50001 - Algorithm Analysis and Design - Lecture 5

Oliver Killane

12/11/21

Lecture Recording

Lecture recording is available here

DLists Continued...

Monoids (again)

A **monoid** is a triple $(M, \diamond, \epsilon)$ where $\diamond$ is associative and of type $M \to M \to M$, and $x \diamond \epsilon \equiv x$.

```haskell
class Monoid m where
    (<>) :: m -> m -> m
    mempty :: m
```

A haskell typeclass can then be instantiated for many other data types. For example the **monoid** $(\mathbb{Z}, +, 0)$ (note that we cannot enforce **monoid** properties through haskell, unlike languages such as **agda**).

```haskell
-- declaring newtype so that many monoid instance on Int do not conflict
newtype PlusInt = PlusInt Int

instance Monoid PlusInt where
    (<>) :: PlusInt -> PlusInt -> PlusInt
    (<>) = (+)

    mempty :: PlusInt
    mempty = 0
```

Likewise we can abstract Lists to a class (which we can instantiate for DLists).

List Class

```haskell
class List list where
    empty :: list a
    single :: a -> list a

    (:) :: a -> lis a -> list a
    snoc :: list a -> list a -> list a

    head :: list a -> a
    tail :: list a -> list a

    last :: list a -> a
    init :: list a -> list a

    (++) :: list a -> list a -> list a

    length :: list a -> Int

    fromList :: [a] -> list a
    toList :: list a -> [a]
```

$[a]$ is out abstract list type, and $lista$ is our concrete type.

It is critical to ensure that $toList \bullet fromList \equiv id$

But in general $fromList \bullet toList \not\equiv id$ (this is as the internal representation may change and much information about the internal representaion cannot be preserved by toList, for example an unbalanced tree changed to a list maybe be balanced when converted back to a tree).

We also included $normalise :: fromList \bullet toList$ as a useful tool to reset the internal structure (for example to rebalanced the tree representation of a list)

Haskell Implementation

To prevent conflicts due to Prelude functions already being defined we can use:

```
import Prelude hiding(head, tail, (++), etc...)
import qualified Prelude
```

To help ensure correctness we can use **Quickcheck** to check properties

```
cabal install --lib QuickCheck
```

Then can use quickcheck to define properties we want to test:

```
import Test.QuickCheck

-- code to test written here ...

prop_propertyname :: InputTypes -> Bool
prop_propertyname = test code

-- example for normalise (takes a list type, that has equality defined for it)
prop_normalise (Eq a, Eq (list a), List list) => list a -> Bool
prop_normalise xs = (toList . fromList) xs == xs

-- Can return properties (requires show) using the triple-equals
prop_assoc :: (Eq (list a), Show (list a), List list)
    => list a -> list a -> list a -> Property
prop_assoc xs ys zs = (xs ++ ys) ++ zs === xs ++ (ys ++ zs)
```

```
ghci file_to_check.hs
*file_to_check> quickCheck (prop_normalise :: [Int] -> Bool)
+++ OK, passed 100 tests.
*file_to_check> quickCheck (prop_normalise :: [Bool] -> Bool)
+++ OK, passed 100 tests.
*file_to_check> verboseCheck (prop_normalise :: [Bool] -> Bool)
Passed:
...

Passed:
...

etc...

+++ OK, passed 100 tests.
```