50001 - Algorithm Analysis and Design - Lecture 4

Oliver Killane

12/11/21

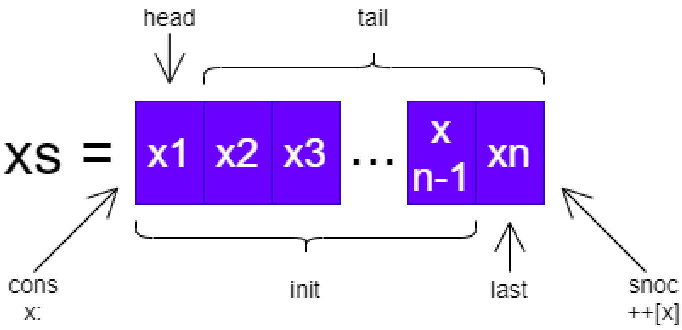
Lecture Recording

Lecture recording is available here

Lists

```haskell
data List [a] = [] | (:) a [a]

-- or...
data List a where
    Empty :: List a
    Cons :: a -> List a -> List a
```

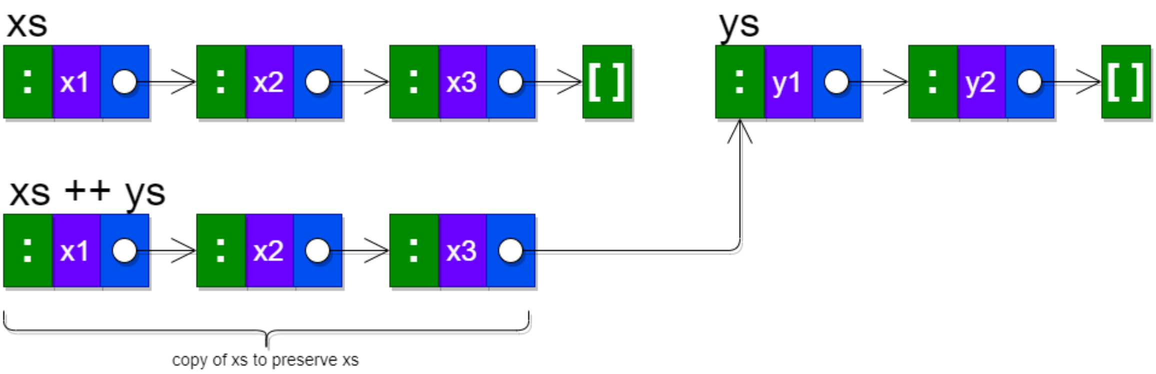


Lists in **Haskell** are a persistent data structure, meaning that when operations are applied to lists the original list is maintained (not mutated).

Append

We can append lists, by traversing over the first list, copying values (this ensures both argument lists are preserved).

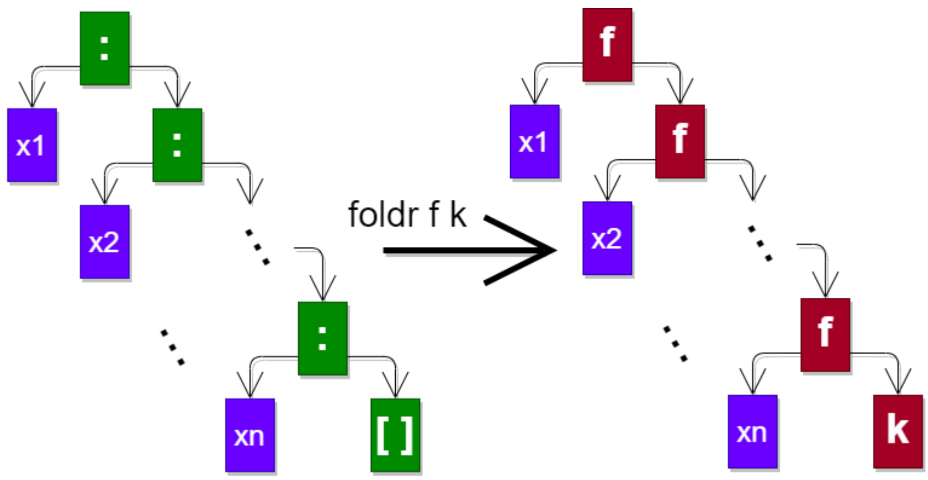
```haskell
(++) :: [a] -> [a] -> [a]
[]      ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```



As the entire first list must be traversed, the cost of $xs{+}{+}ys$ is $T_{(++)}(n) \in O(n)$ where $n = length\ xs$

Foldr

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f k []     = k
foldr f k (x:xs) = f x (foldr f k xs)
```



As you can see, $foldr\ (:)\ [] \equiv id$.
Foldr can also be expressed through bracketing

$$foldr\ f\ k\ [x_1, x_2, \dots, x_n] \equiv f\ x_1\ (f\ x_2\ (\dots(f\ x_n\ k)\dots))$$

Associativity

Associativity determines how operations are grouped in the absence of brackets.

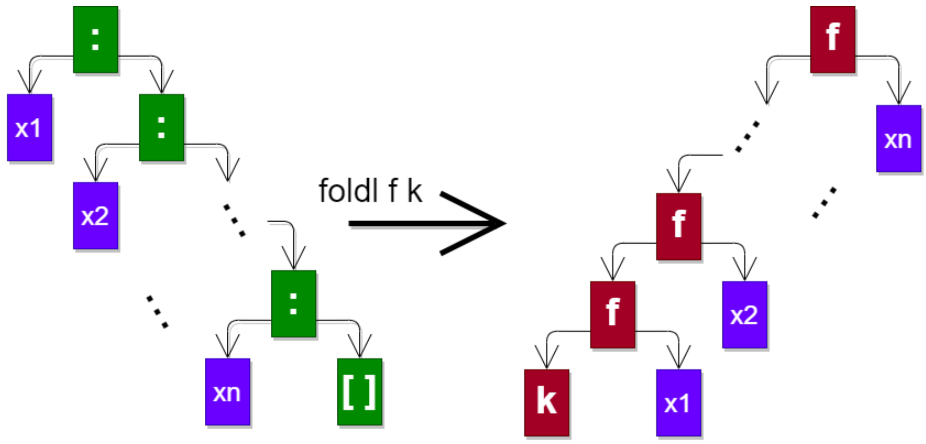
$a \spadesuit b \spadesuit c$	unbracketed statement
$((a) \spadesuit b) \spadesuit c$	$\spadesuit$ is left associative
$a \spadesuit (b \spadesuit (c))$	$\spadesuit$ is right associative

If $\spadesuit$ is associative, then the right & left associative versions are equivalent.

foldr applies functions in a right-associative scheme.

Foldl

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f k []     = k
foldl f k (x:xs) = foldl f (f k x) xs
```



As you can see $foldl\ (snoc)\ [] \equiv id$.
Foldl can be expressed through bracketing

$$foldl\ f\ k\ [x_1, x_2, \ldots, x_n] \equiv f\ (\ldots(f\ (f\ k\ x_1)x_2)\ldots x_n)$$

Monoids

Consider the case when for some $\bigstar$ and ϵ: $foldr\ \bigstar\ \epsilon \equiv foldl\ \bigstar\ \epsilon$. For this to be possible for $\bigstar\ ::\ a \to a \to a$ and $\epsilon\ ::\ a$.

$\bigstar$ must be associative	$x \bigstar (y \bigstar z) \equiv (x \bigstar y) \bigstar z$
ϵ must have no effect	$\epsilon \bigstar n = n$

These properties for a **monoid** $(a, \bigstar, \epsilon)$.
Other example include:

$$\begin{array}{cccc} (lists, ++, []) & (\mathbb{N}, +, 0) & (\mathbb{N}, \times, 1) & (bool, \wedge, true) \\ (bool, \vee, false) & (\mathbb{R}, max, \infty) & (\mathbb{R}, min, -\infty) & (Universal\ set, \cup, \emptyset) \end{array}$$

We can also find monoids of functions:

$$(a \to a, (.), id)$$

as $(id\ .\ g)x \equiv id(g\ x)$ and $((f\ .\ g)\ .h)x = f(g(h\ x))$

Concat

We can easily define concat recursively as:

```
concat :: [[a]] -> [a]
concat []       = []
concat (xs:xss) = xs ++ concat xss
```

We can also notice that $([[a]], (++), [])$ is a monoid, so we can use **foldr** or **foldl**

```
concatr :: [[a]] -> [a]
concatr = foldr (++) []
```

```
concatr :: [[a]] -> [a]
concatr = foldl (++) []
```

as (++) makes a copy of the first argument (to ensure persistent data), if we apply is in a left associative bracketing scheme we will have to make larger & larger copies.

$$(\dots(((([\,]{+}{+}_0\ xs_1){+}{+}_m\ xs_2){+}{+}_{2m}\ xs_3){+}{+}_{3m}\ xs_4\dots){+}{+}_{mn}\ xs_n$$

Hence where $n = length\ xss$ and $m = length\ xs_1 = length\ xs_2 = \dots = length\ xs_n$.

$$T_{concatl}(m,n) \in O(n^2 m)$$
$$T_{concatr}(m,n) \in O(nm)$$

DLists

Instead of storing a list, we store a composition of functions that build up a list.

$$xs_1{+}{+}xs_2{+}{+}xs_3{+}{+}\dots{+}{+}xs_n$$
$$\Downarrow$$
$$f\ xs_1 \bullet f\ xs_2 \bullet f\ xs_3 \bullet \dots \bullet f\ xs_n$$
$$\Downarrow$$
$$(xs_1\ {+}{+}) \bullet (xs_2\ {+}{+}) \bullet (xs_3\ {+}{+}) \bullet \dots \bullet (xs_n\ {+}{+})$$

We can then apply this function on the empty list $[\,]$ to get the resulting list.

```
newtype DList a = DList ([a] -> [a])

instance List DList where
    toList :: DList a -> [a]
    toList (DList fxs) = fxs []

    fromList :: [a] -> DList a
    fromList xs = DList (xs++)

    (++) :: DList a -> DList a -> DList a
    DList fxs ++ DList fys = DList(fxs . fys)
```

We can form a **monoid** of ($DList$,++,$DList\ id$).