Graph Algorithms with a Functional Flavour

John Launchbury

Oregon Graduate Institute
jl@cse.ogi.edu

Abstract. Graph algorithms have long been a challenge to program in a pure functional language. Previous attempts have either tended to be unreadable, or have failed to achieve standard asymptotic complexity measures. We explore a number of graph search algorithms in which we achieve standard complexities, while significantly improving upon traditional imperative presentations. In particular, we construct the algorithms from reusable components, so providing a greater level of modularity than is typical elsewhere. Furthermore, we provide examples of correctness proofs which are quite different from traditional proofs, largely because they are not based upon reasoning about the dynamic *process* of graph traversal, but rather reason about a static *value*.

1 Introduction

Graph algorithms do not have a particularly auspicious history in purely functional languages. It has not been at all clear how to express such algorithms without using side effects to achieve efficiency, and lazy languages by their nature have had to prohibit side-effects. So, for example, many texts provide implementations of search algorithms which are quadratic in the size of the graph (see [Pau91], [Hol91], or [Har93]), compared with the standard linear implementations given for imperative languages (see [Man89], or [CLR90]). What is more, very little seems to have been gained by expressing such algorithms functionally—the presentation is sometimes *worse* than the traditional imperative presentation!

In these notes we will explore various aspects of expressing graph algorithms functionally with one overriding concern—we refuse to give ground on asymptotic complexity. The algorithms we present have identical asymptotic complexity to the standard presentation.

Our emphasis is on depth-first search algorithms. The importance of depth-first search for graph algorithms was established twenty years ago by Tarjan and Hopcroft [Tar72, HT73] in their seminal work. They demonstrated how depth-first search could be used to construct a variety of efficient graph algorithms. In practice, this is done by embedding code-fragments necessary for a particular algorithm into a depth-first search procedure skeleton in order to compute relevant information while the search proceeds. While this is quite elegant it has a number of drawbacks. Firstly, the depth-first search code becomes intertwined with the code for the particular algorithm, resulting in monolithic programs. The code is not built by re-use, and there is no separation between logically distinct phases. Secondly, in order to reason about such depth-first search algorithms we have to reason about a dynamic *process*—what happens and when—and such reasoning is complex.

Occasionally, the ***depth-first forest*** is introduced in order to provide a ***static*** value to aid reasoning. We build on this idea. If having an explicit depth-first forest is good for reasoning then, so long as the overheads are not unacceptable, it is good for programming. In this paper, we present a wide variety of depth-first search algorithms as combinations of standard components, passing explicit intermediate values from one to the other. The result is quite different from traditional presentations of these algorithms, and we obtain a greater degree of modularity than is usually seen.

Of course, the idea of splitting algorithms into many separate phases connected by intermediate data structures is not new. To some extent it occurs in all programming paradigms, and is especially common in functional languages. What is new, however, is applying the idea to graph algorithms. The challenge is to find a sufficiently flexible intermediate value which allows a wide variety of algorithms to be expressed in terms of it.

In our work there is one place where we do need to use destructive update in order to gain the same complexity (within a constant factor) as imperative graph algorithms. We make use of recent advances in lazy functional languages which use monads to provide updatable state, as implemented within the Glasgow Haskell compiler. The compiler provides extensions to the language Haskell providing updatable arrays, and allows these state-based actions to be encapsulated so that their external behaviour is purely functional (a summary of these extensions is given in the Appendix). Consequently we obtain linear algorithms and yet retain the ability to perform purely functional reasoning on all but one fixed and reusable component.

Most of the methods in this paper apply equally to strict and lazy languages. The exception is in the case when depth-first search is being used for a true *search* rather than for a complete *traversal* of the graph. In this case, the co-routining behaviour of lazy evaluation allows the search to abort early without needing to add additional mechanisms like exceptions.

2 Representing graphs

There are at least three rather distinct ways of representing (directed) graphs in a language like Haskell. For example:

1. as an element of an algebraic datatype containing cycles constructed using laziness;
2. as an (immutable) array of edges; or
3. as explicit mutable nodes in the heap (working within the state monad).

The first of these is the most "functional" in its flavour, but suffers from two serious defects. First, cyclic structures are isomorphic to their unrolled counterparts[1], but graphs are not isomorphic to their unrolling. Each node of the graph could be tagged explicitly, of course. But this still leaves us with the second defect: cyclic structures are hard to preserve and modify. Hughes proposed lazy memo functions as a means of preserving cycles [Hug85], but these have not been adopted into any of the major

[1] In languages like Scheme which have object identity this is not the case, but this is at the (semantic) cost of tagging each cons-cell with a unique identifier.

lazy languages. And without them, something as simple as mapping a function over the graph will cause it to unfurl. In addition within any cycle, the graph structure is monolithic: any change to a part of it will force the whole cycle to be rebuilt. An exception to this may occur if the compiler manages to deduce some sort of linearity property which allows update-in-place, but then the efficiency of the algorithm may become a very delicate matter.

The second representation method lies somewhere on the border between "functional" and "imperative". Using arrays to store edge lists is a common practice in the imperative world, but the only array facility used is constant-time read-access (if the graph is static), so purely functional arrays are appropriate. This is the method we will focus on.

The final method is highly imperative, and is most appropriate when it is vital to be able to change the graph in-place (i.e. when a local modification should be globally visible).

2.1 Adjacency Lists

We represent a graph as a standard Haskell immutable array, indexed by vertices, where each component of the array is a list of those vertices reachable along a single edge. This gives constant time access (but not update—these arrays may be shared arbitrarily). By using an indexed structure we are able to be explicit about the sharing that occurs in the graph. In addition, this structure is linear in the size of the graph, that is, the sum of the number of vertices and the number of edges.

We can use the same mechanism to represent *undirected* graphs as well, simply by ensuring that we have edges in both directions. An undirected graph is a symmetric directed graph. We could also represent *multi-edged* graphs by a simple extension, but will not consider them here.

Graphs, therefore, may be thought of as a table indexed by vertices.

```
type Table a = Array Vertex a
type Graph   = Table [Vertex]
```

The type `Vertex` may be any type belonging to the Haskell index class `Ix`, which includes `Int`, `Char`, tuples of indices, and more. For now we will assume:

```
type Vertex = Char
```

We will make the simplifying assumption that the vertices of a graph are contiguous in the type (e.g. numbers 0 to 59, or characters 'a' to 'z', etc.). If not then a hash function will need to be introduced to map the actual names into a contiguous block. Because we assume contiguity, we commonly represent the list of vertices by a pair of end-points:

```
type Bounds = (Vertex,Vertex)
```

Haskell arrays come with indexing (`!`) and the functions `indices` (returning a list of the indices) and `bounds` (returning a pair of the least and greatest indices).

To further manipulate tables (including graphs) we define a generic function `mapT` which applies its function argument to every table index/entry pair, and builds a new table.

```
mapT :: (Vertex -> a -> b) -> Table a -> Table b
mapT f t = array (bounds t) [(v, f v (t!v)) | v<-indices t]
```

The Haskell function **array** takes low and high bounds and a list of index/value pairs, and builds the corresponding array in linear time.

Finally, it is sometimes useful to translate an ordered list of vertices into a lookup table which shows the position of the vertex in the list. For this we could use the function **tabulate**:

```
tabulate :: Bounds -> [Vertex] -> Table Int
tabulate bnds vs = array bnds (zip vs [1..])
```

which zips the vertices together with the positive integers *1*, *2*, *3*,..., and (in linear time) builds an array of these numbers, indexed by the vertices.

2.2 Edges

Sometimes it is convenient to extract a list of edges from the graph. An edge is a pair of vertices. But, because some graphs are sparse, we also need to know separately what the vertices are.

```
type VE = (Bounds,[(Vertex,Vertex)])

edges :: Graph -> VE
edges g = (bounds g, [(v,w) | v <- indices g, w <- g!v])
```

To build up a graph from a list of edges we define a function buildG.

```
buildG :: VE -> Graph
buildG (bnds,es) = accumArray snoc [] bnds es
  where snoc xs x = x:xs
```

Like array the Haskell function accumArray builds an array from a list of index/value pairs, with the difference that accumArray accepts possibly many values for each indexed location, which are combined using the function provided as accumArray's first argument. Here we simply build lists of all the values associated with each index. Again, constructing the array takes linear time with respect to the length of the adjacency list. So in linear time, we can convert a graph defined in terms of edges to a graph represented by a vertex table.

2.3 Simple operations

Following edges

To find the immediate successors to a vertex v in a graph g we simply compute g!v, which returns a list of vertices.

Transposing a graph
Combining the functions edges and buildG gives us a way to reverse all the edges in a graph giving the *transpose* of the graph:

```
transposeG :: Graph -> Graph
transposeG g = buildG (vs, [(w,v) | (v,w) <- es])
  where (vs,es) = edges g
```

We extract the edges from the original graph, reverse their direction, and rebuild a graph with the new edges.

OutDegree and InDegree
Using mapT we could define,

```
outdegree :: Graph -> Table Int
outdegree g = mapT numEdges g
  where  numEdges v ws = length ws
```

which builds a table containing the number of edges leaving each vertex.

Now by using transposeG we can immediately define an indegree table for vertices:

```
indegree :: Graph -> Table Int
indegree g = outdegree (transposeG g)
```

This example gives an early feel for the approach we develop in these notes. Rather than defining indegree from scratch by, for example, building an array incrementally as we traverse the graph, we simply reuse previously defined functions, combining them in a fresh way. The result is shorter and clearer, though potentially more expensive (an intermediate array is constructed).

There are two things to say about this additional cost. Firstly, the additional cost only introduces a constant factor into the complexity measure, so the essence of the algorithm is preserved. Secondly, recent work in the automatic removal of intermediate structures promises to come a long way to removing this problem. We will come back to this in Section 7.

3 Depth-First Search

The traditional view of depth-first search is as a *process* which may loosely be described as follows. Initially, all the vertices of the graph are deemed "unvisited", so we choose one and explore an edge leading to a new vertex. Now we start at this vertex and explore an edge leading to another new vertex. We continue in this fashion until we reach a vertex which has no edges leading to unvisited vertices. At this point we backtrack, and continue from the latest vertex which does lead to new unvisited vertices.

Eventually we will reach a point where every vertex reachable from the initial vertex has been visited. If there are any unvisited vertices left, we choose one and begin the search again, until finally every vertex has been visited once, and every edge has been examined.

In this paper we concentrate on depth first search as a specification for a *value* rather than on a process. The specified value is the *spanning forest* defined by a depth-first traversal of a graph, that is, a particular sub-graph of the original which, while it contains all the vertices, typically omits many of the edges.

The edges of the graph that are included in the spanning forest are (quite naturally) called *tree edges*. The omitted edges may be classified further with respect to the forest. Thus an edge in the graph which goes in the opposite direction to the tree edges is called a *back edge*. Conversely, a *forward edge* jumps more than one level from a vertex to one of its descendents (in the spanning forest). Finally, a *cross edge* is an edge which connects vertices *across* the forest—but always from right to left, there are no left-right cross edges. This standard classification is useful for thinking about a number of algorithms, and later we give an algorithm for classifying edges in this way.

3.1 Specification of depth-first search

As the approach to depth-first search algorithms which we explore in these notes is to manipulate the depth-first forest *explicitly*, the first step, therefore, is to construct the depth-first forest from a graph. To do this we need an appropriate definition of trees and forests.

A forest is a list of trees, and a tree is a node containing some value, together with a forest of sub-trees. Both trees and forests are polymorphic in the type of data they may contain.

```
data Tree a   = Node a (Forest a)
type Forest a = [Tree a]
```

A depth-first search of a graph takes a graph and an initial ordering of vertices. All graph vertices in the initial ordering will be in the returned forest.

```
dfs :: Graph -> [Vertex] -> Forest Vertex
```

This function is the pivot of these notes. For now we restrict ourselves to considering its properties, and will leave its Haskell implementation until Section 5.

Sometimes the initial ordering of vertices is not important. When this is the case we use the related function

```
dff :: Graph -> Forest Vertex
dff g = dfs g (indices g)
```

which arbitrarily uses the underlying order of indices as an initial vertex ordering.

3.2 Properties

What are the properties of depth-first forests? They can be completely characterised by the following two conditions.

(i) The depth-first forest of a graph is a spanning subgraph, that is, it has the same vertex set, but the edge set is a subset of the graph edge set.
(ii) The graph contains no left-right cross edges with respect to the forest.

Later on in the paper, we find it convenient to talk in terms of *paths* rather than single edges: a path being made up of zero or more edges joined end to end. We will write $v \longrightarrow w$ to mean that there is an edge from v to w; and $v \twoheadrightarrow w$ to mean that there is a path of zero or more edges from v to w. It should be clear from the context which graph is being discussed.

The lack of left-right cross edges translates into paths. At the top level, it implies that there is no path from any vertex in one tree to any vertex in a tree that occurs later in the forest. Thus[2],

Lemma 1. *If* `(ts++us=dff g)`, *then*

$$\forall v \in \texttt{ts} \,.\, \forall w \in \texttt{us} \,.\, \neg(v \twoheadrightarrow w)$$

Deeper within each tree of the forest, there *can* be paths which traverse a tree from left to right, but the absence of any graph edges which cross the tree structure from left to right implies that the path has to follow the tree structure. That is:

Lemma 2. *If the tree* `(Node x (ts++us))` *is a subtree occurring anywhere within* `dff g`, *then*

$$\forall v \in \texttt{ts} \,.\, \forall w \in \texttt{us} \,.\, v \twoheadrightarrow w \Rightarrow v \twoheadrightarrow x$$

So the only way to get from v to w is via (an ancestor of) x, the point at which the forests that contain v and w are combined (otherwise there would be a left-right cross edge). Thus there is also a path from v to x.

The last property we pick out focusses on `dfs`, and provides a relationship between the initial order, and the structure of the forest[3].

Lemma 3. *Let a and b be any two vertices. Write $\twoheadrightarrow$ for paths in the graph* `g`, *and $\leq$ for the ordering induced by the list of vertices* `vs`. *Then*

$$\begin{aligned} &\exists t \in \texttt{dfs g vs} \,.\, a \in t \wedge b \in t \\ \Leftrightarrow\ &\exists c \,.\, c \twoheadrightarrow a \wedge c \twoheadrightarrow b \wedge (\forall d \,.\, d \twoheadrightarrow a \vee d \twoheadrightarrow b \Rightarrow c \leq d) \end{aligned}$$

This Property says that:

$\Rightarrow$ given two vertices which occur within a single depth-first tree (taken from the forest), then there is a predecessor of both (with respect to $\twoheadrightarrow$) which occurs earlier in `vs` than any other predecessor of either. (If this were not the case, then a and b would end up in different trees).

$\Leftarrow$ if the earliest predecessor of either a or b is a predecessor of them both, then they will end up in the same tree (rooted by this predecessor).

While these three properties are true of depth-first search spanning forests, but they are not complete. There are other useful properties not derivable from these.

[2] We use the notation `ts++us` to indicate any division of the list of trees in the forest, such that the order of the trees is preserved. Note that either `ts` or `us` could be empty. Also, we use $\in$ to indicate list membership and not purely for set membership.

[3] We further overload the $\in$ notation, to mean that both a and b occur as vertices within the tree t.

4 Depth-first search algorithms

Having specified depth-first search (at least partly) we turn to consider how it may be used.

Algorithm 1. Depth-first numbering

The first algorithm is straightforward. We wish to assign to each vertex a number which indicates where that vertex came in the search. A number of other algorithms make use of this *depth-first search number*, including the biconnected components algorithm that appears later, for example.

We can express depth-first ordering of a graph g most simply by flattening the depth-first forest in *preorder*. Preorder on trees and forests places ancestors before descendants and left subtrees before right subtrees[4]:

```
preorder :: Tree a -> [a]
preorder (Node a ts) = [a] ++ preorderF ts

preorderF :: Forest a -> [a]
preorderF ts = concat (map preorder ts)
```

Now obtaining a list of vertices in depth-first order is easy:

```
preOrd :: Graph -> [Vertex]
preOrd g = preorderF (dff g)
```

For many situations this is eactly what we want. However, it is sometimes useful to translate the ordered list into actual numbers. To obtain such a table of depth-first search numbers we can simply use the `tabulate` function from earlier:

```
preNums :: Graph -> Table Int
preNums g = tabulate (bounds g) (preOrd g)
```

Algorithm 2. Topological sorting

The dual to preorder is postorder, and unsurprisingly this turns out to be useful in its own right. Postorder places descendants before ancestors and left subtrees before right subtrees:

```
postorder :: Tree a -> [a]
postorder (Node a ts) = postorderF ts ++ [a]

postorderF :: Forest a -> [a]
postorderF ts = concat (map postorder ts)
```

So, like with preorder, we define,

[4] The use of repeated appends (++) caused by `concat` introduces an extra logarithmic factor here, but this is easily removed using standard transformations.

```
postOrd :: Graph -> [Vertex]
postOrd g = postorderF (dff g)
```

Again, using `tabulate` we could construct (in linear time) a table containing the post-order numbers of each of the vertices.

```
postNums :: Graph -> Table Int
postNums g = tabulate (bounds g) (postOrd g)
```

The absence of left-right cross edges in depth-first search forests leads to a pleasant property when any depth-first search forest is flattened in postorder. If there is a path from some vertex v to a vertex w later in the ordering, then there is also an even later vertex u (beyond w) which, like w is also reachable by a path from v. In addition, however, there is also a path in the other direction, going from u to v. We can make this precise as follows.

Definition 4. A linear ordering $\leq$ on vertices is a *post-ordering* with respect to a graph g exactly when,

$$v \leq w \wedge v \twoheadrightarrow w \Rightarrow \exists u \,.\, v \twoheadleftarrow\!\!\twoheadrightarrow u \wedge w \leq u$$

where $v \twoheadleftarrow\!\!\twoheadrightarrow u$ means $v \twoheadrightarrow u$ and $u \twoheadrightarrow v$.

This property is so-named because post order flattening of depth first forests have this property.

Theorem 5. *The order in which the vertices appear in* `postOrd g` *is a post-ordering with respect to* g.

Proof. If v comes before w in a post order flattening of a forest, then either w is an ancestor of v, or w is to the right of v in the forest. In the first case, take w as u. For the second, note that as $v \twoheadrightarrow w$, by Property 1, v and w cannot be in different trees of the forest. Then by Property 2, the lowest common ancestor of v and w will do.

We can apply all this to topological sorting. A topological sort is an arrangement of the vertices of a directed acyclic graph into a linear sequence $v_1, \ldots, v_n$ such that there are no edges (and hence no paths) from later (greater numbered) to earlier (lesser numbered) vertices.

This problem arises quite frequently, where the graph represents a set of tasks need to be scheduled, with the edges representing inter-task dependencies (an edge $v \longrightarrow w$ is interpreted as "task v must be done before task w"). The topological sort produces a linear ordering of the tasks in which none of the tasks depend on earlier tasks.

We define,

```
topSort :: Graph -> [Vertex]
topSort g = reverse (postOrd g)
```

Theorem 6. *If* g *is an acyclic graph, then* `topSort g` *produces a topological sorting of* g.

Proof. We write $\geq$ for the order of vertices from `topSort g`. Then $\leq$ is the reverse ordering, that is, the ordering given by `postOrd g`. Now, suppose that $v \geq w \wedge v \twoheadrightarrow w$. Then,

$$\begin{aligned} w \geq v \wedge v \twoheadrightarrow w &\Rightarrow v \leq w \wedge v \twoheadrightarrow w \\ &\Rightarrow \exists u \,.\, v \twoheadleftarrow\!\!\twoheadrightarrow u \wedge w \leq u \\ &\Rightarrow v = w \end{aligned}$$

as g is acyclic.

Algorithm 3. Connected components

Two vertices in an undirected graph are *connected* if there is a path from the one to the other. In a directed graph, two vertices are connected if they would be connected in the graph made by viewing each edge as undirected. Finally, with an undirected graph, each tree in the depth-first spanning forest will contain exactly those vertices which constitute a single component.

We can translate this directly into a program. The function `components` takes a graph and produces a forest, where each tree represents a connected component.

```
components :: Graph -> Forest Vertex
components g = dff (undirected g)
```

where a graph is made undirected by:

```
undirected :: Graph -> Graph
undirected g = buildG (vs, concat [[(v,w),(w,v)] | (v,w)<-es])
   where (vs,es) = edges g
```

The undirected graph we actually search may have duplicate edges, but this has no effect on the structure of the components. Furthermore, as the number of edges is at most doubled, neither is there any effect on the asymptotic complexity.

Algorithm 4. Strongly connected components

Two vertices in a directed graph are said to be *strongly connected* if each is reachable from the other. A strongly connected component is a maximal subgraph, where all the vertices are strongly connected with each other. The problem of determining strongly-connected components is well known to compiler writers as the dependency analysis problem—separating procedures/functions into mutually recursive groups. We implement the double depth-first search algorithm of Kosaraju (unpublished), and [Sha81].

```
scc :: Graph -> Forest Vertex
scc g = dfs (transposeG g) (reverse (postOrd g))
```

The vertices of a graph are ordered using `postOrd` (which, recall, includes a call to `dfs`). The reverse of this ordering is used as the initial vertex order for a depth-first traversal on the transpose of the graph. The result is a forest, where each tree constitutes a single strongly connected component.

The algorithm is simply stated, but its correctness is not at all obvious. However, it may be proved as follows.

Theorem 7. *Let a and b be any two vertices of* $\mathtt{g}$. *Then*

$$(\exists t \in \mathtt{scc\ g} \,.\, a \in t \wedge b \in t) \Leftrightarrow a \twoheadleftarrow\!\!\twoheadrightarrow b$$

Proof. The proof proceeds by calculation. We write $\mathtt{g}^{\mathrm{T}}$ for the transpose of $\mathtt{g}$. Paths $v \twoheadrightarrow w$ in $\mathtt{g}$ will be paths $v \twoheadleftarrow w$ in $\mathtt{g}^{\mathrm{T}}$. Further, let $\leq$ be the post-ordering defined by `postOrd g`. Then its reversal induces the ordering $\geq$. Now,

$$\begin{array}{ll}
 & \exists t \in \mathtt{scc\ g} \,.\, a \in t \wedge b \in t \\
\Leftrightarrow & \{\text{Definition of } \mathtt{scc}\} \\
 & \exists t \in \mathtt{dfs}\ \mathtt{g}^{\mathrm{T}}\ (\mathtt{reverse\ (postOrd\ g)}) \,.\, a, b \in t \\
\Leftrightarrow & \{\text{By Property 3}\} \\
 & \exists c \,.\, c \twoheadleftarrow a \wedge c \twoheadleftarrow b \wedge (\forall d \,.\, d \twoheadleftarrow a \vee d \twoheadleftarrow b \Rightarrow c \geq d) \\
\Leftrightarrow & \exists c \,.\, a \twoheadrightarrow c \wedge b \twoheadrightarrow c \wedge (\forall d \,.\, a \twoheadrightarrow d \vee b \twoheadrightarrow d \Rightarrow d \leq c)
\end{array}$$

From here on we construct a loop of implications.

$$\begin{array}{ll}
 & \exists c \,.\, a \twoheadrightarrow c \wedge b \twoheadrightarrow c \wedge (\forall d \,.\, a \twoheadrightarrow d \vee b \twoheadrightarrow d \Rightarrow d \leq c) \\
\Rightarrow & \{\text{Consider } d = a \text{ and } d = b\ \} \\
 & \exists c \,.\, a \twoheadrightarrow c \wedge a \leq c \wedge b \twoheadrightarrow c \wedge b \leq c \wedge (\forall d \,.\, a \twoheadrightarrow d \vee b \twoheadrightarrow d \Rightarrow d \leq c) \\
\Rightarrow & \{\leq \text{ is a post-ordering}\} \\
 & \exists c \,.\ (\exists e \,.\, a \twoheadleftarrow\!\!\twoheadrightarrow e \wedge c \leq e) \wedge \\
 & \qquad (\exists f \,.\, b \twoheadleftarrow\!\!\twoheadrightarrow f \wedge c \leq f) \wedge (\forall d \,.\, a \twoheadrightarrow d \vee b \twoheadrightarrow d \Rightarrow d \leq c) \\
\Rightarrow & \{e = c \text{ and } f = c \text{ using } (\forall d \ldots)\} \\
 & \exists c \,.\, a \twoheadleftarrow\!\!\twoheadrightarrow c \wedge b \twoheadleftarrow\!\!\twoheadrightarrow c \\
\Rightarrow & \{\text{Transitivity}\} \\
 & a \twoheadleftarrow\!\!\twoheadrightarrow b
\end{array}$$

which gives us *one* direction. But to complete the loop:

$$\begin{array}{ll}
 & a \twoheadleftarrow\!\!\twoheadrightarrow b \\
\Rightarrow & \{\text{There is a latest vertex reachable from } a \text{ or } b\} \\
 & a \twoheadleftarrow\!\!\twoheadrightarrow b \wedge \exists c \,.\, (a \twoheadrightarrow c \vee b \twoheadrightarrow c) \wedge (\forall d \,.\, a \twoheadrightarrow d \vee b \twoheadrightarrow d \Rightarrow d \leq c) \\
\Rightarrow & \{\text{Transitivity of } \twoheadrightarrow\} \\
 & \exists c \,.\, a \twoheadrightarrow c \wedge b \twoheadrightarrow c \wedge (\forall d \,.\, a \twoheadrightarrow d \vee b \twoheadrightarrow d \Rightarrow d \leq c)
\end{array}$$

as required, and so the theorem is proved.

To the best of our knowledge, this is the first calculational proof of this algorithm. Traditional proofs (see [CLR90], for example) typically take many pages of wordy argument. In contrast, because we are reusing an earlier algorithm, we are able to reuse its properties also, and so obtain a compact proof. Similarly, we believe that it is because we are using the depth-first search forest as the basis of our program that our proofs are simplified as they are proofs about *values* rather than about *processes*.

A minor variation on this algorithm is to reverse the roles of the original and transposed graphs:

```
scc' :: Graph -> Forest Vertex
scc' g = dfs g (reverse (postOrd (transposeG g)))
```

The advantage now is that not only does the result express the strongly connected components, but it is also a valid depth-first forest for the original graph (rather than for the transposed graph). This alternative works as the strongly connected components in a graph are the same as the strongly connected components in the transpose of the graph.

Algorithm 5. Classifying edges

We have already discussed the classification of graph edges with respect to a given depth-first search. Here we codify the idea by building subgraphs of the original containing all the same vertices, but only a particular kind of edge.

Tree edges are easiest, these are just the edges that appear explicitly in the spanning forest. The other edges may be distinguished by comparing preorder and/or postorder numbers of the vertices of an edge.

Only back edges go from lower postorder numbers to higher, whereas only cross edges go from higher to lower in *both* orderings. Forward edges, which are the composition of tree edges, cannot be distinguished from tree edges by this means—both tree edges and forward edges go from lower preorder numbers to higher (and conversely in postorder)—but as we can already determine which are tree edges there is no problem in extracting the remaining forward edges. The implementation of these principles is now immediate[5].

```
tree :: Bounds -> Forest Vertex -> Graph
tree bnds ts = buildG (bnds, concat (map flat ts))
  where  flat (Node v ts) = [(v,w) | Node w us <- ts]
                               ++ concat (map flat ts)

back :: Graph -> Table Int -> Graph
back g post = mapT select g
  where  select v ws = [ w | w <- ws, post!v<post!w ]

cross :: Graph -> Table Int -> Table Int -> Graph
cross g pre post = mapT select g
  where
    select v ws = [ w | w <- ws, post!v>post!w, pre!v>pre!w]

forward :: Graph -> Graph -> Table Int -> Graph
forward g tree pre = mapT select g
  where  select v ws = [ w | w <- ws, pre!v<pre!w] \\ tree!v
```

To classify an edge we generate the depth-first spanning forest, and use this to produce tables of preorder and postorder numbers. We then have all the information required to construct the appropriate subgraphs corresponding to the various sorts of edges.

[5] The use of (quadratic) list difference in `forward` is a minor infelicity—the second list is an ordered subsequence of the first so can be removed by a linear traversal of the first.

Algorithm 6. Finding reachable vertices

Finding all the vertices that are reachable from a single vertex v demonstrates that the `dfs` doesn't have to take all the vertices as its second argument. Commencing a search at v will construct a tree containing all of v's reachable vertices. We then flatten this with preorder to produce the desired list.

```
reachable :: Graph -> Vertex -> [Vertex]
reachable g v = preorderF (dfs g [v])
```

We could have used either flattening (pre- or post-order) but using preordering does not require any buffering of vertices—the vertices are placed in the list as soon as `dfs` places them in the spanning forest.

One application of this algorithm is to test for the existence of a path between two vertices:

```
path :: Graph -> Vertex -> Vertex -> Bool
path g v w = w 'elem' (reachable g v)
```

The `elem` test is lazy: it returns `True` as soon as a match is found. Thus the result of `reachable` is demanded lazily, and so only produced lazily. As soon as the required vertex is found the generation of the depth-first search forest ceases. Thus `dfs` implements a true *search* and not merely a complete *traversal*. To achieve this in a strict language like ML, for example, would require modifications to `dfs` to enable it to raise an exception at an appropriate time.

5 Implementing depth-first search

In order to translate a graph into a depth-first spanning tree we make use of a technique common in lazy functional programming: generate then prune. Given a graph and a list of vertices (a root set), we first generate a (potentially infinite) forest consisting of all the vertices and edges in the graph, and then prune this forest in order to remove repeats. The choice of pruning pattern determines whether the forest ends up being depth-first (traverse in a left-most, top-most fashion) or breadth-first (top-most, left-most), or perhaps some combination of the two.

5.1 Generating

We define a function `generate` which, given a graph g and a vertex v builds a tree rooted at v containing all the vertices in g reachable from v.

```
generate :: Graph -> Vertex -> Tree Vertex
generate g v = Node v (map (generate g) (g!v))
```

Unless g happens to be a tree anyway, the generated tree will contain repeated subtrees. Further, if g is cyclic, the generated tree will be infinite (though rational). Of course, as the tree is generated on demand, only a finite portion will be generated. The parts that `prune` discards will never be constructed.

5.2 Pruning

The goal of pruning the (infinite) forest is to discard subtrees whose roots have occurred previously. Thus we need to maintain a finite set of vertices (traditionally called "marks") of those vertices to be discarded. The set-operations we require are initialisation (the empty set), a membership test, and addition of an extra element. While we are prepared to spend linear time in generating the empty set (as it is only done once), it is essential that the other operations can be performed in constant time (otherwise we lose linearity of `dfs`).

The easiest way to achieve this is to make use of *state transformers*, and mimic the imperative technique of maintaining an array of booleans, indexed by the set elements. This is what we do. We provide an explanation of state-transformers in the Appendix, but as they have already been described in a number of papers [Mog89, Wad90, LPJ94], and already been implemented in more than one Haskell variant, we avoid cluttering the main text.

The implementation of vertex sets is easy:

```
type Set s = MutArr s Vertex Bool

mkEmpty :: Bounds -> ST s (Set s)
mkEmpty bnds = newArr bnds False

contains :: Set s -> Vertex -> ST s Bool
contains m v = readArr m v

include :: Set s -> Vertex -> ST s ()
include m v = writeArr m v True
```

A set is represented as a mutable array, indexed by vertices, containing booleans. To generate an empty finite set we allocate an appropriately sized array with every element initialised to `False`. Set membership, and augmenting the set with a new member are just done using array reading and writing.

Using these, we define `prune` as follows.

```
prune :: Bounds -> Forest Vertex -> Forest Vertex
prune bnds ts = runST (mkEmpty bnds `thenST` \m ->
                           chop m ts)
```

The `prune` function begins by introducing a fresh state thread, then generates an empty set within that thread and calls the "procedure" `chop`. The final result of `prune` is the value generated by `chop`, the final state being discarded.

```
chop :: Set s -> Forest Vertex -> ST s (Forest Vertex)
chop m [] = returnST []
chop m (Node v ts : us)
  = contains m v  `thenST` \visited ->
    if visited then
      chop m us
    else
```

```
include m v       `thenST` \_  ->
chop m ts         `thenST` \as ->
chop m us         `thenST` \bs ->
returnST ((Node v as) : bs)
```

When chopping a list of trees, the root of the first is examined. If it has occurred before, the whole tree is discarded. If not, the vertex is added to the set represented by m, and two further calls to chop are made in sequence.

The first, namely, chop m ts, prunes the forest of descendants of v, adding all these in turn to the set of marked vertices. Once this is complete, the pruned sub-forest is named as, and the remainder of the original forest is chopped. The result of this is named bs, and a forest is constructed from the two.

All this is done lazily, on demand. The state combinators force the computation to follow a predetermined linear sequence, but exactly where in that sequence the computation is, is determined by external demand. Thus if only the top-most left-most vertex were demanded, then that is all that would be produced. On the other hand, if only the final tree of the forest is demanded, then because the set of marks is single-threaded, all the previous trees will be produced. However, this is demanded by the very nature of depth-first search anyway, so it is not as restrictive as it may at first seem.

At this point one may wonder whether any benefit has been gained by using a functional language. After all, the code looks fairly imperative. To some extent such a comment would be justified, but it is important to note that this is the *only* place in the development that destructive operations have to be used to gain efficiency. In addition, the complete encapsulation provided by runST guarantees that dfs has a purely functional exterior—the state cannot escape, not even to repeat calls of dfs. As far as the rest of the program is concerned, dfs *is purely functional*. Thus we have the flexibility to gain the best of both worlds: where destructive update is vital we use it, where it is not vital we can encapsulate it and use the full power of the lazy functional languages.

5.3 Depth-first Search

The components of generate and prune are combined to provide the definition of depth-first search.

```
dfs g vs = prune (bounds g) (map (generate g) vs)
```

The argument vs is a list of vertices, so the generate function is mapped across this (having been given the graph g). The resulting forest is pruned in a left-most top-most fashion by prune.

5.4 Is State Essential?

If paying an extra logarithmic factor is acceptable, then it is possible to dispense completely with the imperative features used in prune, and to use an implementation of sets based upon balanced trees, for example. Then set membership and adding elements to sets become logarithmic operations, hence the extra factor.

Even in this case, however, the set of marks has to be passed around *in a state-like manner*: when pruning a tree it is vital to know which vertices occurred in the earlier trees. Thus the code for `dfs` may remain entirely unchanged, and simply the definitions of the plumbing combinators and the set operations would be changed to reflect the alternative implementation.

Interestingly there is an alternative to using state ***without losing asymptotic complexity***, and that is to use lazy arrays (as were implemented in LML for example, and discussed in the context of state transformers in Launchbury and Peyton Jones [LPJ]). In this case the `dfs` code would have to be altered, as the set membership test is combined in a single operation with adding a new element. That is, the test means "add a new element, and tell me if it was already in the set or not". Details of this technique have been exlpored by Johnsson [Joh].

6 Complexity Analysis

Models for complexity analysis of imperative languages have been established for many years, and verified with respect to reality across many implementations. Using these models it is possible to show that traditional implementations of the various depth-first search algorithms are linear in the size of the graph (that is, run in $O(V + E)$ time).

Corresponding models for lazy functional languages have not been developed to the same level, and where they have been developed there has not yet been the same extensive verification. Using such models (based on counting function calls) we believe our implementation of the depth-first search algorithms to be linear, but because these models have not been fully tested, we also ran empirical tests.

We took measurements on the strongly connected components algorithm, which uses two depth-first searches. Timings were taken on randomly generated graphs with up to 5000 vertices and edges, and we plotted the results. They are quite clear: the plotted points all lie on a plane, indicating the linearity of the algorithm.

As for constant factors, we currently estimate that we lose a factor of about 6 compared with coding in C by (a) using Haskell, and (b) using multi-pass algorithms. However, such figures are notoriously slippery, especially as the quality of the underlying implementation continues to improve.

7 Fusing the Phases

There has been a lot of recent work on program fusion (also known as deforestation) in the past few years. Most of this work finds its roots in Burstall and Darlington's fold/unfold transformations [DB76]. As the various depth-first search algorithms presented earlier are built component-wise in a multi-pass fashion, it makes sense to ask whether we can fuse the various phases. If we can, how similar are the results to traditional depth-first search algorithms?

7.1 Fusing the components of depth-first search

Even `dfs` itself was defined in two separate phases: generate an infinite (potentially) forest, and then prune it in a depth-first manner. The relevant code is as follows.

```
dfs g vs = prune (bounds g) (map (generate g) vs)

prune bnds ts = runST (mkEmpty bnds `thenST` \m ->
                       chop m ts)

chop m [] = returnST []
chop m (Node v ts : us)
  = contains m v  `thenST` \visited ->
    if visited then
      chop m us
    else
      include m v        `thenST` \_  ->
      chop m ts          `thenST` \as ->
      chop m us          `thenST` \bs ->
      returnST ((Node v as) : bs)

generate g v = Node v (map (generate g) (g!v))
```

To fuse this into a single phase we first unfold the definition of prune.

```
dfs g vs = runST (mkEmpty (bounds g) `thenST` \m ->
                  chop m (map (generate g) vs))
```

Secondly, we invent a new function snip which satisfies

```
snip m g vs = chop m (map (generate g) vs)
```

Now using fold/unfold steps we can transform this as follows (starting with a case analysis on the list argument):

```
snip m g [] = chop m (map (generate g) [])
            = chop m []
            = returnST []
```

and

```
snip m g (v:vs)
  = chop m (map (generate g) (v:vs))

  = chop m (Node v (map (generate g) (g!v))
                           : map (generate g) vs)

  = contains m v  `thenST` \visited ->
    if visited then
      chop m (map (generate g) vs)
    else
      include m v                         `thenST` \_  ->
      chop m (map (generate g) (g!v))     `thenST` \as ->
      chop m (map (generate g) vs)        `thenST` \bs ->
      returnST ((Node v as) : bs)
  = contains m v  'thenST' \visited ->
    if visited then
      snip m g vs
    else
      include m v      'thenST' \_  ->
      snip m g (g!v)   'thenST' \as ->
      snip m g vs      'thenST' \bs ->
      returnST ((Node v as) : bs)
```

Collecting all the pieces together we have the new definitions:

```
dfs g vs = runST (mkEmpty (bounds g) 'thenST' \m ->
                  snip m g vs)

snip m g [] = returnST []
snip m g (v:vs)
  = contains m v  'thenST' \visited ->
    if visited then
      snip m g vs
    else
      include m v      'thenST' \_  ->
      snip m g (g!v)   'thenST' \as ->
      snip m g vs      'thenST' \bs ->
      returnST ((Node v as) : bs)
```

This is much more like the traditional coding. Which of the versions is better? It depends what is wanted. Factorising the definition into components promises to make proofs about depth-first search itself easier, but having the two components fused is likely to be (marginally) more efficient, and does not rely of laziness. This latter point is important if these techniques are to be used in a strict language.

7.2 Moving Operations Across State Boundaries

In the previous section, we successfully moved a purely functional operation (`map (generate g vs)`) into the scope of a state thread simply by unfolding definitions. It was so easy because it was an *input* to the state operation that was being affected.

When we want to manipulate the *output* of a state thread we have to call on a little theory. To take a program of the form `f (runST m)` and move the `f` inside the state-thread, we use the following rule (from parametricity):

```
f (runST m) = runST (st f m)
```

where `st` is the function-part of the `ST` functor. That is

```
st f m = \s -> let (a,s')=m s in (f a,s')
```

Perhaps more convenient, however is to give an axiomatization with respect to `thenST` and `returnST`, which goes as follows.

```
st f (m 'thenST' (\v -> n)) = m 'thenST' (\v -> st f n)
st f (returnST a) = returnST (f a)
```

7.3 Topological sort

We take topological sort as an example. It was defined as a depth-first search followed by a flattening of the tree and a reversal of the list. Taking the definition and expanding out the definition of postOrd and dff gives:

```
topSort g = reverse (postorderF (dfs g (indices g)))

dfs g vs = runST (mkEmpty (bounds g) `thenST` \m ->
                  snip m g vs)

snip m g [] = returnST []
snip m g (v:vs)
  = contains m v  `thenST` \visited ->
    if visited then
      snip m g vs
    else
      include m v       `thenST` \_  ->
      snip m g (g!v)    `thenST` \as ->
      snip m g vs       `thenST` \bs ->
      returnST ((Node v as) : bs)
```

We will write revPost for the composition of reverse and postOrderF. Pushing revPost into the state thread gives

```
topSort g = runST (mkEmpty (bounds g) `thenST` \m ->
                   st revPost (snip m g (indices g)))
```

Again, we invent a new function definition. Let

```
revPostSnip m g vs = st revPost (snip m g vs)
```

Then, performing a case analysis on the list argument:

```
revPostSnip m g []
  = st revPost (snip m g [])
  = st revPost (returnST [])
  = returnST (revPost [])
  = returnST []
```

and

```
revPostSnip m g (v:vs)
      = st revPost (snip m g (v:vs))

      = st revPost
          (contains m v  `thenST` \visited ->
           if visited then
             snip m g vs
           else
             include m v       `thenST` \_  ->
          snip m g (g!v)    'thenST' \as ->
          snip m g vs       'thenST' \bs ->
          returnST ((Node v as) : bs))
```

Push the revPost all the way to the leaves of the state thread:

```
    = contains m v  'thenST' \visited ->
      if visited then
        st revPost (snip m g vs)
      else
        include m v       'thenST' \_  ->
        snip m g (g!v)    'thenST' \as ->
        snip m g vs       'thenST' \bs ->
        returnST (revPost ((Node v as) : bs))
```

Use the definition of postorderF, and the fact that reverse (xs++ys) = reverse ys ++ reverse xs (so long as xs is finite):

```
    = contains m v  'thenST' \visited ->
      if visited then
        revPostSnip m g vs
      else
        include m v       'thenST' \_  ->
        snip m g (g!v)    'thenST' \as ->
        snip m g vs       'thenST' \bs ->
        returnST (revPost bs ++ [v] ++ revPost as)
```

Now we introduce auxilliary names (ps and qs) for the result of applying revPost to as and bs, and express these *renamings* using returnST:

```
    = contains m v  'thenST' \visited ->
      if visited then
        revPostSnip m g vs
      else
        include m v                           'thenST' \_  ->
        (snip m g (g!v) 'thenST' \as ->
         returnST (revPost as))              'thenST' \ps ->
        (snip m g vs    'thenST' \bs ->
         returnST (revPost bs))              'thenST' \qs ->
        returnST (qs ++ [v] ++ ps)
```

Finally, pull the revPost operation across the calls to snip)

```
    = contains m v  'thenST' \visited ->
      if visited then
        revPostSnip m g vs
      else
        include m v                              'thenST' \_  ->
        st revPost
          (snip m g (g!v) 'thenST' \as ->
          returnST as))                           `thenST` \ps ->
        st revPost
          (snip m g vs    `thenST` \bs ->
          returnST bs))                           `thenST` \qs ->
        returnST (qs ++ [v] ++ ps)

    = contains m v  `thenST` \visited ->
      if visited then
        revPostSnip m g vs
      else
        include m v                               `thenST` \_  ->
        st revPost
          (snip m g (g!v))                        `thenST` \ps ->
        st revPost
          (snip m g vs)                           `thenST` \qs ->
        returnST (qs ++ [v] ++ ps)

    = contains m v  `thenST` \visited ->
      if visited then
        revPostSnip m g vs
      else
        include m v                               `thenST` \_  ->
        revPostSnip m g (g!v)                     `thenST` \ps ->
        revPostSnip m g vs                        `thenST` \qs ->
        returnST (qs ++ [v] ++ ps)
```

Putting all this together gives the following definition for topological sort:

```
topSort g = runST (mkEmpty (bounds g) `thenST` \m ->
                   revPostSnip m g (indices g))

revPostSnip m g [] = returnST []
revPostSnip m g (v:vs)
  = contains m v  `thenST` \visited ->
    if visited then
      revPostSnip m g vs
    else
      include m v                               `thenST` \_  ->
      revPostSnip m g (g!v)                     `thenST` \ps ->
      revPostSnip m g vs                        `thenST` \qs ->
      returnST (qs ++ [v] ++ ps)
```

We have successfully eliminated the intermediate spanning forest, but the result is still rather unlike a traditional implementation. The lists of vertices built up in the recursive calls are decidedly non-standard. A typical imperative solution would introduce a stack and push the vertices on to the stack as it went along. For example, it may be something like the following (assuming suitable definitions for `mkStack`, `stackToList` and `push`):

```
topSort g
  = runST (mkEmpty (bounds g)            'thenST' \m ->
           mkStack []                    'thenST' \s ->
           revPostSnip m s g (indices g) 'thenST' \_ ->
           stackToList s)

revPostSnip m s g [] = returnST ()
revPostSnip m s g (v:vs)
  = contains m v  'thenST' \visited ->
    if visited then
      revPostSnip m s g vs
    else
      include m v                        'thenST' \_  ->
      revPostSnip m s g (g!v)            'thenST' \_ ->
      push s v                           'thenST' \_  ->
      revPostSnip m s g vs               'thenST' \_ ->
      returnST ()
```

Again, which of these is "better"? The positioning of the push is rather subtle, and the choice of stack rather than queue even more so. If we had wanted the vertices in the other order (i.e. not reversed) then we would have had to make a different choice.

Can the second version be obtained from the first by techniques similar to those used earlier? If so, can the earlier techniques be automated? The answers to these questions are still far from clear.

8 Acknowledgements

This paper bears heavily on work by King and Launchbury [KL95] and much has been lifted verbatim. The additional material has benefitted from discussions with Alex Bunkenburg, and Tim Sheard.

Appendix

Imperative features were initially introduced into the Glasgow Haskell compiler to perform input and output. The approach is based on monads, and can easily be extended to achieve *in-situ* array updates and to allow the imperative actions to be delayed until their results are required. This is the model we use. The notation comes from [LPJ94]

We use the monad of state-transformers with type constructor ST which is defined:

```
type ST s a = a -> (a,s)
```

So elements of type `ST s Int`, say, are functions which, when applied to the state, return a pair of an integer together with a new state. As usual we have the unit `returnST` and the sequencing combinator `thenST`:

```
returnST :: a -> ST s a
returnST a s = (a,s)

thenST :: ST s a -> (a -> ST s b) -> ST s b
(m `thenST` k) s = k a t  where  (a,t) = m s
```

The ST monad comes equipped with three basic array operations:

```
newArr  ::Ix i=> (i,i) -> a ->ST s (MutArr s i a)
readArr ::Ix i=> MutArr s i a -> i -> ST s a
writeArr::Ix i=> MutArr s i a -> i -> a ->ST s ()
```

The first, newArr, takes a pair of index bounds (the type a must lie in the index class Ix) together with an initial value, and returns a reference to an initialised array. The time this operation takes is linear with respect to the number of elements in the array. The other two provide for reading and writing to an element of the array, and both take constant time.

Finally, the ST monad comes equipped with a function runST.

```
runST :: (\/s . ST s a) -> a
```

This takes a state-transformer function, applies it to an initial state, extracts the final value and discards the final state. The type of runST is not Hindley-Milner because of the nested quantifier, so it must be built-in to Haskell. The universal quantifier ensures that in a state thread variables from other state threads are not referenced. For details of this see [LPJ94].

So, for example,

```
runST (newArr (1,8) 0     `thenST` (\nums ->
       writeArr nums 5 42 `thenST` (\_ ->
       readArr nums 5     `thenST` (\v ->
       returnST v))))
```

will return 42. This can be read as follows: run a new state thread extracting the final value when finished; create a new array indexed from 1 to 8 with components all 0; then bind this array to nums; write to array nums at index 5 the value 42; then read the component in nums at index 5 and bind this value to v; finally return value v. Note that the final expression returnST v is unnecessary as readArr returns a value. The parentheses immediately after `thenST` are also unnecessary, as Haskell's grammar binds lambda expressions tighter than infix functions.

References

[CLR90] T.H.Corman, C.E.Leiserson, and R.L.Rivest, *Introduction to Algorithms.* MIT Press, MA, 1990.

[DB76] J.Darlington and R.Burstall, *A System which Automatically Improves Programs.* Acta Informatica, 6(1), pp 41-60, 1976.

[Har93] R.Harrison, *Abstract Data Types in Standard ML.* John Wiley and Sons, 1993.

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

[HT73] J.E.Hopcroft and R.E.Tarjan, *Algorithm 447: Efficient Algorithms for Graph Manipulation.* Communications of the ACM, 16(6), pp 372-378.

[Hug85] R.J.M.Hughes, *Lazy Memo Functions.* Proc. FPCA 85, Nancy, LNCS 201, Springer-Verlag, 1985.

[Joh] T.Johnsson, *Efficient Graph Algorithms Using Lazy Monolithic Arrays.* unpublished.

[KL95] D.King and J.Launchbury, *Structuring Depth-First Search Algorithms in Haskell.* Proc. POPL, San Francisco, CA, 1995.

[LPJ94] J.Launchbury and S.Peyton Jones, *Lazy Functional State Threads.* Proc. PLDI, Orlando, FL, 1994.

[LPJ] J.Launchbury and S.Peyton Jones, *State in Haskell.* LASC Special issue on State in Programming Languages, to appear.

[Man89] U.Manber *Introduction to Algorithms—A Creative Approach.* Addison-Wesley, MA, 1989.

[Mog89] E.Moggi, *Computational Lambda-Calculus and Monads.* Proc LICS, Asilomar, CA, 1989.

[Pau91] L.C.Paulson, *ML for the working programmer.* Cambridge University Press, 1991.

[Sha81] M.Sharir, *A Strong-Connectivity Algorithm and its Application in Data Flow Analysis.* Computers and Mathematics with Applications, 7(1), pp 67-72.

[Tar72] R.E.Tarjan, *Depth-first Search and Linear Graph Algorithms.* SIAM J. of Computing, 1(2), pp 146-160.

[Wad90] P.Wadler, *Comprehending Monads*, Proc. L&FP, Nice, France, 1990.