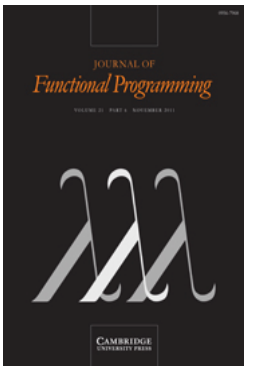

Sorted

WOUTER SWIERSTRA

Journal of Functional Programming / Volume 21 / Issue 06 / November 2011, pp 573 - 583
DOI: 10.1017/S0956796811000207, Published online: 12 October 2011

JFP **21** (6): 573–583, 2011.
doi:10.1017/S0956796811000207 First published online 12 October 2011

FUNCTIONAL PEARL

Sorted

Verifying the Problem of the Dutch National Flag in Agda

WOUTER SWIERSTRA

Heijendaalseweg 135, 6525 AJ Nijmegen, the Netherlands
(*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 minicomputer 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 minicomputer 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 satisfy the following two non-functional requirements:

- the minicomputer 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 solution to the problem that is guaranteed to terminate (Sections 3–5). Although this pearl does not cover the proof of functional correctness in detail, we will step through the key lemmas and definitions that are necessary (Section 6) and discuss how this solution may be extended to handle the case for three colours and also verify the non-functional requirements (Section 7).

This paper uses the dependently typed programming language Agda (Norell, 2007). Readers without any prior exposure to programming with dependent types,

may want to consult one of the many tutorials that are currently available (McBride, 2004; Norell, 2008; Oury & Swierstra, 2008; Bove & Dybjer, 2009).

2 A functional specification of the minicomputer

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

The primitive commands with which we can program the minicomputer 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. Several examples of such finite types should be familiar: *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, any unbound variables in type signatures are implicitly universally quantified, just as in Haskell (Peyton Jones, 2003), Epigram (McBride & McKinna, 2004) and Idris (Brady, 2011). For example, the variable n used in both constructors of the *Index* type is implicitly quantified at the start of both type declarations.

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)

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

The code presented in the remainder of this paper will use Haskell's notation for the unit (*return*) and bind (≫=) operations.

We can now define the *read* function, which 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:

```
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)

write : Index n → Pebble → State n Unit
write i p bs = (unit, update i p bs)
```

Calling *write i p* replaces the pebble stored in bucket number i with the pebble p. Although the interface of the minicomputer 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 minicomputer. This specification is in fact a degenerate case of the functional specification of mutable state (Swierstra, 2008). As the minicomputer cannot allocate new buckets, it is considerably simpler.

3 A first attempt

It is 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 undetermined 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:

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

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

- 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 initialise 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. 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*?

An obvious, but incorrect, choice of increment operation is the *Next* constructor. Recall that *Next* has type *Index n* → *Index* (*Succ n*), whereas we would like to have a function of type *Index n* → *Index n*. Similarly, 'peeling off' a *Next* constructor does not yield a decrement operation of the desired type.

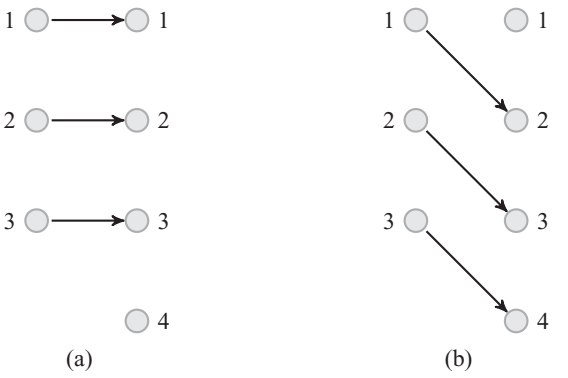

Fig. 2. The graph of the *inj* function (a) and the *Next* constructor (b) on *Index* 3.

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:

$$
\begin{aligned}
&\mathit{inj} : \mathit{Index}\ n \to \mathit{Index}\ (\mathit{Succ}\ n)\\
&\mathit{inj}\ \mathit{One} = \mathit{One}\\
&\mathit{inj}\ (\mathit{Next}\ i) = \mathit{Next}\ (\mathit{inj}\ i)
\end{aligned}
$$

Morally, the *inj* function is the identity, even if it maps $\mathit{One} : \mathit{Index}\ n$ to $\mathit{One} : \mathit{Index}\ (\mathit{Succ}\ n)$, 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) illustrates the graph of the *inj* function. The elements of *Index* 3 and *Index* 4 are enumerated in the left and in the right, respectively. The *inj* function maps the *One* element of *Index* 3 to the *One* element of *Index* 4; similarly, *Next i* is mapped to *Next* (*inj i*). As Figure 2(b) illustrates, the *Next* constructor behaves quite differently. It increments all the indices in *Index* 3, freeing space for a new index, $\mathit{One} : \mathit{Index}\ 4$.

From this picture, we can make the central observation: *Next i* is the successor of *inj i*, and correspondingly, *inj i* is the predecessor of *Next i*.

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* will terminate provided $r \leqslant w$. The usual choice of order on inhabitants of *Index n* is given by the following data type:

$$
\begin{aligned}
&\textbf{data}\ \_ \leqslant \_ : (i\ j : \mathit{Index}\ n) \to \mathit{Set}\ \textbf{where}\\
&\quad \mathit{Base} : \mathit{One} \leqslant i\\
&\quad \mathit{Step} : i \leqslant j \to \mathit{Next}\ i \leqslant \mathit{Next}\ j
\end{aligned}
$$

```
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.

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

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.

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, available online as supplementary material to this paper at journals.cambridge.org/jfp, defines *sort* using a fold over the *Pebble* type. This fold has been typeset 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:

```
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) >>= λ c →
  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.

```
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 as follows:

```
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)
```

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. More generally, this is an instance of the Bove–Capretta method (Bove & Capretta, 2005), which calculates such a type from any non-structurally recursive definition.

Of course, we can show this definition to be equivalent to the original *Difference* type using the *nextDiff* and *injDiff* lemmas. The name *SortT* for this data type should suggest that it encodes the conditions under which the *sort* function will terminate.

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
```

The easiest way to prove termination is by exploiting the equivalence between $i \leqslant j$ and *SortT i j*, witnessed by the function *toSortT*, whose definition is uninteresting enough to omit. 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, we need to 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, we will not do so. Instead this section outlines the key definitions and lemmas, and sketches their proofs.

Many of the proofs rely on the following two lemmas:

```
lookupUpdated : (p : Pebble) (i : Index n) (bs : Buckets n) → (update i p bs ! i) ≡ p
swapPreservation : (i : Index n) (x y : Index n) (bs : Buckets n) → i ≢ x → i ≢ y →
  exec (swap x y) bs ! i ≡ bs ! i
```

These two properties state that the operation *update i p bs* modifies bucket number *i*, overwriting the previous pebble to *p*, but leaves all other buckets unchanged. Recall that the (!) operator looks up the pebble stored at a particular index.

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

```
Invariant : (r w : Index n) → Buckets n → Set
Invariant r w bs = (∀ i → i < r → (bs ! i) ≡ Red) ∧ (∀ i → w < i → (bs ! i) ≡ White)
```

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:

```
sortInv : Invariant r w bs → ∃m : (Index n), Invariant m m (exec (sort r w d) 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. Furthermore, there is a single bucket *m* that separates the red pebbles from the white pebbles.

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 is 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 *swapPreservation* lemma.
- the pebbles in buckets numbered from $w - 1$ onwards are all white. This case closely mimics the branch in which no swap occurred. Using our induction hypothesis and the *swapPreservation* lemma, we know that all pebbles in buckets from w onwards are white. After executing the swap, we also know that the bucket numbered $w - 1$ has a white pebble, and hence, our invariant still holds.

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

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

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

Non-functional requirements. Although this proves that the solution is functionally correct, we have not verified the non-functional requirements. One way to do so is to make a deeper embedding of the language used to program the minicomputer. For instance, we could define the following data type that captures the instructions that may be issued to a minicomputer responsible for sorting n buckets:

$$\begin{aligned}&\textbf{data}\ Instr\ (n : Nat)\ (a : Set) : Set\ \textbf{where}\\ &\quad Return : a \to Instr\ n\ a\\ &\quad Swap : Index\ n \to Index\ n \to Instr\ n\ a \to Instr\ n\ a\\ &\quad Read : Index\ n \to (Pebble \to Instr\ n\ a) \to Instr\ n\ a\end{aligned}$$

It is easy to show that *Instr n* is a monad. Instead of writing programs in the *State* monad that we have done up till now, we could redefine *read* and *swap* to create instructions of this *Instr* type. This has one important advantage: it becomes possible to inspect the instructions for the minicomputer that our *polish* function generates. In particular, we can establish a bound on the maximum number of *Read* operations of any set of instructions:

```
reads : ∀ {a n} → Instrs n a → Nat
reads (Return x) = 0
reads (Swap i j p) = reads p
reads (Read c p) = Succ (max (reads (p Red)) (reads (p White)))
```

Given these definitions, we can prove the following statement of our *polish* function:

```
nonFunctional : (reads (polish n)) ⩽ n
```

The proof follows immediately from a more general lemma, stating that the *sort* function performs at most $w - r$ read operations. The proof of this statement is done by straightforward induction over the *SortT* data type.

Of course, this does not show that our program uses only a constant amount of memory. Perhaps a similar technique could make explicit that Agda values (and in particular the two indices r and w) need to be allocated by the minicomputer.

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. With three colours, one choice of type for the *sort* function is:

```
sort : (r w b : Index n) → r ⩽ w → SortT w b → State n Unit
```

With this choice, we divide the buckets into four distinct zones: those buckets known to store red pebbles, those buckets known to store white pebbles, those buckets storing a pebble of undetermined colour and those buckets storing blue pebbles. In each iteration, we ensure that the number of buckets storing pebbles of undetermend colour decreases by performing induction on *SortT w b*. We can define the invariant our *sort* function maintains:

```
Invariant : (r w b : Index n) → Buckets n → Set
Invariant r w b bs =
  (∀ i → i < r → (bs ! i) ≡ Red)
  ∧ (∀ i → r ⩽ i → i < w → (bs ! i) ≡ White)
  ∧ (∀ i → b < i → (bs ! i) ≡ Blue)
```

The proof that the *sort* function for three colours maintains this invariant is much longer than the proof for two colours. The only real change is that the number of cases grows from two to three – but in every branch, we also need to establish three conjuncts instead of two. As a result, the proof is considerably longer, even if it is not much more complex.

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, the solution presented here is perhaps truer to Dijkstra's original.

Acknowledgments

The author would like to thank Edwin Brady, Jeremy Gibbons, Andres Löh, James McKinna, Ulf Norell and the anonymous reviewers for their valuable feedback on earlier versions of this paper.

References

Bove, A. & Capretta, V. (2005) Modelling general recursion in type theory. *Math. Struct. Comput. Sci.* **15**(4), 671–708.

Bove, A. & Dybjer, P. (2009) Dependent types at work. In *Language Engineering and Rigorous Software Development*, Bove, A., Barbosa, L., Pardo, A. & Pinto, J. (eds), Lecture Notes in Computer Science, vol. 5520. Springer, pp. 57–99.

Brady, E. (2011) Idris-Systems programming meets full dependent types. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Programming Languages meets Programming Verification (PLPV '11)*.

Dijkstra, E. W. (1976) *A Discipline of Programming*. Prentice-Hall.

McBride, C. (2004) Epigram: Practical programming with dependent types. In *Advanced Functional Programming*. Vene, V. & Uustalu, T. (eds), LNCS-Tutorial, vol. 3622. Springer-Verlag, pp. 130–170.

McBride, C. & McKinna, J. (2004) The view from the left. *J. Funct. Program.* **14**(1), 69–111.

Nordstrom, B., Petersson, K. & Smith, J. M. (1990) *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press.

Norell, U. (2007) *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD. thesis. Chalmers University of Technology.

Norell, U. (2008) Dependently typed programming in Agda. In *Advanced Functional Programming*, Koopman, P., Plasmeijer, R., & Swierstra, D. (eds), LNCS-Tutorial, vol. 5832. Springer-Verlag, pp. 230–266.

Oury, N. & Swierstra, W. (2008) The power of Pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*, pp. 39–50.

Peyton Jones, S. (ed) (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

Swierstra, W. (2008) *A Functional Specification of Effects*. PhD. thesis. University of Nottingham.