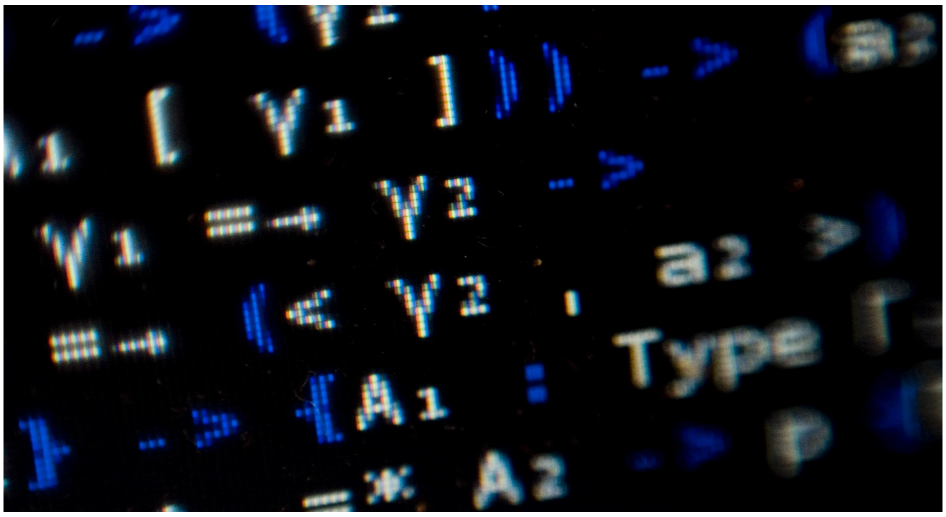
CHALMERS



Towards Reasoning about State Transformer Monads in Agda

Master of Science Thesis

in Computer Science: Algorithm, Language and Logic

Viet Ha Bui

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, July 2009

CHALMERS

Towards Reasoning about State Transformer Monads in Agda

Viet Ha Bui

Examiner: Peter Dybjer

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden July 2009

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, July 2009

Abstract

Wouter Swierstra showed in his PhD thesis how to implement stateful computations in the dependently typed functional programming language Agda. In particular he defined a notion of state which is parameterized by a list of types indicating what kind of data are to be stored in the respective locations. He also showed how to define monadic state transformation over this notion of state. In this thesis we extend Swierstra's work with two new contributions. The first is to implement a stateful version of Dijkstra's algorithm for the Dutch National Flag in Agda. We prove some properties of a function which swaps the contents of two locations, an important step towards showing full correctness of the algorithm in Agda. The second contribution is to formally prove (in Agda) some properties about monads suggested by Plotkin and Power.

Contents

1. Introduction **4**

2. Introduction to the dependently typed language Agda **5**

3. Monads and side effects **9**
3.1. Monad in Haskell 9
3.2 Side effects in Agda 11

4. The Dutch national flag problem and algorithm **16**
4.1 Dutch national flag algorithm in Haskell 16
4.2 Simplify model of references 19
4.3 The Dutch national flag algorithm in Agda. 21
4.4 Swaps properties 23
4.5. Introduction to the correctness of the algorithm 28

5. Proving some axioms of Plotkin and Power **31**

6. Conclusion **39**

Acknowledgements

I would like to thank Professor Peter Dybjer for his excellent support and guidance during my time in Sweden. I would also like to thank Dr Wouter Swierstra for his wise advice and help during my thesis work.

1. Introduction

Dependently typed programming languages, which are based on ideas from Martin Löf Type Theory and Lambda Calculus, have become well-known in the programming language community. Their most interesting feature is the program as a proof principle [6]. If we can express a programming problem as a type in Martin Löf Type Theory, the solution of the problem consists of giving a witness or proof of the specification.

This thesis will examine and illustrate these ideas in the dependently typed programming language Agda [10]. The motivation to choose this subject is both that the demand of knowledge of dependent types has increased and that proving properties with dependent types is hard and fascinating.

Programming with dependent types is hard because it requires many logical manipulations, much technical jargon, and different philosophical perspectives. It is also not a main stream language such as C++ or Java.

Agda 2 is a current dependently typed programming language developed at Chalmers [6]. Previous similar languages are Agda 1, ALF, or even Cayenne [3, 7, 8]. It is also similar to Haskell, a typed functional programming languages. However, Agda has an even richer type system than Haskell.

The author of this thesis expects that readers have general knowledge of type theory and functional programming languages. We will not be able to give a full description of Agda. The reader is refered to Bove and Dybjer [3] and Norell [8] for further understanding of Agda. The thesis will be organized in several parts. Part 2 reviews the necessary syntax and functionalities of Agda. Part 3 shows current developments of modeling imperative programming languages inside Agda. It will introduce monads and Wouter Swierstra's model of dependently typed memory. Part 4 implements the Dutch national flag algorithm using this model and proves some of its properties. Part 5 proves that the model validates several monadic Plotkin-Power's axioms [1].

2. Introduction to the dependently typed language Agda

As stated above, Agda is a dependently typed programming language with a rich type system. This part will give a short guide to its use and syntax.

With dependent types, more conditions about a function can be described in the type checking stage of the compiler. For example, a matrix $A_{m,n}$ can be understood as having a type which depends on the number of rows n and number of columns m. If we want to perform a multiplication `mul` of two matrices $A_{m,n}$ and $B_{u,v}$, then we need to make sure that n equals u. By using the dependently typed language Agda, we can define a type for the `mul` function as follows to guarantee that n equals u.

```
mul : forall {m, n, v} -> Matrix m n -> Matrix n v -> Matrix m v
```

A first construct to notice when we program in Agda is inductive data types [4]. To give an example, we look at the definition of the natural numbers:

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

This declaration can be interpreted as: the data type `Nat` (natural numbers) consists of constructors `Zero` and `Suc`. Thus 0 equals `Zero`, 1 equals `Succ Zero`, 2 equals `Succ (Succ Zero)`, etc. We can see that the base case here is `Zero`, and that the step case is `Succ`. Therefore, if we want to define a function or prove some properties that depends on `Nat`, we can use pattern matching to break it down into two cases.

In the above, we declared a data type in Agda by an inductive definition, Agda also supports dependent data types. For a simple example, we can declare a dependent data type as follows:

```
data Fin (n : Nat) : Set where
 Base : forall {n} -> Fin (Succ n)
 Step : forall {n} -> Fin n -> Fin (Succ n)
```

`Fin n` is a data type depending on a natural number `n` and it is a finite set with `n` elements.

We can also choose implicit and explicit declaration technique by using brackets and parenthesis respectively. For example, we have a explicit natural number `n` in data type `Fin (n : Nat)`. However, for the constructor `Base : forall {n} -> Fin (Succ n)`, we do not need to provide the natural number `n` as an explicit argument of `Base`. The compiler can infer it later [8].

Beside inductive dependent data types as above, we also have dependently typed functions. We can take a polymorphic `Identity` function as an example:

```
Identity : (A : Set) -> A -> A
Identity A x = x
```

If we use an implicit argument, we can define the `Identity` function as:

```
Identity : {A : Set} -> A -> A
Identity x = x
```

Both `Identity` functions above depend on the set `A`, a set which an element `x` belongs to.

We can also use lambda notation as in Haskell. The `Identity` function above, for instance, can be defined as:

```
Identity : {A : Set} -> A -> A
Identity = \x -> x
```

Agda also has another interesting feature, called case analysis by pattern matching. The pattern matching structure always requires that we check all the cases of the inductive definition of the data type. If we miss even one case, the compiler does not allow us to pass the type check.

We can define a `min` function, a function that finds a minimum value of two natural numbers as a simple example to illustrate the point here. If we want to compare two parameters in the `min` function, we must declare all 4 cases:

```
min : Nat -> Nat -> Nat
min Zero Zero = Zero
min (Succ x) Zero = Succ x
min Zero (Succ x) = Succ x
min (Succ x) (Succ y) = min x y
```

One of the best ways to structure the program in Agda is using a `with` construction. We use it to analyze the intermediate results from parameters. Its role is similar to the `case` structure in Haskell. The syntax of `with` looks as follows:

```
f p with d
f p | q1 = e1
:
f p | qn = en
```

`q1,.., qn` are all cases of `d`. In the general case, we can do case analysis with respect to several things:

```
f p with d1 | ...| dm
f p1 | q11 | ... | q1m = e1
:
f pn | qn1 | ... | qnm = en
```

For example, we can build a function `filter` that produces a satisfied condition list in a given list by using the `with` construction as follows:

```
data List (A : Set) : Set where
 Nil : List A
 Cons : A -> List A -> List A

filter : {A : Set} -> (A -> Bool) -> List A -> List A
filter p Nil = Nil
filter p (Cons x xs) with p x
filter p (Cons x xs) | True = Cons x (filter p xs)
filter p (Cons x xs) | False = filter p xs
```

Another important feature of Agda is a goal. It can only be used at the right hand side of the function. By using a goal, it means that you put a question mark `?` on the right hand side of the functions for parts of the term which are unknown yet. It deduces correct instances under the goals. In my experience, this feature is extremely important when you need to work with complicated programs in Agda.

We also have other features such as records and modules, wild cards, etc. If the reader wants to have a deeper knowledge of Agda, he/she can look at its documentations in the references [3, 8].

3. Monads and side effects

3.1 Monads in Haskell

The main advantage of functional programming languages compared to other ones is their purity. There are no side effects. Thus, programs are easier to understand and reason about.

However, purity also causes big problems in functional programming. By using pure functions only, we cannot always execute programs efficiently. This problem increases significantly if we build a large program where execution time is an important factor. On the another hand, impure code and side effects, such as direct interface functions to low level memory, deal very well with this problem. In order to implement impure code in functional programming languages, monads have been introduced to capture side effects. So we can have both pure and impure code in functional programming.

Monads originated in category theory. In 1991, Moggi used monads to structure program semantics [16].

In order to give a simple introduction to this idea, I will show the definition of a monad as a Haskell class.

```
class Monad m where
(>>=) :: m a -> (a -> m b) -> m b
(>>) :: m a -> m b -> m b
return :: a -> m a
fail :: String -> m a
```

The monad consists of a type construction `m`, a `return` function and a bind operation `>>=` . A type constructor `m` means that if `a` is a data type, then the corresponding monadic type is `m a`. A `return` function lifts a specific type to a corresponding monadic type. It has a polymorphic type :

```
return :: a -> m a
```

A bind operation has a polymorphic type:

```
(>>=) :: m a -> (a -> m b) -> m b
```

It means that if we have a value of monadic type `m a`, and we have a function that transforms an element of type `a` to a monadic type `m b`. Then the result is a value in the new monadic type `m b`.

Except the type construction, the `return` function, and the bind operation in the class, we also have a `>>` operation . The meaning of the `>>` operation is exactly the same as the bind operation, except that the function, the second parameter of the bind operation, ignores the value of the first monad operation. We also remark that the monad class should satisfy the following monad laws, called left identity, right identity, and associativity respectively:

```
return a >>= k = k a
m >>= return = m
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

To have a concrete understanding of this idea, we show an example of the `Maybe` monad in Haskell.

```
data Maybe t = Just t | Nothing
```

Then we can declare a `return` function as:

```
return x = Just x
_>>=_ : Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= _  = Nothing
Just x >>= f = f x
_>>_ : Maybe a ->  Maybe b -> Maybe b
Nothing >> _  = Nothing
Just x >>  f = f
```

Another useful example is the state transformer monad [19].

```
Type statetrans s v = s -> (v, s).
```

In here, `s` and `v` stand for state and value respectively. Then we can define the return function and bind operation in the state transformers monad as:

```
return x = \s -> (s, x)
m >>= f = \r -> let (x, s) = m r in (f x) s
```

Hence, we can define several useful state transformers functions:

```
readState = \s -> (s, s).
```

The `readState` function returns the current state of a computation as a value.

```
writeState x = \s  ->  ((), x).
```

The `writeState` function replaces a current state by the state `x`.

In Haskell, we can replace the bind operator `>>=` by `do` notation in order to have sugared syntax in a program. For example, the following code can be understood as: taking a result called `a` from monad `m`, passing `a` to a function `f`, then returning a result which is the sum of `a` and the return value of `f a`.

```
m >>= \ a ->
f a >>= \ b ->
return (a + b)
```

Instead of using the bind operation `>>=` , we can use the `do` notation as follows with the same meaning:

```
do
a <- m
b <- f a
return a + b
```

Another thing to note is that you cannot safely escape from the monad. We have a `return` function that lifts a type to a monadic type. But we do not have a safe function that changes the value from monadic value to the underlying value of the monad.

3.2 Side effects in Agda

Influenced by the way side effects are introduced by using monads in the functional programming Haskell, Wouter Swierstra introduced his model of side effects in Agda by using monads in his thesis in 2008 at Nottingham University [2].

First, he declares a reference monad module which is parameterized by a universe. A universe consists of a pair `U` and a function `el` from `U` to a set.

```
module Refs (U : Set) (el : U -> Set) where
```

We can think of `U` as a set of reference types we want to store while `el a` is the set of elements of reference type `a`. For example, if the universe with `U` contains type `Bool` then `el Bool` may have true and false values. Notice that we declared this module as a dependent module. Therefore, if we want to use this module, we need to define and instantiate a universe.

Another example which we will need for the Dutch national flag algorithm, we have references containing objects of type `Colour` which has three elements: red, white, blue. Then we can define a universe `U` with one constructor called `COLOUR`, and an `el` function which transfers `U` to `Colour`. So the universe should be:

```
data Colour : Set where
 red : Colour
 white : Colour
 blue : Colour

data U : Set where
 COLOUR : U

el : COLOUR -> Set
el _ = Colour
```

After we defined a universe in the module, we can define a notation of shape that a heap and a reference depends on. A shape is a list of types in the universe.

```
Shape : Set
Shape = List U
```

After we defined a shape, we can define what it means to be a heap of a certain shape. A heap is a list of elements that have exactlyte types that we declared in a shape.

```
data Heap : Shape -> Set where
 Empty : Heap Nil
 Alloc : forall {u s} -> el u -> Heap s -> Heap (Cons u s)
```

The `Empty` heap has a shape which is an empty list. If we want to add a new element of type `u` to the heap, we need to allocate a new location of the right type `u` in a shape for it. Thus, we can see the dependent relation between a heap and a shape in here. For example, a heap contains 3 colors red, white, blue can be defined as : `Alloc red (Alloc white (Alloc blue Empty))`. Then, the corresponding shape would be `Cons COLOUR (Cons COLOUR (Cons COLOUR Nil))`.

After constructing the heap, another question is how we can access the elements of the heap. We can use references which is modeled as a stack for this purpose. Thus, we can define a reference as below. The reference data type is essentially the same as `Fin n` if all elements of the shape have the same type.

```
data Ref : U -> Shape -> Set where
 Top : forall {u s} -> Ref u (Cons u s)
 Pop : forall {u v s} -> Ref u s -> Ref u (Cons v s)
```

We can now define syntactic operations on the heap called `IO` operations. An element of the `IO a s t` data type is an operation that changes a heap depending on a shape `s` to a heap depending on a shape `t` with a return value of type `a`.

We define four operations: `Return, Write, Read, and New`. A `Return` function lifts a value to the corresponding `IO` value. A `Read` function reads a value from a specific reference without changing the `IO`. A `Write` function replaces a value from a specific location. And finally, the `New` function adds a new reference to a shape and then links it to a new value.

```
data IO (a : Set) : Shape -> Shape -> Set where
  Return : forall {s} -> a -> IO a s s
  Write : forall {s t u} -> Ref u s -> el u -> IO a s t ->
  IO a s t
  Read : forall {s t u}  -> Ref u s -> (el u -> IO a s t) ->
 IO a s t
  New : forall {s t u} -> el u -> (Ref u (Cons u s) ->
 IO a (Cons u s) t) -> IO a s t
```

After defining the IO data type, we can define the monad operations, which consists of return function and bind operation, as follows:

```
return : forall {s a} -> a -> IO a s s
return = Return

_>>=_ : forall {s t u a b} -> IO a s t -> (a -> IO b t u) ->
IO b s u
Return x >>= f = f x
Write l d wr >>= f = Write l d (wr >>= f)
Read l rd >>= f = Read (\d -> rd d >>= f)
New d io >>= f = New d (\l -> io l >>= f)
infixr 10 _>>=_
```

The return function and bind operation have the same meaning in section 3.1. A return function lifts a value to an IO operation and leaves the shape of the heap unchanged. The bind operation composes two monadic computations.

For the next part, I will explain the direct computations to the heap. If readers want to know more about this module, such as automatic weakening, restriction in this model, they can look at chapter 6 of Wouter Swierstra's thesis [2].

```
_!_ : forall {s u} -> Heap -> Ref u s -> el u
_!_ Empty ()
Alloc x _ ! Top = x
Alloc _ h ! Pop k = h ! k

update : forall {s u} -> Heap s -> Ref u s -> el u -> Heap s
update Empty () d
update (Alloc _ heap) Top d = Alloc d heap
update (Alloc x heap) (Pop i) d = Alloc x (update heap i d)

run : forall {a s t} -> IO a s t -> Heap s -> Pair a (Heap t)
run (Return x) h = (x , h)
run (Write loc d io) h = run io (update h loc d)
run (Read loc io) h = run (io (h ! loc)) h
run (New d io) h = run (io Top) (Alloc d h)
```

The look up (`!`) function reads and returns a value in the heap at the specific reference. The `update` changes a value stored at the specific location. The `run` function executes an `IO` operation of type `IO a s t` in a given initial heap of shape `s` and returns a pair consisting of an element of type `a` and a heap of shape `t`.

4. The Dutch national flag problem and algorithm

The Dutch national flag problem is a famous programming problem proposed by Dijkstra [3]. The Dutch flag has three colors: red, white and blue. We have objects of these colors in a list randomly. The question is, how can we sort them so that objects of the same color are adjacent, with the colors in the order red, white, and blue respectively. We pick this algorithm because it is a good example to illustrate the use of invariant properties and side effects. The algorithm can be extended to find the best k values in an unsorted array [23]. However, to illustrate the side effects, the sorting algorithm for three colors is enough.

We can split an array `a` into 3 sections: `a [0.. lo-1]` is a space for red objects, `a[lo.. mid -1]` is a space for white objects , and `a[Hi .. N]` is a space for blue objects. Note that `a[mid .. Hi-1]` contain arbitrary colors. Then our solution algorithm can be described with pseudo code as follows:

```
Lo := 0; Mid := 0; Hi := N;
while Mid leq Hi do
  case a[Mid] of
    Red: swap a[Lo] and a[Mid]; Lo++; Mid++
    White: Mid++
    Blue: swap a[Mid] and a[Hi]; Hi--
```

We assume that at the initial state, we have no red, white and blue objects in their correct places, i.e. `Lo = Mid = 0` and `Hi = N`. Then we check the objects simultaneously, from `a[0]` to `a[N]` by the indexed variant called `Mid`. We stop our algorithm when `Mid` is greater than `Hi`. At each step, we either increase `Mid` by one or decrease `Hi` by one. Therefore, this algorithm has totally `N` checking steps.

4.1 The Dutch national flag algorithm in Haskell

I will implement the Dutch national flag algorithm in Haskell. We already have a monad of references implemented as a module in Haskell. For using this, we just need to import it.

```
import Data.IORef
```

For short, we can declare data color as:

```
data Colour = Red | White | Blue
```

Then we can declare two generic functions: one swaps the contents of two references, and another one swaps two indexed values in an array as follow:

```
swap :: IORef a -> IORef a -> IO ()
swap m n = do
 valueM <- readIORef m
 valueN  <- readIORef n
 writeIORef m valueN
 writeIORef n valueM

swapList :: [IORef a] -> Int -> Int -> IO [IORef a]
swapList io x y = do
 swap a b
 return io
  where
   a :: IORef a
   a = io !! x
   b :: IORef a
   b = io !! y
```

The `swap` function swaps the content of two given references under the assumption that both of them have the same type `a`. A `readIORef` reads the content at the specified reference, and the `writeIORef` writes the content to the location as discussed in section 3.2.

The `swapList` function swaps the values of two given locations where parameters are the array of object references and two specified positions. As we expected, we just use the `swap` function to define this function.

By using the `swapList` function, we can simulate the Dutch national flag algorithm with the condition that we should give `lo, mid`, and `hi` values in order to help this algorithm working recursively.

```
continueSort :: [IORef Colour] -> Int -> Int -> Int -> IO ()
continueSort as lo mid hi = do
   if (mid > hi)
    then return
   else do
      tmp <- readIORef (as !! mid)
      case tmp of
         Red -> do
            as <- swapList as lo mid
            x <- continueSort as (lo+1) (mid+1) hi
            return
          White -> do
            y <- continueSort as lo (mid+1) hi
            return
          Blue -> do
            as <- swapList as mid hi
            z <- continueSort as lo mid (hi -1)
            return
```

The `continueSort` function takes a list of color references and `lo, mid` and `hi` as parameters. It works recursively by changing three parameters `lo, mid` and `hi` and used the `swapList` function if needed. The function stops when the `mid` value is greater than the `hi` value.

Thus, the Dutch national flag problem can be solved by instantiating the `continueSort` function. `Lo` and `mid` values are `Zero` while the `hi` value is the number of element in the array. We also need to defined a function `listToIORef` that lifts colors from an array input to a list of `IORef` monad in a `continueSort` function.

```
dutchNationalFlag :: [Colour] -> IO ()
dutchNationalFlag xs = do
 as <- listToIORef xs
 continueSort as 0 0 ((length xs)-1)

listToIORef :: [Colour] -> IO [IORef Colour]
listToIORef [] = return []
listToIORef (x :xs) = do
```

```
 a <- newIORef x
 as <- listToIORef xs
 return (a : as)
```

4.2 Simplified model of references in Agda

I will rewrite the reference module for a short implementation of Dutch national flag algorithm in Agda. Since our heap only contains one type of values (colors), we do not need to parameterize our heap by a shape of different types.

```
module RefsToDutchFlag where
```

Instead of using the shape, we use a natural number as every color has the same amount of storage space. The heap should be a list of colors, and the type of references should be indexed by a natural number which is the number of elements in the heap.

```
data Colour : Set where
 red : Colour
 white : Colour
 blue : Colour

data Heap : Nat -> Set where
 Empty : Heap Zero
 Alloc : forall {m} -> Colour -> Heap s -> Heap (Succ m)

data Ref : Nat -> Set where
 Top : forall {m} -> Ref (Succ m)
 Pop : forall {m} -> Ref s -> Ref (Succ m)
```

Thus we define the data type `IO` again as follows:

```
data IO (a : Set) : Nat -> Nat -> Set where
 Return : forall {s} -> a -> IO a s s
 Write : forall {s t} -> Ref s -> Colour -> IO a s t ->
 IO a s t
 Read : forall {s t} -> Ref s -> (Colour -> IO a s t) ->
 IO a s t
 New : forall {s t} ->
(u : Colour) -> (Ref (Succ s) -> IO a (Succ s) t) ->
 IO a s t
```

The monad laws are still the same.

```
return : forall {s a} -> a -> IO a s s
return = Return

_>>=_ : forall {s t u a b} -> IO a s t-> (a -> IO b t u) ->
 IO b s u
Return x >>= f = f x
Write l d wr >>= f = Write l d (wr >>= f)
Read l rd >>= f = Read l (\d -> rd d >>= f)
New d io >>= f = New d (\l -> io l >>= f)
infixr 10 _>>=_
```

In order to manipulate the heap, such as reading a value at a specific location in the heap or updating a value in the heap at a specific location, we need to redefine a lookup and update functions again because it affects the heap.

```
_!_ : forall {s} -> Heap s -> Ref s -> Colour
_!_ Empty ()
Alloc x _ ! Top = x
Alloc _ h ! Pop k = h ! k

update : forall {s} -> Heap s -> Ref s -> Colour -> Heap s
update Empty () d
update (Alloc _ heap) Top d = Alloc d heap
update (Alloc x heap) (Pop i) d = Alloc x (update heap i d)
```

We can execute an `IO` monad with an initial heap by the `run` function.

```
run : forall {a s t} -> IO a s t -> Heap s -> Pair a (Heap t)
run (Return x) h = (x , h)
run (Write loc d io) h = run io (update h loc d)
run (Read loc io) h = run (io (h ! loc)) h
run (New d io) h = run (io Top) (Alloc d h)
```

In order to program with the monadic function, we declare `read` and `write` functions. They are special cases of `Read` and `Write` where the previous `IO` is `Return`

```
write : forall {s} -> Ref s -> Colour -> IO Unit s s
write ref d = Write ref d (Return unit)

read : forall {s} -> Ref s -> IO Colour s s
read ref = Read ref Return
```

4.3 The Dutch national flag algorithm in Agda.

Now we present the Dutch national flag algorithm in Agda by using the monadic `IO` and references as we presented in section 4.2. First, we can define a `swap` function in the same way as the `swap` function in Haskell. Note that the type of a reference depends on a size of the heap in order to keep the reference not out of bound. Further note that we do not have `do` notation in Agda. However, the `read` and `write` functions have nearly the same syntax as those one in Haskell.

```
swap : (s : Nat) -> Ref s -> Ref s -> IO Unit s s
swap s ref1 ref2 = read ref1 >>= \val1 ->
read ref2 >>= \val2 ->
write ref1 val2 >>= \_ ->
write ref2 val1
```

There are two kinds of reference traversals. One is increasing and one is decreasing a reference location. The decreasing function has a simple formula. It takes a reference in a shape, and `Pop` it if it is not the end of shape or the shape has only one element.

```
dec : (s : Nat) -> Ref s -> Ref s
dec Zero ()
dec (Succ Zero) Top = Top
dec (Succ (Succ x)) Top = Pop Top
dec (Succ s) (Pop r) = Pop (dec s r)
```

The increasing function is a little bit harder, as each time we remove a `Pop` from a reference, the shape of the reference is also changed. If we want to keep the shape constant, we need to use a `stepUp` function to change the shape to the original status again.

```
stepUp : (s : Nat) -> Ref s -> Ref (Succ s)
stepUp Zero ()
stepUp (Succ s) Top = Top
stepUp (Succ s) (Pop xs) = Pop (stepUp s xs)

inc : (s : Nat) -> Ref (Succ s) -> Ref (Succ s)
inc Zero x = x
inc (Succ s) Top = Top
inc (Succ s) (Pop xs) = stepUp (Succ s) xs
```

After defining these help functions, we can define a sorting algorithm that is similar to the `continueSort` function in the Dutch national flag algorithm. We need two conditions to make sure that the algorithm works correctly, namely, terminating when `mid > hi`, and sometimes performing the `swap` function in each case of reading the value of the heap at a reference `mid` .

There are several ways to solve the problem. I follow a simple one. We add an extra parameter to make sure that the function terminates. I call it a `counter` here. Each time by running the `continueSort` function one, the `counter` automatically reduces one. We can see that the number of checking steps of the Dutch national flag algorithm is invariant to `N`. Therefore, if we set the `counter` to be n at the beginning, we can ensure the `continueSort` function terminates and the sorting algorithm finishes when `counter` equals `Zero`.

```
continueSort : (s : Nat) (lo mid hi : Ref (Succ s)) -> Nat
-> Heap (Succ s) -> IO Unit (Succ s) (Succ s) ->
 IO Unit (Succ s) (Succ s)

continueSort s lo mid hi counter h io with (h ! mid)
continueSort s lo mid hi Zero h io | _ = Return unit
continueSort s lo mid hi (Succ counter) h io | red =
swap (Succ s) lo mid >>= \ _ ->
continueSort s (dec (Succ s) lo) (dec (Succ s) mid)
 hi counter h io
```

```
continueSort s lo mid hi (Succ counter) h io | white =
continueSort s lo ((dec (Succ s)) mid) hi counter h io
continueSort s lo mid r (Succ counter) h io | blue =
swap (Succ s) mid r >>= \ _ ->
continueSort s lo mid (inc s r) counter h io
```

Hence, we can write the Dutch national flag algorithm as follows. We store the result directly on the heap.

```
dutchNationalFlag : (s : Nat) -> Heap (Succ s) ->
IO Unit (Succ s) (Succ s)
dutchNationalFlag Zero _ = return unit
dutchNationalFlag (Succ xs) h =
continueSort (Succ xs) Top Top ( heapToRef (Succ xs) h)
(Succ xs) h (Return unit)
```

For the final Dutch national flag algorithm, we need to initialize the `continueSort` function. First, the `lo` and `mid` values, they should be both the `Top` reference. Then the `counter`, and the `hi` value are the number of elements in the heap. Both of them will be defined by using a function `RefToNat` that changes a reference to a corresponding natural number.

```
refToNat : {a : Nat} -> Ref a -> Nat
refToNat Top = Zero
refToNat (Pop i) = Succ (refToNat i)
```

4.4 Properties of swap function

As we have seen in the previous parts that we can construct general functions and algorithms in Agda. This part will be about proving properties of algorithms in Agda. First we use the definition of equality as the follows in Agda:

```
data _==_ {A : Set} (x : A) -> (y : A) -> Set where
 Refl : x == x
```

Two elements `x, y` of the same set `A` are equal (==) if we can use `Refl` to construct a canonical proof of `x == y`. For example, the witness, or program for `Zero == Zero, One == One`, should be `Refl` when `A` is `Nat`. We cannot construct a witness for `Zero == One` since it is not true.

As we can define an equality data type, then we can define a difference as a function from equality to an `Empty` set. This is the intuitionistic notation of negation.

```
_<>_ : {A : Set} -> A -> A -> Set
x <> y = (x == y) -> Empty
```

There are several properties of equality such as transitivity, symmetry, congruence or substitution. However, at this point, we just need to use congruence.

```
cong : {A B: Set} -> {x y : A} -> (f : A -> B) -> x == y ->
 f x == f y
```

This `cong` function means that if `x` and `y` are two elements of the set `A`, `f` is function from `A` to `B`, then `f x` should equal to `f y` if `x` equals to `y` in A. The witness is simple :

```
cong f Refl = Refl
```

For the convenience of the reader, we define several help functions. First, we can define an executing function `exec` which runs an `IO` operation with an initial heap. It take the second argument of the result, i.e the heap, after running the `run` function by the `IO` and the initial heap.

```
exec : forall {s t : Nat} -> IO Unit s t -> Heap s -> Heap t
exec io h = snd (run io h)
```

We can also define an `and` function with the symbol `/\` . It is another name of a `Pair` function in the standard Agda library.

```
_/\_ : Set -> Set -> Set
_/\_ a b = Pair a b
infixr 20 _/\_
```

Next, we need to prove the property called `trivialProp` which states that if two references are different, then their increased values are also different. More explicitly, if `Pop ref1` and `Pop ref2` are two different references, then `ref1` is different from `ref2`:

```
trivialProp : {s : Nat} -> {r1 r2 : Ref s} -> Pop r1 <> Pop r2
-> r1 <> r2
trivialProp f Refl = f Refl
```

Finally, we have the property called `alwayTrue` which expresses that, if `Top` is different with `Top` then everything is equal.

```
alwayTrue : {s : Nat} {a : Set} -> ( x y : a) -> Top <> Top
 -> x == y

alwayTrue x y f = magic (f Refl)
```

Let us come back to `swap`'s properties. We prove one property of the `update` function. It can be stated that the given color should equals one we read from a specific location. The location is a place where we just updated it with a given color. We see this statement translated into the dependent type as follows:

```
updateProp : (s : Nat) -> (h : Heap s) -> (r : Ref s) ->
 (c : Colour) -> c == ((update h r c) ! r)
```

For the proof of this property, we just pattern match on the heap and the location of the reference.

```
updateProp Zero h () c
updateProp (Succ s) (Alloc x h) r c with r
updateProp (Succ s) (Alloc x h) r c | Top = Refl
updateProp (Succ s) (Alloc x h) r c |  (Pop k) =
updateProp s h k c
```

We also need another property of the `update` function. It can be expressed as if we have two different references in a heap and we update the first reference with a specific value, then the value of the second reference is unchanged. We can translate this statement into Agda as follow:

```
updateProp' : (s : Nat) -> (h : Heap s) -> (r r1 : Ref s) ->
 r <> r1 ->(c : Colour) -> let h' = update h r1 c in
(h ! r) == (h' ! r)
```

We solved it by pattern matching on two references. When we pattern match on two references, there is a case where both references are `Top` as the requirement that all cases should be listed. For the solution at this case, we use the `alwayTrue` function.

```
updateProp’ Zero h () () f c
updateProp’ (Succ s) (Alloc x h) Top (Pop k) f c = Refl
updateProp’ (Succ s) (Alloc x h) Top Top f c = alwayTrue x c f
updateProp’ (Succ s) (Alloc x h) (Pop k) Top f c = Refl
updateProp’ (Succ s) (Alloc x h) (Pop k) (Pop k’) f c =
updateProp’ s h k k’ (trivialProp f) c
```

Finally, we have another property of the `update` function. It can be expressed as if we update a heap in two other different references than the specified one, then the value at the specified reference is not changed. It is an extension of the previous property. We can state in Agda as follow:

```
updateProp’’ : (s : Nat) -> ( h : Heap s) -> (r r1 r2 : Ref s) ->
 r <> r1 ->  r <> r2 -> (c1 c2 : Colour) -> let h’ =
 update h r1 c1 in let h’’ = update h’ r2 c2 in
(h ! r) == (h’’ ! r)
```

The only way we can prove it is to divide it into cases using pattern matching. However, The proof of this property is quite long due to we need to deal with all the cases of pattern matching parameters. In this property, we need 8 cases.

```
updateProp’’ Zero h () _ _  _ _  _ _
updateProp’’ (Succ s) (Alloc x h) Top Top f1 f2 c1 c2 =
alwayTrue x c2 f2
updateProp’’ (Succ s) (Alloc x h) Top Top (Pop k) f1 f2 c1 c2 =
 alwayTrue x c1 f1
updateProp’’ (Succ s) (Alloc x h) Top (Pop k) Top f1 f2 c1 c2 =
alwayTrue x c2 f2
updateProp’’ (Succ s) (Alloc x h) Top (Pop k) (Pop k’) f1 f2 c1
 c2 = Refl
updateProp’’ (Succ s) (Alloc x h) (Pop k) Top Top f1 f2 c1 c2
 = Refl
updateProp’’ (Succ s) (Alloc x h) (Pop k) Top (Pop k’) f1 f2 c1
```

```
 c2 = updateProp' s h k k' (trivialProp f2) c2
updateProp'' (Succ s) (Alloc x h) (Pop k) (Pop k') Top f1 f2 c1
c2 = updateProp' s h k k' (trivialProp f1) c1
updateProp'' (Succ s) (Alloc x h) (Pop k) (Pop k1) (Pop k2) f1
 f2 c1 c2 =
updateProp'' s h k k1 k2 (trivialProp f1) (trivialProp f2) c1 c2
```

From the above update properties, we can prove the first property of of the `swap` function: if we swap two references `ref1` and `ref2` with the property that both of them differ from reference `r`,then the value at reference `r` is invariant.

```
swapProp1 : (s : Nat) -> (ref1 ref2 : Ref s) -> (h : Heap s) ->
let h' = exec (swap s ref1 ref2) h in
(r : Ref s) -> r <> ref1 -> r <> ref2 -> (h ! r) == (h' ! r)
```

The solution for this lemma is quite simple, we just apply the third property of the `update` function.

```
swapProp1 Zero () () _ _ _ _
swapProp1 (Succ s) ref1 ref2 h r f1 f2 = let c' = h ! ref1 in
let c'' = h ! ref2 in
updateProp'' (Succ s) h r ref1 ref2 f1 f2 c'' c'
```

We can also prove a second property of the `swap` function. If we swap the content of two references `ref1` and `ref2`, then the contents at these two references should be exchanged.

```
swapProp2 : (s : Nat) -> (ref1 ref2 : Ref s) -> (h : Heap s) ->
let h' = exec (swap s ref1 ref2) h in
(h ! ref1) == (h' ! ref2) /\ (h ! ref2) == (h' ! ref1)
```

We prove this property by pattern matching on the heap and the explicit value of `ref1` and `ref2`. Therefore, we need 8 cases to match.

```
swapProp2 Zero () () _
swapProp2 (Succ Zero) ref1 ref2 (Alloc x Empty) with ref1 | ref2
swapProp2 (Succ Zero) ref1 ref2 (Alloc x Empty) | Top | Top
= Refl , Refl
swapProp2 (Succ Zero) ref1 ref2 (Alloc x Empty) | Top | Pop ()
```

```
swapProp2 (Succ Zero) ref1 ref2 (Alloc x Empty) | Pop () | Top
swapProp2 (Succ Zero) ref1 ref2 (Alloc x Empty) | Pop () | Pop ()

swapProp2 (Succ (Succ s)) ref1 ref2 (Alloc x h) with ref1 | ref2
swapProp2 (Succ (Succ s)) ref1 ref2 (Alloc x h) | Top | Top =
Refl , Refl
swapProp2 (Succ (Succ s)) ref1 ref2 (Alloc x h) | Top | Pop k =
updateProp (incS s) h k x , Refl
swapProp2 (Succ (Succ s)) ref1 ref2 (Alloc x h) | Pop k | Top =
 let h'' = update h k x in let c' = h ! k in
let h''' = update h'' Top c' in Refl , updateProp (incS s) h k x
swapProp2 (Succ (Succ s)) ref1 ref2 (Alloc x h) | Pop k | Pop k'
= swapProp2 (incS s) k k' h
```

As we can see, the properties of `swap` function are a combination of those two previous properties. More clearly, if we swap two references `ref1` and `ref2` such that both of them differ from a reference `r`, then the value at a reference `r` is invariant. On the another hand, if we swap the content of two references `ref1` and `ref2`, then the contents at two references should be exchanged.

```
swapProp : (s : Nat) -> (ref1 ref2 : Ref s) -> (h : Heap s) ->
let
h' = exec (swap s ref1 ref2) h in
(r : Ref s) -> r <> ref1 -> r <> ref2 -> (h ! r) == (h' ! r)
/\ (h ! ref1) == (h' ! ref2) /\ (h ! ref2) == (h' ! ref1)
```

The solution is simple, as we expected, we just combined the two properties of the `swap` function above.

```
swapProp s ref1 ref2 h r f1 f2 =
 swapProp1 s ref1 ref2 h r f1 f2 , swapProp2 s ref1 ref2 h
```

4.5 Correctness of the algorithm

In the previous section, we constructed proofs of some properties of the `swap` function. In this section, I will introduce the correctness property of the sorting algorithm. First of all, I will introduce some help functions.

```
replicateWhite : (n : Nat) -> Heap n
replicateWhite Zero = Refs.Empty
replicateWhite (Succ m) = Alloc white (replicateWhite m)

replicateBlue : (n : Nat) -> Heap n
replicateBlue Zero = Refs.Empty
replicateBlue (Succ m) = Alloc blue (replicateBlue m)

replicateRed : (n : Nat) -> Heap n
replicateRed Zero = Refs.Empty
replicateRed (Succ m) = Alloc red (replicateRed m)
```

These replicate functions make a heap with all elements in the heap to be white, red, and blue respectively.

In contrast to the replicate functions, we defined `numred, numblue` and `numwhite` functions to count the number of red, blue, and white elements in the heap.

```
numRed : (s : Nat) -> Heap s -> Nat
numRed Zero _ = Zero
numRed (Succ s) (Alloc red h) = Succ (numRed s h)
numRed (Succ s) (Alloc _ h) = numRed s h

numWhite : (s : Nat) -> Heap s -> Nat
numWhite Zero _ = Zero
numWhite (Succ s) (Alloc white h) = Succ (numWhite s h)
numWhite (Succ s) (Alloc _ h) = numWhite s h

numBlue : (s : Nat) -> Heap s -> Nat
numBlue Zero _ = Zero
numBlue (Succ s) (Alloc blue h) = Succ (numBlue s h)
numBlue (Succ s) (Alloc _ h) = numBlue s h
```

The heap is correctly sorted if it can be divided into red, white and blue parts respectively. Therefore, we can declare a new data type to make sure that a heap is correct sorted.

```
 infixr 90 _+++_
_+++_ : {s1 s2 : Shape} -> Heap s1 -> Heap s2 ->
Heap (s1 ++ s2)
Refs.Empty +++ h = h
(Alloc h hs) +++ h2 = Alloc h (hs +++ h2)

data Issorted : {s : Nat} -> Heap s -> Set where
Sorted : ( m n k : Nat) ->
 Issorted ((replicateRed m) +++ (replicateWhite n)
+++ (replicateBlue k))
```

Thus, we can state the correctness of the sorting algorithm as:

```
correctness : (s : Nat) -> (h : Heap (Succ s)) ->
Issorted (snd ( run ( dutchNationalFlag s h ) h ))
```

The witness for the above statement would be:

```
correctness s h =
Sorted (numRed (Succ s) h) (numWhite (Succ s) h)
 (numBlue (Succ s) h)
```

However, I was not able to prove this property.I could not prove that the length n of the heap h is not equals to the sum of the length of the number of red, white, blue elements in the heap h.

I however believe that the algorithm is correct because i have run it on a number of test cases and it has always performed correctly.

5. Proving some axioms of Plotkin and Power

In the previous chapter, we see the advantages of Agda in proving the `swap` properties. We prove it by using the idea of Martin Löf type theory. We formulate the properties in the dependent typed constraints. Then, the body of the type is a proof for the properties. In this part, I will use a proof system in Agda to prove some axioms stated in the article about computation determine monads by Gordon Plokin and John Power [1] for Swierstra's notion of dependently type state transformer monad.

These axioms have quite strong results in the monadic theory. For an arbitrary monadic system, if we prove that it satisfies these axioms, then any theorem you can prove about state monadic properties of the system will follow from these axioms. It is a consequence from several theorems in the paper. Therefore, instead of designing properties to prove as we have in `swap` function, we just need to prove that the system satisfy these given axioms.

For this part, I will prove that the reference `IO` monad above satisfies the first 7 axioms of look up and update functions in state transformer monads.

First of all, we need to extend the equality module in Agda. As we have seen in section 4.4, an equality of two elements in the same set is a data type with one constructor named `Refl`. Here, we extend an equality (heterogenous equality) to two elements in different sets. The reason to modify the equality data type is that we need to compare reference locations. The reference data type is a dependent function that depends on the type of the containing element and the shape. Therefore, even they are just represent by `Top` and `Pop` constructors, they may belong to different sets. Thus, the data type equality defines as follows:

```
data _==_ {A : Set} (x : A) : {B : Set} -> (y : B) -> Set where
Refl : x == x
```

As we have a different equality, we can redefined distinctness as a function from an equality to an `Empty` set.

```
_<>_ : {A B : Set} -> A -> B -> Set
x <> y = (x == y) -> Empty
```

5.1 Proof for Plotkin and Power axioms in the model

There are totally 7 axioms about look up and update functions in this paper. These axioms are listed below, the proof will be discussed latter on.

```
1) l(loc, u(loc, v, a)) = a
2) l(loc, (l(loc, a v v) = l(loc, a v v)
3) u(loc, v, u(loc, v, a) = u(loc, v, a)
4) u(loc, v, l(loc, a v)) = u (loc, v, a v)
5) l(loc, (l(loc', a v v')) = l(loc', l(loc, a v v'))
 where loc is different with loc'
6) u(loc, v, u(loc', v', a) = u(loc', v', u(loc, v, a)
where loc is different with loc'
7) u(loc, v, l(loc', a v')) = l(loc', (u(loc, v, a v'))
where loc is different with loc'
```

`u` means an update function and `l` means a look up function.

As we prove in the general case of references, we need to embed a universe into the proof. We can do this by putting it into the declaration of the module.

```
module Axioms (U : Set) (el : U -> Set) where
import Refs
open Refs U el public
```

Similar to the proof of the properties of the `swap` function, we also need several support functions such as `trivialProp` and `alwayTrue`.

```
trivialProp :{u1 u2 u3 u4 u : U} {s : Shape} {r1 : Ref u1 s}
 {r2 : Ref u2 s} -> Pop r1 <> Pop r2 -> r1 <> r2
trivialProp f Refl = f Refl
```

The `trivialProp` states that if `r1` is a reference for a type `u1`, `r2` is a reference for a type `u2` in a shape `s` and `Pop r1` is different from `Pop r2` then `r1` is distinct from `r2`.

The `alwayTrue` and `updateProp` functions have the same meaning as we had in the section 4.4. Except that a dependent shape is in a general case rather than in a specific case `Nat`.

```
alwayTrue: {A : Set} (a b : A) -> (Top <> Top) -> a == b
alwayTrue a b f = magic (f Refl)

updateProp : {u : U} (s : Shape) -> (h : Heap s) ->
 (r : Ref u s) -> (c : el u) -> c == ((update h r c) ! r)
updateProp Nil h () c
updateProp (Cons u s) (Alloc x h) r c with r
updateProp (Cons u s) (Alloc x h) r c | Top = Refl
updateProp (Cons u s) (Alloc x h) r c | (Pop k) =
 updateProp s h k c
```

The next support functions are called `equal` and `equal'`. the `equal` function states that if two heaps `h1` and `h2` are equal, then we still have the equal result if we run it in an arbitrary `IO`. The `equal'` function has a similar meaning as the `equal` function except that we change the role of an `IO` and a heap. We define the `equal'` function in order to prove axiom 4.

We also need to define the `run'` function in order to support the `equal'` function. The `run'` functionality is exactly the same as `run` function, except that we permute the order of two parameters in the `run` function.

```
equal : forall {a s t} -> {h1 h2 : Heap s} -> (io : IO a s t)
-> h1 == h2 -> run io h1 == run io h2

equal io v = cong (run io ) v

run' : forall {a s t} -> (h : Heap s) -> IO a s t ->
Pair a (Heap t)

run' h io = run io h

equal' : forall {a s t} -> {h : Heap s} -> (io1 io2 : IO a s t)
 -> io1 == io2 -> run io1 h == run io2 h

equal' {a} {s} {t} {h} io1 io2 v = cong (run' h) v
```

We can also extend an equality of two elements in sets to the equality of two `IO` monads. Two `IO` monad `io1` and `io2` are called equal (represent by symbol ===) if and only if we have `run io1 h == run io2 h` for every heap `h`.

```
data _===_ {a : Set} {s t : Shape} : IO a s t -> IO a s t -> Set
 where
IOEqual : forall {a s t h io1 io2} -> run io1 h == run io2 h ->
 io1 === io2
```

Using this definition, the axiom1: l(loc, u(loc, v, a)) = a can be stated as:

```
lemma1 : forall {u : U} (s t : Shape) (h : Heap s) ->
 (ref : Ref u s) -> (io : IO (el u) s t) ->
Read ref (\v -> Write ref v io) === io

lemma1 s t h ref io = IOEqual (equal io (lemma11 s h ref))
```

It means that when we look up the value in the location `loc` after we update this location by the value `a`, then the result should be `a`. The proof for this lemma is simple. We prove an equality in data type `===` by constructing a constructor `IOEqual` from them. Then the problem reduces to constructing the equality of `run io1 h` and `run io2 h`, or `run io1 h == run io2 h` where `io1, io2` monads are `Read ref ( v -> Write ref v io)` and `io`. This problem can be constructed by using the `equal` function, the function translates from `h1 == h2` to `run io h1 == run io h2`, where `h1, h2` are two equal heaps. However, `h1 == h2` at this `lemma1` can be proved by using the supporting function `lemma11`.

```
lemma11 :{u : U} (s : Shape) -> (h : Heap s) ->
 (ref : Ref u s) -> update h ref (h ! ref) == h
```

The `lemma11` means that if we update the heap `h` at location `ref` with the value that we have read from `ref`, then the heap is unchanged. We prove this lemma by pattern matching on `ref`.

```
lemma11 Nil h ()
lemma11 (Cons u s) (Alloc x h) ref with ref
lemma11 (Cons u s) (Alloc x h) ref | Top = Refl
lemma11 (Cons u s) (Alloc x h) ref | (Pop k) =
cong (Alloc x) (lemma11 s h k)
```

Next, we can restate the axiom 2 l(loc, (l(loc, a v v) = l(loc, a v v) as:

```
lemma2 : forall {u : U} (s t : Shape) (h : Heap s) ->
 (ref : Ref u s)  -> (io : el u -> el u -> IO (el u) s t) ->
Read ref (\v' -> Read ref (\ v -> io v v' )) ===
 Read ref (\v -> io v v)

lemma2 s t h ref io =
 IOEqual (equal (io (h ! ref) (h ! ref)) Refl)
```

Axiom 2 states that if we look up at the location `loc` twice then it is the same as we look up it once. So, we translate it to Agda as follows: for the reference `ref` of type `u` in a shape `s, io` is a function from pair of elements of `u` to the `IO` monad `IO (el u) s t`. If the `IO` monad is a double reading from the location `ref`, then it should equal to the one if we read it once. The proof for this lemma is similar to `lemma1`

With the similar expression as axiom 2, the axiom 3 u(loc, v, u(loc, v, a) = u(loc, v, a) can be stated as :

```
lemma3 : forall {u : U} (s t : Shape) (h : Heap s) ->
 (ref : Ref u s) -> (v1 v2 : el u) -> (io : IO (el u) s t) ->
Write ref v1 (Write ref v2 io) === Write ref v2 io

lemma3 s t h ref v1 v2 io =
IOEqual (equal io (lemma31 s h ref v1 v2))
```

The type condition means that for every location `ref`, and two value `v1, v2` that can be stored in `ref`. If we write to the `ref` value `v1` and then `v2`, then it is the same as we write to the `ref` location value `v2` only.

We prove it by using a supporting `lemma31` and using the `equal` function as for two previous axioms. `lemma31` can be stated as: if we update the heap in a location `ref` twice by values `v1` and `v2`, then the heap is the same as we update the heap at `ref` by the last value.

```
lemma31 : {u : U} (s : Shape) -> (h : Heap s) -> (ref : Ref u s)
 -> (v1 v2 : el u) ->
update (update h ref v1) ref v2 == update h ref v2
```

As it is similar to `update` property, we prove it by pattern matching on `ref`.

```
lemma31 Nil h () _  _
lemma31 (Cons u s) (Alloc x h) ref v1 v2 with ref
lemma31 (Cons u s) (Alloc x h) ref v1 v2 | Top = Refl
lemma31 (Cons u s) (Alloc x h) ref v1 v2 | (Pop k) =
cong (Alloc x) (lemma31 s h k v1 v2)
```

In a similar manner, we can express axiom 4, i.e u(loc, v, l(loc, a v)) = u (loc, v, a v) in Agda as:

```
lemma4 : forall {u : U} (s t : Shape) (h : Heap s) ->
 (ref : Ref u s) -> (v : el u) -> (io : el u ->
 IO (el u) s t) ->
Write ref v (Read ref (\ v -> io v)) ===
 Write ref v (io v)
```

The axiom means that if we update the reference `ref` with the value `v` then we read the result from this location. it equals the action that we just update the location `ref` with the value `v`. The solution for this axiom is quite interesting. The solution is similar to the `lemma1`. However, instead of using a heap equality and the `equal` function, we use an `IO` monad equality and the `equal'` function.

With similar proof as for the above axioms, we reduce the === relation to the `equal'` function. We use the `equal'` function instead of the `equal` function because of the direct interface of the `IO` function and `Write, Read` monadic functions to the monad. We cannot have the same monad in order to use the `equal` function here. An `equal'` function takes 3 argument, two monads and one equality between them. Here, two `IO` monads are `(io (update h ref v !  ref))` and `(io v)`. The element of the equality data type between two monads has been constructed by using the congruence function from an `IO` function and an equality in `updateProp` function followed by a symmetry function to permute the order of two monads. We use the `sym` function to avoid redeclaring a function, which has exactly the same meaning as an `updateProp` function except the order of values in the equality module.

```
lemma4 s t h ref v io =
IOEqual (equal' (io (update h ref v ! ref)) (io v)
(sym (cong io (updateProp s h ref v))))
```

Axiom 5, l(loc, (l(loc', a v v')) = l(loc', l(loc, a v v')) means that if two references `loc` and `loc'` are different, then the `IO` monad obtained by reading from the `loc` and `loc'` respectively is equal to the `IO` monad by reading in the opposite order, i.e reading from `loc'` then from `loc`. We also can prove it in a similar way to axiom1.

```
lemma5 : forall {u : U} (s t : Shape) (h : Heap s) ->
(ref1 ref2 : Ref u s) -> ref1 <> ref2 ->
(io : el u -> el u -> IO (el u) s t) ->
Read ref1 (\ v -> Read ref2 (\ v' -> io v v')) ===
 Read ref2 (\ v' -> Read ref1 (\ v -> io v v'))
lemma5 s t h ref1 ref2 f io =
 IOEqual (equal (io (h ! ref1) (h ! ref2)) Refl)
```

For the axiom 6, we need another property of `update`, I call it `lemma6'`.

```
lemma6' : forall {u1 u2 : U} (s t : Shape) (h : Heap s) ->
 (ref1 : Ref u1 s) -> (ref2 : Ref u2 s) -> ref1 <> ref2 ->
 (v : el u1)
-> (v' : el u2 ) -> update (update h ref1 v) ref2 v' ==
update (update h ref2 v') ref1 v
```

The above type means that the result of updating the heap with two separate references are the same, regardless of the order of the updating. The proof uses pattern matching on two reference.

```
lemma6' Nil t _ () () f v v'
lemma6' (Cons u s) t (Alloc x h) ref1 ref2 f v v'
with ref1 | ref2
lemma6' (Cons u s) t (Alloc x h) ref1 ref2 f v v'
 | Top | Top =
alwayTrue (update (update (Alloc x h) Top v) Top v')
 (update (update (Alloc x h) Top v')Top v) f
lemma6' (Cons u s) t (Alloc x h) ref1 ref2 f v v'
| Top | Pop k
= Refl
lemma6' (Cons u s) t (Alloc x h) ref1 ref2 f v v'
| Pop k | Top
 = Refl
```

```
lemma6' (Cons u s) t (Alloc x h) ref1 ref2 f v v'
 | Pop k | Pop k'
 = cong (Alloc x) (lemma6' s t h k k' (trivialProp f ) v v')
```

Thus, the axiom 6 can be stated as:

```
lemma6 : forall {u1 u2 u : U} (s t : Shape) (h : Heap s) ->
 (ref1 : Ref u1 s) -> ( ref2 : Ref u2 s) -> ref1 <> ref2 ->
(io : IO (el u) s t ) -> (v : el u1) -> (v' : el u2 ) ->
Write ref1 v (Write ref2 v' io) ===
Write ref2 v' (Write ref1 v io)
```

The proof of this lemma is similar to the `lemma1`, except that we use the support function `lemma6'`.

```
lemma6 s t h ref1 ref2 f io v v' =
IOEqual (equal io (lemma6' s t h ref1 ref2 f v v'))
```

Finally, the axiom7 u(loc, v, l(loc', a v')) = l(loc', (u(loc, v, a v')), means that if two location `loc` and `loc'` are different then the `update` and look up functions at two location giving the same `IO` monad, regardless of order. In a similar way to axiom 1, we can state and proved the axiom 7 as follows:

```
lemma7 : forall {u : U} (s t : Shape) (h : Heap s) ->
(ref1 ref2 : Ref u s) -> (io : el u -> IO (el u) s t) ->
 (v : el u) ->
Write ref1 v (Read ref2 \v' -> io v') ===
Read ref2 (\v' -> Write ref1 v (io v'))

lemma7 s t h ref1 ref2 io v =
IOEqual (equal (io (h ! ref2)) Refl)
```

6. Conclusion

In conclusion, there are several contributions in this thesis. By extending the work in Swierstra's thesis [2] the author gave the Dutch national flag algorithm as an example of Swierstra's model of functional specification of side effects in Agda. The author also proved some properties of the `swap` function. Furthermore, the author validates some of Plotkin and Power's [1] axioms for the model.

As dependently typed languages are still under development, it is quite hard for beginners to use them. The supporting environment, for example, is quite limited at the moment. Therefore, setting up and running test cases in Agda takes more time than in usual programming languages.

However, we can see the advantage of programs as proofs as a philosophical idea in dependently typed languages. In functional programming languages such as Haskell, we can model some side effects by using monad. However, we cannot prove the properties of the system, or guarantee that the system works correctly. In contrast, we can do this using dependent types.

References

[1] Gordon Plotkin and John Power (2003). Notions of computation determined monads. *FOSSACS 2002. Lecture Notes in Computer Science.*

[2] Wouter Swierstra (2008). *A functional specification of effects.* PhD thesis. Nottingham university.

[3] Ana Bove, Peter Dybjer. *Dependent type at work.* Summer School on Language Engineering and Rigorous Software Development (LerNet). Piriapolis. Uruguay. February - March 2008.

[4] Peter Dybjer (1994). Inductive Families. *Formal Aspect of Computing.* 6(4). 440-465.

[5] Thorsten Altenkirch, Conor McBride, Wouter Swierstra (2007). Observational equality, now!. *Proceedings of the 2007 workshop on Programming Languages Meets Program Verification.* 57 - 68.

[6] Bengt Nordström,Kent Petersson, Jan M. Smith (1990). *Programming in Martin Lof type theory.* Oxford University Press.

[7] Ana Bove (2002). *General Recursion in Type Theory.* PhD thesis. Chalmers university.

[8] Ulf Norell (2007). *Towards a practical programming language based on dependently type theory.* PhD thesis. Chalmers University of Technology.

[9] Epigram home page. http://www.e-pig.org/

[10] Agda home page. http://wiki.portal.chalmers.se/agda/

[11] Simon Thompson (1999). *Haskell The Craft of Functional Programming.* Addison-Wesley. ISBN 0-201-34275-8.

[12] Benjamin C. Pierce (2002). *Types and Programming Languages.* The MIT Press. ISBN 0-262-16209-1.

[13] Qiao Haiyan (2003). *Testing and Proving in Dependent Type Theory.* Phd thesis. Chalmers university.

[14] Peter Morris (2007). *Constructing universes for generic programming.* PhD thesis. Nottingham university.

[15] John Robert Harrison (1996). *Theorem Proving with the Real Numbers.* PhD thesis. Cambridge university. 1996

[16] Eugenio Moggi (1991). Notions of Computation and Monads. *Information and Computation.* 93(1): 5592.

[17] Roland Backhouse (2001). *The Dutch national flag problem.* From http://www.cs.nott.ac.uk/ rcb/G51MPC/slides/DutchNationalFlag.pdf

[18] Colin L. McMaster (1978). An analysis of algorithms for the Dutch National Flag Problem. *Communications of the ACM.* 842 - 846. Volume 21. Issue 10.

[19] Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. *First International Spring School on Advanced Functional Programming Techniques.* Sweden. Springer-Verlag Lecture Notes in Computer Science 925. May 1995