50001 - Algorithm Analysis and Design - Lecture 16

Oliver Killane

18/12/21

Lecture Recording

Lecture recording is available here

Mutable Algorithms

We can use **STRef s a** to hold a mutable reference to **a** that can be created, read and modified.

```haskell

-- State Transformer (ST) takes a state s and return value a.
-- "Give me any program with any state s, and if it returns an a, so will I".
runST :: (forall s . ST s a) -> a

-- Take a value a, produces a program with some state s, that returns a
-- reference with that state and the value stored.
newSTRef :: a -> ST s (STRef s a)

-- Takes in a reference in some state s, performs a computation to get a
readSTRef :: STRef s a -> ST s a

-- Take in a reference, and a new value a, performing a computation to update
-- the referenced value (update does not return anything itself, hence ()).
writeSTRef :: STRef s a -> a -> ST s ()

-- Takes a value, returns a computation that executes in any state s to return
-- an a.
return :: a -> ST s a
```

We can use this to create a mutable version of fibonacci.

```haskell
import Data.STRef (newSTRef, readSTRef, writeSTRef)
import Control.Monad.ST (runST)

-- immutable looping fibonacci
fib :: Int -> Integer
fib n = loop n 0 1
  where
    loop :: Int -> Integer -> Integer -> Integer
    loop 0 x y = x
    loop n x y = loop (n-1) y (x+y)

-- mutable looping fibonacci
fib0 :: Int -> Integer
fib0 n = runST $ do
  rx <- newSTRef 0
  ry <- newSTRef 1
  let loop 0 = do readSTRef rx
      loop n = do {
        x <- readSTRef rx;
        y <- readSTRef ry;
        writeSTRef rx y;
        writeSTRef ry (x + y);
        loop (n - 1);}
  loop n
```

Mutable Datastructures

Array

```haskell
import Control.Monad.ST
import Data.Array.ST

-- Use indexable i, range, default to get a mutable array in some state
newArray :: Ix i => (i,i) -> a -> ST s (STArray s i a)

-- Use indexable i, a mutable array and return the value a with state s
readArray :: Ix i -> STArray s i a -> i -> ST s a

-- Use indexable i, a mutable array, and the value to place, return a
-- computation that updates the array (new state) but returns no value.
-- note:
writeArray :: Ix i => STArray s i a -> i -> a -> ST s ()
```

Each operation is assumed to take constant time.

For example, an algorithm to find the smallest natural number not in a list.

```haskell
import Data.Array.MArray (MArray(newArray))
import Data.Array.ST
import Control.Monad.ST
import Data.List ((\\))

-- immutable
minfree :: [Int] -> Int
minfree xs = head ( [0..] \\ xs )

-- effectively the same as minfree
minfree' :: [Int] -> Int
minfree' xs = head . filter(not . (`elem` xs)) $ [0..]


-- Builds up an array of which are present,
minfreeMut :: [Int] -> Int
minfreeMut = length . takeWhile id . checklist

{-
Build an array, at each index True/False for if the index is in xs
We only need to use an array of size (length xs) as we do not care about
natural numbers larger than this (if they are in the list, then a smaller
natural number was missed).

xs [0,1,2,3,6,7,8]
ys [T,T,T,T,F,F,T] ... (don't care about 7 or 8)

-}
checklist :: [Int] -> [Bool]
checklist xs  = runST $ do {
  ys <- newArray (0,l - 1) False :: ST s (STArray s Int Bool);
  sequence [writeArray ys x True | x <- xs, x < l];
  getElems ys;
  }
  where
    l = length xs
```

Hash

```haskell
class Hashable a where
    hash :: a -> Int
```

A hash generates an integer from some data. Typically range restricted (e.g hashmap can hold a finite number of entries), and the hash function should be designed to reduce collisions (two distinct data having the same hash).

Below is an example of a bucket based hash map, using linked list buckets.