Data Structures with Sharing

• We talked a lot about Trees in the previous lecture - a defining characteristic of Trees is that every node has a unique parent.
• In Graph terms, for each node there is at most one edge with that node as a target.
• What about structures where this doesn’t hold? Where there are more than two ways of addressing a structure, e.g. Graphs in fact.
• Functional languages seem particularly ill-suited to this.
• This is largely due to the lack of pointers or object references in the languages.
Three approaches to Graphs

• In this lecture we will look at three alternative approaches to constructing graph like data structures in Haskell.

Using indexed collections of Nodes, Edges

Using a structured data type with cyclic dependencies

An inductive approach using graph constructors

This is a non-structural approach

This has limitations for modifying the graphs

This provides inductive structure and pattern matching
Graphs using Indexed Collections of Nodes and Edges
The obvious approach

- In some sense, representing graphs simply as a list of nodes paired up somehow with a list of nodes each node is connected to.
- This requires that the nodes can act as indices in to a table like structure.
- We could just fix nodes to be of type Int, then a graph could be represented by a table of type graph :: Int \rightarrow [ Int ]. However, that is not really an efficient representation.
- Instead, Haskell has a package called Data.Array that provides generic indexing types
  - There is a class called I x whose instances (Nums, Chars etc) can act as indexes.
  - Data.Array is implemented so that Arrays are both monolithic and incremental.
  - This means memory is allocated completely before the array is filled but update is done incrementally by not creating new arrays.
Data.Graph

• Haskell has a stable Data.Graph package that is stable and dates from 20 years ago.

\[
\text{type Table } a = \text{Array Vertex } a
\]

\[
\text{type Graph } = \text{Table [ Vertex ]}
\]

• There are functions that allow you to build graphs from lists of edges, to fetch all vertices and edges, to reverse all edges and to count in/out degree.

• There are also functions that implement well known graph algorithms such as Depth First Search, Topological Sort, Reachability, Spanning Forests etc

• You have studied these in Algorithmics so we won’t repeat this here.

Vertex is any instance of class \( I x \) - e.g. Int

So a graph is a Vertex-indexed array of lists of Vertex values.
Some Comments on Table Approach

- The graphs are intended to be immutable
  - No changing the structure of them
- The types doesn’t allow for weighted graphs
- The performance is very slow compared to C
  - Can use a library of Haskell bindings to C graph library instead but this is no longer pure Haskell
- The graphs are not structural so all algorithms are implemented essentially as traversal across Arrays.
- See the paper “Structuring Depth-First Search Algorithms in Haskell”, by David King and John Launchbury 1995 for a fuller description.
Graphs using Cyclic Dependencies
Loopy List

Consider the example of a simple List structure with a loop in it so that the tail of the first node is the initial list.

In C

```c
struct List {
    int *head;
    List *tail;
}
List* repeat(void *x) {
    List *xs = new List;
    xs->head = x;
    xs->tail = xs;
    return xs;
}
```

In Java

```java
class List {
    int head;
    List tail;
}
List repeat(int x) {
    List xs = new List();
    xs.head = x;
    xs.tail = xs;
    return xs;
}
```

In Haskell

```haskell
repeat x =
  let xs = x : xs in
  xs
```

Okay, but that won’t work for more complicated examples will it?
Tying the Knot

- There is something odd going on in the Haskell definition of repeat
- The value \( xs \) is defined in terms of itself
- Lazy evaluation allows us to get away with this
- Further, this can be done for mutually recursive values also
- This actually defines an infinite list with alternating 0:1:0:1:0: \( \ldots \) values
- In fact, in memory it creates the structure
  \[ 0 : 1 : \text{thunk} \]
  where \( \text{thunk} \) is a means of calculating the rest of the list

```haskell
repeat x = let xs = x : xs in xs

cyclic = let x = 0 : y
         y = 1 : x
         in x
```

Using mutually recursion to define cyclic values lazily is known as “Tying the Knot”
Doubly Linked Lists

- Let’s try use this idea to build Doubly Linked Lists

\[
data \text{ DLLList} \ a = \text{Empty} \mid \text{Node} \ a \ (\text{DLLList} \ a) \ (\text{DLLList} \ a)
\]

- The nodes do not persist well and we cannot build incrementally.
- To build X and Z we need to know Y as it is the next/prev Node
- **Problem**: how do we construct nodes X and Z before Y?
- **Solution**: build them all at once from a regular list!
First steps at mkDLLList

```haskell
mkDLLList :: [a] -> DLList a
mkDLLList [] = Nil
mkDLLList (x:xs) = ???

We need something like this

mkDLLList [] = Empty
mkDLLList [x] = Node a Empty Empty
mkDLLList [x1, x2] = let node1 = Node x1 Nil node2
                     node2 = Node x2 node1 Nil
                     in node1
mkDLLList [x1, x2, x3] = let node1 = Node x1 Nil node2
                          node2 = Node x2 node1 node3
                          node3 = Node x3 node2 Nil
                          in node1
```

etc
General definition of mkDLLList

- To define the general case we use an auxiliary function `mkDLLList'` that accepts an extra argument, being the previous Node.

```haskell
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
```

The current node is the “previous” node in the next node.
Cyclic Definitions for Graphs

- Let’s move to more complicated structures.
- Doubly linked lists have sharing but the traversal pattern is straightforward and reaches every node.
- We built the structure by tracing a path right through it.
- For general graphs the pattern of links is not linear and is unpredictable.
- To get started we’ll restrict ourselves to graphs where each node has a single outgoing edge and generalise from this later.

<table>
<thead>
<tr>
<th>Node Value</th>
<th>Outgoing Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘a’</td>
<td>3</td>
</tr>
<tr>
<td>‘b’</td>
<td>0</td>
</tr>
<tr>
<td>‘c’</td>
<td>3</td>
</tr>
<tr>
<td>‘d’</td>
<td>2</td>
</tr>
<tr>
<td>‘e’</td>
<td>2</td>
</tr>
</tbody>
</table>
Cyclic Definitions for Graphs

• First let’s define a suitable data type

```haskell
data Graph a = GNode a (Graph a)
```

• We want to build a Graph structure from a table of nodes with their outgoing edge

```haskell
mkGraph :: [(a, int)] → Graph a
```

• For Doubly Linked lists we used a let binder to name each node and traversed the input list.

• We can’t do that here as we have no predictable path and the graph may not be connected.

• We can instead use a let binder to name the input table of links and tie the knot over that.
The outgoing edge is simply the result of making that node by mapping across the table.

Let's generalise this to multiple outgoing edges now:

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

```
mkGGraph :: [ (a, [Int]) ] -> GGraph a
mkGGraph table = table' !! 0
  where table' =
    map (\(x,ns) -> GNode x (map (table'!!) ns ) ) table
```
Some Comments on Cyclic Dependency Approach

• Advantages of cyclic approach
  • No explicit naming of nodes needed
  • Fast traversal - just follow the next link

• Disadvantages of cyclic approach
  • Cannot “escape” from the structure quickly, access via traversal
  • Cannot build structures incrementally

• How are the cyclic structures stored in memory
  • A lot of trust is being placed in the implementation that nodes are not duplicated on traversal - otherwise looping round a graph will cause memory leaks.
Inductive Graphs
FGL - The Functional Graph Library

- This is the library Data.Graph.Inductive.Graph
- It provides an inductive (structural) way of building graphs.
- Nodes in the graph are represented by Int values
- Every graph is either the empty graph or
- A graph extended by a new node V along with the incoming and outgoing edges on V
- So we build graphs by inserting one node at a time in to a previously built graph - like (:) adds elements to a list.
- However, unlike (:) and lists, the build order does not matter.
- Graphs cannot be extended with an existing node, nor can edges refer to non-existing nodes.
FGL build operations

empty :: Graph a b

Analagous to [], (:

embed :: Context a b → Graph a b → Graph a b

This concept is key

A context represents a detached node along with all of its incoming edges and outgoing edges and a value to store in the node.

type Adj b = [(b, Node)]

type Context a b = (Adj b, Node, a, Adj b)

List of labelled edges

Node id

label

Incoming

Outgoing
It is important to see that there are alternative ways to construct this graph - one node at a time.
Graph Decomposition

• An important benefit of building graphs inductively is the ability to reverse the process by decomposing graphs one node at a time.

• FGL provides a function

\[
\text{match} :: \text{Node} \rightarrow \text{Graph a b} \rightarrow \text{Decomp a b}
\]

\[
\text{type Decomp a b = ( Maybe (Context a b) , Graph a b )}
\]

where

• \text{match } v \text{ g} tries to find node } v \text{ in a graph and, if successful, it returns } v \text{’s context along with the remaining graph.}
  
  • The remaining graph is } g \text{ without any edges involving } v
  
  • Graphs may be decomposed in a different order in which they are built.

• \text{matchAny g} is similar but matches \textbf{any} node
Example using Decomposition

- Decomposition is used in FGL in order to provide many of the usual Graph algorithms: depth-first search etc
- To see how it is useful I’ll show one example: a map function for graphs.
- This function takes a function that transforms a context by relabelling the context node and recursively applies the relabelling to every node in the graph.

```haskell
gmap :: (Context a b) → (Context c b) → Graph a b → Graph c b

gmap f g | isEmpty g = empty
           | otherwise = embed (f c) (gmap f g')
           where (c,g') = matchAny g
```

```plaintext
relabel
```

```plaintext
rest of graph
```
Some Comments on the Inductive approach

- Advantages of the inductive approach:
  - Provides an inductive structure that almost, but not quite supports pattern matching for defining the graph functions.
  - Implementations of functions that are easier to reason about.

- Disadvantages:
  - Not the most efficient of implementations.
  - Not all graphs can be sensibly realised this way - dense graphs lose a lot of edges on decomposition.
YOUR QUESTIONS

Next Lecture:
Equational Reasoning