COMP50001: Algorithm Design & Analysis

Sheet 2 (Week 3)

Exercise 2.1

Find a binary operation $(\diamond) :: (a \to a) \to (a \to a) \to (a \to a)$ and an element $\epsilon :: a \to a$ such that the set of functions of type $a \to a$ with $\diamond$ and ϵ forms a monoid.

Exercise 2.2

Given any two monoids $(M_1, \diamond_1, \epsilon_1)$ and $(M_2, \diamond_2, \epsilon_2)$, a *monoid homomorphism* from M_1 to M_2 is a function $h :: M_1 \to M_2$ such that

$$\begin{aligned} h\ (x \diamond_1 y) &= (h\ x) \diamond_2 (h\ y) \\ h\ \epsilon_1 &= \epsilon_2 \end{aligned}$$

Give three monoid homomorphisms from $([Int], +\!\!+, [\,])$ to $(Int, +, 0)$.

Exercise 2.3

Calculate the asymptotic time complexity of *concatl xs* below in terms of n and m where xs contains n lists, each containing m elements.

$$\begin{aligned} &concatl :: [[a]] \to [a] \\ &concatl = foldl\ (+\!\!+)\ [\,] \end{aligned}$$

Exercise 2.4

The *List* type class is shown in Figure 2.4. Complete the specification of the *List* type class by providing a default implementation for all the operations other than *fromList* and *toList*.

```
class List list where
  fromList :: [a] → list a
  toList :: list a → [a]
  normalize :: list a → list a

  empty :: list a
  single :: a → list a
  cons :: a → list a → list a
  snoc :: list a → a → list a

  head :: list a → a
  tail :: list a → list a
  init :: list a → list a
  last :: list a → a

  isEmpty :: list a → Bool
  isSingle :: list a → Bool
  length :: list a → Int
  (++) :: list a → list a → list a
  (!!) :: list a → Int → a
```

Figure 1: List class definition

Exercise 2.5

Implement an instance of *List* using standard lists $[a]$ without using functions from the *Prelude* other than the list constructors, and give the time complexities of each operation.

Exercise 2.6

Implement an instance of *List* using the following *Tree* type:

data *Tree a* = *Tip* | *Leaf a* | *Fork* (*Tree a*) (*Tree a*)

Ensure that the worst case complexity of $(+\!\!+)$ is $O(1)$. What is the worst case complexity of *head*?

Exercise 2.7

Define an instance of *List* using *DList* below, and give the complexities of all operations in terms of the length of the input list (assume all *DList* arguments to functions are built using the operations in *List*).

$$\textbf{newtype}\ \mathit{DList}\ a = \mathit{DList}\ ([a] \to [a])$$

Hint: $\mathit{fromList}\ \mathit{xs}\ =\ \mathit{DList}\ (\mathit{xs}{+\!\!+})$. Consider carefully whether the time complexity is affected by strict or lazy evaluation.

Exercise 2.8

Explain why the following implementation of fromList is undesirable in the last exercise:

$$\mathit{fromList}\ \mathit{xs} = \mathit{DList}\ ({+\!\!+}\mathit{xs})$$

Exercise 2.9

Prove or disprove the following assertions for the *DList* instance of *List* from Exercise (2.7).

1. $\mathit{fromList}\ (\mathit{toList}\ \mathit{dxs}) = \mathit{dxs}$ for any $\mathit{dxs} :: \mathit{DList}\ a$.
2. $\mathit{toList}\ (\mathit{fromList}\ \mathit{xs}) = \mathit{xs}$ for any $\mathit{xs} :: [a]$.

Solutions to the Exercises

Solution 2.1

Define $(\diamond)\ f\ g\ x = f\ (g\ x)$, i.e., $(\diamond)$ is function composition, and $\epsilon\ x = x$, i.e., ϵ is the identity function *id*. For any $x :: a$,

$$(\epsilon \diamond f)\ x = \epsilon\ (f\ x) = f\ x = f\ (\epsilon\ x) = (f \diamond \epsilon)\ x$$

so $\epsilon \diamond f = f \diamond \epsilon$. Similarly, for any $x :: a$,

$$(f \diamond (g \diamond h))\ x = f\ (g\ (h\ x)) = ((f \diamond g) \diamond h)\ x$$

so functions of $a \to a$ with $\diamond$ and ϵ forms a monoid.

Solution 2.2

1. The constant function mapping all lists *xs* to 0 is a monoid homomorphism.
2. The function $length :: [Int] \to Int$ is a monoid homomorphism.
3. The function $sum :: [Int] \to Int$ defined by

   $$\begin{aligned} &sum\ [\,] &&= 0 \\ &sum\ (x : xs) &&= x + sum\ xs \end{aligned}$$

   is a monoid homomorphism.

Solution 2.3

We define a ternary recurrence relation $T(k, n, m)$ to compute the asymptotic complexity of $foldl\ (+\!\!+)\ ys\ xss$ where $ys :: [a]$ contains k elements and $xss :: [[a]]$ contains n lists of m a-elements. Because $foldl\ (+\!\!+)\ ys\ [\,] = ys$,

$$T(k, 0, m) = 1.$$

Also we have $foldl\ (+\!\!+)\ ys\ (xs : xss) = foldl\ (+\!\!+)\ (ys +\!\!+ xs)\ xss$. In strict time analysis, argument $ys +\!\!+ xs$ must be computed before recursive call to *foldl*. Since the time complexity of computing $ys +\!\!+ xs$ is $O(length\ ys)$,

$$T\ (k, n, m) = k + T\ (k + m, n - 1, m).$$

Then the time complexity of $concatl\ xss = foldl\ (+\!\!+)\ [\,]\ xss$ is

$$\begin{aligned} &T(0, n, m) \\ =\ &0 + T(m, n-1, m) \\ =\ &0 + m + T(2m, n-2, m) \\ =\ &0 + m + 2m + T(3m, n-3, m) \\ =\ &\ldots \\ =\ &(\sum_{k=0}^{n-1} k * m) + T(nm, n-n, m) \\ \in\ &\Theta(n^2 m) \end{aligned}$$

Solution 2.4

It is not desirable to provide a default implementation of *fromList* and *toList*, since serve as the bridge between the abstract datatype $[a]$ and the concrete type *list a*.

```
class List list where
  fromList :: [a] → list a
  toList :: list a → [a]
  normalize :: list a → list a
  normalize xs = fromList (toList xs)

  empty :: list a
  empty = fromList [ ]

  single :: a → list a
  single x = fromList [x]

  cons :: a → list a → list a
  cons x xs = fromList (x : toList xs)

  snoc :: list a → a → list a
  snoc xs x = fromList (toList xs ++ [x])

  head :: list a → a
  head xs = head (toList xs)

  tail :: list a → list a
  tail xs = fromList (tail (toList xs))

  init :: list a → list a
  init xs = fromList (init (toList xs))

  last :: list a → a
  last xs = last (toList xs)

  isEmpty :: list a → Bool
  isEmpty xs = isEmpty (toList xs)

  isSingle :: list a → Bool
  isSingle xs = isSingle (toList xs)

  length :: list a → Int
  length xs = length (toList xs)

  (++) :: list a → list a → list a
  xs ++ ys = fromList (toList xs ++ toList ys)

  (!!) :: list a → Int → a
  xs !! i = toList xs !! i
```

Solution 2.5

```
instance List [] where
    -- fromList xs: O(1)
  fromList = id
    -- toList xs: O(1)
  toList    = id
    -- normalize xs: O(1)
  normalize = id
    -- empty: O(1)
  empty :: [a]
  empty = []
    -- single x: O(1)
  single :: a -> [a]
  single x = [x]
    -- cons x xs: O(1)
  cons :: a -> [a] -> [a]
  cons = (:)
    -- snoc xs x: O(n) where n = length xs
  snoc :: [a] -> a -> [a]
  snoc xs x = xs ++ [x]
    -- head xs: O(1)
  head :: [a] -> a
  head []       = error "head: empty list"
  head (x : xs) = x
    -- tail xs: O(1)
  tail :: [a] -> [a]
  tail []       = error "tail: empty list"
  tail (x : xs) = xs
    -- init xs: O(n) where n = length xs
  init :: [a] -> [a]
  init []       = error "init: empty list"
  init [x]      = []
  init (x : xs) = x : init xs
    -- last xs: O(n) where n = length xs
  last :: [a] -> a
  last []       = error "last: empty list"
  last [x]      = x
  last (x : xs) = last xs
    -- isEmpty xs: O(1)
  isEmpty :: [a] -> Bool
  isEmpty [] = True
  isEmpty _  = False
    -- isSingle xs: O(1)
  isSingle :: [a] -> Bool
  isSingle [x]  = True
  isSingle _    = False
    -- length xs: O(n) where n = length xs
  length :: [a] -> Int
  length []       = 0
  length (x : xs) = 1 + length xs
    -- xs ++ ys: O(n) where n = length xs
  (++) :: [a] -> [a] -> [a]
  [] ++ ys       = ys
  (x : xs) ++ ys = x : xs ++ ys
    -- xs !! i: O(n) where n = length xs
  (!!) :: [a] -> Int -> a
  []       !! n = error "(!!): empty list"
  (x : xs) !! 0 = x
  (x : xs) !! n = xs !! (n - 1)
```

Solution 2.6

This is a naive (but complete!) solution that makes no attempt to balance the trees:

```
instance List Tree where
  fromList [] = Tip
```

$$fromList\ (x : xs) = Fork\ (Leaf\ x)\ (fromList\ xs)$$

$$\begin{aligned} &toList\ Tip &&= [\,] \\ &toList\ (Leaf\ x) &&= [x] \\ &toList\ (Fork\ txs\ tys) &&= toList\ txs +\!\!+ toList\ tys \end{aligned}$$

$$txs +\!\!+ tys = Fork\ txs\ tys$$

All the other definitions use the default implementation.

Solution 2.7

Is $empty = DList\ (\lambda xs \to [\,])$ correct?

instance *List DList* **where**

-- *toList*: $O(n)$
$toList :: DList\ a \to [a]$
$toList\ (DList\ dxs) = dxs\ [\,]$

-- *fromList*: $O(1)$
$fromList :: [a] \to DList\ a$
$fromList\ xs = DList\ (xs +\!\!+)$

-- $(+\!\!+)$: $O(1)$
$(+\!\!+) :: DList\ a \to DList\ a \to DList\ a$
$DList\ dxs +\!\!+ DList\ fys = DList\ (dxs \circ fys)$

-- *empty*: $O(1)$
$empty :: DList\ a$
$empty = DList\ (\lambda xs \to xs)$

-- *single x*: $O(1)$
$single :: a \to DList\ a$
$single\ x = DList\ (x{:})$

-- *cons x xs*: $O(1)$
$cons :: a \to DList\ a \to DList\ a$
$cons\ x\ (DList\ dxs) = DList\ ((x{:}) \circ dxs)$

-- *snoc xs x*: $O(1)$
$snoc :: DList\ a \to a \to DList\ a$
$snoc\ (DList\ dxs)\ x = DList\ (dxs \circ (x{:}))$

-- *head xs*: $O(n)$, where $n = length\ xs$
$head :: DList\ a \to a$
$head\ xs = head\ (toList\ xs)$

-- *tail xs*: $O(n)$, where $n = length\ xs$
$tail :: DList\ a \to DList\ a$
$tail\ xs = fromList\ (tail\ (toList\ xs))$

-- *init xs*: $O(n)$, where $n = length\ xs$
$init :: DList\ a \to DList\ a$
$init\ xs = fromList\ (init\ (toList\ xs))$

-- *last xs*: $O(n)$, where $n = length\ xs$
$last :: DList\ a \to a$
$last\ xs = last\ (toList\ xs)$

-- *isEmpty xs*: $O(n)$, where $n = length\ xs$
$isEmpty :: DList\ a \to Bool$
$isEmpty\ xs = isEmpty\ (toList\ xs)$

-- *isSingle xs*: $O(n)$, where $n = length\ xs$
$isSingle :: DList\ a \to Bool$
$isSingle\ xs = isSingle\ (toList\ xs)$

-- *length xs*: $O(n)$, where $n = length\ xs$
$length :: DList\ a \to Int$
$length\ xs = length\ (toList\ xs)$

-- *xs !! i*: $O(n)$, where $n = length\ xs$
$(!!) :: DList\ a \to Int \to a$
$xs\ !!\ n = toList\ xs\ !!\ n$

In the above implementation, any operation returning some $DList\ f$ satisfies that $f = (xs +\!\!+)$ for some $xs :: [a]$, or $f = (x{:})$ for some $x :: a$, or f is the composite of two functions inside *DList*. Since $(xs +\!\!+)$ is equal to (and has the same asymptotic time complexity as)

$$(x_0:) \circ (x_1:) \ \ldots \circ (x_n:)$$

for xs = $[x_0, x_1, \ldots, x_n]$. It follows that for any *DList* $f :: DList\ \ a$ built from the interface of *List*, f is equal to (and have the same time complexity as)

$$(y_0:) \circ (y_1:) \ \ldots \circ (y_m:)$$

for a set of elements $y_0, y_1, \ldots, y_m :: a$. Thus in a lazy semantics, $toList\ (DList\, f) = f\ [\,]$ takes constant time and furthermore *head*, *tail*, *isEmpty*, and *isSingle* take constant time too.

Solution 2.8

With this definition, we can construct

$$\begin{aligned} d &= fromList\ xs_1 +\!\!+ (fromList\ xs_2 +\!\!+ (\ldots\ fromList\ xs_n)) \\ &= DList((+\!\!+xs_1) \circ (+\!\!+xs_2) \circ \ldots \circ (+\!\!+xs_n)) \end{aligned}$$

and evaluting $toList\ d = concatl\ [xs_1, xs_2, \ldots, xs_n]$. In Exercise (2.3), we have demonstrated that it takes $O(n^2 m)$ time to evaluate it with strict semantics, as opposed to $O(nm)$ where m is the maximum number of elements in xs_i.

Solution 2.9

1. Letting $dxs = DList\ reverse$,

   $$fromList\ (toList\ dxs) = fromList\ (reverse\ [\,]) = fromList\ [\,] = DList\ (\lambda xs \rightarrow [\,] +\!\!+ xs)$$

   which is different from *dxs*, so this property does not hold.

2. for any list $xs :: [a]$,

   $$toList\ (fromList\ xs) = toList\ (DList\ (xs +\!\!+)) = xs +\!\!+ [\,] = xs$$

   so this property holds.