Monad P3 : Mutable Data Structures (1D)

Young Won Lim
12/26/19

Please send corrections (or suggestions) to youngwlim@hotmail.com.

Based on

Haskell in 5 steps

https://wiki.haskell.org/Haskell_in_5_steps

Mutable data structures

Arrays, hash tables and any other mutable data structures

are defined in the same way - for each of them,

there's an operation that creates new "mutable values"

and returns a reference to it.

Then special read and write operations in the IO monad are used.

https://wiki.haskell.org/IO_inside#IO_actions_as_values

Mutable arrays

```
mport Data.Array.IO
main = do arr <- newArray (1,10) 37 :: IO (IOArray Int Int)
          a <- readArray arr 1
          writeArray arr 1 64
          b <- readArray arr 1
          print (a, b)
```

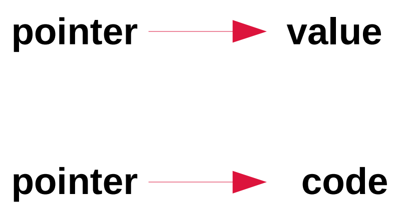
Here, an array of 10 elements with 37 as the initial value
at each location is created.
After reading the value of the first element (index 1)
into 'a' this element's value is changed to 64
and then read again into 'b'
As you can see by executing this code,
'a' will be set to 37 and 'b' to 64.

https://wiki.haskell.org/IO_inside#IO_actions_as_values

Box

In most implementations of **lazy evaluation**,
values are represented at runtime as **pointers**
to either their **value**, or **code** for computing their value.

This extra level of indirection,
together with any extra tags needed by the runtime,
is known as a **box**.



https://wiki.haskell.org/Arrays

Boxed Arrays

The default **boxed arrays** consist of many of these boxes,
each of which may compute its value separately.

This allows for many neat tricks,
like recursively defining an array's elements
in terms of one another, or
only computing the specific elements of the array
which are ever needed.

However, for large arrays, it costs a lot in terms of overhead,
and if the entire array is always needed, it can be a waste.

https://wiki.haskell.org/Arrays

UnBoxed Arrays

Unboxed arrays are more like arrays in C -

they contain just the plain **values**

without this extra level of indirection,

for example, an array of 1024 values of type Int32

will use only 4 kb of memory.

Moreover, indexing of such arrays can be significantly faster.

https://wiki.haskell.org/Arrays

UnBoxed Arrays – only for simple types

First, **unboxed arrays** can be made
only of plain **values** having a fixed size
- **Int**, **Word**, **Char**, **Bool**, **Ptr**, **Double**, etc.

custom unboxed arrays
for other simple types, including enumerations.

But **Integer**, **String** and
any other types defined with variable size
cannot be elements of **unboxed arrays**.

https://wiki.haskell.org/Arrays

UnBoxed Arrays – all elements are evaluated

Second, without that extra level of indirection,

all of the elements in an **unboxed array**

must be evaluated when the **array** is evaluated,

so you lose the benefits of **lazy evaluation**.

Indexing the array to read just one element

will construct the entire array. (all elements evaluated)

This is not much of a loss if you will eventually need the whole array,

may be too expensive if you only ever need specific values.

https://wiki.haskell.org/Arrays

UnBoxed Arrays – no recursive definition

Accessing only one element

will construct the entire array

this fact prevents unboxed arrays

from recursively defining the array elements

in terms of each other

Nevertheless, unboxed arrays are

a very useful optimization instrument,

and are recommended to be used as much as possible.

https://wiki.haskell.org/Arrays

Array Library Types

array library supports two array varieties -

lazy boxed arrays

strict unboxed arrays

A parallel array implements something intermediate:

strict boxed immutable arrays

This keeps the flexibility of
using any data type as an array element
while making both creation of and access to
such arrays much faster.

https://wiki.haskell.org/Arrays

Parallel Arrays – creation and access

Parallel array creation is implemented as

one **imperative loop**

that fills all the array elements,

while **accesses** to array elements

don't need to check the **box**

https://wiki.haskell.org/Arrays

Parallel Arrays – drawbacks

It should be obvious that parallel arrays
are not efficient in cases where
the calculation of array elements is relatively complex
and most elements will not be used.

parallel arrays don't support the **IArray** interface,
which means that you can't write generic algorithms
which work both with **Array** and the **parallel array constructor**.

https://wiki.haskell.org/Arrays

Array Types

	Immutable	Mutable	
	Immutable Instance **Iarray a e**	IO Monad Instance **MArray a e IO**	ST Monad Instance **MArray a e ST**
Standard	**Array** **DiffArray**	**IOArray**	**STArray**
Unboxed	**UArray** **DiffUArray**	**IOUArray** **StorableArray**	**STUArray**

lazy boxed arrays

strict unboxed arrays

https://wiki.haskell.org/Arrays

Array constructor

Haskell'98 supports just one **array constructor type**, namely **Array**, which gives you **immutable boxed arrays**.

https://wiki.haskell.org/Arrays

Immutable Array

Immutable means that these arrays,
like any other pure functional data structure,
have contents fixed at construction time.
You can't modify them, only query.

There are **modification** operations,
but they just return new arrays and
don't modify the original one.

This makes it possible to use Arrays
in **pure functional** code along with **lists**.

https://wiki.haskell.org/Arrays

Boxed Array

Boxed means that array elements
are just ordinary Haskell (**lazy**) values,
which are evaluated on demand,
and can even contain **bottom** (**undefined**) **values**.

https://wiki.haskell.org/Arrays

Immutable Array

the typeclass **Iarray** (**immutable array**)

Data.Array.Iarray

defines the same operations that were defined

for **Array** in Haskell'98

Data.Array

The big difference is that

it is now a typeclass and

there are 4 array type constructors,

each implements these interface:

Array, **UArray**, **DiffArray**, and **DiffUArray**.

https://wiki.haskell.org/Arrays

Mutable Array

the type class **MArray** (**mutable array**)

Data.Array.MArray

contains operations to update array elements in-place.

Mutable arrays are very similar to **IORefs**,

only they contain multiple values.

Type constructors for **mutable arrays** are

IOArray and **IOUArray**

operations which create, update and query these arrays

all belong to the **IO monad**:

https://wiki.haskell.org/Arrays

Mutable Array

In the same way that **IORef** has its more general cousin **STRef**,
IOArray has a more general version **STArray**
(and similarly, **IOUArray** corresponds to **STUArray**).

These array types allow one to work with
mutable arrays in the ST monad:

https://wiki.haskell.org/Arrays

Immutable non-strict arrays

Haskell provides indexable arrays,
which may be thought of as functions
whose domains are isomorphic to contiguous subsets of the integers.

Functions restricted in this way can be implemented efficiently;
in particular, a programmer may reasonably expect
rapid access to the components.

To ensure the possibility of such an implementation,
arrays are treated as data, not as general functions.

https://www.haskell.org/hugs/pages/libraries/base/Data-Array.html

Immutable non-strict arrays

Since most array functions involve the class Ix,

this module is exported from Data.Array

so that modules need not import both Data.Array and Data.Ix.

https://www.haskell.org/hugs/pages/libraries/base/Data-Array.html

Ix class

The Ix class is used to map a contiguous subrange of values
in a type onto integers.
It is used primarily for array indexing
(see Data.Array, Data.Array.IArray and Data.Array.MArray).

The first argument (l,u) of each of these operations is
a pair specifying the lower and upper bounds
of a contiguous subrange of values.

https://www.haskell.org/hugs/pages/libraries/base/Data-Ix.html

UArray (1)

Arrays with **unboxed** elements. Instances of **IArray** are provided for **UArray** with certain element types (**Int**, **Float**, **Char**, etc.; see the UArray class for a full list).

A **UArray** will generally be more efficient (in terms of both time and space) than the equivalent Array with the same element type.

http://hackage.haskell.org/package/array-0.4.0.0/docs/Data-Array-Unboxed.html

UArray (1)

However, **UArray** is **strict** in its elements -
so don't use UArray if you require
the non-strictness that Array provides.

Because the **IArray** interface provides operations
overloaded on the type of the array,
it should be possible to just change the array type
being used by a program from say Array to UArray
to get the benefits of unboxed arrays
(don't forget to import Data.Array.Unboxed instead of Data.Array).

http://hackage.haskell.org/package/array-0.4.0.0/docs/Data-Array-Unboxed.html

Mutable arrays

	Immutable Instance **Iarray a e**	IO Monad Instance **MArray a e IO**	ST Monad Instance **MArray a e ST**
Standard	**Array** **DiffArray**	**IOArray**	**STArray**
Unboxed	**UArray** **DiffUArray**	**IOUArray** **StorableArray**	**STUArray**

https://wiki.haskell.org/Arrays

Index Types

The Ix library defines a type class of array indices:

```
class  (Ord a) => Ix a  where
    range       :: (a,a) -> [a]
    index       :: (a,a) a -> Int
    inRange     :: (a,a) -> a -> Bool
```

https://www.haskell.org/tutorial/arrays.html

Index Types (1)

The range operation takes a bounds pair and produces the list of indices lying between those bounds, in index order. For example,

range (0,4) => [0,1,2,3,4]

range ((0,0),(1,2)) => [(0,0), (0,1), (0,2), (1,0), (1,1), (1,2)]

The inRange predicate determines whether an index lies between a given pair of bounds. (For a tuple type, this test is performed component-wise.)

https://www.haskell.org/tutorial/arrays.html

Index Types (2)

Finally, the index operation allows a particular element of an array to be addressed: given a bounds pair and an in-range index, the operation yields the zero-origin ordinal of the index within the range; for example:

```
index (1,9) 2 => 1

index ((0,0),(1,2)) (1,1) => 4
```

https://www.haskell.org/tutorial/arrays.html

Array Creation (1)

Haskell's monolithic array creation function forms
an array from a pair of bounds and a list of index-value pairs
(an association list):

```
array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

Here, for example, is a definition of an array of
the squares of numbers from 1 to 100:

```
squares = array (1,100) [(i, i*i) | i <- [1..100]]
```

https://www.haskell.org/tutorial/arrays.html

Array Creation (2)

Array subscripting is performed with the infix operator !,
and the bounds of an array can be extracted with the function bounds:

squares!7 => 49

bounds squares => (1,100)

We might generalize this example by parameterizing
the bounds and the function to be applied to each index:

```
mkArray                 :: (Ix a) => (a -> b) -> (a,a) -> Array a b
mkArray f bnds          =  array bnds [(i, f i) | i <- range bnds]
```

Thus, we could define squares as mkArray (\i -> i * i) (1,100).

https://www.haskell.org/tutorial/arrays.html

Array Creation (3)

Many arrays are defined recursively; that is,

with the values of some elements depending on the values of others.

Here, for example,

we have a function returning an array of Fibonacci numbers:

```
fibs    :: Int -> Array Int Int
fibs n  =  a  where a = array (0,n) ([(0, 1), (1, 1)] ++
                                  [(i, a!(i-2) + a!(i-1)) | i <- [2..n]])
```

https://www.haskell.org/tutorial/arrays.html

Array Creation (4)

Another example of such a recurrence is the n by n wavefront matrix, in which elements of the first row and first column all have the value 1 and other elements are sums of their neighbors to the west, northwest, and north:

```
wavefront       :: Int -> Array (Int,Int) Int
wavefront n     =  a  where
                   a = array ((1,1),(n,n))
                        ([((1,j), 1) | j <- [1..n]] ++
                         [((i,1), 1) | i <- [2..n]] ++
                         [((i,j), a!(i,j-1) + a!(i-1,j-1) + a!(i-1,j))
                                     | i <- [2..n], j <- [2..n]])
```

https://www.haskell.org/tutorial/arrays.html

Accumulation (1)

We can relax the restriction that an index appear at most once
in the association list by specifying how to combine
multiple values associated with a single index;
the result is called an accumulated array:

```
accumArray :: (Ix a) -> (b -> c -> b) -> b -> (a,a) -> [Assoc a c] -> Array a b
```

https://www.haskell.org/tutorial/arrays.html

Accumulation (2)

```
hist            :: (Ix a, Integral b) => (a,a) -> [a] -> Array a b
hist bnds is    =  accumArray (+) 0 bnds [(i, 1) | i <- is, inRange bnds i]
```

Suppose we have a collection of measurements on the interval [a,b),
and we want to divide the interval into decades and count
the number of measurements within each:

```
decades         :: (RealFrac a) => a -> a -> [a] -> Array Int Int
decades a b     =  hist (0,9) . map decade
                   where decade x = floor ((x - a) * s)
                         s        = 10 / (b - a)
```

https://www.haskell.org/tutorial/arrays.html

Incremental Update (1)

In addition to the monolithic array creation functions,
Haskell also has an incremental array update function,
written as the infix operator //; the simplest case,
an array a with element i updated to v, is written a // [(i, v)].
The reason for the square brackets is
that the left argument of (//) is an association list,
usually containing a proper subset of the indices of the array:

```
(//)            :: (Ix a) => Array a b -> [(a,b)] -> Array a b
```

https://www.haskell.org/tutorial/arrays.html

Incremental Update (2)

As with the array function, the indices in the association list must be unique for the values to be defined.

For example, here is a function to interchange two rows of a matrix:

```
swapRows :: (Ix a, Ix b, Enum b) => a -> a -> Array (a,b) c -> Array (a,b) c
swapRows i i' a =  a // ([((i ,j), a!(i',j)) | j <- [jLo..jHi]] ++
                         [((i',j), a!(i ,j)) | j <- [jLo..jHi]])
                 where ((iLo,jLo),(iHi,jHi)) = bounds a

swapRows i i' a =  a // [assoc | j <- [jLo..jHi],
                                 assoc <- [((i ,j), a!(i',j)),
                                           ((i',j), a!(i, j))] ]
                 where ((iLo,jLo),(iHi,jHi)) = bounds a
```

https://www.haskell.org/tutorial/arrays.html

Matrix Multiplication (1)

```
matMult         :: (Ix a, Ix b, Ix c, Num d) =>
                 Array (a,b) d -> Array (b,c) d -> Array (a,c) d
matMult x y     =  array resultBounds
                    [((i,j), sum [x!(i,k) * y!(k,j) | k <- range (lj,uj)])
                              | i <- range (li,ui),
                                j <- range (lj',uj') ]
      where ((li,lj),(ui,uj))         =  bounds x
            ((li',lj'),(ui',uj'))     =  bounds y
            resultBounds
              | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
              | otherwise             = error "matMult: incompatible bounds"
```

https://www.haskell.org/tutorial/arrays.html

Matrix Multiplication (2)

```
matMult x y     =  accumArray (+) 0 resultBounds
                            [((i,j), x!(i,k) * y!(k,j))
                                    | i <- range (li,ui),
                                      j <- range (lj',uj')
                                      k <- range (lj,uj)  ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: incompatible bounds"
```

https://www.haskell.org/tutorial/arrays.html

Matrix Multiplication (3)

```
genMatMult      :: (Ix a, Ix b, Ix c) =>
                   ([f] -> g) -> (d -> e -> f) ->
                   Array (a,b) d -> Array (b,c) e -> Array (a,c) g
genMatMult sum' star x y  =
      array resultBounds
            [((i,j), sum' [x!(i,k) `star` y!(k,j) | k <- range (lj,uj)])
                                 | i <- range (li,ui),
                                   j <- range (lj',uj') ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: incompatible bounds"
```

https://www.haskell.org/tutorial/arrays.html

Matrix Multiplication (3)

```
genMatMult      :: (Ix a, Ix b, Ix c) =>
                   ([f] -> g) -> (d -> e -> f) ->
                   Array (a,b) d -> Array (b,c) e -> Array (a,c) g
genMatMult sum' star x y  =
      array resultBounds
            [((i,j), sum' [x!(i,k) `star` y!(k,j) | k <- range (lj,uj)])
                                | i <- range (li,ui),
                                  j <- range (lj',uj') ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: incompatible bounds"
```

https://www.haskell.org/tutorial/arrays.html

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf