Building recursive data structures in Haskell

Duncan Coutts

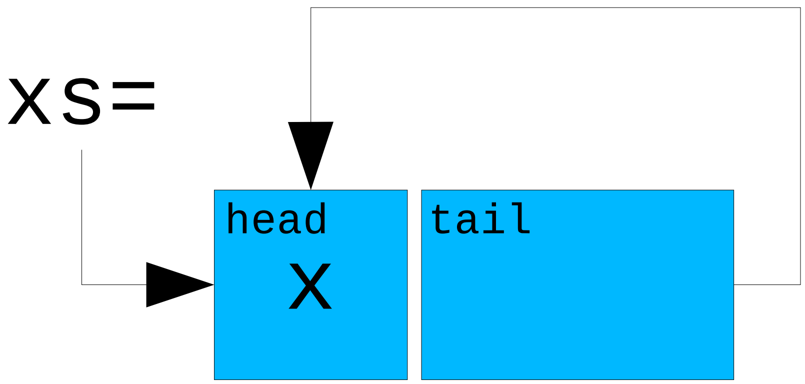
4/12/03

Infinite Values

- Haskell allows us to build 'infinite values' with finite representation
- For example the prelude function `repeat` returns an infinite list of the same element

```
repeat :: a -> [a]
repeat x = x : x : x : ...
```

Representation



- recursive structures can be represented using pointers

Representation

- In traditional imperative languages we would explicitly use pointers

```
struct List {
    void *head;
    List *tail;
}

List* repeat(void *x) {
    List *xs = new List;
    xs->head = x;
    xs->tail = xs;          /* close the loop */
    return xs;
}
```

Haskell version using let

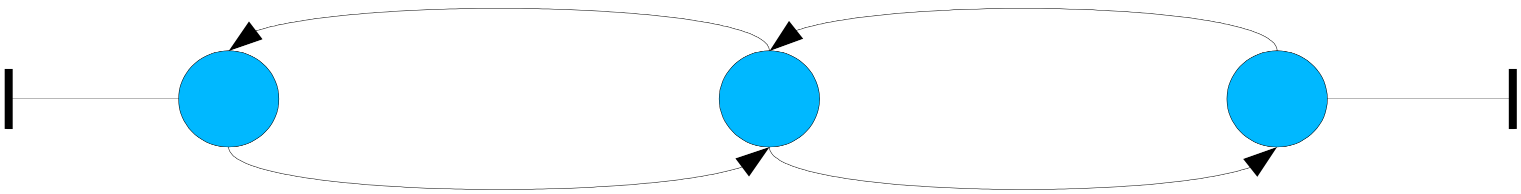
- In Haskell we don't have mutable references but we do have lazy evaluation and recursive let

```
repeat :: a -> [a]
repeat x = let xs = x:xs
            in xs
```

That's nice but how do we build more complicated structures?

Doubly linked lists

- As a first example of building more complex cyclic structures we'll look at doubly linked lists



Doubly linked lists

- The data type

```
data List a = Node a (List a) (List a)
            | Nil
```

- Values of this type will not persist well, we will not be able to build them incrementally.
- We'll have to build them all in one go

```
mkList :: [a] -> List a
mkList [] = Nil
mkList (x:xs) = ???
```

Doubly linked lists

- Some special cases

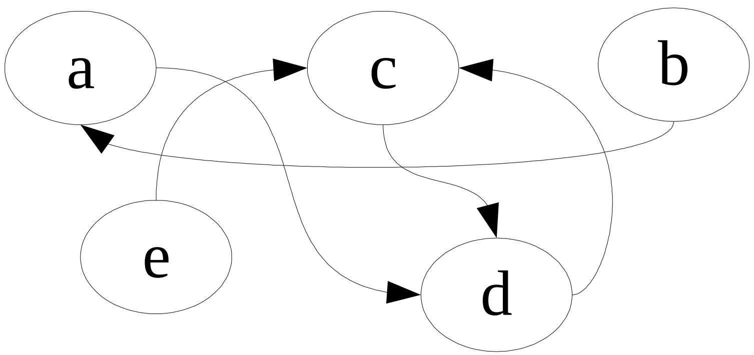
```
mkList :: [a] -> List a
mkList []       = Nil
mkList [x]      = Node a Nil Nil

mkList [x1, x2] = let node1 = Node x1 Nil   node2
                      node2 = Node x2 node1 Nil
                   in node1

mkList [x1, x2, x3] = let node1 = Node x1 Nil   node2
                          node2 = Node x2 node1 node3
                          node3 = Node x3 node2 Nil
                       in node1
```

Doubly linked lists

- For the general case we add an extra argument `prev` which is the previous node

```
mkList' :: [a] -> List a -> List a
mkList' []     prev = Nil
mkList' (x:xs) prev = let cur = Node x prev (mkList' xs cur)
                          in cur

mkList :: [a] -> List
mkList xs = mkList' xs Nil
```

Graphs



node	link
a	3
b	0
c	3
d	2
e	2

- Next we'll look at graphs
- For starters we'll consider directed graphs where each node has exactly one outgoing edge

Graphs

- The data type
  `data Graph a = GNode a (Graph a)`
- We want a function that builds a `Graph` from the table of nodes with explicit integer links

```
mkGraph :: [(a, Int)] -> Graph a
```

Graphs

node	link
a	1
b	2
c	3
d	4
e	0

node	link
a	3
b	0
c	3
d	2
e	2

- Last time we built the structure by tracing a path through it.
- With the graph, the pattern of links is not linear or predictable.
- What recursive value can we name?

```
Let x = ...x...
```

Graphs

- We can name the table!
- We can build all the links 'simultaniously' by using a collection

```
mkGraph :: [(a, Int)] -> Graph a
mkGraph table  = table' ! 0
  where table' = listArray (0, length table - 1) $
          map (\(x, n) -> GNode x (table' ! n)) table
```

This example uses a Haskell array, but any collection implementation that is lazy in its elements would do.

General Directed Graphs

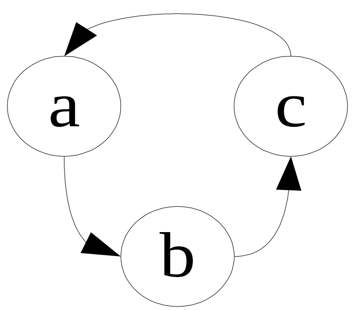
- We can easily generalise the last example to general directed graphs

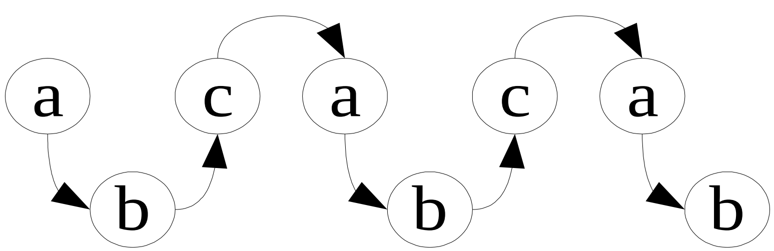
```
data GGraph a = GGNode a [GGraph a]

mkGGraph :: [(a, [Int])] -> GGraph a
mkGGraph table = table' ! 0
  where table' = listArray (0, length table - 1) $
                 map (\(x, ns) ->
                  GGNode x (map (table' !) ns)) table
```

Advantages & Disadvantages

- Advantages of cyclic representations over representations with explicit links
  - No need to deal with node names
  - Faster structure traversal
- Disadvantages
  - Cannot “escape” structure
  - Cannot update structure incrementally

Thinking about sharing





- Once we've built one of these recursive values does traversing it really take constant space?
- Can we be sure that we're not allocating new nodes as we traverse the structure?

Thinking about sharing

- For example, earlier we defined

```
repeat :: a -> [a]
repeat x = let xs = x:xs
            in xs
```

- Would this definition be 'the same'?

```
repeat' x = x:repeat' x
```

- We can easily prove them equal.

So they're equal but not the same huh?

Thinking about sharing

- I don't know of a useful semantics that allows one to reason about sharing. Remember that Haskell is specified as non-strict, not lazy.
- We can hand wave and make assumptions about how our compiler implements things.
- We can experiment by looking inside the evaluation using `Debug.trace`