we use the fact that the *sort* function respects our *Invariant*. All that remains to be done is to show that the invariant is trivially true for the initial call to the *sort* function.

7 Discussion

Two colours or three? There is still some work to be done to verify the problem of the Dutch National Flag. The good news is that the *structure* of the algorithm is almost identical. Specifically, we can use the same termination argument: in every step of the algorithm the difference between two indices decreases. The only real change is that the number of cases grows from two to three. Consequently, the invariant that must be maintained and the corresponding proofs grow.

There is one detail I have swept under the rug. The final *correctness* result states that all pebbles to the left of some index m are red and all pebbles to the right of m are white. It does not say anything about m itself. The reason is that the *sort* function as defined here ends when its two arguments are equal. That is fine for the case with two colours; when there are three colours this may lead to incorrect results. Fixing this entails duplicating a bit more code: in the base case we should also check if there is a swap necessary.

Related work The Problem of the Dutch National Flag is also covered as one of the final examples in *Programming in Martin-Löf's Type Theory* (Nordstrom *et al.*, 1990). The program presented there is a bit different from Dijkstra's original solution: it does not use an in-place algorithm that swaps pebbles as necessary, but instead solves the problem using bucket sort. While this does produce correctly sorted results, I would like to think that the solution presented here is truer to Dijkstra's original.

Acknowledgements

I would like to thank Andres Löh, James McKinna and Ulf Norell for their valuable feedback on a draft version of this paper.

References

Bove, Ana, & Capretta, Venanzio. (2005). Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, **15**(04), 671–708.

Bove, Ana, & Dybjer, Peter. (2008). Dependent types at work. *Summer School on Language Engineering and Rigorous Software Development*.

Dijkstra, Edsger W. (1976). *A discipline of programming*. Prentice-Hall, Inc.

McBride, Conor. (2004). Epigram: Practical programming with dependent types. *Pages 130–170 of:* Vene, Varmo, & Uustalu, Tarmo (eds), *Advanced Functional Programming*. LNCS-Tutorial, vol. 3622. Springer-Verlag.

McBride, Conor, & McKinna, James. (2004). The view from the left. *Journal of Functional Programming*, **14**(1).

Nordstrom, Bengt, Petersson, Kent, & Smith, Jan M. (1990). *Programming in Martin-Löf's type theory: An introduction*. Oxford University Press.

Norell, Ulf. (2007). *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology.

Norell, Ulf. (2008). Dependently typed programming in Agda. *Pages 230–266 of:* Koopman, Pieter, Plasmeijer, Rinus, & Swierstra, Doaitse (eds), *Advanced Functional Programming*. LNCS-Tutorial, vol. 5832. Springer-Verlag.

Oury, Nicolas, & Swierstra, Wouter. (2008). The power of Pi. *ICFP '08: Proceedings of the Thirteenth ACM SIGPLAN International Conference on Functional Programming*.

Swierstra, Wouter. (2008). *A functional specification of effects*. Ph.D. thesis, University of Nottingham.