Dynamic Cyclic Data Structures in Lazy Functional Languages

Chris Clack, Stuart Clayman, David Parrott *
Department of Computer Science, University College London.
e-mail: {clack,sclayman}@cs.ucl.ac.uk

October 24, 1995

Abstract

Many popular myths have evolved about functional programming languages and their power of expression; one such myth is that "functional languages cannot construct arbitrary cyclic data structures at run-time" and therefore that functional languages are not appropriate for problems which require efficient representations of dynamically-specified cyclic graph structures (such as a route-finding robot in a maze). This myth persists despite ample published work to the contrary (see for example [All89]) and is a recurring subject of debate [USE].

We provide a survey of some of the related work in this area; we then build on this work by presenting methods for both the dynamic construction of complex cyclic structures and their direct manipulation using circular programming techniques.

1 Introduction

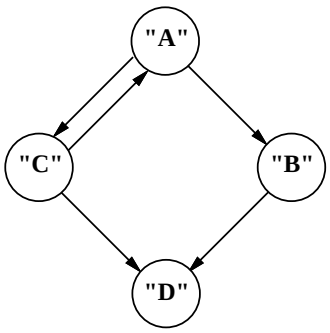

Figure 1: A small, directed, cyclic graph

If one were asked to use a functional language to construct the directed graph given in Figure 1 then the resulting code might typically resemble:[1]

```
> data Graph a = Node a [Graph a]
>
> graph::Graph [Char]
> graph = node0
>
> node0,node1,node2,node3 :: Graph [Char]
> node0 = Node "A" [node2, node1]
> node1 = Node "B" [node3]
> node2 = Node "C" [node0, node3]
> node3 = Node "D" []
```

In the above example, a graph and a graph node are the same thing. The graph data structure is defined recursively as a Node which holds a value and a list of other graph nodes. The definitions of the four graph nodes are mutually recursive, with each node using the *names* of the graph nodes to which it points.

1.1 Dynamic non-cyclic representations

It may be easy to create cyclic structures statically in a functional language, but it is difficult to create them "on-the-fly" at run-time. The static names that can be bound to data objects at compile time are no longer available at run-time, and it is not possible to introduce new names to label new graph nodes as they are encountered.

Traditionally, a functional programmer would solve this problem by constructing a *representation* of a graph, rather than a truly cyclic structure. Here are some alternative representations for the graph in Figure 1:

Nodes and Edges

Graphs may be represented by *a list of nodes and a separate list of edges*, thus circumventing the problem of physically connecting edges to nodes. For example, in [Hol91, page 95] a graph is represented as a pair containing a list of nodes and a list of edges, where an edge is defined by the values of the two nodes which it connects (this representation cannot, of course, be used where two or more

*During the course of this work David Parrott was supported by a SERC studentship

[1] In this paper we choose the Haskell language [HPJW+91] **(version 0.41)** for our examples.

nodes have the same node value—however, a node tag may be used instead of the node value):

```
> type Node  = [Char]
> type Edge  = (Node, Node)
> type Graph = ([Node], [Edge])
>
> buildgraph :: [Node]->[Edge]->Graph
> buildgraph ns es = (ns, es)
>
> nodes = ["A","B","C","D"]
> edges = [("A","B"),("A","C"),
>          ("B","D"),("C","A"),("C","D")]
>
> graph :: Graph
> graph = buildgraph nodes edges
```

Functional representation

Reade [Rea89, page 172] demonstrates how a graph may be represented as *a function which maps from a given node to a list of the successors of that node*, i.e. a list of the nodes pointed to by the given node:

```
> data Graph a = Agraph (a -> [a])
>
> next :: [Char] -> [[Char]]
> next "A" = ["B","C"]
> next "B" = ["D"]
> next "C" = ["A","D"]
> next "D" = []
>
> graph :: Graph [Char]
> graph =  Agraph next
```

Dynamic graph creation is also possible:

```
> data Graph a =  Agraph (a -> [a])
> type Node = [Char]
> type Nodes = [Node]
> type SuccessorMap = (Node,Nodes)
>
> next::[SuccessorMap]->Node->Nodes
> next l x
>  = (snd . head)
>     (filter (((==) x).fst) l)
>
> buildgraph :: [SuccessorMap]
>                -> Graph [Char]
> buildgraph succlist
>  = Agraph (next succlist)
>
> fig1_succs :: [ SuccessorMap ]
> fig1_succs
>  = [("A",["B","C"]),("B",["D"]),
>     ("C",["A","D"]), ("D",[]) ]
>
> graph :: Graph [Char]
> graph = buildgraph fig1_succs
```

Here the function `next` searches the successor list to find the connected nodes.

Virtual Heap

In the above example of a successor list, the ordering of the pairs in the list is immaterial because the successor values are the actual node values rather than references to the graph nodes. However, if the order is carefully chosen then the successor list itself can be used as *a virtual heap*, using list indexing to access individual nodes (and, of course, their successors which are identified by their index). We use the term "virtual heap" because the location of each cell is used as a virtual address:

```
> type Indices = [Int]
> type Node a  = (a, Indices)
> type Graph a = [Node a]
>
> buildgraph::[a]->[Indices]->Graph a
> buildgraph [] elist = []
> buildgraph (val:vals) (edges:elist)
>   = (val, edges) :
>     (buildgraph vals elist)
>
> nodes = ["A", "B", "C", "D"]
> edges = [[2,1],[3],[0,3],[]]
>
> graph :: Graph [Char]
> graph = buildgraph nodes edges
```

In this code, the `Indices` indicate virtual addresses in the `graph` virtual heap. To follow an index i, an expression such as `graph!!`i is used to return a value of type `Node [Char]`. The `Indices` type will therefore always be a list of `Int` although a graph might be described with node values of any type.

Burton [BY90] extends work in this area to show how to simulate a *mutable* graph, using a virtual heap with "pointers" (which are heap indices, as above). Burton's technique relies on "plumbing" a data structure throughout all of the functions which need access to it; this is uncomfortable for the programmer and enforces an almost total ordering on evaluation, which may be a disadvantage for a parallel implementation.

Arrays

If a programmer is lucky enough to have access to a lazy functional language which provides *arrays* (such as Haskell [HPJW+91]), then it is possible [BY90] to create a virtual heap representation which is potentially highly efficient.

```
> type Index = Int
> type Graph a = Array Index (a,[Index])
>
> buildgraph::[a]->[[Index]]->Graph a
> buildgraph vals elist
>  = listArray (0, (length vals)-1)
```

```
>               (zip vals elist)
>
> nodes = ["A", "B", "C", "D"]
> edges = [[2,1],[3],[0,3],[]]
>
> graph :: Graph [Char]
> graph = buildgraph nodes edges
```

King and Launchbury [KL94] use Haskell arrays, extended with update-in-place, to represent a graph which supports linear time depth-first search.

Summary

None of the above methods use machine pointers to represent the edges of graphs, all the non-array representations are subject to unwelcome time and/or space overheads, and none are as elegant as a truly cyclic structure. The array representation has the potential to be the most efficient of the above methods; if constant-time array indexing is supported then following an edge can be almost as fast with an array representation as it is with a cyclic structure (depending on the addressing modes of the underlying processor). However, very few lazy functional languages currently provide arrays and we therefore concentrate on truly cyclic graphs rather than array representations of graphs.

2 Dynamic Cyclic Graph Structures

The method of representing graphs given in this section combines the virtual heap and adjacency list approaches; it relies on lazy constructors to create a truly cyclic data structure which uses machine pointers to represent the edges of the graph and which can be built "on the fly" (though we delay discussion of dynamic construction until the next section). Both the adjacency list and virtual heap are intermediate tools for constructing the real graph: they will be garbage collected as soon as the graph is traversed.

The list-based "virtual heap" representation requires the space overhead of a top-level list of nodes and the time overhead to access nodes using the `!!` list indexing operator. Fortunately, *both* of these can be overcome as follows:

```
> data Graph a =  Node a  [Graph a]
> glist :: [Graph [Char]]
> glist = [ Node "A" [glist!!2, glist!!1],
>           Node "B" [glist!!3],
>           Node "C" [glist!!0, glist!!3],
>           Node "D" [] ]
> graph :: Graph [Char]
> graph = head glist
```

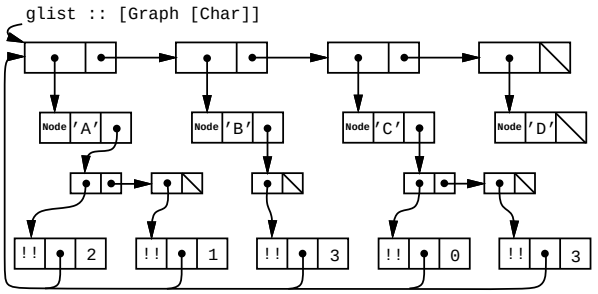

Figure 2: Representing a simple graph using embedded virtual addressing.

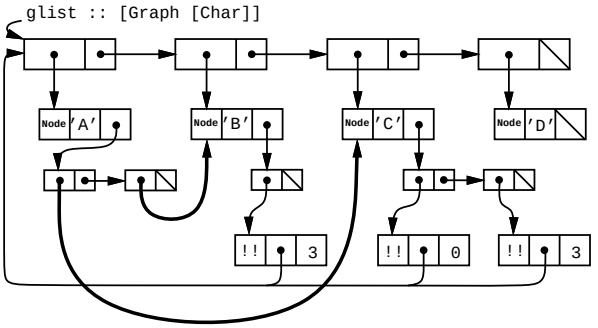

Figure 3: Following the arcs of the graph deletes references to the list structure.

The arcs of the graph are now represented by direct references to nodes. If every element of `graph` is visited without following any of the arcs then the resulting data structure will be that shown in Figure 2. At first glance it appears that the space overhead of the top-level list `glist` and the time overhead of the `!!` operator are still present. However, the `!!` operator is now embedded in the representation of the graph so, when an arc is followed, the index expression is re-written to point directly at the relevant element of the list. Figure 3 shows the re-written data structure after the two arcs of the initial node have been visited.

In Figure 3 there are fewer references to `glist` than there are in Figure 2. When *all* of the references to `glist` have been evaluated then the list structure is no longer required and can be garbage collected. At this point we are left with just the required data items, arranged as a graph with machine pointers representing the arcs. This is illustrated in Figure 4 where the final data structure compares favourably with the original graph of Figure 1. The space overhead of the enclosing list has

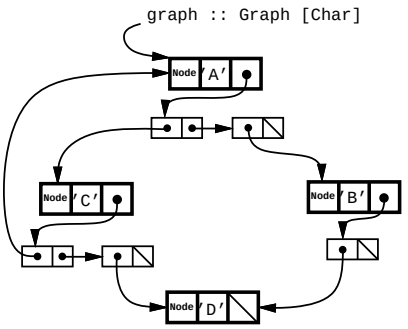

Figure 4: Eventually, the top-level list is garbage collected, leaving just the graph structure.

therefore been overcome, and future traversals of the graph are efficient in time because the pointers representing the arcs are followed directly.

It is clear that the above method can represent a rooted, connected graph (in the above code we assume that the first element of `glist` is the root; this is not compulsory, but it will be assumed throughout the rest of this paper). If it is desired to represent a non-rooted graph or a graph with many unconnected components, then the above technique can still be employed but the top-level `glist` must be retained in order to maintain access to each component. This implies a space overhead, but the individual components will still benefit from short-circuiting of the list references.

2.1 Dynamic Construction of a Cyclic Graph

A truly cyclic graph structure such as the one given above can be constructed "on the fly" by a function which takes a description of the graph and returns a real graph. An adjacency list is a convenient way to describe a graph: for example, the graph in Figure 1 can be described by the list `[("A",[2,1]), ("B",[3]), ("C",[0,3]), ("D",[])]`, where the numbers are self-referential indices into the adjacency list.

In order to construct the graph from its description, it is necessary to use a "circular" programming technique [Bir84]. This is a self-referential definition, where the result of the definition is used in the definition itself. A simple example is the following recursively-defined data structure where all three items in the list are the value `"datum"`:

```
> cyclic = [(cyclic !! 2),
>           (cyclic !! 0), "datum" ]
> graph  = head cyclic
```

Thus, the graph in Figure 1 might be described by the following code (we use an algebraic type to stop the type checker from looping when determining the type of the second element of the tuples):

```
> data Graph a = Node (a, [Graph a])
> cyclic :: [Graph [Char]]
> cyclic
>  = [Node ("A",[(cyclic !! 2),(cyclic !! 1)]),
>     Node ("B",[(cyclic !! 3)]),
>     Node ("C",[(cyclic !! 0), (cyclic !! 3)]),
>     Node ("D",[]) ]
> graph  = head cyclic
```

The above example can now be expanded into a function application:

```
> cyclic
>  = f1 cyclic
>    where
>    f1 x = [Node ("A",[(x !! 2),(x !! 1)]),
>            Node ("B",[(x !! 3)]),
>            Node ("C",[(x !! 0), (x !! 3)]),
>            Node ("D",[]) ]
> graph  = head cyclic
```

In both of the above examples, the result is defined in terms of itself: lazy evaluation ensures that the each list item (and hence the indexing into the result) remains unevaluated until it is needed.

The above function application can be parameterised with respect to a list of descriptions of the nodes (where each description is a tuple containing the node value and a list of successor indices), but care is needed. Here is the wrong way to do it:

```
> wrong_gBuild::[([Char],[Int])]->[Graph [Char]]
> wrong_gBuild ndesc
>  = f2 (wrong_gBuild ndesc) ndesc
>    where
>    f2 x [] = [ ]
>    f2 x ((v,ilist):rest)
>     = (Node (v,map ((!!)x) ilist)):(f2 x rest)
```

The function `wrong_gBuild` will give correct results but will not build any cycles (instead, repeated instances of the application `(f2 (wrong_gBuild ndesc) ndesc)` will be created). In order to build a truly cyclic structure it is necessary to give a single name to the result and use that name as the basis for indexing (this is essential to get the sharing behaviour that will form the cycle):

```
> gBuild :: [([Char],[Int])] -> [Graph [Char]]
> gBuild ndesc
>  = result
>    where
>    result = f3 result ndesc
>    f3 x [] = [ ]
>    f3 x ((v,ilist):rest)
>     = (Node (v,map ((!!)x) ilist)):(f3 x rest)
```

```
> cyclic :: [Graph [Char]]
> cyclic = gBuild [("A",[2,1]),("B",[3]),
>                  ("C",[0,3]),("D",[])]
> graph = head cyclic
```

The function `f3` above takes two arguments. The first argument refers to the final result and will be used as a basis for indexing operations embedded in the resultant data structure (thus making the result self-referential): we rely on the laziness of the list constructor to delay the evaluation of the cyclic references into `x` and ensure that such indexing does not cause an infinite evaluation loop. Shared references in the final graph are a natural result of the sharing which is inherent in a lazy evaluator. The second argument is a list of numbers representing indices. The definition for `cyclic` demonstrates how this function may be used to build the data structure in the previous example. Notice in particular that the list of indices could be generated dynamically at run-time.

In order to utilise the above circular programming technique to build a cyclic graph data structure dynamically, we must first establish an appropriate representation as follows:

```
> type Index = Int
> type Adjacency a = (a,[Index])
> data Graph a = EGraph | Node a [Graph a]
```

Thus, we establish the type `Graph a` which is polymorphic in the values held at each graph node; this graph is either empty or is a graph node with a value and a list of successors (each successor is a graph node). We also establish a type `Adjacency a` which is a two-tuple indicating a value and a list of successor indices; we intend that an adjacency list will be of type `[Adjacency a]` and that successor indices will be indices into the adjacency list.

We may now re-write `gBuild` and its auxiliary function `f4` as follows:

```
> gBuild :: [Adjacency a] -> Graph a
> gBuild descriptions
>  = case result of
>         [] -> EGraph
>         _  -> head result
>    where result = f4 result descriptions
>
> f4 :: [Graph a]->[Adjacency a]->[Graph a]
> f4 result [] = []
> f4 result ((value,links): rest)
>  = (Node value (map (gIndex result) links))
>    : f4 result rest
>
> gIndex :: [Graph a] -> Index -> Graph a
> gIndex result n = result !! n
```

2.1.1 Success!

If the above function `gBuild` is applied to the adjacency list `t1` in the following example, it will construct the cyclic graph shown in Figure 1.

```
> t1 :: [Adjacency [Char]]
> t1 = [("A", [1,2]),("B", [0,3]),
>       ("C", [3]),("D", [])]
>
> graph :: Graph [Char]
> graph = gBuild t1
```

It is possible to verify that the above code truly builds a cyclic structure simply by evaluating it! Try to evaluate `graph` and the evaluation mechanism should print the nodes of the graph endlessly, without ever running out of heap space (because exactly the same memory locations are visited over and over again).

We have now achieved our aim: **The function `gBuild` takes the adjacency list and returns a cyclic graph: the adjacency list may be constructed "on the fly", and does not rely on any names in the program!**

3 Graph Traversal

One of the most common operations that is performed on a graph is that of traversal — i.e. performing an action on each node in the graph exactly once. The action may simply be to inspect the value of a node, or it may be to apply a function to the value at that node. There are many operations that are based on traversal — for example, a depth-first traversal can be used to search for membership of a graph (whether a given value occurs in a graph), or to determine the minimum spanning tree of a connected graph, or to "flatten" a graph into its adjacency list representation.

In this section we consider the implementation of a graph flattening function call `gToAdjList`; first we discuss the problem of traversing an immutable graph (together with a solution, which requires another modification to the `Graph` data type), and then we present the code for `gToAdjList`.

3.0.2 Keeping Track of Nodes as they are Visited

The graph construction function `gBuild` returns a truly cyclic data structure. Thus, an attempt to traverse the graph using a naïve recursive technique might loop indefinitely due to cyclic pointers in the graph.

Imperative programs commonly solve this problem by including with each node a bit that is set

when the node is visited. This bit is used by graph traversal functions in order to ensure that nodes are not visited more than once. However, the pure functional paradigm requires that the data structure is immutable and therefore prevents the in-place update of a visited bit.

The functional programming solution to this problem is to augment each node of the graph with a unique tag and to provide the traversal function with a list of tags which have previously been visited (thus, the tag of a node is checked before operating on the value of the node). This is similar to the imperative technique of using a stack to provide a non-recursive graph traversal function.

The functional programming solution (which requires an immutable data structure) may be less efficient than the imperative solution, but it is far more flexible: it permits sharing of the data structure by multiple traversal functions and concurrent traversal by multiple tasks (in a parallel processing system).

We illustrate the technique with an example — deriving a list of all the node values in a graph.

We first expand the `Graph` data type as follows:

```
> type Tag = Int
> data Graph a
>   = EGraph | Node a Tag [Graph a]
```

where the `Tag` data entry is unique with respect to every other node in the same graph.

It is now a simple matter to define a support function to extract the tag from a graph node:

```
> gTagOfNode :: Graph a -> Tag
> gTagOfNode EGraph
>   = error "Can't take tag of empty graph"
> gTagOfNode (Node v t l) = t
```

Modifying the graph building function to add unique tags automatically is similarly straightforward:

```
> gBuild :: [Adjacency a] -> Graph a
> gBuild descriptions
>   = case result of
>           [] -> EGraph
>           _  -> head result
>     where
>     result = f5 result 0 descriptions
>
> f5 :: [Graph a] -> Tag -> [Adjacency a]
>       -> [Graph a]
> f5 res tag [] = []
> f5 res tag ((value,links): rest)
>   = (Node value tag (map (gIndex res) links))
>     : f5 res (tag + 1) rest
```

Example: produce a list of all the node values

The function `gVisit` returns a list of all the node values in the graph *ordered according to the sequence in which they would be encountered during a pre-order traversal of the graph from its root node.* Notice that the function is defined recursively using `foldl` and so `gVisit` must have the type `a -> b -> a`.

```
> gVisit :: Graph a -> [a]
> gVisit EGraph = []
> gVisit g
>   = reverse (fst (gVisit' ([],[]) g))
>
> gVisit' :: ([a],[Tag])
>             -> Graph a -> ([a],[Tag])
> gVisit' (vals,tags) EGraph
>   = (vals, tags)
> gVisit' (vals,tags) (Node v t glist)
>   = if (or (map ((==) t) tags))
>     then (vals, tags) else
>     foldl gVisit' ((v:vals),(t:tags)) glist
```

When visiting the graph a list of tags is constructed to record the nodes already visited. Prior to visiting a node, the list of tags is checked to see if it already contains the tag of the node to be visited, if it does then the node is not revisited. Unfortunately, the list-of-tags technique introduces a searching overhead of $O(n^2)$ time-complexity where n is the number of nodes to be visited. This is expensive in comparison to the constant overhead of checking a single bit; the functional version could quite easily be adapted to use a tree of tags (giving performance of $O(nlog(n))$), and we believe that $O(n)$ traversal is possible. However, this is left to further work.

The function which does most of the work is `gVisit'`, which produces a two-tuple as its result. The first item in this tuple is a list of node values and the second item is a list of the corresponding tags. In this simple version of the function the lists are built in reverse order: `gVisit` then reverses the list of node values.

A simple application of `gVisit` to a graph built from `t1` gives a pre-order list of the nodes in Figure 1:

```
gVisit (gBuild t1)   =>  ["A","C","D","B"]
```

A similar function, which provides useful support for our examples in later sections, is `gtagvisit`:

```
> gtagvisit :: Graph a -> [Tag]
> gtagvisit EGraph = []
> gtagvisit g = reverse (tagvisit [] g)
>
```

```
> tagvisit :: [Tag] -> Graph a -> [Tag]
> tagvisit tags EGraph
>     = tags
> tagvisit tags (Node v t glist)
>     = if (or (map ((==) t) tags))
>       then tags
>       else foldl tagvisit (t:tags) glist
```

3.0.3 Flattening a graph

The basic structure for building a graph dynamically is the adjacency list (as previously demonstrated). Once a cyclic structure has been constructed, it is possible to define many useful functions which operate on the cyclic structure. In the next section we will address the problem of how to write such functions directly; however, in this section we note that any graph operation which has a direct analogy with an operation on a list can be implemented by translating the graph into its adjacency list, operating on the adjacency list with the analogue of the required graph operation, and then translating back into a graph. For example, to map a function over all the values in a graph:

```
> gmap :: (a -> b) -> Graph a -> Graph b
> gmap f g
>  = (gBuild . (map h). gToAdjList) g
>    where
>    h (val, indices) = (f val, indices)
```

The above example assumes the existence of a function called `gToAdjList`, which traverses a cyclic graph and returns the corresponding adjacency list for that graph. We may think of this as "flattening" the graph.

We define the graph flattening function to provide the adjacency list of *all reachable nodes from the start node*. Thus, if the graph is not rooted, or if the start node is not the root node, then this function will not provide an adjacency list representation of the whole graph!

In order to implement the function `gToAdjList` we utilise the tagging technique demonstrated in the previous section:

```
> gToAdjList :: Graph a -> [Adjacency a]
> gToAdjList EGraph = []
> gToAdjList g
>  = reverse adj
>    where
>    (_,_,adj) = gToAdjList' (indexl,[],[]) g
>    indexl = zip (gtagvisit g) [0..]
>
> gToAdjList'
>  :: ([(Tag,Int)], [Tag], [Adjacency a])
>     -> Graph a
>     -> ([(Tag,Int)], [Tag], [Adjacency a])
> gToAdjList' arg@(indexl,tags,adj) EGraph
>  = arg
> gToAdjList' arg@(indexl,tags,adj)
>              g@(Node v t glist)
>  = if (or (map ((==) t) tags)) then arg
>    else
>    foldl gToAdjList' (new_adjacency arg) glist
>    where
>    new_adjacency (indexl,tags,adj)
>     = (indexl, (t:tags),
>        ((v , adjacency_vals) : adj))
>    adjacency_vals
>     = (map (index_of indexl) .
>        map gTagOfNode) glist
>
> index_of :: [(Tag,Int)] -> Tag -> Int
> index_of ilist tag
>  = head [ i | (t,i) <- ilist, t == tag]
```

The above function `gToAdjList` calls a subsidiary function `gToAdjList'`; it is the subsidiary function which does most of the work.

`gToAdjList'` has been designed very carefully to have the type `(a -> b -> a)` so that `foldl` can be used on the list of successors. It takes as its first argument a three-tuple containing:

1. A list of all the tags in the graph, together with their indices. This list never changes — it is created at the beginning by `gToAdjList` and is then passed down to `gToAdjList'`. The indices are calculated separately in order to allow the tagging mechanism some freedom in allocating tags (e.g. the tags may not start at zero).

2. A list of tags that have already been visited by previous calls to `gToAdjList'`.

3. An accumulating parameter containing the current Adjacency List.

The second argument to `gToAdjList'` is the graph, and the result is a three-tuple containing (as its third element) the completed Adjacency List.

4 Direct Manipulation of Graphs

Translation both to and from an adjacency list is straightforward, and any operation which we wish to perform on a graph can be performed instead on its adjacency list. However, the overheads involved in such a translation might outweigh the performance advantages of using a cyclic structure. It is therefore important to show that it is possible to manipulate a cyclic graph structure *directly*.

Direct manipulation of a cyclic data structure is unavoidably complex. The following discussion demonstrates the coding technique for a function

called `gMap`, which maps a function over the value held in every node of a graph (this is analogous to the `map` function over lists). In order to illustrate a common difficulty with circular programming, first we give the *wrong* way to code this function and then we present the correct code.

The wrong way

The function `gMap` uses a subsidiary function `gMap'`; this latter function avoids creating an intermediate adjacency list and creates a new graph structure "on the fly". It only constructs two intermediate lists; a list which maps tags to indices and a list of tags which have already been visited.

As with the function `gBuild`, a circular programming style has been used; the name `result` appears inside `gMap` both as a definition and as an argument to the function which defines it. The code for `gMap'` is complex — it is responsible for constructing `result`, but it may embed references to the final `result` inside the expression which it is constructing (as long as the evaluation of these references are delayed by a constructor — for example, either `Node` or the standard list constructor).

```
> gMap :: (a -> b) -> Graph a -> Graph b
> gMap f EGraph = EGraph
> gMap f g
>  = head (reverse result)
>    where
>    (_, _, _, _, result)
>     = gMap' (f, result, indexl, [], []) g
>    indexl= zip (gtagvisit g) [0..]
>
> type MapState a b = ((a -> b), [Graph b],
>                      [(Tag,Int)],[Tag],
>                      [Graph b])
>
> -- MapState consists of
> --      * a function to map over items
> --      * an indexing reference point
> --      * a list of tags and indices
> --      * a list of visited tags
> --      * the current reversed list
> --        of new nodes
>
> gMap'::MapState a b->Graph a->MapState a b
> gMap' arg EGraph
>       = arg
> gMap' arg@(f,ref,indexl,tags,result)
>         g@(Node v t glist)
>  = if (or (map ((==) t) tags)) then arg
>    else foldl gMap' newarg glist
>    where
>    newarg = (f, ref, indexl, (t:tags),
>              (newnode : result))
>    newnode
>     = Node (f v) t
>       [gIndex ref i | i <- index_vals]
>    index_vals
>        = (map (index_of indexl) .
>           map gTagOfNode) glist
```

In the above example, the self reference occurs in the local definition block for the function `gMap`. In that block, the first local definition is:

```
> (_, _, _, _, result)
>  = gMap' (f, result, indexl, [], []) g
```

Thus, the reference point used for indexing is the identifier `result`. However, the actual graph is created by indexing into `(reverse result)` because `result` is built backwards (as can be seen by inspection of the function `gMap'`).

Unfortunately, there is an error in the design of this function. The indices in `indexl` refer to the original graph, but they are used to index into the reversed list of graph cells (`result`). Thus, the wrong reference point is being used for indexing and the graph will be constructed incorrectly, with incorrect edges.

The right way

It is important that the reference point that is used for indexing is the list of graph nodes which is the reverse of `result`. This can be achieved by substituting the following definition for `gMap`:

```
> gMap :: (a -> b) -> Graph a -> Graph b
> gMap f EGraph = EGraph
> gMap f g
>  =  head ref
>     where
>     (_, _, _, _, result)
>            = gMap' (f, ref, indexl, [], []) g
>     indexl = zip (gtagvisit g) [0..]
>     ref    = reverse result
```

4.1 Cyclic programming and reversed lists

The above code builds two lists: first the list `result`, and then the reversed copy of that list, `ref`. This is both inefficient and confusing. It is not clear from inspecting the expression `(newnode :  result)` that the list is being built in reverse order and therefore *must* be reversed before being used. It is easy to forget to use `reverse` and it is easy to attempt future modifications without realising the reversed nature of the list.

Reverse-constructed lists arise frequently when using cyclic programming techniques, where the first element of a list is found at the end of a recursion. We have therefore adopted a functional representation for reversed lists. We illustrate this

new approach by employing three subsidiary functions which manipulate reversed lists. First, we define the type of a reversed list using a function representation as:

```
> type RList a = [a] -> [a]
```

Next we define the following three functions:

```
> -- rlistToList converts from a reversed
> -- list into a real list
>
> rlistToList :: RList a -> [a]
> rlistToList r = r []
>
> -- rnil is the empty reversed list
>
> rnil :: RList a
> rnil x = x
>
> -- rcons is the reverse list cons function
>
> rcons :: a -> RList a -> RList a
> rcons v g init = g (v : init)
```

Thus, the list

```
(1 : (2 : (3 : [])))
```

can be constructed as:

```
rlistToList (rcons 3 (rcons 2 (rcons 1 rnil)))
```

The following code reworks the `gMap` function using this new representation (note the use of the Haskell infix notation `` `rcons` ``):

```
>
> gMap :: (a -> b) -> Graph a -> Graph b
> gMap f EGraph = EGraph
> gMap f g
>  = head ref
>    where
>    (_, _, _, _, result)
>          = gMap' (f, ref, indexl, [], rnil) g
>    indexl= zip (gtagvisit g) [0..]
>    ref   = rlistToList result
>
> type MapStateR a b
>       = ((a -> b), [Graph b], [(Tag,Int)],
>          [Tag], RList (Graph b))
>
> gMap'::MapStateR a b->Graph a->MapStateR a b
> gMap' arg EGraph
>  = arg
> gMap' arg@(f,ref,indexl,tags,result)
>         g@(Node v t glist)
>  = if (or (map ((==) t) tags)) then arg
>    else foldl gMap' newarg glist
>    where
>    newarg= (f, ref, indexl, t:tags,
>               newnode `rcons` result)
>    newnode
>          = Node
>            (f v) t
>            [gIndex ref i | i <- index_vals]
>    index_vals
>          = (map (index_of indexl) .
>             map gTagOfNode) glist
```

The differences between this new version and the previous version of the code are:

1. The function `gMap` uses `rlistToList` instead of `reverse` and `rnil` instead of `[]`.
2. The function `gMap'` has a slightly modified type (to use `RList (Graph b)` instead of `[Graph b]`).
3. The function `gMap'` uses `(newnode `rcons` result)` instead of `(newnode : result)`.

The construction of the old list `result` has been replaced by the construction of nested closures which are finally evaluated (by `rlistToList` to produce the final list. The closures provide the items to the list in the reverse order: when evaluated, the closures build the list in the correct order. The self-referential indices will index directly into the result of the delayed closures.

This new technique provides a new way to express the construction of a reversed list; the fact that an explicit function is used instead of the built-in list constructor (`:`) means that the code displays better self-documentation — it is evident that the data structure is being constructed in reverse order. This is an important programming technique that guards against inappropriate code modifications in the future.

The above functional representation is also slightly more efficient than building a list then reversing it. Assuming an appropriate (linear) definition for `reverse` is employed, there is a constant-factor performance difference between (i) building a list then reversing it, and (ii) using delayed evaluation of closures. However, that constant factor can be significant for large lists; with the two-list technique, the time required to achieve the final result is roughly proportional to `3*n` (where `n` is the number of items in the list), whereas with the functional representation the result is achieved in time roughly proportional to `2*n`.

5 Summary

We have refuted the myth that "functional languages cannot construct arbitrary cyclic structures at run-time". We have summarised previous work in this area; we have demonstrated the simplicity of

creating static cyclic structures and we have then built on previous work by presenting methods for the dynamic construction of complex cyclic structures at run-time. We have provided practical advice on the traversal and direct manipulation of such structures using circular programming techniques, including the use of a new construction for reversed lists which improves performance and provides better documented (and therefore more robust) code.

6 Further Work

This paper reports the foundation of work that is currently proceeding as follows:

1. We are improving the performance of the graph traversal methods; we aim for a linear time depth-first traversal.

2. The circular programming style used in this paper is necessary in order to create a cyclic structure dynamically, but is admittedly somewhat opaque. Bird [Bir84] has shown how circular programs may be derived from their non-circular counterparts via program transformation; this might be a route to simplifying direct manipulation of cyclic structures, but it is not immediately clear how to provide a non-circular version.

   We are investigating ways by which we may make circular programming easier for mortal programmers. In particular, we are searching for a way to separate the issues of traversal, calculation and construction.

3. We are writing further graph manipulation functions for efficient addition and deletion of nodes and edges; this is not as straightforward as it sounds.

4. In order to relieve programmers from the substantial burden of implementing circular code, we aim to package the cyclic data structure and its associated operations into an abstract data type which will be made available to other researchers.

   In particular, we are investigating the use of a graph monad to guide the selection of ADT primitives and the establishment of a graph algebra.

5. We are extending our graph metaphor to consider unconnected components, weighted edges, undirected graphs, and graphs with no (or multiple) roots.

6. We are interested in the work of Klarlund [KS93], who suggests that type systems might in future be extended to encompass general-purpose "graph-shaped" user-defined recursive types which contain both values and routing information, and he provides a logic for expressing properties of such types. So far, our aim has been to use existing functional languages, so that our results will be immediately useful to a large number of programmers. However, we will be keen to experiment with any language that implements an experimental version of Klarlund's graph types.

Our work in this area was initially motivated by Clayman's PhD research [Cla94] over two years ago; our research is showing promise in at least three of the above areas and we hope very soon to report our progress and achievements in subsequent papers.

References

[All89] L. Allison. Circular programs and self-referential structures. *Software—Practice and Experience*, 19(2):99–109, 1989.

[Bir84] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[BY90] F.W. Burton and H-K Yang. Manipulating multilinked data structures in a pure functional language. *Software—Practice and Experience*, 20(11):1167–1185, 1990.

[Cla94] S.Clayman. Developing and Measuring Parallel Rule-Based Systems in a Functional Programming Environment. *PhD Thesis*, Dept. Computer Science, University College London, 1994.

[Hol91] I. Holyer. *Functional Programming with Miranda*. Pitman, 1991.

[HPJW+91] P. Hudak, S.L. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M.M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. *Report on the Programming Language Haskell. A Non-strict, Purely Functional Language.*, version 1.1 edition, August 1991.

[KL94] David J. King and John Launchbury. Lazy depth-first search and linear graph algorithms in Haskell. Research Report FP-94-06, Computing Science Department, University of Glasgow, March 1994.

[KS93] N. Klarlund and M.I. Schwartzbach. Graph types. *Proceedings of the 20th Symposium on Principles of Programming Languages*, pages 196–205, 1993.

[Rea89] C. Reade. *Elements of Functional Programming*. Addison Wesley, 1989.

[USE] Archives, `comp.lang.functional`, USENET news group, 1992–95.