The Monad.Reader/Issue5/Practical Graph Handling

From HaskellWiki

< The Monad.Reader | Issue5

This article needs reformatting! Please help tidy it up.--WouterSwierstra 14:26, 9 May 2008 (UTC)

Contents

1 A Practical Approach to Graph Manipulation

by JeanPhilippeBernardy for The Monad.Reader Issue 5 BR *Date(2005-07-08T20:48:51Z)*

Abstract.

Tree-based data structures are easy to deal with in Haskell. However, working with

graph-like structures in practice is much less obvious. In this article I present a solution that has worked for me in many cases.

1.1 Introduction

I always found that dealing with graphs in Haskell is a tricky subject. Even something like a implementing a depth-first search, which is trivially achieved in an imperative language, deserves an article on its own for Haskell #dfs 4. A PhD thesis has been written on the subject of graphs and functional programming #king-thesis 2, and it seems that it still doesn't exhaust the design-space: radically different ideas have been proposed afterwards #induct 3.

In this article I'll present (a simplified version of) a solution that I think deserves more coverage #cycle-therapy 1. The idea is to abstract graph manipulation by anamorphisms and catamorphisms. This approach features "separation of concerns" and "composability", hence it can be readily applied to practical problems.

- Section 2 shows how anamorphisms and catamorphisms can be generalised to graphs.
- Section 3 details the data structures used to represent graphs
- Section 4 discusses various problems where cata/anamorphisms can be applied
- Section 5 gives a sample implementation for the catamorphism and anamorphism
- Section 6 concludes

1.1.1 Nota

This article has been generated from a literate haskell source. So, although the text of this wiki page will not compile, all the examples are real and run. The source can be accessed here: attachment:PracticalGraphHandling.lhs

We will assume you know the hierarchical libraries. Refer to http://haskell.org/ghc/docs/latest/html/libraries/index.html in case of doubt.

1.2 Origami with Graphs

1.2.1 Fold & Unfold (the big deal)

Most of you probably know what a "fold" (also known as catamorphism) is. For those who don't, intuitively, it's an higher-order operation that reduces a complex structure to a single value. It applies a function given as parameter to each node, propagating the results up to the root. This is a highly imprecise definition, for more details please read #bananas-lenses 5.

For example, the fold operation on lists can be typed as follows:

```
foldr :: (a -> b -> b) -> -- ^ operation to apply
```

```
        b ->            -- ^ initial value
        [a] ->          -- ^ input list
        b               -- ^ result
```

Conversely, "unfold" builds a complex structure out of a building function, applying it iteratively.

```
unfoldr :: (b -> Maybe (a, b)) ->  -- ^ building function (Nothing => end of list)
           b ->                    -- ^ seed value
           [a]                     -- ^ result
```

The second argument is the initial value from which the whole resulting list will be derived, by applying the 1st argument. In the following we'll refer to it as the "seed".

The catamorphism/anamorphism abstractions have proven to be very useful in practise. They're ubiquitous to any haskell programming, either explicitly, or implicitly (hidden in higher-level operations). In this article I'll show how those abstractions can be generalised to graph structures, and argue that they are equally useful in this case.

The rest of the article assumes the reader is fairly familiar with fold and unfold. Fortunately there are many articles on the subject. For example you can refer to #bananas-lenses 5 if you ever feel uncomfortable.

1.2.2 Generalisation

Let's examine how fold/unfold can be generalized for graphs. Since we are working on graphs instead of lists, we must account for

```
1. Any number of children for a node;
2. "Backwards" arcs (cycles);
3. Labelled edges.
```

The most relevant point being 2, of course.

1.2.2.1 unfoldG

From the above, we can deduce that the type of unfoldG will be:

```
unfoldG :: (Ord s) => (s -> (n, [(e, s)])) -> s -> (Vertex, LabGraph n e)
unfoldG f r = (r', res)
  where ([r'], res) = unfoldGMany f [r]
```

where `s` is the seed type, `n` is the node labels, `e` the edges labels.

The `Ord s` constraint reflects point 2 above. It is needed because the unfoldG function must record every seed value encountered. Whenever a seed is seen a second time, `unfoldG` will recognize it and create a "backward arc". We use `Ord` instead of `Eq` because a mere equality test rules out using `Data.Map`.

The attentive reader will note that we return an additional Vertex value. This is needed to identifty which node the root seed corresponds to.

In order to get an intuitive feeling of how `unfoldG` works, let's examine a simple example.

```
gr1 :: LabGraph Int Char
(_,gr1) = unfoldG gen (0::Int)
    where gen x = (x,[('a',(x+1) `mod` 10), ('b', (x+2) `mod` 10)])
```

`gr1` being defined as above, its structure is:

attachment:gr1.png

Because we might want to build a graph from a set of seeds instead of a single one, we will also need the following function:

```
unfoldGMany :: (Ord s) => (s -> (n, [(e, s)])) -> [s] -> ([Vertex], LabGraph n e)
unfoldGMany f roots = runST ( unfoldGManyST f roots ) -- detailed later
```

`unfoldG`, alone, is already very a practical tool, because it lets you reify a function (`a -> a`) graph. It then can be examined, processed, etc. whereas the function can only be evaluated.

1.2.2.2 foldG

On a graph, the catamorphism (fold) type will become:

```
foldG :: (Eq r) => r -> (Vertex -> [(e, r)] -> r) -> Graph e -> Vertex -> r
foldG i f g v = foldGAll i f g ! v
```

As for `unfoldG`, the `foldG` function must include a special mechanism to handle cycles. The idea is to apply the operation iteratively until the result converges. It's the purpose of the first parameter is to "bootstrap" the process: it will be used as an initial value.

Thus, `foldG i f g ! v` will iteratively apply `f` on nodes of graph `g`, using `i` as "bottom" value. It will return the value computed at vertex `v`. Of course, this will work only if `f` is well-behaved: it must converge at some point. I won't dwelve in to the theoretical details here, see #cycle-therapy 1 for a formal explanation.

Notice that `foldG` can work on a graph without node labels. If the parameter function needs to access node labels, it can do so without `foldG` needing to know.

It's also worth noticing that, in our implementation, the information will be propagated in the reverse direction of arcs.

It's very common to need the result value for each vertex, hence the need for :

```
foldGAll :: (Eq r) => r -> (Vertex -> [(e, r)] -> r) -> Graph e -> Table r
```

The implementation of these functions doesn't matter much. The point of the article is not how these can be implemented, but how they can be used for daily programming tasks. For completeness though, we'll provide a sample implemenation at the end of the article.

1.3 Data Structure & Accessors

Without further ado, let's define the data structures we'll work on.

```
type Vertex = Int
type Table a = Array Vertex a
type Graph e = Table [(e, Vertex)]
type Bounds  = (Vertex, Vertex)
type Edge e = (Vertex, e, Vertex)
```

A graph is a mere adjacency list table, tagged with edge labels.

The above structure lacks labels for nodes. This is easily fixed by adding a labeling (or coloring) function.

```
type Labeling a = Vertex -> a
data LabGraph n e = LabGraph (Graph e) (Labeling n)

vertices (LabGraph gr _) = indices gr

labels (LabGraph gr l) = map l (indices gr)
```

The above departs slightly from what's prescribed in #cycle-therapy 1. Instead of a *true graph* built by knot-tying, we chose to use an `Array` with integers as explicit vertex references. This is closely follows Data.Graph in the hierarchical libraries, the only difference being that we have labelled edges.

Not only this is simpler, but it has the advantage that we can reuse most of the algorithms from Data.Graph with only minor changes:

```
-- | Build a graph from a list of edges.
buildG :: Bounds -> [Edge e] -> Graph e
buildG bounds0 edges0 = accumArray (flip (:)) [] bounds0 [(v, (l,w)) | (v,l,w) <- edges0]

-- | The graph obtained by reversing all edges.
transposeG  :: Graph e -> Graph e
transposeG g = buildG (bounds g) (reverseE g)

reverseE    :: Graph e -> [Edge e]
reverseE g   = [ (w, l, v) | (v, l, w) <- edges g ]
```

However, as previously said, we'll try to abstract away from the details of the structure. This is not always possible, but in such cases, I believe the array representation to be a good choice, because it's easy to work with. If anything, one can readily use all the standard array functions.

For example, here's the function to output a graph as a GraphViz file:

```
showGraphViz (LabGraph gr lab)  =
    "digraph name {\n" ++
    "rankdir=LR;\n" ++
    (concatMap showNode $ indices gr) ++
    (concatMap showEdge $ edges gr) ++
    "}\n"
    where showEdge (from, t, to) = show from ++ " -> " ++ show to ++
                                   " [label = \"" ++ show t ++ "\"];\n"
          showNode v = show v ++ " [label = " ++ (show $ lab v) ++ "];\n"

edges :: Graph e -> [Edge e]
edges g = [ (v, l, w) | v <- indices g, (l, w) <- g!v ]
```

1.4 Applications

I'll now enumerate a few problems where the "origami" approach can be applied successfully.

1.4.1 Closure

A simple application (special case) of "unfoldG" the computation of the transitive closure of a non-deterministic function.

```
closure :: Ord a => (a -> [a]) -> (a -> [a])
closure f i = labels $ snd $ unfoldG f' i
    where f' x = (x, [((), fx) | fx <- f x])
```

In this context, "non deterministic" means that it yields many values, as a list. As noted before, this will work only when everything remains finite in size.

For example, if we define

```
interleave (x1:x2:xs) = (x1:x2:xs) : (map (x2:) (interleave (x1:xs)))
interleave xs = [xs]

interleave "abcd" ==> ["abcd","bacd","bcad","bcda"]
```

a very bad way to compute the permutations of list can be

```
permutations = closure interleave
```

```
permutations "abcd" ==> ["abcd","bacd","acbd","cabd","abdc","badc",
                         "adbc","dabc","dbac","bdac","dacb","adcb",
                         "dcab","cdab","cadb","acdb","cdba","dcba",
                         "cbda","bcda","bdca","dbca","bcad","cbad"]
```

But sometimes the function to 'close' is more complicated than `interleave` and then `closure` becomes really useful.

1.4.2 Shortest Path

Let us now examine the toy problem of finding the distance to a given node from all the other nodes of the graph. Most readers probably know the Dijkstra's algorithm to compute the solution to the problem. We will not try to reproduce it here, instead we will define the computation in terms of `foldG`.

Here it goes:

```
-- | Compute the distance to v for every vertex of gr.
distsTo :: Vertex -> Graph Float -> Table Float
distsTo v gr = foldGAll infinite distance gr
    where infinite = 10000000 -- well, you get the idea
          distance v' neighbours
              | v == v' = 0
              | otherwise = minimum [distV+arcWeight | (distV, arcWeight) <- neighbours]
```

So clear that it barely needs to be explained. :) Just notice how the minimize function assumes that the distance is already computed for all its neighbours. This works because `foldG` will iterate until it finds the fixed point.

On this simple graph,

```
grDist = buildG (1,5) [(1,5.0,2), (2,5.0,3), (2,7.0,4), (3,5.0,4), (4,5.0,5), (4,3.0,1)]
```

attachment:grdist.png

the result of

```
dists = distsTo 5 grDist
```

is

attachment:grdist2.png

(labeling each node with the its result, ie. distance to vertex `5`)

1.4.3 Finite Automaton

Finite automatons are basically graphs, so let's see how we can apply the framework to their analysis.

First, let's define an automaton. For our purposes, it is a graph of states/transitions, some of the states being marked as initial or final.

```
type Automaton t = (Vertex, Graph t, Set Vertex) -- ^ Initial, transitions, finals
```

For starters, here is how the `showGraphViz` function can be applied to automaton display:

```
automatonToGraphviz (i, gr, fs) = showGraphViz (LabGraph gr lab)
    where lab :: Labeling String
          lab v = (if v == i then (">"++) else id) $
                  (if v `Set.member` fs then (++"|") else id) []
```

Nothing ground breaking. We only label the nodes accordingly to their final or initial status.

```
aut1 = (1, buildG (1,3) [(1,'a',2),(2,'a',2),(2,'b',2),(2,'c',3),(1,'a',3)], Set.fromList [3])
```

attachment:aut1.png

A more interesting example is how to transform a non-deterministic automaton to an equivalent deterministic one. The underlying idea is that non-deterministic execution of the automaton is equivalent to deterministic execution on all possible transitions at once. Refer to #hop&ull 6 for details. This is relatively easily done using `unfoldG`.

```
simpleGenerator f x = (x, f x)

nfaToDfa :: Ord e => Automaton e -> Automaton e
nfaToDfa (initial1, aut1, finals1) = (initial2, aut2, finals2)
    where (initial2, LabGraph aut2 mapping) = unfoldG (simpleGenerator build) seed
          seed = Set.singleton initial1
          build state = Map.toList $ Map.fromListWith Set.union $ map lift $
                        concat $ map (aut1 !) $ Set.toList state
          lift (t,s) = (t, Set.singleton s)
          isFinal = setAny (`Set.member` finals1) . mapping
          finals2 = Set.fromList $ filter isFinal $ indices aut2
          setAny f = any f . Set.toList
```

The 'build' function is the tricky part. Yet, it's not as complicated as it seems: all it does is

1. Find all reachable nodes from a set of nodes;
2. Classify them by transition label
3. Build target state-sets accordingly.

```
aut2 = nfaToDfa aut1
```

attachment:aut2.png

Another thing we possibly wish to compute is the set of strings accepted by the automaton, (aka. the language it defines). Most of the time this will be infinite, so we will limit ourselves to strings of length `n` maximum. We need finiteness because otherwise `foldG` would not find a fixed point: string sets would keep growing idefinitely.

```
accepted n (initial1, aut1, finals1) = Set.unions [resultTable ! v | v <- Set.toList finals1]
                                       -- gather what's accepted at all final states
    where resultTable = foldGAll Set.empty step (transposeG aut1)
          step v trans = Set.unions ((if v == initial1 then Set.singleton [] else Set.empty) :
                                     [Set.map ((++[t]) . take (n-1) ) s | (t,s) <- trans])
```

Notice that we need to reverse the graph arcs, otherwise the information propagates in the wrong direction.

With

```
accAut1 = accepted 4 aut1
accAut2 = accepted 4 aut2
```

we have

```
accAut1 == accAut2 == {"a","aaac","aabc","aac","abac","abbc","abc","ac"}
```

1.4.4 LALR Automaton

Another area where I applied graph (un)folding is LALR(1) parser generation. The detailed code depends on just too many things to fit in this paper, thus we will only sketch how pieces fit together. Also, since a course on parsing is clearly beyond the scope of this article, please refer to local copy of the dragon book #dragon 7 for details on the method.

In the process of generating tables for a LALR automaton, there are three steps amenable to implementation by `foldG` and `unfoldG`.

1. Construction of the closure of a LR-items kernel. This one is very similar to the `closure` function described above, except that we don't discard the graph structure. It'll be of use for step 3. 2. LR(0) automaton generation. Then again a use for `unfoldG`. 3. Propagation of the lookahead. It is a fold over the whole graph of LR-items, basically using set union as coalescing operation. It is very similar to computation of acceptable strings above.

1.5 Implementation

1.5.1 UnfoldG

For the sake of completeness, here's how to implement the `unfoldG` function.

The algorithm effectively a depth-first search, written in imperative style. The only difference is that the search graph is remembered and returned as result.

```
unfoldGManyST :: (Ord a) => (a -> (c, [(b, a)]))
              -> [a] -> ST s ([Vertex], LabGraph c b)
```

```
unfoldGManyST gen seeds =
    do mtab <- newSTRef (Map.empty)
       allNodes <- newSTRef []
       vertexRef <- newSTRef firstId
       let allocVertex =
               do vertex <- readSTRef vertexRef
                  writeSTRef vertexRef (vertex + 1)
                  return vertex
       let cyc src =
            do probe <- memTabFind mtab src
               case probe of
                  Just result -> return result
                  Nothing -> do
                    v <- allocVertex
                    memTabBind src v mtab
                    let (lab, deps) = gen src
                    ws <- mapM (cyc . snd) deps
                    let res = (v, lab, [(fst d, w) | d <- deps | w <- ws])
                    modifySTRef allNodes (res:)
                    return v
       mapM_ cyc seeds
       list <- readSTRef allNodes
       seedsResult <- (return . map fromJust) =<< mapM (memTabFind mtab) seeds
       lastId <- readSTRef vertexRef
       let cycamore = array (firstId, lastId-1) [(i, k) | (i, a, k) <- list]
       let labels = array (firstId, lastId-1) [(i, a) | (i, a, k) <- list]
       return (seedsResult, LabGraph cycamore (labels !))
  where firstId = 0::Vertex
        memTabFind mt key = return . Map.lookup key =<< readSTRef mt
        memTabBind key val mt = modifySTRef mt (Map.insert key val)
```

Notice how every time a seed is encountered, its corresponding vertex number stored. Whenever the seed is encountered again, the stored is just returned.

1.5.2 FoldG

```
foldGAllImplementation bot f gr = finalTbl
    where finalTbl = fixedPoint updateTbl initialTbl
          initialTbl = listArray bnds (replicate (rangeSize bnds) bot)

          fixedPoint f x = fp x
              where fp z = if z == z' then z else fp z'
                        where z' = f z
          updateTbl tbl = listArray bnds $ map recompute $ indices gr
              where recompute v = f v [(b, tbl!k) | (b, k) <- gr!v]
          bnds = bounds gr
```

The proposed implementation for foldG is rather bold. It just applies the coalescing function repeatedly till it converges.

While this is not an ideal situation, it's perfectly suited for a first-trial implementation, or when performance is not crucial.

If execution time becomes critical, then more specialized versions can be crafted. In the case of the shortest path algorithm, for example, it could take advantage of the nice properties of the coalescing function to use a priority queue and greedily find the fixed point. This would restore the optimal O(n * log n) complexity.

1.6 Conclusion

The approach presented may not be excellent for controlling details of implementation and tuning run-time performance, but I think that's not the point of haskell programming anyway. On the other hand, it is very good for quick implementation of a large range of graph algorithms. The fact that it's mostly based on a generalisation on fold and unfold should appeal to haskell programmers.

1.7 References

- Anchor(cycle-therapy) [1] *Cycle Therapy: A Prescription for Fold and Unfold on Regular Trees*, F. Turbak and J.B. Wells, http://cs.wellesley.edu/~fturbak/pubs/ppdp01.pdf
- Anchor(king-thesis) [2] *Functional Programming and Graph Algorithms*, D. J. King, http://www.macs.hw.ac.uk/~gnik/publications
- Anchor(induct) [3] *Inductive Graphs and Functional Graph Algorithms*, Martin Erwig, http://web.engr.oregonstate.edu/~erwig/papers/abstracts.html
- Anchor(dfs) [4] , D. J. King and John Launchbury, http://www.cse.ogi.edu/~jl/Papers/dfs.ps
- Anchor(bananas-lenses) [5] *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*, Erik Meijer, Maarten Fokkinga, Ross Paterson. http://citeseer.ist.psu.edu/meijer91functional.html
- Anchor(hop&ull) [6] *Introduction to Automata Theory, Languages, and Computation*, JE Hopcroft, and JD Ullman, http://www-db.stanford.edu/~ullman/ialc.html
- Anchor(dragon) [7] *Compilers: Principles, Techniques and Tools*, Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. (Addison-Wesley 1986; ISBN 0-201-10088-6)

CategoryArticle