UNIVERSITY
of
GLASGOW

Department of Computing Science

Functional Programming and Graph Algorithms

David Jonathan King

A thesis submitted for a Doctor of Philosophy Degree in Computing Science at the University of Glasgow

March 1996

BLANK IN ORIGINAL

Abstract

Functional languages are renowned for their mathematical tractability, clarity of expression, abstraction powers, and more. There are problem domains, however, that still present real challenges to functional languages. One notoriously difficult problem domain is graph algorithms.

Graph algorithms have been studied for a long time with conventional von Neumann languages. The emphasis has primarily been on the efficiency of the algorithm. Concerns such as clarity of the algorithm have been secondary. Although the underpinnings of algorithms generally have a solid theoretical foundation, there is still some distance between computer program and proof of correctness.

This thesis is an investigation of graph algorithms in the non-strict purely functional language Haskell. Emphasis is placed on the importance of achieving an asymptotic complexity as good as with conventional languages. This is achieved by using the monadic model for including actions on the state. Work on the monadic model was carried out at Glasgow University by Wadler, Peyton Jones, and Launchbury in the early nineties and has opened up many diverse application areas. One area is the ability to express data structures that require sharing. Although graphs are not presented in this style, data structures that graph algorithms use are expressed in this style. Several examples of stateful algorithms are given including union/find for disjoint sets, and the linear time sort binsort.

The graph algorithms presented are not new, but are traditional algorithms recast in a functional setting. Examples include strongly connected components, biconnected components, Kruskal's minimum cost spanning tree, and Dijkstra's shortest paths. The presentation is lucid giving more insight than usual. The functional setting allows for complete calculational style correctness proofs — which is demonstrated with many examples.

The benefits of using a functional language for expressing graph algorithms are quantified by looking at the issues of execution times, asymptotic complexity, correctness, and clarity, in comparison with traditional approaches. The intention is to be as objective as possible, pointing out both the weaknesses and the strengths of using a functional language.

BLANK IN ORIGINAL

Contents

BLANK IN ORIGINAL

List of algorithms

Preface

This thesis is submitted in partial fulfilment of the requirements for a Doctor of Philosophy Degree at the University of Glasgow. It comprises a study of graph algorithms in a lazy functional language; with the thesis that they may be implemented in such languages without loss of asymptotic complexity and, furthermore, that the abstraction powers of these languages allows the algorithms to be expressed so that their structure is more apparent than is commonly the case.

All the work carried out herein is original material except where otherwise stated. Chapter 6 is an extension of the work presented by King and Launchbury (1995) at the 22'nd Conference on Principles of Programming Languages. Preliminary work on functional binomial queues in Chapter 3 was reported at the seventh Glasgow Workshop on Functional Programming, see King (1995).

The programs in this thesis are written in standard Haskell, Version 1.2 (Hudak et al. 1992), and have been executed with the Glasgow Haskell compiler (Peyton Jones et al. 1993). Knowledge of Haskell is assumed, but most of the concepts should be familiar to anyone that has a passing knowledge of functional languages, see Hudak and Fasel (1992) for a comprehensive tutorial on Haskell.

Précis of thesis

The principal issues covered in the thesis are established in Chapter 1. First it is explained why efficient implementations of graph algorithms have alluded purely functional programming languages. With examples of graph algorithms, a comparison is made between the traditional imperative approach and a functional approach, and then with an imperative functional approach. A major claim of this thesis is that

the high-level abstraction powers of functional languages can offer new insights into algorithms. Chapter 3 justifies this claim with two purely functional examples. First, program transformation is used to show the equivalence of two well-known sorting algorithms: treesort and functional quicksort. Then a functional implementation and correctness proof is given for a priority queue algorithm.

Sometimes purely functional algorithms are not enough. Chapter 4 explains how the monadic model may be used to express algorithms that require mutable state for their efficiency. Several examples of stateful algorithms are given, some of which are important for later graph algorithms. The chapter concludes with a discussion of the merits and otherwise of the imperative functional style of programming. Then returning to the purely functional world, graphs are introduced in Chapter 5. Using the typical methods of representation many examples of simple functions on graphs are given.

In Chapter 6 the work of the previous two chapters is brought together as the algorithms that are based on depth-first search are explored in detail. This chapter epitomises the thesis. Traditional algorithms are expressed in a modular way, which is good for both code reuse, and program verification. Mutable state is used, but only where it is essential for efficiency.

More graph algorithms are given in Chapter 7 including Kruskal's minimum cost spanning forest algorithm, Dijkstra's shortest paths, and Floyd's all-pairs shortest paths algorithm. Some of these algorithms are ones that seem intrinsically to require mutable state for their efficiency.

Chapter 8 examines the advantages and drawbacks of expressing graph algorithms in a purely functional language, compared with traditional approaches. Empirical comparisons are made between the same algorithms in a functional and an imperative language. The aspects of analysing the complexity of a functional algorithm are made, and some examples are given. The differences between the approach taken in the thesis and conventional approaches are quantified objectively. Finally, Chapter 9 reviews the thesis, and discusses directions for future research.

Contributions of thesis

- It is shown how graph algorithms may be expressed in purely functional languages with no loss of efficiency. Several examples of traditional graph algorithms are presented.

- The high-level abstraction powers for functional languages are shown to offer insights to the algorithms presented.

- Many examples of correctness proofs are given, and shown in all detail. For example, a functional implementation and correctness proof, of binomial functional queues is given. Moreover, several proofs of graph algorithms are given in all detail, such as a proof of a strongly connected components algorithm.

- Examples of imperative functional algorithms are given, and are shown to be superior in some ways to conventional imperative approaches.

- An extensive survey of numerous approaches for expressing graph algorithms is given.

- A comparison is made between functional and imperative presentations with regard to expressibility, demonstrating correctness, demonstrating complexity, and time and space performance.

Acknowledgements

This work was supported for three years by a studentship from the Engineering and Physical Sciences Research Council, and was undertaken at the Department of Computing Science, Glasgow University. I am indebted to many people who have helped me throughout my period of research, and I gratefully thank them all, whether named here or not.

This thesis has been greatly influenced by my supervisor, John Launchbury. John is a very encouraging and patient teacher, and is never short of good ideas. I'm very grateful for his kindness and friendship throughout my studies.

I would like to thank the examination committee: Chris Reade, John O'Donnell and David Watt for giving extensive comments which have helped to improve the final copy of this thesis.

The Glasgow Functional Programming Group is an extremely fruitful and friendly environment to work in. I have learnt a great deal from the weekly seminars and annual workshop, not only about other people's work, but how to present my own material. I thank Simon Peyton Jones for reading through a draft of this thesis and giving extensive comments. Phil Wadler for inspiring my interest in many topics. John O'Donnell for his encouragement and interest in my work. Kieran Clenaghan for his knowledge of graph algorithms, and willingness to listen to my ideas. Others in the group that I would like to thank are the Glasgow Haskell compiler gurus, Will Partain and Jim Mattson. Both always ready to answer my questions.

Keith Van Rijsbergen deserves credit for his guidance and confidence in me at critical times during my period of study.

I would also like to thank my fellow students in the Department. Especially my office mates in G162: Andy Gill, Simon Marlow, and André Santos. Life would have been much duller without them.

Finally, I would like to thank my family, especially my parents for their encouragement, and everlasting faith in me.

March 1996, Glasgow

David J. King

Chapter 1

Introduction

Computing languages are constantly being designed with the goal that they should aid the quick and accurate development of diverse software. In the opinion of many language designers (for example, John Backus (Backus 1978) and Robin Milner (Frenkel and Milner 1993)) functional languages have more potential to succeed in meeting this goal than conventional programming languages. There are two quite strong arguments, however, that can be levelled against functional languages. The first is that they are too inefficient, and the second is that there are many problem domains that these languages do not solve well. The latter issue is addressed in this thesis with an investigation of graph algorithms, which have until now proved to be incompatible with purely functional languages.

Graph theory has been studied since Euler's (1736) paper on the famous seven bridges of Königsberg problem. The problem is to determine if a tour is possible crossing each of the seven bridges no more than once. Today graph theory and its algorithms are widely used in computer software and hardware, but perhaps more importantly they have many real world applications. A survey of graph theory applications will not be given here, partly as it is not the topic of the thesis, but also because it has been done extensively by others. For example, the following three books cover a wide spectrum of applications: Wilson and Beineke (1979); Temperley (1981); and Walther (1984).

Graph algorithms can be complex, making their implementation difficult to comprehend and non-trivial to prove correct. Functional languages are acknowledged to be good at expressing problems clearly, and for providing a good framework for correctness proofs. So why have graph algorithms frustrated programmers of purely functional languages? The reason is fundamental — graphs do not have a recursive

data structure. Recursive data structures are tree shaped, and can be recursively traversed from their root to their leaves. Graphs do not conform to this structure — there may be a cycle from a vertex to itself. A traversal, therefore, will re-visit old vertices, and this has to be dealt with.

Our interest is with non-strict, purely functional languages like Haskell, rather than strict mostly functional languages like Standard ML. There are several advantages with using a pure language. The lack of side effects makes mathematical reasoning about programs more straightforward. Lazy languages also provide for greater expressiveness which will be illustrated throughout the thesis. Haskell was chosen because it is the *standard* non-strict functional language. Besides this, Haskell was an obvious choice at Glasgow, where research is being undertaken with the language. Although Haskell is used throughout, many of the points made also apply to other functional languages, including strict mostly pure languages like Standard ML.

Although all the examples will be expressed in Haskell, there are three minor deviations from the standard: (i) the monad of state transformers requires some functions to be built into the language (`runST`, for example); (ii) the `do` notation is used for including imperative actions on state; and (iii) pairs are used instead of the `Assoc` type. These deviations are expected to exist in version 1.3 of Haskell.

The terminology for graphs used here is not completely standard, mainly because there is no standard terminology for graphs. The term *graph* will be used to mean a directed graph (some authors abbreviate this to *digraph*; and some use the term graph to refer to an undirected graph). The points will be called *vertices* and the arrows *edges* (these are called *arcs*, by many authors to distinguish them from undirected edges). Graphs considered will always be finite in the number of vertices and edges. Unless otherwise stated our graphs do not have multiple edges, multiple vertices, or self-loops. Nonetheless, such graphs as well as other kinds of graph can be represented, and are in this thesis.

1.1 Graph algorithms and functional languages

Let's start with a simple example: detecting if a graph has a cycle. One algorithm is to follow graph edges leaving a stone at each vertex that is passed through. When a dead-end is reached, we retrace our steps picking up the stones until a new path is

found to follow. If a new path leads to a vertex with a stone, then the graph has a cycle. On the other hand, if we tour the whole graph without returning to a vertex with a stone then the graph has no cycles.

To implement this algorithm we have to mimic placing stones on vertices. The most suitable representation for this is an array mapping vertices to stones. This makes it easy to access and change the stone component. The algorithm's time complexity is dependent on these array operations as they will be performed for each vertex. Writing to an array in constant time is not straightforward in a language without side effects. This is well known with respect to graph problems, for instance, Zimmermann (1990) in his book on automatic complexity analysis of functional programs says the following:

> *"functional programming is not well suited for algorithms of graph theory as these usually make frequent use of side effects"*

In other words access to some form of mutable state is required in order to achieve a good asymptotic complexity for implementations of graph algorithms.

In many ways the elegance of a functional language comes from not having access to mutable state, so it should only be used when there is no alternative. It is a view strongly held by programming language researchers, that with a von Neumann machine architecture, access to the state is essential for efficient graph algorithms. In some ways this is obvious, since information about a vertex needs to be updated instantly for an efficient algorithm. Nevertheless, even this may be possible by encapsulating the updating operations into a special combinator. The combinator would be purely functional, in the sense that it would take and return purely functional values. However, the function itself would be implemented in an imperative style. In this thesis such methods are not dismissed. In Chapter 7 it is shown how breadth-first search may be expressed with a special combinator. Often, as is the case with breadth-first search, that the implementation becomes more complex than an imperative implementation.

1.2 Why graph algorithms can be complex

The most efficient graph algorithms traverse the graph the fewest number of times. The fastest linear graph algorithm will traverse the graph once. A linear graph algorithm is one whose running time is proportional to the size of the graph, that is to say, the sum of the number of vertices(V) and edges(E), i.e. $O(V+E)$. Most graph algorithms require many pieces of information to be calculated for each vertex. For a single pass algorithm many calculations will be performed at once. This is part of the reason why traditional presentations of graph algorithms are difficult to follow.

1.3 Traditional approaches

1.3.1 English style pseudo-code presentations

The traditional way of expressing graph algorithms is to give English style pseudo-code. This is usually a mixture of English statements and Algol-like imperative code. The resulting algorithms cannot be executed and usually do not contain enough detail to make them easy to code. Here's an example of an algorithm for finding the connected components of an undirected graph. This is taken verbatim from Manber's (1989) book, p. 192.

```
Algorithm Connected_Components(G);
Input: G=(V,E) is an undirected graph
Output: v.Component is set to the number of the component
        containing v, for every vertex v.

begin
     Component_Number := 1;
     while there is an unmarked vertex v do
          Depth_First_Search(G,v);
          (using the following preWORK;
                v.Component := Component_Number;)
          Component_Number := Component_Number + 1
end
```

Figure 1.1 English style pseudo-code description of connected components.

The algorithm in Figure 1.1 makes use of a depth-first search algorithm augmented with some code specifically to annotate vertices with their connected component number. A depth-first search is performed starting at vertex v, and exactly all the vertices that are in the same component as v will be annotated. The component number is then incremented, and another depth-first search commences starting with an unvisited vertex. This process is repeated until all the vertices have been visited, and hence annotated with a component number. Here is Manber's (1989, p. 191) description of depth-first search:

```
Algorithm Depth_First_Search(G,v);
Input: G=(V,E) is an undirected graph, v is a vertex in V
Output: depends on the application

begin
      mark v;
      perform preWORK on v;
      for all edges (v,w) do
            if w is unmarked then Depth_First_Search(G,w);
            perform postWORK for (v,w)
end
```

Figure 1.2 English style pseudo-code description of depth-first search.

The algorithm for depth-first search presented in Figure 1.2 is given as a skeleton description with `preWORK` and `postWORK` changing for particular algorithms. This is a useful approach as depth-first search is used for many other graph algorithms. This programming idiom, however, is not supported by conventional compilers — it is not possible to pass fragments of code for `preWORK` and `postWORK` to the depth-first search procedure. Instead the depth-first search fragment has to be re-written for each algorithm. In a functional language there is no problem: common programming idioms are just *higher-order functions* which are passed fragments of code in the form of functions. The ability to name and reuse programming idioms is one of the great strengths of functional languages. This is the approach taken here, and these concepts will now be demonstrated with the above example.

1.3.2 Functional language presentations

The typical functional programming approach is quite different. Programs are structured as a sequence of transformations on the input data. The focus is on what the intermediate data should be at each stage. For example, the program to separate vertices that are in different components may be expressed functionally as:

```
vertex_components :: Graph -> [[Vertex]]
vertex_components g = map flatten (dff g)
```

A depth-first search is performed on the graph, returning in this case a depth-first spanning forest of the graph. This is a list of trees where each tree contains the vertices of one component. Finally each tree is flattened using `flatten` returning a list of lists.

Not all the details will be given here (such as the representation used for `Graph`, `Vertex` etc.), since they are described and motivated in later chapters. Instead, just enough detail is given so that the examples can be used to substantiate some of the claims made.

The English style pseudo-code (Figure 1.2) can be mimicked by taking the result of `vertex_components` and doing the following:

```
component_table :: Graph -> [(Vertex, Int)]
component_table g = [ (v,n) | (vs,n)<-ps, v<-vs]
      where ps = zip (vertex_components g) [1..]
```

This generates a table (actually a list) mapping each vertex to its component number. The components could just as easily have been generated the in the form of subgraphs by the following:

```
components :: Graph -> [[(Vertex,Vertex)]]
components g = [ [ (v,w) | v<-vs, w<-g!v] | vs<-vcs]
      where vcs = vertex_components g
```

Instead of augmenting a skeleton algorithm with fragments of code, the algorithm is built by *gluing* together simpler parts. Structuring programs in this way often allows

```
dff :: Graph -> [Tree Vertex]
dff g = fst (dfs g [] (vertices g))

dfs :: Graph -> [Vertex] -> [Vertex] -> ([Tree Vertex], [Vertex])
dfs g ms []     = ([], ms)
dfs g ms (v:vs) = if v `elem` ms
                    then dfs g ms vs
                  else let (ts,as) = dfs g (v:ms) (g!v)
                           (us,bs) = dfs g as vs
                       in (Node v ts: us, bs)
```

Figure 1.3 Purely functional implementation of depth-first search.

for greater understanding of the algorithm. Figure 1.3 shows a purely functional implementation of depth-first search.

The function `vertices` returns a list of all the vertices contained in the graph. The function `dfs` takes three arguments: the graph; a list of all the vertices that have been visited before; and an ordering of vertices that are used as positions to start searching. The expression `(g!v)` returns a list of all vertices that are adjacent to `v` in graph `g`. For each vertex `dfs` checks to see if it has been visited before by looking it up in the visited list. Since doing a lookup in a list of length n takes $O(n)$ time, the asymptotic complexity of `dfs` is $O(V(V+E))$. The English style pseudo-code determines if a vertex has been visited before by extracting from the field component of the vertex in constant time. Consequently, for depth-first search, there is an unfortunate disparity between the complexity of the functional algorithm $O(V(V+E))$ and the imperative algorithm $O(V+E)$.

One of the main reasons people have persisted with functional languages is *provability*. The style shown above of expressing algorithms as the composition of smaller units, whilst being good for structuring programs, is also helpful in structuring proofs. Pure functional programs are *referentially transparent*, which roughly means that the same expression can be replaced with the same value. In other words, pure programs are side effect free. This makes the mathematics for reasoning about a program's execution more tractable, making it realistic to prove a program's correctness in all detail.

As well as provability, the functional implementation has more potential for optimisations. Again because of the mathematical tractability of the code it makes it straightforward to apply simple transformations. For instance, the dataflow for the function `vertex_components` which is: *Graph* → *Forest* → *List*, can be reduced to the dataflow: *Graph* → *List*. This is known as *code fusion*, one of many transformations that are realistic to include in an optimising functional language compiler.

Another important advantage highlighted in the examples above is code *reuse*. The function `vertex_components` was reused in the definitions of `component_table`, and `components`. Furthermore, the function `dff` can be freely reused for expressing many algorithms. Code reuse is more prevalent in functional languages than conventional languages, in part because of the *transformations on data* style of programming. But more importantly because functions can be *polymorphic*, meaning that they may take values of many different types. An example used above, is the function `zip`, that zips two lists together regardless of their type. Functional algorithms are commonly expressed as the composition of simple reusable components like `dff`. Code reuse comes hand-in-hand with *modularity*, which is beneficial for programming and proof.

Referential transparency outlaws *destructive updating*. For instance, when you execute `x := 8`, in an imperative language, the contents of `x` is destroyed and replaced with `8`. Unfortunately, efficient implementations of graph algorithms seem inherently to require some form of destructive updating (Section 1.1). Figure 1.2 illustrates this: during the course of the depth-first search vertices are marked. Marking is carried out for each and every vertex so it has a direct impact on the complexity of the algorithm.

There are several excellent expositions on the merits of functional programming languages including: Backus (1978); Hughes (1989); and Pountain (1994). Many of the points made above have been drawn from this material.

1.4 Imperative functional approach

This thesis explores the use of state in a functional language. There are several ways of introducing state, some of which are reviewed later (Chapter 4, p. 66). The method chosen here is to use the monad of state transformers which is fully supported in the Glasgow Haskell compiler.

Surprisingly, in the depth-first search algorithm presented above, the marking of vertices is the only place where destructive update is necessary for an $O(V + E)$ time implementation. So the implementation uses an updatable array (just a normal imperative array which has $O(1)$ time array update) to represent the set of visited vertices (Figure 1.4). Hence the functional implementation has asymptotic complexity $O(V + E)$.

Introducing state into a functional language, no matter how elegantly, is fraught with danger. The difference between the provability of functional and imperative functional code is marginal. The code itself even looks imperative. Here is an example of depth-first search expressed in a functional language, using the monad of state transformers (Figure 1.4). This is meant to give you a flavour of what to expect. The details about introducing state into a functional language are not given until Chapter 4.

```
dff :: Graph -> [Tree Vertex]
dff g = runST (do { marks <- newArr (bounds g) False;
                    dfs g marks (vertices g)
                  })

dfs :: Graph -> ST s (MutArr s Vertex Bool) -> [Vertex] ->ST s [Tree Vertex]
dfs g marks []     = return []
dfs g marks (v:vs) = do { visited <- readArr marks v;
                          if visited then dfs g marks vs
                          else do { ts <- dfs g marks (g!v);
                                    us <- dfs g marks vs;
                                    return (Node v ts: us)
                                  }
                        }
```

Figure 1.4 Imperative functional description of depth-first search.

This is a good example of state being *encapsulated*, purely functional values are taken and returned. Other algorithms like the functional components algorithm now have an acceptable time complexity as well as being purely functional. Throughout this thesis achieving an acceptable asymptotic complexity with respect to conventional languages is of paramount concern. Nevertheless, as much as possible the algorithms will be expressed purely functionally, although sometimes this is unavoidable.

BLANK IN ORIGINAL

Chapter 2

Literature survey

Graph theory and its algorithms is a huge topic. This chapter reviews some of the many diverse methodologies for the design, implementation, and verification of graph algorithms.

2.1 Graph algorithm design

Graph problems are so diverse that a unifying approach to the design of graph algorithms is not feasible. The style of presentation of graph algorithms over the last twenty years has been to present the final algorithm usually with pseudo-code (see Section 1.3). Typically, the derivation of the algorithm and intuition as to why it works are not clear. There has been a potpourri of approaches for expressing graph algorithms to give more insight, some of which are now reviewed.

2.1.1 Mathematical induction

Mathematical induction is not only useful for proving the correctness of an algorithm, but can be instrumental in algorithm development. As an example take the problem of sorting a list of numbers. The base case of an inductive proof is the empty list, which requires no sorting. Let's assume that $n-1$ elements are already sorted, then n elements can be sorted by inserting the nth element in its correct position. We have proved that n elements can be sorted, by using insertion — thus we have the algorithm *insertion sort*. The performance of this $O(n^2)$ time sort can be improved

by using a different inductive principle. Instead of extending a solution of $n-1$ to n, a solution of $n/2$ is extended to a solution for n. The base case is again trivial, the inductive case is to merge two sets of $n/2$ numbers together — this leads to the $O(n \log n)$ algorithm *merge sort.*

Manber (1989) describes mathematical induction as a general method for developing combinatorial algorithms. As is clear from the above examples, the philosophy gives insight and understanding of the algorithms. Graphs, however, are not well suited to the inductive approach. They do not have an inductive structure with neat sub-components. In general it is not possible to derive a graph algorithm by extending a solution of a small graph to a solution of a larger graph. There are, however, cases where it is possible to induct on the number of vertices or edges. One example given in Manber (1989, Chapter 7) is Dijkstra's single-source shortest paths algorithm (further explained in Section 7.2). The induction hypothesis is: given a graph and a source vertex v, we know the closest k vertices to v, and the lengths of the shortest paths to them. So induction is on the vertices whose shortest paths have been computed. Initially, the first shortest path is the closest vertex from v and this is the base case for the induction. Having an inductive principle doesn't guarantee a good or the best algorithm, and not all graph problems can be expressed in this way. Nevertheless, where applicable this offers insight to the algorithm, as well as formally proving it correct.

2.1.2 Using libraries

With many graph algorithms several efficient routines are essential to achieve the best asymptotic complexity. An obvious approach to the fast development of graph algorithms is to maintain a library of highly-tuned, reusable routines. Recently this approach has been taken by LEDA (Mehlhorn and Naher 1989) and Stanford GraphBase (Knuth 1993). In both cases the graph algorithms and related data structures are implemented imperatively, using C++ for LEDA, and C for GraphBase. Having such libraries does prevent the *re-invention of the wheel*; but having them in C is not ideal. While C is widely used, the language does not provide a good setting for clarity and proof. Furthermore, it is not feasible to provide a complete set of routines for every graph problem. For example, different representations of the graph are needed for different algorithms. With Knuth's GraphBase the style of presentation is a lit-

erate one — documents are written in CWEB which can be translated to C and/or TEX. This style encourages a far better presentation than usual, and is a great aid to understanding and maintaining the code. GraphBase provides several examples of non-random graphs, with the intention to provide standards to empirical compare different algorithms. Example non-random graphs used in GraphBase include: graphs of character acquaintances in classic works of literature; cross references in Roget's Thesaurus; mileage between North American cities; and many more.

2.1.3 Graph languages

Another approach to the design of graph algorithms is to use a specialised language for graphs. Examples include GRAMAS (Pape 1979) a graph manipulation system which provides an Algol-like language; and GRAPL (Nagl 1979) which is mainly concerned with dynamic algorithms. The language GEL (Graph Exploration Language) of Erwig (1992) provides exploration operators, which give a concise way of expressing many algorithms. GEL will be discussed in some detail since it is based on a lazy functional language. In GEL depth-first search and breadth-first search are expressed:

```
dfs v = explore v:Stack; suc
bfs v = explore v:Queue; suc
```

Here `explore` denotes tree exploration, and takes a *data structure*, an *expansion*, and a *computation*. The data structures have associated *get* and *put* operations, for taking a single element from and inserting multiple elements into the data structure. In the expressions above the *get* operation will return an element only if it was not returned by a previous call of get. The *put* operation is expressed by `(:)` above (note this is an overloaded operator). For stacks, `v:Stack` means that `v` is initially pushed onto the stack. The expansion is expressed by the function `suc`, which gives the association list for the current graph element. Computations were not used in the above; computations are actions taken during graph exploration. The explore operators work well for many algorithms, reducing the gap between specification and implementation.

Although a graph based language can be expressive for some problems, there are likely to be problems that cannot be expressed well in the language, and it seems less than ideal to add new language concepts for every new problem tackled. Ever

changing languages tend not to be widely used, and having too many special features tends to make them more difficult to learn.

Mathematica

Mathematica is an environment/workbench for experimenting with discrete mathematics, and is available on many platforms (Wolfram 1991). It provides a high-level applicative programming language with an extensive amount of mathematics under the hood. The language features include list processing, algebraic simplification, pattern matching, and looks like a traditional functional language. A high-level graphics description language allows graphs to be displayed interactively. The main drawback of Mathematica is that the model of computation makes it difficult to get the right complexity for some traditional algorithms. Skiena (1990) shows how traditional graph algorithms may be implemented in Mathematica. His emphasis is on conciseness of code rather than efficiency. This is fine for experimentation, but when considering real problems on a large scale, efficiency becomes crucial.

2.1.4 Program derivation

The advantages of deriving a program rather than inventing it, then proving it correct, are quite clear. The derivation gives a correctness proof for free, whereas it may be extremely difficult to show the correctness of an arbitrary program. Moreover, the design decisions are pinpointed during derivation, shedding more light on the resulting program.

Bird-Meertens's calculational style

The *Bird-Meertens* formalism (also known as *squiggol*) embodies the transformational approach (Meertens 1986, Bird (1987, 1988), Backhouse 1989). It is particularly suited to the functional paradigm, although it is claimed not to be language dependent. Starting with a mathematical specification, a more efficient algorithm is developed by successive program transformations. The methodology has been applied and developed on many diverse problems including some graph problems (Bird 1984*a*).

But here Bird does not come up with algorithms that have the best asymptotic complexity. For example, he derives an algorithm to test if a graph has a cycle that runs in $O(V^2)$ time whereas this problem is possible in $O(V + E)$ time (note $E \leq V^2$).

In the Bird-Meertens formalism, algebraic properties of datatypes are used in the development of algorithms. Most of the work done so far has been with the datatypes in the Boom hierarchy, namely: lists, sets, bags, and to a lesser extent trees. See Jeuring's (1992) thesis or Hoogerwoord's (1989) thesis for an abundance of problems solved in this style. Examples include finding the minimum sum over all segments of an integer list, and Eratosthenes's sieve for computing prime numbers. There has been limited work applying the approach to arrays (Wright 1988, Jeuring 1991). Gibbons's (1991) thesis is about applying the approach to trees. He used higher-order combinators, *catamorphisms*, to express upward and downward *accumulations* on trees. Applying an accumulation to a tree replaces every node with some 'accumulated information' about other tree nodes. An upwards accumulation replaces every node with some function applied to its descendants; downward accumulations replace every node with some function applied to its ancestors. An example is an algorithm to label every node with the smallest and largest elements of the node's subtree; this is simply expressed using an upward accumulation.

Derivation specifically for efficient graph algorithms was investigated by Reif and Scherlis (1984). They worked with a high-level specification of an algorithm and developed a lower-level efficient implementation. Their approach gives insight into the algorithms they develop; but the development can be long and tedious and relies somewhat on knowing the final algorithm. To overcome this tedium they propose to make the development semi-automatic. Their main example is the biconnected components algorithm of Tarjan (1972).

The transformational approach can work extremely well especially on structures that have a well known algebra (lists and trees, for example). Graphs, however, do not have an obvious algebra. Developing a graph algorithm from its specification is a good way of gaining deeper insight and understanding into the algorithm; but to do the development a *eureka factor* plays a strong rôle. For large graph problems, the transformational approach can be laborious. Moreover, the transformation rules are not complete — new algorithms commonly need new transformations.

Dijkstra's calculational style

The *calculational style* of programming described by Dijkstra (1976) and others, that has its origins with the axiomatisation of programs (Hoare 1969) and stepwise-refinement (Wirth 1971), is loosely analogous in the imperative world to the Bird-Meertens formalism. Efficient algorithms are derived by using pre-conditions, post-conditions, and loop invariants of program fragments. Problems tackled by this approach are often to do with array manipulation, for example: sorting, searching, or partitioning an array so that sections have certain properties. More recently some graph problems have been tackled in this style (Gries and Schneider 1993, Chapter 19). Problems that were previously considered hard to solve are derived systematically from their specification by following the rules of the calculus. This is not purely mechanical though, occasionally design decisions (eureka steps) are needed. Again this approach can be tedious for large problems and often a certain amount of insight is needed to derive an algorithm.

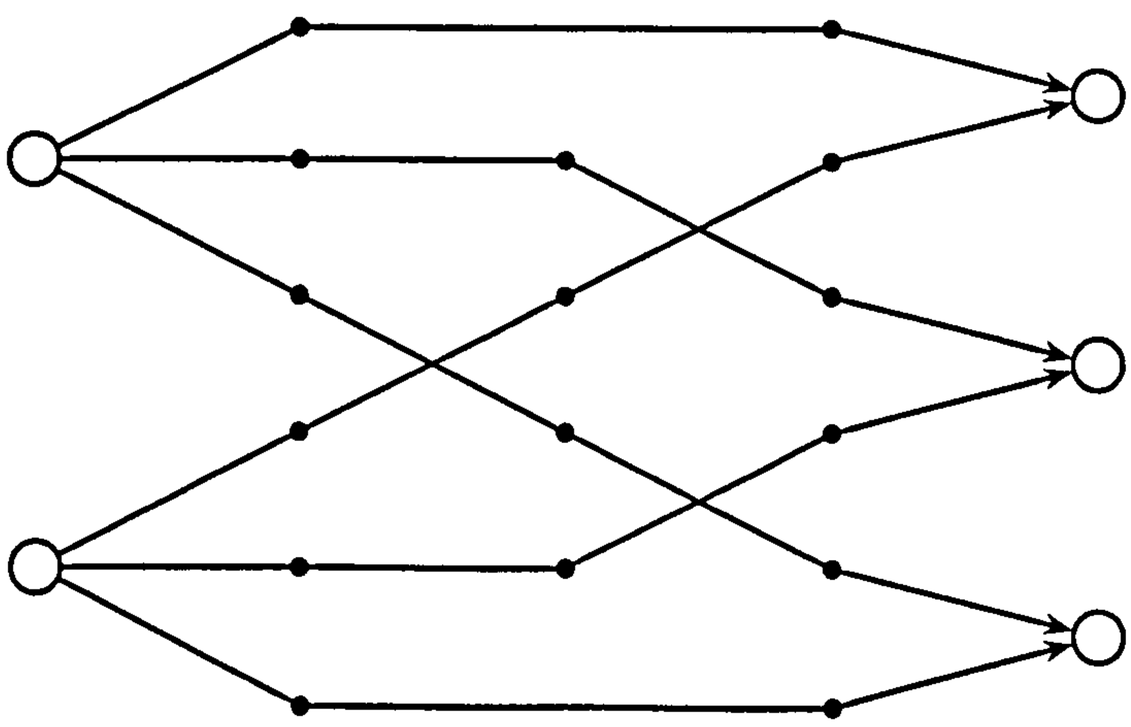
2.1.5 Graph algebras

Algebras for graphs have been studied in the context of graph rewriting, see Bauderon and Courcelle (1986), for example. There is no universally accepted graph algebra for expressing or developing algorithms. It is not obvious how such an algebra should be expressed. Other common structures such as trees and lists have a well understood algebra. The reasons are similar to why graph algorithms do not lend themselves to inductive proofs, and are problematical for lazy functional languages — graphs do not have a recursive data structure. Recursive data structures are always tree shaped. Klarlund and Schwartzbach (1993) use *graph types* to overcome this weakness. Datatypes are extended so that graph structures can be expressed without using explicit pointers. They do this by using routing fields in datatypes, that contain navigation directives which lead to a node in the tree. For example, a directive might be 'move up to a specific child'. The advantages are that graph copying and comparing can be derived by the compiler, and it becomes easier to verify, and statically analyse programs. The use of directives, however, can make the graph type descriptions obscure, and several graph shapes cannot be expressed.

Möller and Russling (1992), and Möller (1993*a*, 1993*b*), and Russling (1994, 1995) use an algebra of formal languages and relations to model graphs. They show how

some traditional graph problems (shortest paths, cycle detection, reachability, and Hamiltonian paths) are derived using algebraic laws. Their language does not use predicate calculus (that is, quantifiers), and is therefore more compact than usual derivations (for example, the Bird-Meertens formalism).

Gibbons (1994) presents an initial-algebra approach for modelling directed acyclic graphs. Defining an initial algebra for datatypes consists of giving an object *constructors* for building larger objects, and *laws* for algebraically manipulating the objects. Gibbons's current work has a number of caveats: for example, he can only represent directed acyclic graphs, and the edges must be ordered. The notation is also quite cumbersome for representing graphs. Here's an example of a simple five vertex graph with six edges:



$$(2 \times vert_{0,3}) \fatsemi (edge \parallel ((edge \parallel swap_{1,1} \parallel edge) \fatsemi (2 \times swap_{1,1})) \parallel edge) \fatsemi (3 \times vert_{2,0})$$

Where $vert_{m,n}$ is a vertex with m incoming edges and n outgoing edges; *edge* is a directed edge; $x \parallel y$ places x beside y, but with no connection; $x \fatsemi y$ means place x before y, formed by connecting the outgoing edges of x to incoming edges of y; $m \times x$ produces m copies of x all of which are placed beside each other; $swap_{m,n}$ consists of m edges connecting the first m outgoing edges with the last m incoming edges, and connecting the last n outgoing edges with the first n incoming edges. Although there are difficulties with the work, this seems a reasonable continuation of the Bird-Meertens formalism for graphs.

Algebras for path problems

An algebra for paths to aid the derivation of path algorithms is a more plausible proposition. This approach has been explored by Backhouse and Carré (1975), Carré (1979) and Tarjan (1981), amongst others. In their approach a general algorithm for solving path problems on directed graphs is defined. Different problems are solved by using different interpretations of the operators in the path algebra. For example, the solution to a set of linear equations by Gauss-Jordan elimination may be interpreted as a version of Floyd's (1962) shortest path algorithm. A shortcoming of this work is that the emphasis is on manipulating symbols, which are written in a concise mathematical notation. Thus an insight into an algorithm is not gained.

2.1.6 Functional approaches

Some of the difficulties in the design of graph algorithms can be overcome by using a functional language. The essentials of an algorithm can be expressed without so much of the baggage (such as memory management) that is typical with an imperative language. As well as making the development easier, this provides a framework for reasoning and yields deeper algorithm insight. Some of the typical functional approaches that have been taken in the past will now be reviewed.

In Standard ML it is common practice (for example, Harrison (1993) or Paulson (1991)) to represent a graph by a list of pairs. In order to test if a vertex has been visited before, a list of all visited vertices is held, and a list membership test performed. This leads to graph traversal algorithms having a quadratic asymptotic complexity in the number of vertices $O(V^2)$. Holyer (1991), and Thompson (1995) do just the same in a lazy functional language. Wikström (1987) using Standard ML notes that the best implementations of graphs use arrays; but at the time Standard ML didn't have arrays, so he commented that balanced binary trees could be used instead. Arrays have since been added to Standard ML; but to my knowledge no one has exploited this for graphs, though their imperative nature would make it quite possible to do so. Reade (1989) working with Standard ML uses a more functional method of representation. He represents a graph by a function which computes the successors of each vertex. Again a list of visited vertices is maintained to ensure termination, so the algorithms presented are not optimal.

The parallel non-strict functional language Id (Nikhil and Arvind 1990) provides *M-structures* which are particularly well suited to express state based computations. Barth et al. (1991) show how M-structures provide a way of efficiently expressing graph traversal. An M-structure array has operations *take* and *put*. A *take* operation will either suspend if there is no value to take; or read the value and reset the position to empty. A *put* operation writes a new value to an empty (that is, taken) position. If there are suspended *take* operations when doing a *put* then the value is communicated to one of them and the array component remains empty. This M-structure array is particularly suited for holding marks to express whether a vertex has been visited before or not during a traversal. The disadvantage of using M-structures is that referential transparency can be lost. Currently a new language is being designed which combines Id's parallel evaluation strategy and features such as M-structures, with the syntax and type system of Haskell. The language is to be called *pH* which stands for parallel Haskell. It is not clear that there will be any benefits in using pH for the implementation of graph algorithms.

Meira (1985*b*) working with the functional language KRC gives three possible representations for graphs. The first is to use a list of lists where `xss!!i` represents vertices adjacent to `i`. The second is to use a list of pairs, where each pair represents an edge. The third is to represent the graph by a successor function, from vertex to its immediate neighbours in the same way as Reade (1989). For marking visited vertices he again uses a list holding visited vertices.

Launchbury (1989) using Lazy ML gave a succinct implementation of the strongly connected components algorithm of Kosaraju (unpublished), and Sharir (1981). Again this algorithm wasn't linear; but it was clear where the inefficiency lay — a membership test on a list was used to check if a node had been visited before or not.

Burton and Yang (1990) using a lazy functional language represent their graphs by heaps. The heaps are implemented with balanced binary trees. The heaps are also used for holding visited markings on vertices, which leads to having logarithmic time graph traversal. One drawback of this is that each function must take a heap and return an updated heap.

Kashiwagi and Wise (1991) express their graph algorithms in Haskell. To overcome the problem of requiring side effects they present graph algorithms as the fixed point of a set of recursive equations. The recursive equations are derived directly from the formal specification of the problem. This makes the proof of correctness of the

program almost transparent. Graphs are represented by lists, so the algorithms are not optimal. Nevertheless, unlike the usual imperative algorithms, these are suitable for parallel evaluation. Unfortunately, the algorithms presented become long and unreadable, which makes it hard to gain any insight from them.

Schoenmakers (1992) in his thesis does not cover graphs, but uses a functional notation with added pointer and array operators to explore the amortised complexity behaviour of many data structures. The imperative features such as arrays and pointers are encapsulated as much as possible by using intermediate algebras. Data structures covered are: lists, trees, skew heaps, Fibonacci heaps, and more. The emphasis is firmly on formally showing the amortised behaviour of these structures.

Hartel and Glaser (1994) implement the resource constrained shortest path problem, which is NP-complete, in the lazy functional language Intermediate. The problem is to find the shortest path in a network such that certain constraints are satisfied. They develop three variants of a solution and give a critique on the usefulness of laziness and functional programming compared to more traditional approaches to the problem.

2.2 Algorithm correctness

Algorithm correctness can be divided into two categories: *program verification*, and *program derivation*. Verification is done after the program has been written; derivation from a specification results in a program, and correctness proof. A correctness proof is a formal demonstration that a program meets its specification; not that it is guaranteed to execute correctly in all circumstances. Such a guarantee would require proving the correctness of the hardware and software used in all detail, which is currently infeasible.

Several methods of program derivation have already been briefly discussed: the Bird-Meertens formalism (Section 2.1.4); the Dijkstra calculus (Section 2.1.4); and the algebraic approach (Section 2.1.5). Program derivation can overcome many of the software development problems: hacking up the solution to a problem is a sure way of introducing subtle bugs; it is not obvious from the result, where the important design decisions were made; because the program was not written with correctness in mind, verification is extremely difficult. Program derivation is no panacea, it is

often an effort to derive the smallest of algorithms. It is not just a mechanical process: experience and skill play a big part in program derivation, just like with programming.

Program verification is more common; it is generally quicker to write a program, than to derive it formally. Books on algorithmic graph theory usually do not demonstrate correctness in all detail. Theorems are stated/proved, but with an informal connection to the algorithm. Verifying a functional algorithm is far easier because of their mathematical tractability (see Bird and Wadler (1988) for many examples of reasoning about functional programs). Furthermore, functional languages encourage styles of programming, such as modularity, which is good for both programming and proof.

2.3 Complexity analysis of algorithms

When looking at algorithms, of any sort, one of the most important topics to consider is complexity analysis. Almost every book on algorithms has a chapter, or more, on complexity: starting with Knuth (1973*a*), and continuing with Aho et al. (1983), Sedgewick (1988), Kingston (1990), and Corman et al. (1990) amongst others. These all cover the analysis of imperative algorithms, which is in many ways easier than analysing functional algorithms. This is because of the close correspondence between imperative algorithm, and the method of evaluation. Assignments, loops, conditionals, and arithmetic operations are all compiled to similar machine instructions.

The typical presentation of complexity analysis for imperative algorithms is not done in all detail. The analysis is often literate and informal, instead of mathematical. Common sense leaps are made from pseudo-code to the analysis. This is not surprising as the code is not amenable to mathematical manipulation.

Functional languages are amenable to mathematical manipulation, so showing the analysis of a functional algorithm in all detail is plausible. The mapping from a functional program to machine instructions, however, is not as direct as with imperative languages. Lazy functional languages pose further problems as the evaluation order is not fixed. Sands (1990) in his thesis developed a simple calculus for time analysis of strict functional languages; and more recently Sands (1995) has extended this for non-strict functional languages. Bjerner and Holmström (1989) also looked at the complexity of lazy functional programs, but for a first-order language. Their approach is compositional, and so requires computing information about context. This

becomes impractical for relatively simple problems, and it's not easy to see how it can be extended usefully to higher-order languages. The analysis of functional programs will be discussed further in Chapter 8, where some example calculations will be given for a functional example; a stateful example; and a lazy example.

Chapter 3

Functional algorithms

There are several advantages in expressing algorithms in a functional language. They abstract away from storage details so that little more than the essence of the algorithm is described. Formally manipulating a functional algorithm is far simpler than the imperative equivalent. Consequently, the correctness of a functional program with respect to some specification is often straightforward to prove. Typically the functional program *is* the specification. Lazy functional programs are not evaluated in a fixed (sequential) order, so the algorithm has potential for parallel evaluation. Another pleasant feature of functional languages is the way data structures can be expressed and manipulated. Dealing with lists and trees is often annoying in an imperative language, because of the explicit use of pointers.

Conversely, a high level of programming may also be considered disadvantageous. There is no easy means for expressing storage details. The language is far removed from machine instructions, so deriving the asymptotic complexity is not as easy as it may seem. The object code is generally slower than that of a conventional language. Although once the essence of the algorithm has been designed functionally, with a little effort it can be refined to a sequential imperative implementation.

This chapter looks at some functional algorithms, and demonstrates typical equational reasoning on them. The higher level of abstraction is shown to give deeper insight, as well as making equational reasoning easier.

3.1 Treesort

Treesort is a good example of an algorithm that is expressed well functionally, and a presentation can be found in many introductory texts on functional programming (Field and Harrison 1988, Reade 1989). Treesort works by building intermediate trees, the following (polymorphic) binary tree representation being typical:

```
data Tree a = Tip | Node (Tree a) a (Tree a)
```

A tree is either a tip or a node. A node has three components: a left subtree, an element, and a right subtree. A tree is ordered if at every node the element is greater than all the elements in the left subtree but less than or equal to all the elements in the right subtree.

Treesort relies on the property that flattening an ordered tree in *in-order* produces an ordered sequence. Thus, each element in the original input sequence is inserted into an ordered tree, maintaining the ordering property, and the final tree is flattened.

This is specified by the following functions. First `ins` which inserts a single element into an ordered tree.

```
ins :: Ord a => a -> Tree a -> Tree a
ins y Tip                    = Node Tip y Tip
ins y (Node l x r) | y<x  = Node (ins y l) x r
                   | y>=x = Node l x (ins y r)
```

The type for `ins` has the context `Ord a` which restricts the type `a` to objects that have a defined ordering. This function is simply extended to insert a sequence of elements into an ordered tree:

```
insSeq :: Ord a => Tree a -> [a] -> Tree a
insSeq t []     = t
insSeq t (x:xs) = insSeq (ins x t) xs
```

This function is more concisely expressed with a list fold operation, but for our purposes the above is more convenient.

Finally, flattening the tree in-order is done with the following:

```
flatten :: Tree a -> [a]
flatten Tip          = []
flatten (Node l x r) = flatten l ++ [x] ++ flatten r
```

This implementation of `flatten` is not the most efficient; it runs in $O(n \log n)$ time for the average case, whereas a linear $O(n)$ time algorithm is possible. This has no bearing on the complexity of the following implementation of treesort, since `insSeq` also runs in $O(n \log n)$ time for the average case.

```
treesort :: Ord a => [a] -> [a]
treesort xs = flatten (insSeq Tip xs)
```

3.1.1 Transforming the trees out of treesort

As well as being a good example of the abstraction power of functional languages; treesort is a good example of an algorithm that can be formally manipulated. When treesort is run many intermediate trees are created, none of which outlive the result of the sorting process. These trees may be completely removed from the algorithm by using the *unfold/fold* transformation strategy well established by Burstall and Darlington (1977). The technique works by unfolding function calls to their definitions, then the resulting expression is simplified before being folded back into function calls.

Like many of the examples given by Burstall and Darlington a key *eureka* step is needed in the simplification phase of the transformation. Eureka steps are just those transformations that an automatic system, for example, the deforestation algorithm of Wadler (1988*a*), could not invent. The following lemma describes our eureka step which expresses the property that the root of the tree acts as a pivot for the rest of the input, and that the input could be divided up into two sublists ahead of time. Throughout these transformations it is assumed that all lists and trees are finite and contain defined elements only.

Lemma 3.1
For all trees $(Node\ l\ x\ r)$ and lists xs the following holds,

$$\begin{aligned} &insSeq\ (Node\ l\ x\ r)\ xs \\ &\quad = \ Node\ (insSeq\ l\ (filter\ (<x)\ xs)) \\ &\qquad\quad x\ (insSeq\ r\ (filter\ (\geq x)\ xs)) \end{aligned}$$

Proof

The proof is by induction on the length of the list. The base case is almost immediate:

$$
\begin{aligned}
& insSeq\ (Node\ l\ x\ r)\ [\,] \\
&= \quad Node\ l\ x\ r \\
&= \quad Node\ (insSeq\ l\ [\,])\ x\ (insSeq\ r\ [\,]) \\
&= \quad Node\ (insSeq\ l\ (filter\ (<x)\ [\,]))\ x\ (insSeq\ r\ (filter\ (\geq x)\ [\,]))
\end{aligned}
$$

as $filter\ p\ [\,] = [\,]$ for all predicates p.

For the inductive case, xs is expressed as $y : ys$. First assume that $y < x$.

$$
\begin{aligned}
& insSeq\ (Node\ l\ x\ r)\ (y : ys) \\
&= \quad insSeq\ (ins\ y\ (Node\ l\ x\ r))\ ys \\
&= \quad \{\ \text{By assumption}\ \} \\
& \quad insSeq\ (Node\ (ins\ y\ l)\ x\ r)\ ys \\
&= \quad \{\ \text{Induction}\ \} \\
& \quad Node\ (insSeq\ (ins\ y\ l)\ (filter\ (<x)\ ys)) \\
& \quad\ x\ (insSeq\ r\ (filter\ (\geq x)\ ys)) \\
&= \quad Node\ (insSeq\ l\ (y : filter\ (<x)\ ys)) \\
& \quad\ x\ (insSeq\ r\ (filter\ (\geq x)\ ys)) \\
&= \quad \{\ \text{By assumption}\ \} \\
& \quad Node\ (insSeq\ l\ (filter\ (<x)\ (y : ys))) \\
& \quad\ x\ (insSeq\ r\ (filter\ (\geq x)\ (y : ys)))
\end{aligned}
$$

as required. The case when $y \geq x$ is similar. □

3.1.2 Treesort is equivalent to functional quicksort

With this lemma, the transformation of *treesort* proceeds as follows. The nil and cons cases are done separately.

Case [].

$$\begin{aligned} \mathit{treesort}\ [] &= \mathit{flatten}\ (\mathit{insSeq}\ \mathit{Tip}\ []) \\ &= \mathit{flatten}\ \mathit{Tip} \\ &= [] \end{aligned}$$

Case $(x : xs)$.

$$\begin{aligned} &\mathit{treesort}\ (x : xs) \\ &= \mathit{flatten}\ (\mathit{insSeq}\ \mathit{Tip}\ (x : xs)) \\ &= \mathit{flatten}\ (\mathit{insSeq}\ (\mathit{ins}\ x\ \mathit{Tip})\ xs) \\ &= \mathit{flatten}\ (\mathit{insSeq}\ (\mathit{Node}\ \mathit{Tip}\ x\ \mathit{Tip})\ xs) \\ &= \quad \{\text{Lemma 3.1}\} \\ &\quad \mathit{flatten}\ (\mathit{Node}\ (\mathit{insSeq}\ \mathit{Tip}\ (\mathit{filter}\ (<x)\ xs)) \\ &\quad\ x\ (\mathit{insSeq}\ \mathit{Tip}\ (\mathit{filter}\ (\geq x)\ xs))) \\ &= \mathit{flatten}\ (\mathit{insSeq}\ \mathit{Tip}\ (\mathit{filter}\ (<x)\ xs)) \\ &\quad +\!\!+ [x] \\ &\quad +\!\!+ \mathit{flatten}\ (\mathit{insSeq}\ \mathit{Tip}\ (\mathit{filter}\ (\geq x)\ xs)) \\ &= \mathit{treesort}\ (\mathit{filter}\ (<x)\ xs) \\ &\quad +\!\!+ [x] \\ &\quad +\!\!+ \mathit{treesort}\ (\mathit{filter}\ (\geq x)\ xs) \end{aligned}$$

This provides an alternative recursive definition for *treesort* with no intermediate trees. The recursion is well founded as the length of the list argument decreases with each recursive call. Written without the intermediate transformation steps yields,

```
treesort []     = []
treesort (x:xs) = treesort (filter (<x) xs)
                  ++ [x]
                  ++ treesort (filter (>=x) xs)
```

This is more well-known as functional quicksort. So these two algorithms can be considered equivalent. But aren't all sorting algorithms equivalent in the sense that they all have the same specification? Yes, of course, but the notion of equivalence is stronger here. Both treesort and functional quicksort can be considered as realisations of the same abstract algorithm. This stronger notion of equivalence can be characterised by the following theorem:

Theorem 3.2
Treesort and functional quicksort carry out the same comparisons during the sorting process.

Sketch Proof First both sorting algorithms are re-written so that they return the comparisons undertaken rather than an ordered list. For example, quicksort may be re-written:

$$
\begin{array}{lcl}
\mathit{quicksort}^c & :: & \mathit{Ord}\ a \Rightarrow [a] \rightarrow \{(a,a)\} \\
\mathit{quicksort}^c\ [\,] & = & \emptyset \\
\mathit{quicksort}^c\ (x : xs) & = & \mathit{quicksort}^c\ (\mathit{filter}\ (<x)\ xs) \\
 & & \cup \{(x,k) \mid k \in xs\} \\
 & & \cup\ \mathit{quicksort}^c\ (\mathit{filter}\ (\geq x)\ xs)
\end{array}
$$

This function is then shown to be equivalent with a function that returns the comparisons carried out during treesort. The proof is similar to the transformations carried out above. □

The notion of equivalence of sorts used here concerns comparisons, not the order in which they are performed. One may define a stronger notion of equivalence which also compares the order in which comparisons are made. However, treesort and quicksort do not perform comparisons in the same order.

The similarity between treesort and quicksort has been observed before, Hibbard (1962) showed the connection between the analysis of the two sorts. See Knuth (1973*b*) for a thorough discussion of both sorts. To the best of my knowledge no one has formally demonstrated the similarity of the two sorts. Perhaps this is because one wouldn't consider doing this with imperative code.

3.2 Functional priority queues

A crucial part of many algorithms is the data structure that is used. Frequently, the algorithm needs an abstract datatype providing a number of primitive operations on a data structure. A priority queue is one such data structure that is used by a number of algorithms. Applications include Dijkstra's (1959) algorithm for single-source shortest paths (Section 7.2), and the minimum cost spanning tree problem

(see Tarjan (1983) for a discussion of minimum spanning tree algorithms). See Knuth (1973*b*) and Aho et al. (1983) for many other applications of priority queues.

A priority queue is a set where each element has a key indicating its priority. The most common primitive operations on priority queues are:

`emptyQ`	Return the empty queue.
`isEmpty q`	Return `True` if the queue `q` is empty, otherwise return `False`.
`insertQ i q`	Insert a new item `i` into queue `q`.
`findMin q`	Return the item with minimum key in queue `q`.
`deleteMin q`	Delete the item with minimum key in queue `q`.
`meld p q`	Return the queue formed by taking the union of queues `p` and `q`.

In addition, the following two operations are occasionally useful:

`delete i q`	Delete item `i` from queue `q`.
`decreaseKey i q`	Decrease the key of item `i` in queue `q`.

There are numerous ways of implementing the abstract datatype for priority queues. Using *heap-ordered* trees is one of the most common implementations. A tree is heap-ordered if the item at every node has a smaller key than its descendants. Thus the entry at the root of a heap has the earliest priority. A variety of different trees have been used including: heaps (Knuth 1973*b*), splay trees and skew heaps (Sleator and Tarjan 1983), 2-3 trees (Aho et al. 1983). In addition, lists (sorted or unsorted) are another possible implementation of queues, but will be less efficient on large data sets. For a comparative study of these implementations and others in an imperative paradigm see Jones (1986).

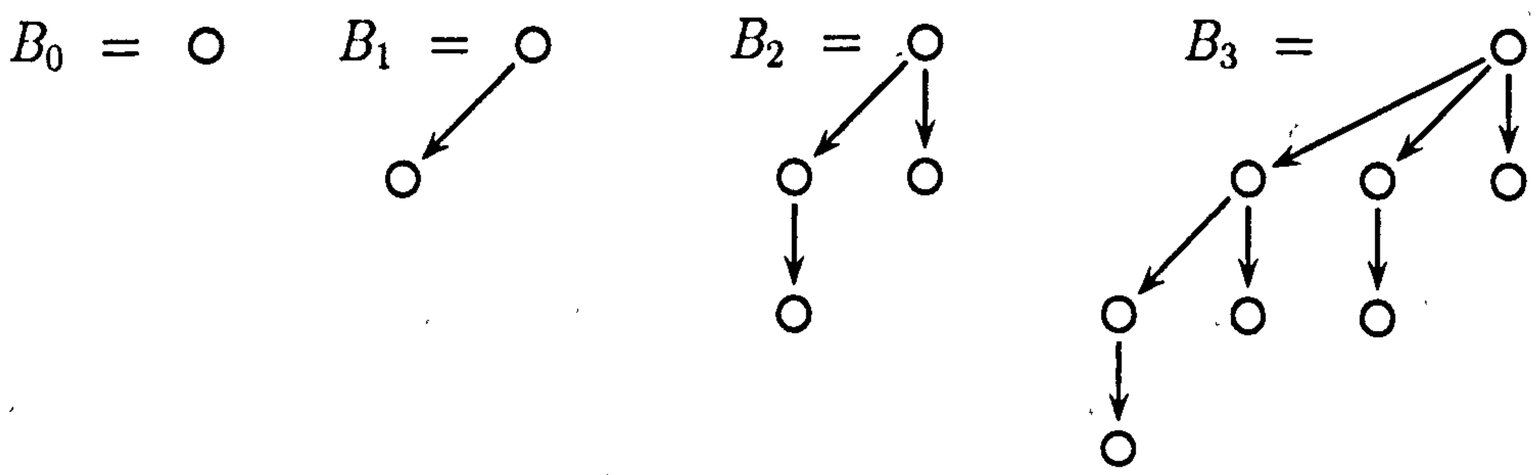
The literature on priority queues in a functional paradigm is sparse. Heaps are the most common functional implementation, see Paulson (1991), for example. Often the disadvantage of using heaps, or balanced trees, is that more bookkeeping is required for balancing. This extra bookkeeping adds to the amount of storage space needed

by the queue, as well as making the implementation of the primitives more complex. We present a functional implementation of priority queues that is based on trees, but does not require as much storage space or balancing code as other implementations. The implementation is far more elegant than a typical imperative implementation, lending itself well to a formal proof of correctness.

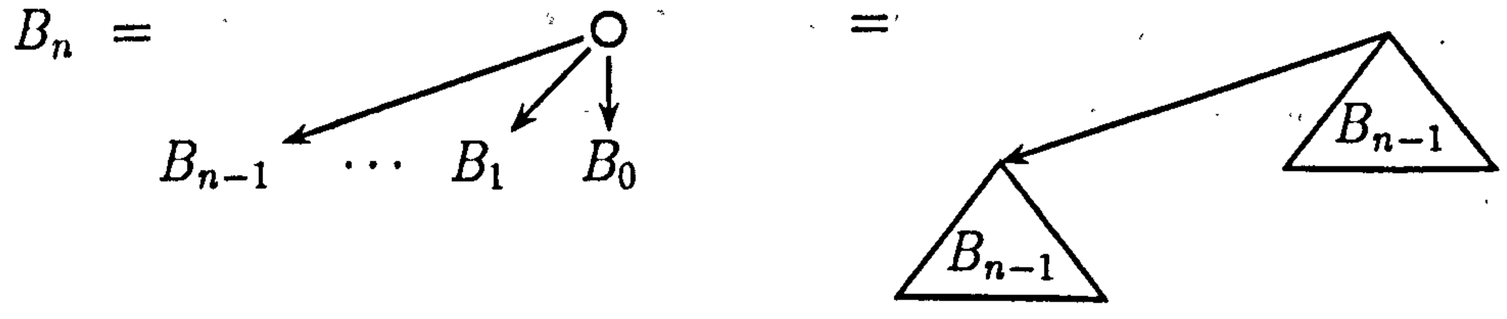
Vuillemin (1978) describes *binomial queues* which support the full complement of priority queue operations in $O(\log n)$ worst-case time. They are based on heap-ordered trees in that a priority queue is represented by a list of heap-ordered trees (that is, a forest), where each tree in the forest is a binomial tree. The remaining following sections present a purely functional implementation of binomial queues expressing the full complement of priority queue operations in Haskell.

3.3 Binomial trees

Binomial trees are general trees that have a regular shape. They are best presented diagrammatically, where circles represent nodes:



There are two equally good ways of expressing the general case, for $n > 0$.



In the rightmost picture the root of a B_{n-1} tree is linked to the root of another B_{n-1} tree, by adding it as the first child.

In Haskell a general tree may be defined with the following datatype:

```
data Tree a = Node a [Tree a]
```

Then using this datatype binomial trees can be defined inductively:

$$B_0 = Node\ x\ [\,]$$
$$B_n = Node\ x\ [\,B_{n-1}, \ldots, B_1, B_0\,], \qquad \text{for } n > 0.$$

Alternatively, the inductive case for $n > 0$ may be defined:

$$B_n = Node\ x\ (B_{n-1} : xs), \qquad \text{where} \quad Node\ x\ xs \quad \text{is a} \quad B_{n-1} \quad \text{tree.}$$

Haskell has no way of enforcing the structure for binomial queues, beyond the programmer using a predicate that verifies it. It is conceivable that a powerful type system could enforce the binomial structure. The Haskell predicate which verifies the structure is defined using the second definition of binomial queues from above.

```
isBinTree :: Int -> Tree a -> Bool
isBinTree k (Node x [])     = k == 0
isBinTree k (Node x (t:ts)) = isBinTree (k-1) t
                              && isBinTree (k-1) (Node x ts)
```

Binomial trees have some pleasing combinatorial properties. For instance, the binomial tree B_k has 2^k nodes, and $\binom{k}{d}$ nodes of depth d, hence their name. See Vuillemin (1978) and Brown (1978) for more properties.

3.4 Implementing binomial queues functionally

Vuillemin (1978) represents a priority queue with a forest of binomial trees. It is important that a list of trees is used to represent the forest because the ordering is important (a set of trees would not do). The first tree in the binomial queue must either be a B_0 tree or just *Zero* meaning no tree, and the second a B_1 tree or just *Zero*, this leads to the following structure for a binomial queue:

$$[\,T_0, T_1, \ldots, T_n\,] \qquad \text{where} \quad T_k = Zero \mid B_k, \qquad \text{for} \quad 0 \leq k \leq n.$$

Vuillemin (1978) and others use an array to represent the forest; moreover, for simplicity, binary trees are used to represent the binomial trees. Imperative implementations of linked structures of this kind usually turn out to be clumsy. Instead the primitives will be expressed as recursive functions on a list of general trees, giving a natural encoding.

So the following datatypes are used:

```
type BinQ     = [BinQTree]
data BinQTree = Zero | One (Tree Item)
```

The constructors `Zero` and `One` were chosen because the queue primitives are analogous with binary arithmetic.

Each item is a pair of entry and key:

```
type Item     = (Entry, Key)
```

Where `Key` is a type with an ordering, that is, it is an instance of the Haskell `Ord` class. The projection functions on items are:

```
entry, key :: Item -> Entry
entry = fst
key   = snd
```

The following predicates may be used to verify that a list of trees has the right structure to be a binomial queue.

```
isBinQ :: BinQ -> Bool
isBinQ q = isBinQTail 0 q


isBinQTail :: Int -> BinQ -> Bool
isBinQTail k []     = True
isBinQTail k (q:qs) = isBinQTree k q && isBinQTail (k+1) qs

isBinQTree :: Int -> BinTree -> Bool
isBinQTree k Zero    = True
isBinQTree k (One t) = isBinTree k t
```

Now we can start to express the priority queue operations. Creating a new empty queue, and testing for the empty queue follow immediately:

```
emptyQ :: BinQ
emptyQ = []

isEmpty :: BinQ -> Bool
isEmpty q = null q
```

Uniting (or melding) two queues together is the most useful of all the primitive operations, because other primitives are defined in terms of it. There is a strong analogy between queue melding and binary addition. Given the two binomial queues $[P_0, P_1, \ldots, P_n]$ and $[Q_0, Q_1, \ldots, Q_m]$ melding is carried out positionally from left to right, using the property that two B_k binomial trees can be linked into a B_{k+1} binomial tree. First P_0 is melded with Q_0, giving one of four possible results. If both P_0 and Q_0 contain trees (that is, they are not `Zero`) they are linked to form a B_1 tree so that the heap-order property is maintained. With just one tree and one `Zero` the result is the tree, and given two `Zero`'s the result is `Zero`. This process of linking is carried out on successive trees. If the result of melding P_k with Q_k results in a B_{k+1} tree then this is carried on (analogous to a carry bit in binary arithmetic) and melded with P_{k+1} and Q_{k+1}.

```
meldC :: BinQ -> BinQ -> BinQTree -> BinQ
meldC [] qs Zero       = qs
meldC [] qs c          = meld [c] qs
meldC ps [] c          = meldC [] ps c
meldC (p:ps) (q:qs) c = sum:meldC ps qs c'
                 where (sum,c') = addC p q c

addC :: BinQTree -> BinQTree -> BinQTree -> (BinQTree,BinQTree)
addC Zero Zero c                                  = (c,Zero)
addC (One (Node x xs)) (One (Node y ys)) c = (c,One t)
      where t | key x < key y = Node x (Node y ys: xs)
              | otherwise     = Node y (Node x xs: ys)
addC p q c                                        = addC q c p
```

```
meld :: BinQ -> BinQ -> BinQ
meld p q = meldC p q Zero
```

Points to note about this definition of `meld` are: that the third argument to `meldC` behaves like a carry; and the function `meld` calls `meldC` with the initial carry of `Zero`; also the third case in `addC` rotates the arguments until the first two are in the same form.

The asymptotic complexity of `meld` is $O(\log n)$ (where n is the number of items in the larger queue). We arrive at this by observing that two B_k trees can be linked in constant time, and the number of these linking operations will be equal to the size of the longest queue, that is $O(\log n)$. For a more detailed analysis of the complexity of meld and the other queue operations see Brown (1978).

Inserting an item into the queue is expressed by melding a B_0 tree holding the item, into the binomial queue.

```
insertQ :: Item -> BinQ -> BinQ
insertQ i qs = meld [One (Node i [])] qs
```

```
insertMany :: [Item] -> BinQ
insertMany is = foldr insertQ [] is
```

Since each binomial tree is heap-ordered the item with the minimum key will be a root of one of the trees. This is found by scanning the list of trees. The item with minimum key is deleted by first extracting the tree that it is the root of, then melding the subtrees back into the binomial queue. This melding is easy as the subtrees themselves form a binomial queue, in reverse order. These subtrees could be stored in this order, which would save doing a reversal, but this would make `meld` slightly more difficult to define.

First the forest is traversed returning the required tree, and replacing it with a `Zero`.

```
removeMinT :: BinQ -> (BinQTree, BinQ)
removeMinT [t]                  = (t,[])
removeMinT (t:ts) | t<mt        = (t,Zero:ts)
                  | otherwise = (mt,t:mts)
      where  (mt,mts) = removeMinT ts
```

The ordering on binomial queue trees used here is defined:

```
instance Ord BinQTree where
      Zero          < t             = False
      t             < Zero          = True
      One (Node x xs) < One (Node y ys) = key x < key y
```

After this the subtrees of the extracted tree are melded back into the queue:

```
deleteMin :: BinQ -> BinQ
deleteMin qs = meld (map One (reverse ts)) qs'
      where
        (One (Node i ts), qs') = removeMinT qs
```

If two tree roots have the same key value, then the latest one occurring in the list is chosen by `removeMinT`.

The total running time of `removeMinT` is $O(\log n)$, since it traverses a list of length $\log n$ carrying out constant time operations. The `deleteMin` operation carries out a meld, as well as `removeMinT`. Since the subtrees being melding back into the queue are smaller, the melding will take $O(\log n)$ time. Hence `deleteMin` will run in $O(\log n)$ time.

The function `removeMinT` may also be used to express `findMin` which again runs in $O(\log n)$ time.

```
findMin :: BinQ -> Item
findMin q = i
      where
        One (Node i ts) = fst (removeMinT q)
```

The two pass algorithm for `deleteMin` can be performed in one pass over the binomial queue (giving a constant time speed-up) by using the standard cyclic programming technique, see Bird (1984*b*). A function is used that both takes the item to be removed as an argument and returns the item with minimum key, as well as the binomial queue without the item. As usual, the efficient algorithm has a more cumbersome implementation, and so is omitted here.

3.5 Correctness of functional binomial queues

To show the correctness of the primitive operations, three properties must be shown: (i) that the binomial queue structure is maintained; (ii) the heap-ordering property is maintained; and (iii) that the primitives satisfy their specification. We may show that the queue primitives maintain the binomial queue structure by using the previously defined functions `isBinQ`, `isBinTree`, `isBinQTail`, and `isBinQTree`. Here we show that `meld` maintains the queue structure, by first proving a property about `meldC`.

3.5.1 Meld maintains the binomial queue structure

Theorem 3.3 (`meld` maintains the binomial queue structure)
After a meld operation the resulting queue is a binomial queue, if and only if meld is given two binomial queues.

$$\forall p, q \;.\; isBinQ\ (meld\ p\ q) \iff isBinQ\ p \wedge isBinQ\ q$$

Proof

Using Lemma 3.4, instantiating n with 0, and c with *Zero*. □

Lemma 3.4 (*meldC*)
For all $n \geq 0$, and assuming that *ps*, *qs* and *c* are well-defined,

$$\forall ps, qs, c \;.\; isBinQTail\ n\ (meldC\ ps\ qs\ c)$$
$$\iff isBinQTail\ n\ ps \wedge isBinQTail\ n\ qs \wedge isBinQTree\ n\ c$$

Proof By induction. If *length ps* = *length qs* there would be fewer cases to show. Furthermore, the melding implementation could be changed so that these lists are always equal in size by appending `Zeros` onto the end of the shorter list. This would make the program less efficient, however, and the extra code would be superfluous. Here we show the correctness of the actual implementation.

Case $ps = [\,]$, $qs = [\,]$, any c.

$$
\begin{aligned}
& \mathit{isBinQTail}\ n\ (\mathit{meldC}\ [\,]\ [\,]\ c) \\
\iff & \quad \left\{ \mathit{meldC} \text{ showing for } c = \mathit{Zero} \text{ and } c = \mathit{One}\ t \right\} \\
& \quad \mathit{isBinQTail}\ n\ (\mathit{meldC}\ [\,]\ [\,]\ \mathit{Zero}) \\
& \quad \wedge\ \mathit{isBinQTail}\ n\ (\mathit{meldC}\ [\,]\ [\,]\ (\mathit{One}\ t)) \\
\iff & \quad \left\{ \mathit{meldC} \right\} \\
& \quad \mathit{isBinQTail}\ n\ [\,]\ \wedge\ \mathit{isBinQTail}\ n\ (\mathit{meldC}\ [\,\mathit{One}\ t\,]\ [\,]\ \mathit{Zero}) \\
\iff & \quad \left\{ \mathit{meldC} \right\} \\
& \quad \mathit{isBinQTail}\ n\ [\,]\ \wedge\ \mathit{isBinQTail}\ n\ [\,\mathit{One}\ t\,] \\
\iff & \quad \left\{ \mathit{isBinQTail},\ \mathit{isBinQTree} \right\} \\
& \quad \mathit{isBinQTail}\ n\ [\,]\ \wedge\ \mathit{isBinQTail}\ n\ [\,]\ \wedge\ \mathit{isBinQTree}\ n\ (\mathit{One}\ t) \\
\iff & \quad \left\{ \text{As } \mathit{isBinQTree}\ n\ \mathit{Zero} \text{ is true for } n \geq 0 \right\} \\
& \quad \mathit{isBinQTail}\ n\ [\,]\ \wedge\ \mathit{isBinQTail}\ n\ [\,]\ \wedge\ \mathit{isBinQTree}\ n\ c
\end{aligned}
$$

Case $ps = [\,]$, $(q : qs)$, any c.

This is shown by induction on the length of qs, that is, we assume for qs and show for $(q : qs)$. The previous case is the base case for the induction.

$$
\begin{aligned}
& \mathit{isBinQTail}\ n\ (\mathit{meldC}\ [\,]\ (q : qs)\ c) \\
\iff & \quad \left\{ \mathit{meldC} \text{ showing for } c = \mathit{Zero} \text{ and } c = \mathit{One}\ t \right\} \\
& \quad \mathit{isBinQTail}\ n\ (\mathit{meldC}\ [\,]\ (q : qs)\ \mathit{Zero}) \\
& \quad \wedge\ \mathit{isBinQTail}\ n\ (\mathit{meldC}\ [\,]\ (q : qs)\ (\mathit{One}\ t)) \\
\iff & \quad \left\{ \mathit{meldC} \right\} \\
& \quad \mathit{isBinQTail}\ n\ (q : qs) \\
& \quad \wedge\ \mathit{isBinQTail}\ n\ (\mathit{meldC}\ [\,\mathit{One}\ t\,]\ (q : qs)\ \mathit{Zero}) \\
\iff & \quad \left\{ \mathit{meldC} \right\} \\
& \quad \mathit{isBinQTail}\ n\ (q : qs)\ \wedge\ \mathit{isBinQTail}\ n\ (\mathit{sum} : \mathit{meldC}\ [\,]\ qs\ c') \\
& \quad \wedge\ (\mathit{sum}, c') = \mathit{addC}\ (\mathit{One}\ t)\ q\ \mathit{Zero} \\
\iff & \quad \left\{ \mathit{isBinQTail} \right\} \\
& \quad \mathit{isBinQTail}\ n\ (q : qs)\ \wedge\ \mathit{isBinQTree}\ n\ \mathit{sum} \\
& \quad \wedge\ \mathit{isBinQTail}\ (n+1)\ (\mathit{meldC}\ [\,]\ qs\ c') \\
& \quad \wedge\ (\mathit{sum}, c') = \mathit{addC}\ (\mathit{One}\ t)\ q\ \mathit{Zero}
\end{aligned}
$$

$$
\begin{aligned}
&\iff \quad \{\text{Inductive hypothesis}\} \\
&\quad isBinQTail\ n\ (q : qs)\ \wedge\ isBinQTree\ n\ sum \\
&\quad \wedge\ isBinQTail\ (n+1)\ []\ \wedge\ isBinQTail\ (n+1)\ qs \\
&\quad \wedge\ isBinQTree\ (n+1)\ c'\ \wedge\ (sum, c') = addC\ (One\ t)\ q\ Zero \\
&\iff \quad \{\text{Lemma 3.5}\} \\
&\quad isBinQTail\ n\ (q : qs)\ \wedge\ isBinQTail\ (n+1)\ [] \\
&\quad \wedge\ isBinQTail\ (n+1)\ qs\ \wedge\ isBinQTree\ n\ (One\ t) \\
&\quad \wedge\ isBinQTree\ n\ q\ \wedge\ isBinQTree\ n\ Zero \\
&\iff \quad \{\ isBinQTail,\ isBinQTree,\ c = Zero \text{ or } c = One\ t\ \} \\
&\quad isBinQTail\ n\ []\ \wedge\ isBinQTail\ n\ (q : qs)\ \wedge\ isBinQTree\ n\ c
\end{aligned}
$$

Case $(p : ps)$, $qs = []$, any c.

$$
\begin{aligned}
&isBinQTail\ n\ (meldC\ (p : ps)\ []\ c) \\
&\iff \quad \{meldC\} \\
&\quad isBinQTail\ n\ (meldC\ []\ (p : ps)\ c) \\
&\iff \quad \{\text{Previous case}\} \\
&\quad isBinQTail\ n\ (p : ps)\ \wedge\ isBinQTail\ n\ []\ \wedge\ isBinQTree\ n\ c
\end{aligned}
$$

Case $(p : ps)$, $(q : qs)$, any c.

This case is also shown by induction, but this time on the length of ps and qs simultaneously. that is. assume for ps and qs and show for $(p : ps)$ and $(q : qs)$. The cases shown above are the base cases.

$$
\begin{aligned}
&isBinQTail\ n\ (meldC\ (p : ps)\ (q : qs)\ c) \\
&\iff \quad \{meldC\} \\
&\quad isBinQTail\ n\ (sum : meldC\ ps\ qs\ c')\ \wedge\ (sum, c') = addC\ p\ q\ c \\
&\iff \quad \{isBinQTail\} \\
&\quad isBinQTree\ n\ sum\ \wedge\ isBinQTail\ (n+1)\ (meldC\ ps\ qs\ c') \\
&\quad \wedge\ (sum, c') = addC\ p\ q\ c
\end{aligned}
$$

$$
\begin{aligned}
\iff & \quad \left\{ \text{Inductive hypothesis} \right\} \\
& \quad isBinQTree\ n\ sum\ \wedge\ isBinQTail\ (n+1)\ ps \\
& \quad \wedge\ isBinQTail\ (n+1)\ qs\ \wedge\ isBinQTree\ (n+1)\ c' \\
& \quad \wedge\ (sum, c') = addC\ p\ q\ c \\
\iff & \quad \left\{ \text{Lemma 3.5} \right\} \\
& \quad isBinQTree\ n\ p\ \wedge\ isBinQTail\ (n+1)\ ps \\
& \quad \wedge\ isBinQTree\ n\ q\ \wedge\ isBinQTail\ (n+1)\ qs\ \wedge\ isBinQTree\ n\ c \\
\iff & \quad \left\{ isBinQTail \right\} \\
& \quad isBinQTail\ n\ (p : ps)\ \wedge\ isBinQTail\ n\ (q : qs)\ \wedge\ isBinQTree\ n\ c
\end{aligned}
$$

□

In the original implementation the function `addC` was not used. It was only later when `meld` was verified that it was introduced as a means of simplifying the proof. Splitting the function in two does make the algorithm more understandable; hence formally proving a program, is not only useful in convincing us that it works, but can improve the program.

Lemma 3.5 (`addC` maintains binomial tree structure)
For all $n \geq 0$, and assuming that p, q, and c are well defined,

$$
\begin{aligned}
\forall p, q, c\ .\ & isBinQTree\ n\ sum\ \wedge\ isBinQTree\ (n+1)\ c' \\
& \wedge\ (sum, c') = addC\ p\ q\ c \\
\iff & isBinQTree\ n\ c\ \wedge\ isBinQTree\ n\ p\ \wedge\ isBinQTree\ n\ q
\end{aligned}
$$

Proof

By case analysis on p, q, and c. There are several cases in the proof, the most interesting being where p and q both contain trees.

Case $p = One\ (Node\ x\ xs)$, $q = One\ (Node\ y\ ys)$, any c.

$$
\begin{aligned}
& isBinQTail\ n\ (addC\ (One\ (Node\ x\ xs))\ (One\ (Node\ y\ ys))) \\
& \wedge\ (sum, c') = addC\ (One\ (Node\ x\ xs))\ (One\ (Node\ y\ ys))\ c \\
\Longleftrightarrow \quad & \left\{ addC \right\} \\
& \quad isBinQTree\ n\ c \\
& \quad \wedge\ (isBinQTree\ (n+1)\ (One\ (Node\ x\ (Node\ y\ ys : xs))) \\
& \quad \vee\ isBinQTree\ (n+1)\ (One\ (Node\ y\ (Node\ x\ xs : ys)))) \\
\Longleftrightarrow \quad & \left\{ isBinQTree, isBinTree \right\} \\
& isBinQTree\ n\ c\ \wedge\ isBinQTree\ n\ (One\ (Node\ x\ xs)) \\
& \wedge\ isBinQTree\ n\ (One\ (Node\ y\ ys))
\end{aligned}
$$

Completing the case. The other cases are similar. □

3.5.2 Meld maintains the heap-ordering property

A tree is heap-ordered if the item at every node has a smaller key than its descendants. Formally, a general tree t is heap-ordered if and only if:

$$\forall x, y \in t\ .\ x \neq y\ \wedge\ key\ x < key\ y\ \wedge\ x \longrightarrow_t y$$

where $x \longrightarrow_t y$ represents a path in t from x to y (a path consists of zero or more edges). The heap-ordering predicate may be expressed with the following recursive function:

```
heapOrdT :: Tree Item -> Bool
heapOrdT (Node x [])                = True
heapOrdT (Node x (Node y ys: xs)) = key x < key y
                                    && heapOrdT (Node x xs)
                                    && heapOrdT (Node y ys)
```

Since we are dealing with binomial trees it is convenient to use a function that deals directly with them:

```
heapOrd :: BinQTree -> Bool
heapOrd Zero     = True
heapOrd (One t) = heapOrdT t
```

This function is then generalised for binomial queues, by using the following recursive definition (it is slightly easier for proof purposes to use this definition rather than a higher-order one).

```
heapOrdQ :: BinQ -> Bool
heapOrdQ []     = True
heapOrdQ (t:ts) = heapOrd t && heapOrdQ ts
```

Theorem 3.6 (Heap-ordering property of `meld`)
Assuming that p, and q are well defined then:

$$\forall ps, qs \;.\; heapOrdQ\ (meld\ ps\ qs) \iff heapOrdQ\ ps \;\wedge\; heapOrdQ\ qs$$

Sketch Proof

The proof follows the same course of procedure as the proof for Theorem 3.3 and relies upon the following lemma for `addC`. □

Lemma 3.7 (Heap-ordering property of `addC`)
Assuming that p, q, and c are well defined:

$$\begin{aligned}\forall p, q, c \;.\; heapOrd\ sum \;\wedge\; heapOrd\ c' \;\wedge\; (sum, c') = addC\ p\ q\ c \\ \iff heapOrd\ p \;\wedge\; heapOrd\ q \;\wedge\; heapOrd\ c\end{aligned}$$

Proof

By case analysis. There are several cases to consider, but here only the most interesting case will be given.

Case $p = One\ p'$, $q = One\ q'$, any c, where $p' = Node\ x\ xs$ and $q' = Node\ y\ ys$.

$$
\begin{aligned}
& heapOrd\ sum\ \wedge\ heapOrd\ c'\ \wedge\ (sum, c') = addC\ (One\ p')\ (One\ q')\ c \\
\iff\quad & \left\{ addC \right\} \\
& \quad heapOrd\ sum\ \wedge\ heapOrd\ c'\ \wedge\ (sum, c') = (c, One\ t) \\
& \quad \wedge\ ((t = Node\ x\ (q' : xs)\ \wedge\ key\ x < key\ y) \\
& \quad \vee\ (t = Node\ y\ (p' : ys)\ \wedge\ key\ x \geq key\ y)) \\
\iff\quad & \left\{ \text{Substituting for } sum \text{ and } c' \right\} \\
& \quad heapOrd\ c \\
& \quad \wedge\ ((heapOrd\ (One\ (Node\ x\ (q' : xs)))\ \wedge\ key\ x < key\ y) \\
& \quad \vee\ (heapOrd\ (One\ (Node\ y\ (p' : ys)))\ \wedge\ key\ x \geq key\ y)) \\
\iff\quad & \left\{ heapOrd, heapOrdT \right\} \\
& \quad heapOrd\ c \\
& \quad \wedge\ ((key\ x < key\ y\ \wedge\ heapOrdT\ p'\ \wedge\ heapOrdT\ q') \\
& \quad \vee\ (key\ y \leq key\ x\ \wedge\ heapOrdT\ q'\ \wedge\ heapOrdT\ p')) \\
\iff\quad & heapOrd\ c\ \wedge\ heapOrdT\ p'\ \wedge\ heapOrdT\ q' \\
\iff\quad & heapOrd\ (One\ p')\ \wedge\ heapOrd\ (One\ q')\ \wedge\ heapOrd\ c
\end{aligned}
$$

Establishing the case. The other cases are no more difficult than this one. □

3.5.3 Meld meets its specification

The third property required to show the correctness of *meld* is that it does a real union of elements:

Theorem 3.8
Assuming that x, p, and q are well defined then:

$$\forall x, p, q\ .\ x \in meld\ p\ q \iff x \in p\ \vee\ x \in q$$

Sketch Proof The proof follows the same structure as the proof for Theorem 3.3 (the proof that `meld` returns a binomial queue structure). Similarly, a lemma that `addC` does a true addition of trees is shown by case analysis. □

3.6 Implementing decreaseKey and delete

The usual way in imperative languages to implement `decreaseKey` and `delete` is to maintain an auxiliary data structure which supports direct access, in constant time, to each item. Usually this is specified by having pointers into the middle of the queue, but this is awkward in a functional setting. One way to achieve a reasonable complexity whilst remaining purely functional is to maintain a set of all items currently in the queue. Instead of physically removing the item from the queue, it is just removed from the set. So binomial queues are extended to a pair containing the queue and a set.

```
type BinQExt = (BinQ, Set Entry)
```

All the priority queue operations must do some extra bookkeeping to maintain the set.

```
emptyPQ :: BinQExt
emptyPQ = (emptyQ, emptySet)

isEmptyPQ :: BinQExt -> Bool
isEmptyPQ (q,s) = isEmptySet s
```

When inserting a new item, it must be inserted into the set. Similarly, when two queues are melded, the union of their sets must be taken:

```
insertPQ :: Item -> BinQExt -> BinQExt
insertPQ i (q,s) = (insertQ i q, insSet (entry i) s)


meldPQ :: BinQExt -> BinQExt -> BinQExt
meldPQ (p,s) (q,t) = (meld p q, unionSet s t)
```

Deleting the minimum item must also delete it from the set:

```
deleteMinPQ :: BinQExt -> BinQExt
deleteMinPQ (p,s) | not (isEmptySet s) = (q, delSet (entry i) s)
      where
        (i,q) = (findMin p, deleteMin p)
```

The `findMinPQ` operation makes no change to the set and is just expressed in terms of `findMin`. When decreasing the key for an item, the item is re-inserted into the queue with its new key. When deleting an item it is removed from the set.

```
decreaseKey :: Item -> BinQExt -> BinQExt
decreaseKey i pq | (entry i) `elemSet` (snd pq) = insertPQ i pq
                 | otherwise                    = pq

delete :: Item -> BinQExt -> BinQExt
delete i (q,s) = (q, delSet (entry i) s)
```

Of course, maintaining a set has an impact on the time and space complexity of the priority queue operations. The set operations may be implemented with balanced trees for a reasonable complexity. The running times of `decreaseKey` and `delete` is $O(\log n)$, both running times being dominated by the set operations. The other operations have the same worst-case complexity as before $O(\log n)$, except `meldPQ` which is now dominated by the complexity of the set union operation $O(n+m)$ (where n and m are the sizes of the two sets). Furthermore, because items are never physically removed from the queue the complexity of the operations is governed by the total number of inserts made. Constant factors may be improved by doing some garbage collection, that is, physically removing items that percolate to the roots of trees.

3.7 Comparison with other priority queues

In an imperative language binomial queues perform better than most other priority queue implementations, see Jones (1986) for an empirical comparison. More recently Fredman and Tarjan (1987) have developed Fibonacci heaps which are based on binomial queues. Fibonacci heaps have a better amortised complexity for many of the operations. Unfortunately, they make heavy usage of pointers, so do not lend themselves to a natural functional encoding.

The usual functional implementation of priority queues is to use heaps, see Paulson (1991), for example. The advantage of binomial queues over heaps is that the meld operation is more efficient (Table 3.1). Jones (1986) reports that in an imperative setting, binomial queues are one of the most complex implementations. In Haskell

Queue	insertQ		deleteMin		meld	
	Lines	Time	Lines	Time	Lines	Time
Binomial	1	$O(\log n)$	6	$O(\log n)$	11	$O(\log n)$
2-3 trees	16	$O(\log n)$	41	$O(\log n)$	26	$O(n)$
Sorted list	5	$O(n)$	1	$O(1)$	6	$O(n)$
Heaps	7	$O(\log n)$	17	$O(\log n)$	21	$O(n)$

Table 3.1 Differences between some Haskell implementations of priority queues.

the operations on tree and list data structures are far cleaner than in an imperative language. Functionally, binomial queues are in many ways more elegant than heaps. They are easier to program and understand, as well as being programmed in fewer lines of code. Similarly binomial queues have the same advantages over 2-3 trees, see Reade (1992) for a functional implementation of 2-3 trees, and Aho et al. (1983) for a description of how they may be used for implementing priority queues. Sorted lists are the simplest of all implementations, and give the best performance for small queues. In spite of this, they have the worst complexity, and will give slower running times for larger queues. Table 3.1 summarises the running times and lines of Haskell code for four different implementations. It should be noted that the asymptotic complexities for the binomial queue operations are all worst-case times. Okasaki (1996) has shown the implementation of binomial queue insertion given here runs in $O(1)$ amortised time.

Independently Fourman (1994) gives a similar implementation of Vuillemin's queues in Standard ML. Unpublished work by Brodal and Okasaki (1995) give a purely functional implementation of optimal priority queues. This together with some other purely functional implementation techniques for data structures are summarised in Table 3.2.

Data structure/author	Description
Queues (Hood and Melville 1981, Gries 1981, Burton 1982)	The queue is represented by a pair of lists (xs, ys) where xs is the front of the queue and ys is the end of the queue in reverse order.
Deques Chuang and Goldberg (1993)	The implementation uses a pair of lists together with their lengths. The heads of the lists represent the two ends of the deque, and the length information is used to achieve a balanced structure.
Deques Okasaki (1994)	The implementation uses a quadruple $(xs, ys, \hat{xs}, \hat{ys})$, where xs and ys are as with queues, and $\hat{xs}$ and $\hat{ys}$ are the tails of xs and ys, indicating which portions of xs and ys have been pre-evaluated. The reverse list operation is done incrementally with laziness. All operations run in $O(1)$ worst-case time.
Priority queues Brodal and Okasaki (1995)	The implementation is an extension of binomial queues. The `findMin` operation is improved to $O(1)$ by maintaining a global root; `insertQ` is improved to $O(1)$ by eliminating cascading carries; and finally `meld` is improved to $O(1)$ by allowing priority queues to contain other priority queues.
Sets Reade (1992) Adams (1993)	There are several good implementations of sets, all of which use a tree data structure. For efficiency, balanced trees are used, for instance, Reade uses 2-3 trees, and Adams uses balanced binary trees.

Table 3.2 Summary of some purely functional data structures.

Chapter 4

Stateful algorithms

A *stateful* algorithm is one in which access is made to the state. Conventional imperative algorithms are stateful, but purely functional algorithms are not. Many of the advantages of functional algorithms come from not having access to the state, however, some algorithms seem inherently to require access to the state in order to reduce their complexity.

This chapter describes the monad of state transformers and with it introduces mutable arrays and mutable variables. This is not new; the approach taken follows closely the work of Launchbury and Peyton Jones (1994, 1996). One difference is the use of the `do` notation to express stateful algorithms. The examples were chosen to illustrate the use of mutable arrays and mutable variables, and because they are useful for later graph algorithms. The chapter finishes with a discussion on the merits and otherwise of the imperative functional approach compared with a traditional imperative approach.

4.1 The need for state

Once we move to data structures that *explicitly require sharing* to achieve an efficient implementation, then the purely functional world becomes less appealing. In this type of structure, the ability of local actions to make global changes on the structure becomes vital. For example, the operations of a deque (double-ended queue) are usually implemented with doubly-linked lists (Knuth 1973*a*), and this method of implementation cannot easily be mimicked in the purely functional world.

Sometimes, as in the case of deques, functional solutions exist. For example, Chuang

and Goldberg (1993) and Okasaki (1994) both give purely functional implementations of deques, see Table 3.2. While such applicative methods are important (and to a wider community than just functional programmers), they can be extremely devious or complex, and there are still a number of problems that have been resistant to efficient functional solutions. Ponder et al. (1988) describe seven such problems, including RAM simulation.

4.2 Including imperative actions in a functional language

Functional languages like Standard ML and Scheme have allowed imperative actions since their conception. In both languages destructive updates can occur as a side effect of evaluation. This forces the evaluation order to be fixed and statically determined (otherwise the program's meaning becomes hard to predict). Amongst other things, this rules out lazy evaluation, or even opportunities for parallel evaluation.

Over the last few years many people have explored various methods of including imperative features in functional languages, culminating in the monadic approach advocated in turn by Moggi (1989), Wadler (1990*a*, 1992), Peyton Jones and Wadler (1993), Launchbury (1993), and Launchbury and Peyton Jones (1994, 1996). This approach has a clear semantics, and can be cleanly combined with lazy functional languages such as Haskell.

In the monadic approach that is explored here, imperative actions are specified as state-transforming functions. In one sense, therefore, adding imperative features provides nothing new: every program presented in this thesis can be written in any purely functional program by simulating the state. The only thing that the imperative features provide is a possible improvement of the complexity of the implementation. That is, rather than representing state changes by replacement of one value with a completely fresh one, true destructive update is used. On the other hand, the laws for reasoning are just those that would be required if the purely functional specification of state provided here were used in reality.

4.3 State transformers

Imperative actions are specified by using (purely functional) state transformers, except that the state argument is implemented by destructive update in the underlying state. For the purposes of this thesis, the implementation of state will be ignored, for details see Launchbury (1993).

State transformers should be viewed as an abstract type defined as follows.

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

return :: a -> ST s a
return 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
```

Elements of type `ST` are functions which, when given a state, produce a value together with a new state. These may be sequenced together using `thenST`. The state argument `s` is given to `m` which produces a value `a` and a new state `t`. These are both passed to `k`, and the result that `k` produces is the result of the whole thing. The function `return` turns a value into a trivial state transformer.

4.3.1 The do notation

For conciseness and clarity the following syntax will be used for sequences of state transformers. Haskell is extended with the syntactic form `do Q`, first used in Launchbury (1993), and implemented in Gofer 2.30 (Jones 1994). The keyword `do` introduces layout, so the following braces and semi-colons can be omitted and inferred automatically (just like in `where` and `case` clauses). Nevertheless, braces and semi-colons will be retained here, to prevent confusion.

```
Q = E                    ⟹    E
  | E ; Q                ⟹    E `thenST` \_ -> do Q
  | P <- E ; Q           ⟹    E `thenST` \P -> do Q
  | let D in ; Q         ⟹    let D in do Q
```

So, for example, we might write the code:

```
.. do { x <- action1;
        let y = x*x in
        action2 (2+y);
        action3 y
      }
```

which expands to:

```
.. (action1 `thenST` (\x->
    let y = x*x in
    action2 (2+y) `thenST` (\_->
    action3 y)))
```

What we have so far only allows us to build state transformers. We have not seen how to apply them to an actual state (that is to run them). Recall that the type `ST` is intended to be abstract, so the programmer cannot merely apply it to a state argument. Hence, for running state threads we use:

```
runST :: (∀s.ST s a) -> a
```

This function takes a state transformer, applies it to an initial (theoretically empty) state, and returns the final value, discarding the state. The type of `runST` is not a Hindley-Milner type, so `runST` must be built in as a language construct. The nested quantifier is sufficient to ensure that the state transformer does not attempt to dereference variables allocated in other, independent, state threads (that is, no *segmentation faults*). See Launchbury and Peyton Jones (1994, 1996) for details.

4.4 Variables

Variables are references into the state. The reference itself is unchanging and unchangeable. The state to which it refers, however, is subject to change by state transformers.

Mutable variables come with the following operations:

```
newVar   :: a -> ST s (MutVar s a)
readVar  :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
(==)     :: MutVar s a -> MutVar s a -> Bool
```

The function `newVar` is a state transformer which creates a new variable, initialises it, and returns a reference to it. The reference type `MutVar` is abstract. The only operations defined on it are those listed above.

Reference types record not only the type of value they store, but also the state in which they were created. This works together with the type of `runST` to allow the typechecker to guarantee that references are only dereferenced in the state thread in which they were created (again, see Launchbury and Peyton Jones (1994, 1996) for the details).

The function `readVar` is used to extract the value of a variable and `writeVar` to assign a new value to a variable. Variables are compared for equality of their values with the overloaded `(==)` operator (that is, `MutVar` is made a member of the equality class Eq).

To see this in action, consider the following procedure `becomes`. It is a polymorphic copying function for variables, which reads the value of its second argument and writes it into the location referenced by its first argument.

```
becomes :: MutVar s a -> MutVar s a -> ST s ()
v `becomes` w = do { val <- readVar w;
                     writeVar v val
                   }
```

The state transformer returns no value of interest, indicated by the type `()`. An example of the use of `becomes` will be given in the next section.

4.5 Explicitly linked lists

Mutable variables can be used to implement explicitly linked lists, that is lists whose links may be changed at will. One way of doing this is to define the following:

```
type LinkedList s a = MutVar s (Link s a)
data Link s a = Nil
              | Item a (LinkedList s a)
```

A linked list is a variable which stores a link. A link is either `Nil`, representing an empty list, or it is an `Item` containing two components: the element stored at this point in the list, and a linked list tail. The definition is like the usual recursive definition of lists except for two aspects. First, the tail of the `Item` node is a *variable in which another item is stored*, rather than the item itself. Second, the type contains an explicit state parameter indicating the presence of state references.

A recursive procedure for (destructively) appending two such lists could be defined as follows.

```
appendL :: LinkedList s a -> LinkedList s a -> ST s ()
appendL v w = do { xs <- readVar v;
                   case xs of
                     Nil      -> v `becomes` w
                     Item x u -> appendL u w
                 }
```

From the type, we see that `appendL` takes two linked lists that (a) must both be in the same state thread, and (b) must contain elements of the same type. Given two such lists, `appendL` returns a state transformer which returns no interesting value — its behaviour is in the state transformations it would enact. That is, it is a procedure.

Similarly, accessing functions `headL` and `tailL` can be defined, the latter destructively chops off the front of the list.

```
headL :: LinkedList s a -> ST s a
headL v = do { Item x w <- readVar v;
               return x
             }

tailL :: LinkedList s a -> ST s ()
tailL v = do { Item x w <- readVar v;
               v `becomes` w
             }
```

The `Link` type provides a pointer-like capability. The variable may contain only `Nil`, or it may contain something more interesting, namely an item with its components.

4.5.1 Queues

Queues are a traditional application of linked lists. They are often implemented using a linked list, together with a pointer to the end of the list to allow for constant time update.

It has been shown by Hood and Melville (1981) that the queue operations can be implemented efficiently without using pointers. This is done by maintaining a pair of lists which contain an initial segment of the queue, and the remaining segment reversed. The head of the reversed segment contains the last item in the queue, therefore it can be accessed in constant time. The amortised time complexity for the complement of queue operations is $O(1)$. This is only an amortised complexity because every so often the reversed segment will become empty, and the other segment will become the reversed segment after a reversal. Okasaki (1994) achieves an $O(1)$ worst case time complexity for the queue operations by using an incremental approach which exploits lazy lists.

While there is no necessity to implement a queue with explicit pointer operations, this will be done here for illustrative purposes. Once it is clear how to express one data structure with pointers it is relatively straightforward to express any data structure in this way. Other examples, such as deques, were implemented in this style, but will not be presented here.

Two alternative implementations will be presented, the first following traditional methods, the second taking advantage of Haskell's ability to return functions as the result of applying a function.

In the first implementation, the queue is a variable containing a pair.

```
type Queue s a = MutVar s (LinkedList s a, LinkedList s a)
```

The first component is a variable containing the first item of a linked list (that is, it points to head of the queue), and the second component is a variable which holds the final `Nil` of the list (that is, it points to the end of the queue).

An empty queue is generated by the state transformer `makeQ` which generates two variables: `v`, initially containing the empty list, and the queue itself containing `v` as both the front and rear ends.

```
makeQ :: ST s (Queue s a)
makeQ = do { v <- newVar Nil;
             newVar (v,v)
           }
```

A queue is empty if the front and rear variables (pointers) are the same (or, equivalently if both contain `Nil`). Elements are added and removed destructively.

```
insert :: Queue s a -> a -> ST s ()
insert q x = do { (f,r) <- readVar q;
                  w <- newVar Nil;
                  writeVar r (Item x w);
                  writeVar q (f,w)
                }

remove :: Queue s a -> ST s a
remove q = do { (f,r) <- readVar q;
                x <- headL f;
                tailL f;
                return x
              }

empty :: Queue s a -> ST s Bool
empty q = do { (f,r) <- readVar q;
               return (f==r)
             }
```

Queues can now be used within any state thread as follows.

```
.. do
      :
      q1 <- makeQ;
      :
      insert q1 5;
      :
      q2 <- makeQ;
      :
      y <- remove q1;
      insert q2 "hello";
      :
```

Each use of `makeQ` generates a new queue which may be used at any type. In the example above, `q1` is a queue of integers, whereas `q2` is a queue of strings. The two queues are independent of each other.

4.5.2 Hiding the queue

The problem with the previous implementation is that the queue is explicit and its structure known. There is nothing to prevent non queue-like operations being applied. This structure may all be hidden from the programmer as follows.

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

An abstract queue is a triple of operations, corresponding precisely to the three abstract operations on queues: insertion, deletion, and testing for being empty. The only thing the user of the queue sees are these operations, no handle on the internal queue structure is given.

The implementation in terms of the previous operations is straightforward.

```
makeAbsQ :: ST s (AbsQueue s a)
makeAbsQ = do { q <- makeQ;
                return (insert q,remove q,empty q)
              }
```

Abstract queues can now be used as follows.

```
.. do
     :
     (insA,remA,empA) <- makeAbsQ;
     :
     insA 5;
     :
     (insB,remB,empB) <- makeAbsQ;
     :
     y <- remA;
     insB "hello";
     :
```

Each time a new queue is generated, names for its operations are provided. These "procedures" access their mutually shared data structure, but do not expose it for unregulated tampering.

It is interesting to observe that this form of encapsulation only becomes viable because we are working with state transformers. Otherwise, each use of the queue operations would have to return a triple of the new operations for future use.

4.6 Mutable arrays

For numerous algorithms it is convenient to have arrays which can be updated in constant time. They can be provided by a similar scheme to mutable variables.

```
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 ()
```

Haskell provides a class `Ix` of types that can be used as array indices. The type `i` is constrained to be in this index class (which includes `Int`, `Char`, pairs of indices, and others).

Like `newVar`, the function `newArr` returns a reference to newly allocated store, only this time it is an initialised array. The index range is given by the pair of values of

type `i`, and the initialisation value by the argument of type `a`. The type of the "array variable" which is returned records the state thread in which it was created, together with the index and element types. Initialisation takes time proportional to the size of the array, the other two operations (for reading and writing) are constant time.

4.7 Binsort

To illustrate the array operations binsort will be expressed which takes $O(n+m)$ time (given n elements to sort which are in a range of size m). With binsort, an array of *bins* is used to sort elements. Each element to be sorted has an associated index in the array. This association is described by the function `key` which takes values to their index position. For example, the function:

```
truncate :: Float -> Int
```

could be used as a key function, to sort floating point numbers with respect to their integer part.

Binsort works by placing elements in the array at an index determined by the `key` function, after which the array is traversed, from the first index to the last, giving the sorted list with respect to the key function.

```
binsort :: Ix i => (i,i) -> (a -> i) -> [a] -> [a]
binsort (l,u) key xs  = runST (do { bin <- newArr (l,u) [];
                                    insert bin key xs;
                                    extract bin [l..u]
                                  })
```

First an array of bins is created with the indices in the range `l..u`. All bins are initialised to the empty list (using lists allows us to handle duplicate elements). Then the insert and extract "procedures" are called (both state transformers, of course), the latter returning a list corresponding to the contents of the array.

```
insert :: Ix i => MutArr s i [a] -> (a -> i) -> [a] -> ST s ()
insert bin key []     = return ()
insert bin key (x:xs) = do { let i = key x in
                             ys <- readArr bin i;
                             writeArr bin i (x:ys);
                             insert bin xs
                           }

extract :: Ix i => MutArr s i [b] -> [i] -> ST s [b]
extract bin []     = return []
extract bin (i:is) = do { xs <- readArr i bin;
                          ys <- extract bin is;
                          return (xs++ys)
                        }
```

Studying the type of `binsort` shows it to be a pure function. For example,

```
binsort (1,5) id [5,2,1,4,2]
```

will return `[1,2,2,4,5]`.

All the state operations are encapsulated within a state thread produced by `runST`. Later in Section 4.9 it is shown how all of the state actions for binsort may be encapsulated using the `accumArray` combinator.

4.8 Disjoint sets (union/find)

Disjoint sets are useful for many algorithms: Tarjan (1974), for example, uses them as part of an algorithm to detect dominators in graphs, and they can be used in the well-known minimum spanning tree algorithm of Kruskal Jr. (1956). Disjoint sets are sets with no elements in common. The operations required are set union and set find operations. When given an element, set find will return the name of the set it is contained in. Each set therefore needs to have a distinct name. The union operation takes the names of two sets and a new name and returns the computed union labelled with the new name. It is crucial to many algorithms to have the

union/find operations computed in near constant time (that is, the complexity is virtually linear in the number of operations).

The most commonly used method of representing disjoint sets is to use *up-trees*, as described by Galler and Fisher (1964). In an up-tree children point to their parents. Each set is represented by an up-tree, where the root node stores the set name.

```
type Set s a = MutArr s a (Node a)
data Node a  = Empty
             | Root Name Int
             | Parent a
```

Mutable arrays are used to store tree nodes, where the set elements are the indices of the array. This restricts us to knowing that the elements are in a certain range, but this is normally the case with the algorithms that use disjoint sets. As well as storing the set name in the root node, the root node also stores the set size, which is useful for the union operation described later.

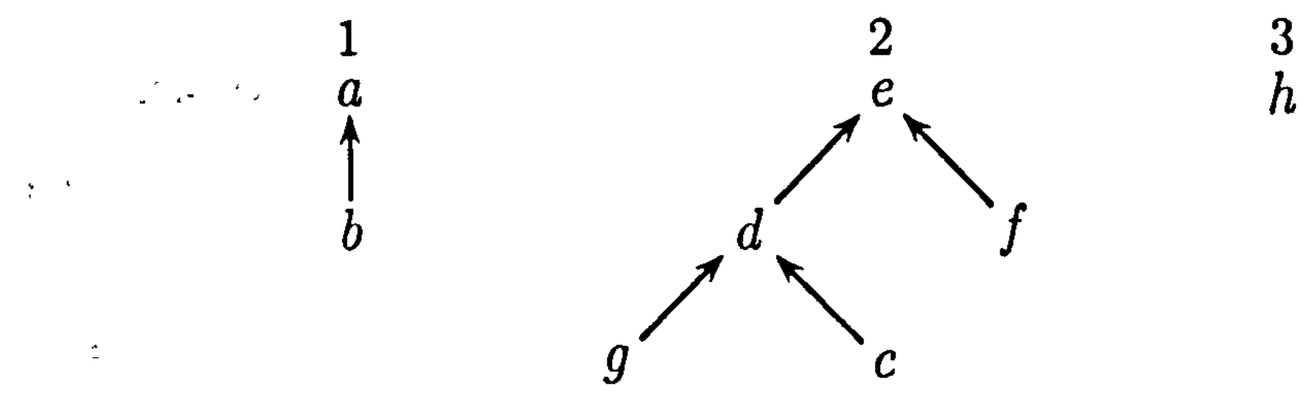


Figure 4.1 The disjoint sets $\{\{a,b\}^1,\{c,d,e,f,g\}^2,\{h\}^3\}$ represented by up-trees. The numerical superscripts are set names.

The `find` operation will follow parent pointers to a tree root, where the set name is contained. For example, in Figure 4.1, find of c will follow the pointers up to the root e, and the set name 2 will be returned.

For the efficiency of later calls to `find` *path compression* is also carried out. This collapses the path to the root by redirecting every node on the path to point to the root. Path compression is implemented using the `fixST` combinator, which takes advantage of laziness:

```
fixST :: (a -> ST s a) -> ST s a
fixST k s = (x,t)  where  (x,t) = k x s
```

```
find :: Ix a => Set s a -> a -> ST s a
find set x = fixST (\p -> compress set x p)

compress :: Ix a => Set s a -> a -> a -> ST s a
compress set x p = do { node <- readArr set x;
                        case node of
                          Root n s -> return x
                          Parent a -> writeArr set x (Parent p);
                                      compress set a p
                      }
```

Figure 4.2 Imperative functional find using path compression.

This provides us with a neat functional way of expressing path compression in one traversal up the tree. The function `compress` takes and returns the pointer to the root. Although this is elegant, it is not essential to use `fixST` here, since the order of the recursive call and `writeArr` could be rearranged to get the same effect. Nonetheless, there are other examples where `fixST` has proved to be extremely useful.

When performing a set union the sizes of each set are compared and the smaller set is linked to the larger. This is known as *union by size* and gives more balanced trees. Again this makes later finds more efficient.

```
union :: Ix a => Set s a -> a -> a -> Name -> ST s a
union set px py nz = do { Root nx sx <- readArr set px;
                          Root ny sy <- readArr set py;
                          if sx>sy
                            then do { writeArr set px (Root nz (sx+sy));
                                      writeArr set py (Parent px);
                                      return px
                                    }
                            else do { writeArr set px (Parent py);
                                      writeArr set py (Root nz (sx+sy));
                                      return py
                                    }
                        }
```

Figure 4.3 Imperative functional union using union-by-size.

Disjoint sets can be constructed with the function `insElem` which creates one set and inserts it into the set of disjoint sets.

```
insElem :: Ix a => Set s a -> Name -> [a] -> ST s ()
insElem ar n []     = return ()
insElem ar n (v:vs) = do { writeArr ar v (Root n (1+length vs));
                           applyST ptrRoot vs
                         }
    where
        ptrRoot x = writeArr ar x (Parent v)
```

This uses the function `applyST` which has the following definition:

```
applyST :: (a -> ST s b) -> [a] -> ST s ()
applyST f []     = return ()
applyST f (x:xs) = do { f x; applyST f xs }
```

For a comparison with traditional imperative code, here's a Pascal implementation of `find` using path compression (Figure 4.4). This was taken verbatim from Kingston's (1990) book, p. 218.

```
procedure Find(x: Entry; var D: DisjointSets): SetType;
var y, z, tmp: Entry;
begin
  y := x;
  while y^.parent ≠ nil do
    y := y^.parent;
  end;
  z := x;
  while z^.parent ≠ nil do
    tmp := z^.parent;
    z^.parent := y;
    z := tmp;
  end;
  return CAST(SetType, y);
end Find;
```

Figure 4.4 Imperative version of `find` using path compression.

This implementation of `find` (Figure 4.4) differs from the Haskell implementation in that it is a two pass algorithm. First the root node is found by chasing pointers, then in the second traversal pointers are made from each node to the known root.

4.9 Stateful combinators

State transformers are first class values, and as with other first class values — lists and trees, for example — there are several useful combining forms. Some of the most useful combinators are now described, which are used in later examples.

There are two obvious ways of combining a list of state transformers. The first `listST` gathers the results of each state action and returns the result in list form; the second `seqST` applies each state action in a list, ignoring the results, and returns the unit state type.

```
listST :: [ST s a] -> ST s [a]
listST = foldr consST nilST
      where
            nilST :: ST s [a]
            nilST = return []

            consST :: ST s a -> ST s [a] -> ST s a
            consST x xs = do { a <- x;
                               as <- xs;
                               return (a:as)
                             }

seqST :: [ST s a] -> ST s ()
seqST = foldr (;) (return ())
```

Just as `map` is useful for lists, `mapST` is useful for state transformers.

```
mapST :: (a -> ST s b) -> [a] -> ST s [b]
mapST f xs = listST (map f xs)
```

Sometimes it is possible to fully encapsulate the state actions in a combinator. Functions like `accumArray` and `lazyArray` do this, and usually give a more appropriate way

of expressing algorithms. For example, the binsort algorithm presented earlier, is far better expressed using `accumArray`.

```
binsort :: Ix i => (i,i) -> (a -> i) -> [a] -> [a]
binsort bnds key xs = flattenArray (accumArray (flip (:)) [] bnds
                                                [ (key x,x) | x<-xs])
    where flattenArray = concat . elems
```

The `accumArray` function has the type:

```
accumArray :: Ix i => (a -> b -> a) -> a -> (i,i) -> [(i,b)] -> Array i a
```

The call `accumArray f e bnds xs` builds an array with bounds `bnds` from a list if index/value pairs `xs`. In this list all the values with the same index are combined with a fold to the left operation, using `f`, starting with the value `e`. An implementation is given in Launchbury and Peyton Jones (1996) which is as follows,

```
accumArray f e bnds xs = runST (do { a <- newArr bnds e;
                                     mapST (update a) xs;
                                     freezeArr a
                                   })
    where
        update :: MutArr s i b -> (i,b) -> ST s ()
        update a (i,v) = do { x <- readArr a i;
                              writeArr a i (f x v)
                            }
```

This definition uses the combinator `freezeArr` which simply takes a mutable array, and returns a standard Haskell monolithic array; it has the type;

```
freezeArr :: Ix i => MutArr s i a -> ST s (Array i a)
```

Haskell arrays are strict in the list of index/value pairs, and in the indices. Johnsson (1995) describes the function `lazyArray`, which has the following specification:

```
lazyArray :: Ix i => (i,i) -> [(i,a)] -> Array i [a]
lazyArray bnds xs = array bnds [ (i, [ v | (j,v)<-xs, i==j])
                                 | i<-range bnds]
```

This is similar to `array` except that the array is created immediately, even before `xs` is evaluated. It is only when the array is indexed that `xs` is searched.

The combinator `lazyArray` will be seen later in an implementation of breadth-first search (Section 7.5).

4.10 Discussion

This chapter presented a number of examples of programming in Haskell with state transformers. The style of programs obtained is an intriguing mix of functional and imperative. This section tries to clarify some of the issues that have been exposed.

The first point is that having imperative features truly increases the power of the language. Any multi-linked data structure can be implemented giving the same asymptotic complexity as in the (sequential) imperative case. This is a big step forward, as previously there were problems for which no efficient solutions were known in Haskell. The big question is, however, whether incorporating the opportunities for imperative actions into a lazy functional language destroys the advantages of the language? Has the baby been thrown out with the bath water?

The example of binsort described earlier in Section 4.7 is interesting in this respect. The algorithm itself seems to require destructive update to be efficient, but its input/output behaviour can be expressed purely functionally. That is, `binsort` is a function which takes a list and returns a list. It has no externally visible state behaviour, and may be treated like a pure function. Thus it is possible to completely encapsulate imperative actions: elsewhere, where imperative actions are not explicitly required, purely functional code may be used.

The reverse inclusion happens all over the place as well. Many of the state-based examples use purely functional values and data structures within the state thread. One concrete effect of this is that even though the *structure* of a given piece of code may mimic imperative code, the *details* may be quite different. With a number of the examples, an implementation was presented at a much higher level than is typical of completely imperative implementations. The code here is more at the level that one expects from pseudo-code.

What of the other features typical in modern functional languages? Again, many of these carry over:

- Almost all of the examples are polymorphic, not in the weak sense of C pointers, but with all the usual guarantees of strong, static typing. This was explicitly drawn out in the case of queues, but it is also there in binsort and in union/find.

- Higher-order programming is used to good effect in encapsulating the queues (quite apart from defining state transformers in the first place!). The value returned by `makeAbsQ` is a triple of functions, each function defined by partial application.

- Laziness shows up in the examples of cyclic programming. This is used in the function `find` from union/find where the well-known technique of cyclic programming is used to reduce a multi-pass algorithm to a single pass.

- In many (though by no means all) imperative languages there is no underlying garbage collection — the programmer has to free space explicitly if there is a high turnover. This is tedious and error prone. In the programs here unreachable storage can simply be ignored, relying on the garbage collector to reclaim it.

Despite all this, using state transformers is no panacea. There are some serious consequences. First of all, we lose much of the structural simplicity common to many purely functional algorithms. Equational reasoning becomes much more complex because of the underlying state — similar techniques that are required for imperative reasoning are required here. Nevertheless, if a program *needs* state operations, then there is no choice. In particular, some programs really are naturally state manipulators in that even functional solutions will plumb extra values through the computation (name-supply programs often have this form). With these nothing is lost by being explicit about state, indeed a better structure may be obtained by so doing.

Because of this it turns out that we are unable to encapsulate state operations as tightly as we might like. Queues and union/find are examples of this. Since their access has an implicit implication for future accesses (i.e. they are state transformers), they have to be used within a state transformer thread, so making the thread quite pervasive, affecting the structure of a large part of the program.

One major difference between traditional imperative languages and the imperative actions described here is in syntax. This is not simply a lexical issue, but intimately involves semantics. In a traditional language the references to `x` in a statement like `x := x+1` have two different meanings. The reference on the left refers to the *location* to which `x` refers, whereas the reference on the right refers to the *value* stored in that location. This lack of distinction is not present in the state transformer idiom. Every reference to a variable refers to its location, making variables first-class (that is pointers, which are not truly first-class in imperative languages). A variable's value can only be accessed by using the "procedure" `readVar`. Unfortunately it seems as though there is no way of avoiding this, without making the state strict in the values it stores. It is of some comfort that despite the syntax being clumsy on occasion, it does at least make the order of state accesses explicit.

A more serious implication of using a state transformer is the sequentialisation of the program (fragment). One of the strengths of non-strict languages is their potential for parallel evaluation, but the more state is used the more potential parallelism is lost.

Finally we ought to refer to the state arguments `s` that seem to pervade the types of state transformers. They are present for technical reasons in order to make encapsulation of state transformers referentially transparent. Nevertheless, they do also play a useful role in alerting the programmer to the existence of state components within data structures.

4.11 Related work

Currently there is no final consensus in the purely functional language community on how arrays should be implemented, but there does seem to be agreement that some problems require constant time update to achieve the same asymptotic efficiency as imperative solutions.

Burton and Yang (1990) experimented with multi-linked data structures in a lazy functional language. The data structures are implemented by using *heaps* which in turn are implemented by using arrays; and the arrays are implemented using balanced trees. So an imperative efficiency wasn't possible, but it would be if the arrays were implemented to provide an update operation in constant time. With their approach

functions are passed a heap and return an updated heap as a result.

A drawback of the imperative functional approach is that it imposes sequentiality on the imperative actions. The dataflow language Id (Nikhil 1991) provides I-structures and more recently M-structures (Barth et al. 1991) which can be updated in constant time whilst fitting well with the parallel evaluation strategy of Id. Although these structures may be the way forward for parallel implementations, they would make a sequential implementation more complex. Moreover, they destroy the semantics of the language — the results of a program which uses M-structures can vary depending on evaluation order.

A shortcoming of using explicit state transformers for state based computations is that we have to be explicit about when state is present, and it is not always possible to encapsulate the state part into one small component. An alternative is somehow to determine when it is safe to do a destructive update. Meira (1985*a*) discusses changing the evaluation scheme for the lazy functional language KRC to determine when it is safe to update objects by overwriting. He then implements a linear time solution to the set union problem.

Gifford and Lucassen (1986) showed how to integrate functional and imperative programming into a single language. They introduced an *effect system* which statically checks for side-effect invariants in a similar manner to type checking. The side-effect invariants are: the ability to read, write, and allocate memory. In essence, the effects system restricts the use of side effects. Advantages are that it's easy to combine programs with different effects, and the programs are suitable for parallel execution. Disadvantages are that equational reasoning may not be used on the program fragments that have side effects, and a predictable order of evaluation is necessary.

There has been an abundance of work on elaborate type systems that reject programs where safe state manipulations cannot be guaranteed. Examples include *linear* type systems (Wadler 1990*c*); the *single-threaded* type systems (Guzmán and Hudak 1990); and the *stratified* type systems (Swarup et al. 1991). With all of these approaches the resulting type system becomes complex, and they have not been fully tested in practice. One type system that has been tested in practice is the *unique* type system of Smetsers et al. (1993). Their system has been implemented in the lazy functional graph rewriting language Concurrent Clean. If the type of an object is unique then there is a guarantee that it will only be accessed once. Hence, destructive updates are safely performed on an object with such a type.

BLANK IN ORIGINAL

Chapter 5

Modelling graphs

This chapter considers various means of modelling a graph in computer store. The particular representation chosen is of great importance, since it can have a profound effect on the complexity of an algorithm. The models chosen here are traditional ones, that is, adjacency lists, and adjacency matrices. Using these representations functions are provided for constructing various types of graph, these functions are useful for testing algorithms. Graph classification functions are also given; most graph algorithms only work on certain types of graph, so it is useful to determine what kind of graph we have.

5.1 Representations of graphs

The most widely-used representations of graphs are adjacency lists and adjacency matrices. Both are typically represented with arrays. In Haskell there are many choices of representation to consider, for example:

- Use a list of pairs to represent the graph edges. This is often chosen in the functional programming literature (for example, Paulson (1991), Holyer (1991)), because it is the simplest. The main shortcoming with this representation is that algorithms do not have the optimal asymptotic complexity.
- Use a function from vertices to their adjacent vertices (see Reade (1989)). The efficiency of algorithms using this representation depends upon the underlying implementation of functions. A shortcoming with this representation is that it is hard to construct arbitrary graphs efficiently during run-time.

- Use a purely functional algebraic datatype utilising laziness to express cycles (Section 5.2).

- Use immutable arrays to represent adjacency lists (Section 5.3) or adjacency matrices (Section 5.5). These are fine as long as we don't need to dynamically change a graph during an algorithm.

- Mimic the conventional approach with the state monad, that is, have explicit pointers in the heap. This representation results in imperative style algorithms. Nevertheless, if parts of a graph need to be dynamically modified, then this is more appropriate than the above purely functional representations.

In other functional languages, one could consider: version arrays (Morrisett (1993) in Standard ML); using reference types in Standard ML; M-structures (Barth et al. (1991) in Id), or using unique types (Smetsers et al. (1993) in Clean). None of these are considered here.

5.2 Cyclic representations

In a lazy language the cyclic nature of a graph can be represented by a cyclic structure. For example, the following cyclic expression is a graph, with one vertex, and one self-looping edge:

```
ones :: [Int]
ones = 1:ones
```

This could be generalised to any directed graph by using a list of vertices (Figure 5.1).

```
graph = [a,b,c,d]
      where a = Vertex "a" [d]
            b = Vertex "b" [a,c]
            c = Vertex "c" []
            d = Vertex "d" [b]
```

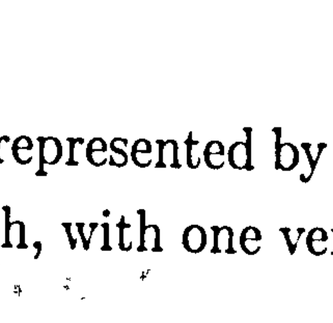


Figure 5.1 A cyclic expression representing a cyclic graph.

As an example the complete graph with vertices in the range described by the bnds pair could be constructed by:

```
completeG bnds = g
      where   g = map constructG vertices
              constructG u = Vertex u [ g!!w | w<-vertices, w/=u]
              vertices = range bnds
```

The function `completeG` creates a cycle by using list lookup (`!!`) on g, and this method may be used to create arbitrary graphs on-the-fly at run-time, see Clack et al. (1995) for an implementation. List lookup is not a constant time operation, so the graph construction algorithm doesn't have linear time complexity. Another difficulty with the cyclic structure is that operationally it's an infinite tree, so care is needed not to loop indefinitely. Try printing out the result of `completeG bnds` and the structure will unravel printing out the same vertices endlessly. The functional programming solution to prevent this endless unravelling, is to label each vertex with a unique name. Then traversal functions will maintain a set of unique names indicating which vertices have been visited before.

5.3 Adjacency lists

For many algorithms the best representation is an array of adjacency lists. The array is indexed by vertices, and each component of the array is a list of those vertices reachable along a single edge. This adjacency structure is linear in the size of the graph. The indexed structure allows us to be explicit about the sharing that occurs in the graph. Thus standard Haskell immutable arrays are chosen here. This gives constant time access (but not update — these arrays may be shared arbitrarily).

The same structure may be used to represent *undirected* graphs as well, simply by ensuring that there are edges in both directions. An undirected graph can be viewed as a symmetric directed graph. *Multi-edged* graphs may also be represented by a simple extension, but these are not considered 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. 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). The function `vertices` is provided as an alternative for `indices`, which returns a list of all the vertices in a graph.

```
vertices :: Graph -> [Vertex]
vertices = indices
```

Sometimes it is convenient to extract a list of edges from the graph, this is done with the function `edges`. An edge is a pair of vertices.

```
type Edge = (Vertex,Vertex)

edges :: Graph -> [Edge]  --  O(V+E)
edges g = [ (v,w) | v<-vertices g, w<-g!v]
```

To manipulate tables (and graphs) the generic function `mapA` is provided which applies its function argument to every array index/entry pair, and builds a new array.

```
mapA :: Ix a => (a -> b -> c) -> Array a b -> Array a c
mapA f a = array (bounds a) [ (i, f i (a!i)) | i<-indices a]
```

The Haskell function `array` takes low and high bounds and a list of index/value pairs, and builds the corresponding array in linear time. Because we are using an array-based implementation we often need to provide a pair of vertices as array bounds. So for convenience we define,

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

Using `mapA` we could define,

```
outdegree :: Graph -> Table Int  --  O(V+E)
outdegree g = mapA numEdges g
  where  numEdges v ws = length ws
```

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

It is often useful to build up a graph from a list of edges, `buildG` is provided for this purpose:

```
buildG :: Bounds -> [Edge] -> Graph  --  O(V+E)
buildG bnds es = accumArray (flip (:)) [] bnds es
```

using `accumArray` described in Section 4.9. Lists are built 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, a graph defined in terms of edges can be converted to the vertex table based graph. For example,

```
graph = buildG ('a','n')
          (reverse [('a','b'), ('a','d'), ('b','e'), ('e','d'),
                    ('e','f'), ('e','g'), ('f','c'), ('g','f'),
                    ('h','j'), ('i','h'), ('j','k'), ('j','i'),
                    ('k','j'), ('l','m'), ('m','n'), ('n','l')])
```

will produce the array representation for the graph shown in Figure 5.2. The function `reverse` is used so that earlier entries will occur earlier in the adjacency list.

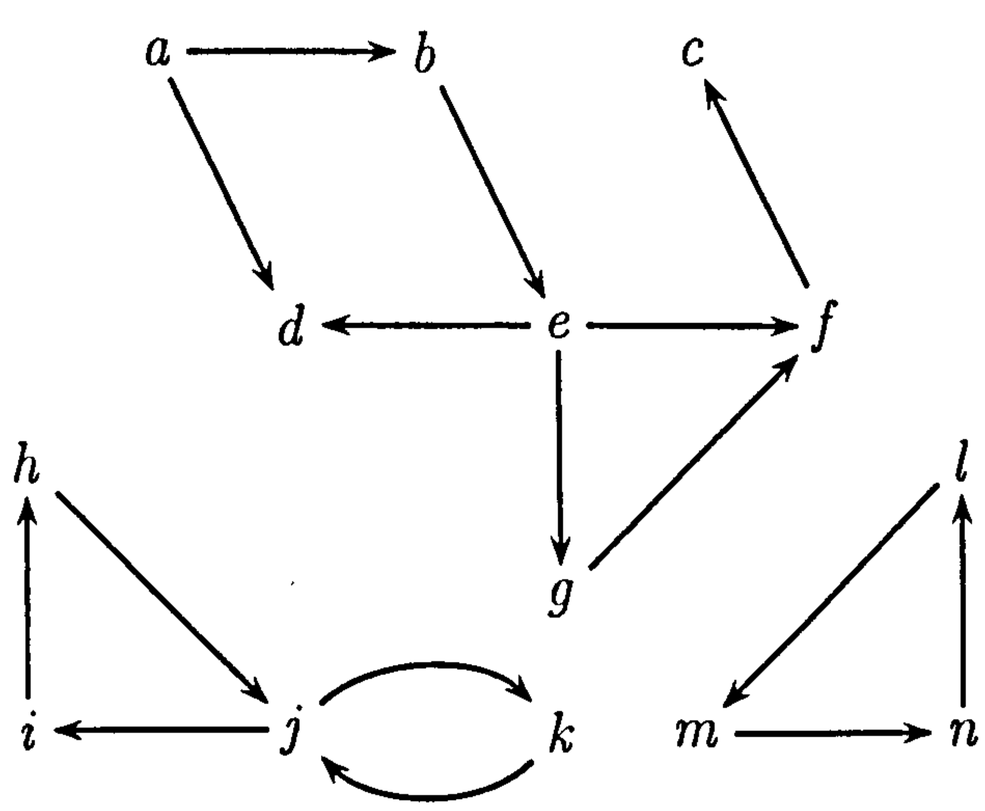


Figure 5.2 A directed graph.

Then, the immediate successors to `'e'` are found by computing:

```
graph ! 'e'
```

which returns `['d', 'f', 'g']`.

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  -- O(V+E)
transposeG g = buildG (bounds g) (map reverseE (edges g))

reverseE :: Edge -> Edge
reverseE (v,w) = (w,v)
```

Edges are extracted from the original graph, their direction reversed, and the graph is rebuilt with the new edges. Then, for example,

```
(transposeG graph) ! 'e'
```

will return `['b']`. Now by using `transposeG`, an in-degree table for vertices may immediately be defined:

```
indegree :: Graph -> Table Int  --  O(V+E)
indegree g = outdegree (transposeG g)
```

5.4 Classifying graphs

It is important to classify graphs for efficient algorithm design. Many algorithms will only work on certain types of graph. Several different classes of graph will now be considered. The *null graph* has no vertices or edges, and the *empty graph* has vertices but no edges. A *simple graph* is one with no self-loops; a *pseudo-graph* contains at least one self-loop. In a *functional graph*, each vertex has out-degree one as the graph is modelling a real function. A graph is *Eulerian* if it is connected and the in-degree and out-degree are the same for every vertex, meaning that there exists a tour which includes each edge exactly once.

These functions are all neatly expressed as one-liners in Haskell and are presented in Figure 5.3. They are expressed with no loss of efficiency, their asymptotic complexity is given as a comment along with their type. Null graphs cannot be modelled with the representation used here, since standard Haskell arrays must have at least one index. Although, it would be quite straightforward to extend our representation to handle null graphs. The function `isEulerian` makes use of `isConnected`; an implementation of `isConnected` will be given later in Section 6.6.3.

```
isEmptyG :: Graph -> Bool  --  O(V)
isEmptyG g = null (edges g)

isPseudoG :: Graph -> Bool  --  O(V+E)
isPseudoG g = or [ v==w | (v,w)<-edges g]

isSimpleG :: Graph -> Bool  --  O(V+E)
isSimpleG g = not (isPseudoG g)

isFunctionalG :: Graph -> Bool  --  O(V)
isFunctionalG g = and [ length (g!v) == 1 | v<-vertices g]

isEulerian :: Graph -> Bool  --  O(V+E)
isEulerian g = isConnected g && (indegree g == outdegree g)
```

Figure 5.3 Some graph classifications.

5.4.1 Classifying undirected graphs

Although the functions above may be applied to undirected graphs; undirected graphs have different properties, and other means of classification. An undirected graph is *regular* if all vertices in the graph have the same degree. An undirected graph is *Eulerian* if it is connected and the degree of each vertex is even. A graph is *complete* if there is an edge between every pair of vertices, the graph must also be simple.

Since undirected edges are represented by two directed edges, the in-degree and out-degree for each vertex in an undirected graph will be equal; here `outdegree` is used as it is more efficient. The function `degreeSeq` sorts all the vertex degree's into ascending order. The function `degreeOrd` orders the vertices in descending order of their out-degrees.

The function `isCompleteG` utilises `binsort` (Section 4.7) for efficiency. The adjacency list for each vertex is ordered and compared with a list of all vertices, except the vertex itself which would form a self-loop. Comparing two lists of size $V-1$ for equality is $O(V)$, and `binsort` on a list of size $V-1$ with index range of size V is $O(V)$, therefore, the algorithm for `isCompleteG` will run in $O(V^2)$ time.

```
isRegularG :: Graph -> Bool  --  O(V+E)
isRegularG g = all (==d) ds
      where (d:ds) = degree g

isEulerianU :: Graph -> Bool  --  O(V+E)
isEulerianU g = isConnected g && all even (degree g)

degree :: Graph -> Table Int  --  O(V+E)
degree g = outdegree g

degreeSeq :: Graph -> [Int]  --  O(V.(log V)+E)
degreeSeq g = quicksort (elems (degree g))

degreeOrd :: Graph -> [Vertex]  --  O(V.(log V)+E)
degreeOrd g = (reverse . map snd . quicksort)
                    [ (length (g!v), v) | v<-vertices g]

isCompleteG :: Graph -> Bool  --  O(V^2)
isCompleteG g = and [ binsort (bounds g) id (g!v)
                      == [ w | w<-vertices g, v/=w] | v<-vertices g]
```

Figure 5.4 Some classifications of undirected graphs.

5.4.2 Generating graphs

It is useful to construct different types of graphs to test algorithms and invariants. The function `buildG` described earlier proves to be invaluable for generating various graphs.

A *simple circuit* is a cyclic path where each vertex appears exactly once except the first and last vertices. In Figure 5.5 the function `simpleCircuit` creates a list of vertices, and generates a graph where each vertex in the list has an edge to the next vertex in the list. The last vertex in the list has an edge in the graph to the first vertex in the list.

The graph in Figure 5.6 is generated by the following function call:

```
functionalG (\x -> 1 + ((x+3) `mod` 12)) (1,12)
```

```
emptyG :: Bounds -> Graph  --  O(V)
emptyG bnds = buildG bnds []

completeG :: Bounds -> Graph  --  O(V^2)
completeG bnds = buildG bnds [ (v,w) | v<-range bnds,w<-range bnds, v/=w]

simpleCircuit :: Bounds -> Graph  --  O(V)
simpleCircuit (l,u) = buildG (l,u) ((u,l):zip rs (tail rs))
      where rs = range (l,u)

functionalG :: (Vertex -> Vertex) -> Bounds -> Graph  --  O(V)
functionalG f bnds = buildG bnds [ (i,f i) | i<-range bnds]
```

Figure 5.5 Generating graphs.

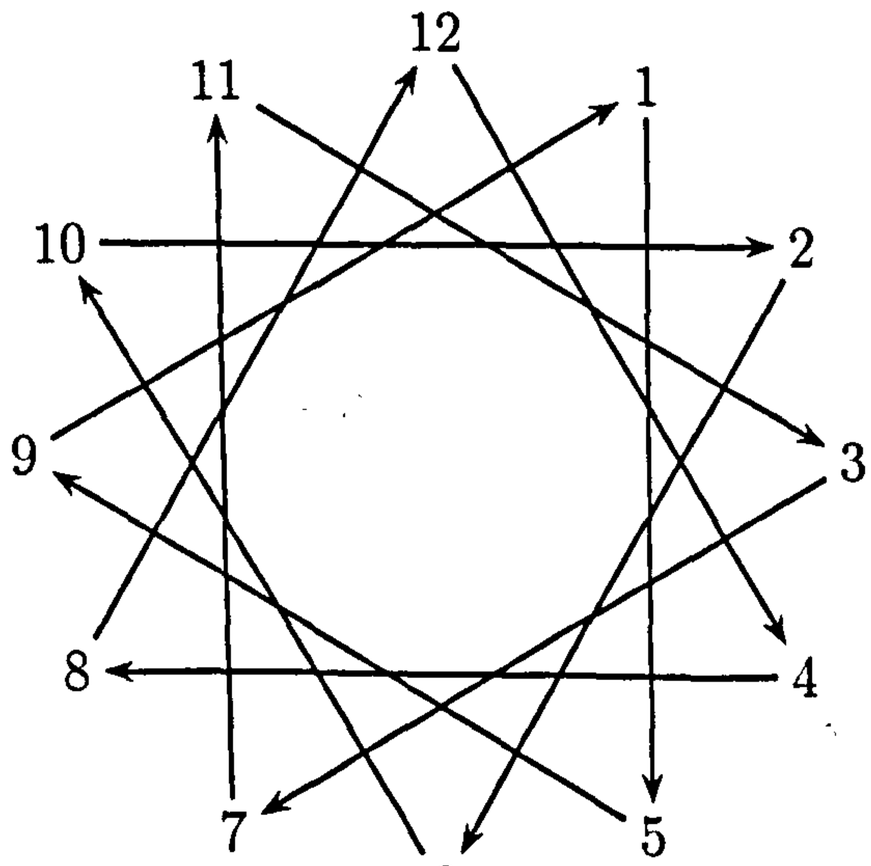


Figure 5.6 An example of a functional graph.

This graph represents an eight hour time difference, for example, between British Summer Time and Pacific Daylight Time.

It is sometimes useful to be able to generate an undirected graph from a directed one. This is done most succinctly by taking the union of a graph with its transpose:

```
undirected :: Graph -> Graph
undirected g = buildG (bounds g) (edges g ++ edges (transposeG g))
```

This will introduce extra edges between vertices v and w if there is already a directed edge from v to w, and from w to v.

5.4.3 Generating random graphs

For measuring the running times of some algorithms, it's convenient to have a large *randomly generated* graph. There are several ways of constructing a graph randomly. The number of edges and vertices could be chosen at random, but it is usually more practical to have control over these values. Here's a straightforward way of generating a random graph where the vertices are integers: first we need a random number generator `randomList` that returns a list of random numbers in a given range. There are several good ways of generating random numbers, a specific implementation is not included here:

```
randomList :: Int -> [Int]
```

A random permutation is generated by constructing an array of integers, and swapping each index once with a random index value. For efficiency, a stateful algorithm is used, which runs in $O(n)$ time. A purely functional solution is not known for this problem (Ponder et al. 1988).

```
randomPerm :: Int -> [Int]
randomPerm n
      = runST (do { r <- newArr (1,n) 0
                  ; applyST (\i -> writeArr r i i) [1..n];
                  ; applyST (swapArr r) (zip [1..n] (randomList n));
                  ; mapST (readArr r) [1..n]
                  })
```

```
where   swapArr :: Ix a => MutArr s a b -> (b,b) -> ST s ()
        swapArr r (x,y) = do { a <- readArr r x;
                               b <- readArr r y;
                               writeArr r x b;
                               writeArr r y a
                             }
```

A permutation of the integers from 1 to v^2 is constructed, and we use the property that there is a one-to-one mapping with graph edges. The function `randomE` takes the number of vertices `v` and edges `e`, and returns a list of `e` edges. These are then converted to a graph with the function `randomG`.

```
randomE :: Int -> Int -> [Edge]
randomE v e = take e [(x+1, y+1) | r<-randomPerm (v*v)
                                 (x,y)<-[r `divMod` v], x/=y]

randomG :: Int -> Int -> Graph
randomG v e = buildG (1,v) (randomE v e)
```

5.5 Adjacency matrices

An adjacency matrix is a $(V \times V)$ matrix, where the edge (v, w) is in the graph, if and only if, row v and column w contains the entry 1. Adjacency matrices are typically represented with a two dimensional array. The advantage over adjacency lists is that we can determine if we have an edge (v, w) in constant time. Adjacency matrices are easily generalised to weighted graphs by storing the weight at the array entry. Like adjacency lists a standard Haskell immutable array is used to represent the matrix.

```
type Matrix a = Array Edge a
type Edge     = (Vertex,Vertex)
```

A graph is now a matrix with labelled entries. This allows multi-edged graphs to be represented, by storing the number of edges in the label. The name `LGraph` will be

used for the type of labelled graphs, and functions will be suffixed with `L` to distinguish from graphs represented with adjacency lists.

```
type LGraph = Matrix Label
type Label  = Maybe Value
```

Here the `Maybe` datatype is used for labels, a label of `Nothing` means there is no edge, and a label of `Just v` is an edge with a label value `v`. Label values may be of any type, but some functions may require equality on the type.

```
verticesL :: LGraph -> [Vertex]  --  O(V)
verticesL g = range (limitsL g)

edgesL :: LGraph -> [Edge]  --  O(V^2)
edgesL g = [ e | e<-indices g, isEdge g e]

limitsL :: LGraph -> (Vertex, Vertex)  --  O(1)
limitsL g = (l,u) where ((l,_),(u,_)) = bounds g

isEdge :: LGraph -> Edge -> Bool  --  O(1)
isEdge g e = weight g e /= Nothing

weight :: LGraph -> Edge -> Label  --  O(1)
weight g e = g ! e
```

Although with adjacency lists it takes $O(1)$ time to return a vertex's successors, with adjacency matrices it takes $O(V)$ time to return all the successors, and predecessor of a vertex.

```
succL :: LGraph -> Vertex -> [Vertex] --  O(V)
succL g v = [ w | w<-verticesL g, isEdge g (v,w)]

predL :: LGraph -> Vertex -> [Vertex]  --  O(V)
predL g v = [ w | w<-verticesL g, isEdge g (w,v)]
```

5.5.1 Classifying edge labelled graphs

Since a different representation is used for edge labelled graphs, most of the classification functions will have different implementations, and different running times. For example, `isEmptyL` has a running time of $O(V^2)$ compared with `isEmptyG` which runs in $O(V)$ time.

```
isEmptyL :: LGraph -> Bool  --  O(V^2)
isEmptyL g = and [ not (isEdge g e) | e<-indices g]

isUnweightedL :: LGraph -> Bool  -- O(V^2)
isUnweightedL g = and [ not (isEdge g e) || g!e==Just 1 | e<-indices g]

isUndirectedL :: LGraph -> Bool  -- O(V^2)
isUndirectedL g = and [ g!(v,w)==g!(w,v) | (v,w)<-indices g]

isCompleteL :: LGraph -> Bool  --  O(V^2)
isCompleteL g = and [ isEdge g (v,w) | (v,w)<-indices g, v/=w]
```

Figure 5.7 Classifying edge labelled graphs.

5.5.2 Generating edge labelled graphs

Just like with adjacency lists it is convenient to have a function that builds a graph from a list of edges:

```
buildL :: Bounds -> [Edge] -> LGraph  --  O(V^2)
buildL (l,u) es = (array i [ (e,Nothing) | e<-range i])
                        // [ (e,Just 1) | e<-es]
    where i = ((l,l),(u,u))
```

If there is an edge between two vertices it is given the label `Just 1` otherwise it is given the label `Nothing`. The operator `(//)` takes an array and a list of index/value pairs, and returns the array from the left argument after it has been updated with the index/value pairs in the right argument. Sometimes it is necessary to specify the edge weights:

```
mkLGraph :: [(Edge,Label)] -> Bounds -> LGraph  --  O(V^2)
mkLGraph els (l,u) = (array i [ (e,Nothing) | e<-range i]) // els
                     where i = ((l,l),(u,u))
```

The functions for generating edge labelled graphs (`emptyL`, `completeL`, `simpleCircuitL`, `functionalL`) are now identical to the functions for graphs modulo `buildL`. Code duplication may be avoided by defining an abstract datatype for graphs that includes functions like `buildL`.

5.6 Discussion

The principal differences between the two graph representations, adjacency lists and adjacency matrices, are summarised in Table 5.1. These differences have an impact on the complexity of graph algorithms. For example, the functions to create degree tables `indegree` and `outdegree` run in $O(V + E)$ time with adjacency lists, but run in $O(V^2)$ time with adjacency matrices (note $E \leq V^2$). With these examples adjacency lists are better suited, and there are several examples where this is the case (another example covered in Chapter 6 is depth-first search). Nevertheless, there are several cases where adjacency matrices are more efficient than adjacency lists. With adjacency matrices, the existence of an edge is determined in $O(1)$ time, hence the representation is more convenient for weighted graph problems — where the edges are annotated. Some examples of weighted graph algorithms are given in Chapter 7.

	Adjacency list	**Adjacency matrix**
Space to represent graph	$O(V + E)$	$O(V^2)$
Time taken to discover the existence of an edge	$O(V)$	$O(1)$
Time taken to return all neighbours of a vertex	$O(1)$	$O(V)$

Table 5.1 Summary of differences between adjacency lists and adjacency matrices.

Chapter 6

Depth-first search based algorithms

Depth-first search (DFS) is a recipe for graph traversal. The recipe being to follow edges deep into the graph before fanning out to other edges. This simple method of traversal is the basis for several algorithms. Tarjan (1972), and Hopcroft and Tarjan (1973) were the first to discover this more than twenty years ago. In their work, and other work since, the DFS algorithm is viewed as a skeleton upon which code fragments are embedded. These code fragments compute information during the traversal process which is relevant to the particular algorithm being expressed. This has proved to be a successful method of designing efficient graph algorithms, but it has a number of drawbacks.

So many calculations are performed during the course of a graph traversal, that it becomes extremely difficult to understand and reason about what is going on. The DFS algorithm is lost as it is intertwined with other code fragments. It cannot be reused without having to duplicate the code. The alternative approach is one that is taken frequently in functional languages: to express the algorithm as the composition of several reusable components.

Given a graph we return a *depth-first spanning forest*, algorithms that use DFS are expressed in terms of this forest. A constant factor in complexity time is lost by doing this, but the gains far out-weigh this slow-down. Algorithms become more lucid, the code for DFS is reused for new algorithms. Since this is good for programming it is also good for reasoning. Static values like the depth-first spanning forest are easier to reason about, rather than dynamic values processed during a traversal.

6.1 Depth-first search

Depth-first search is often viewed 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 like this until we reach a vertex that has no edges leading to unvisited vertices. At this point we backtrack, and continue from the latest vertex that 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, one is chosen and the search commences again, until finally every vertex has been visited once, and every edge has been examined.

The graph in Figure 6.1 shows a depth-first traversal starting at vertex a. If at any vertex there is a choice of edges to follow, the selection is made by using the alphabetical ordering of vertices.

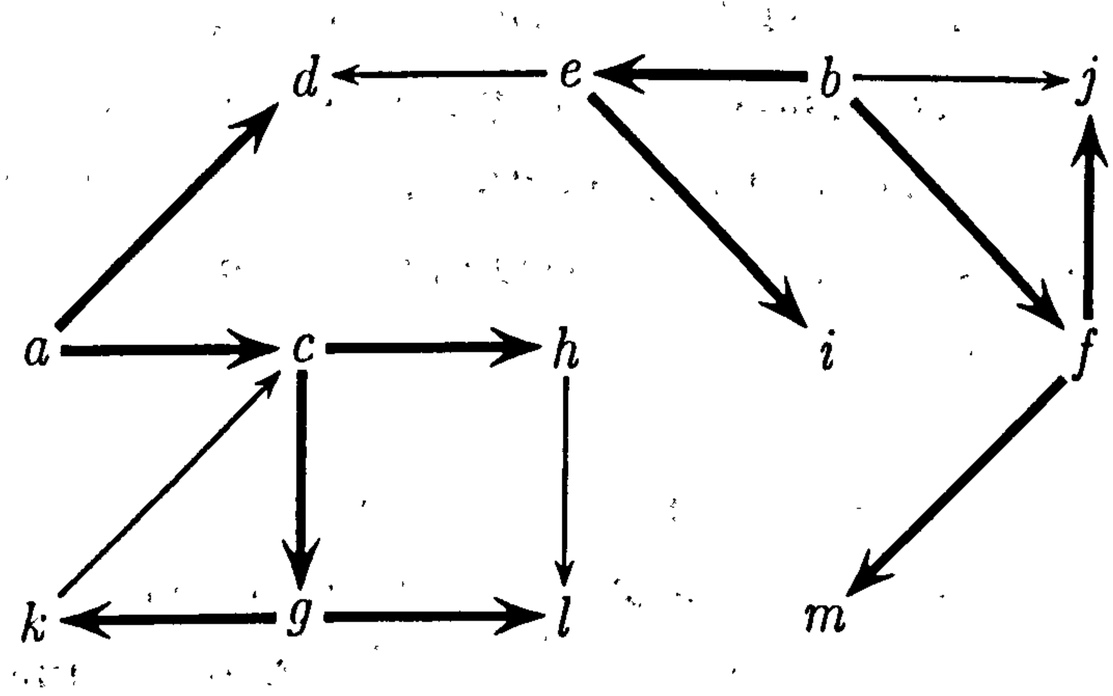


Figure 6.1 A directed graph: bold edges give a depth-first traversal.

6.2 Specification of depth-first search

We will concentrate on depth first search as a specification for a *value*, namely the *spanning forest* defined by a depth-first traversal of a graph. Such a forest for the graph in Figure 6.1 is depicted in Figure 6.2. The (solid) tree edges are those graph

edges that lead to unvisited vertices. The remaining graph edges are also shown, but in dashed lines. These edges are classified according to their relationship with the tree, namely, *forward edges* (which connect ancestors in the tree to descendants), *back edges* (the reverse), and *cross edges* (which connect nodes across the forest, but always from right to left). This standard classification is useful for thinking about a number of algorithms and later, in Section 6.6.5, an algorithm for classifying edges in this way is given.

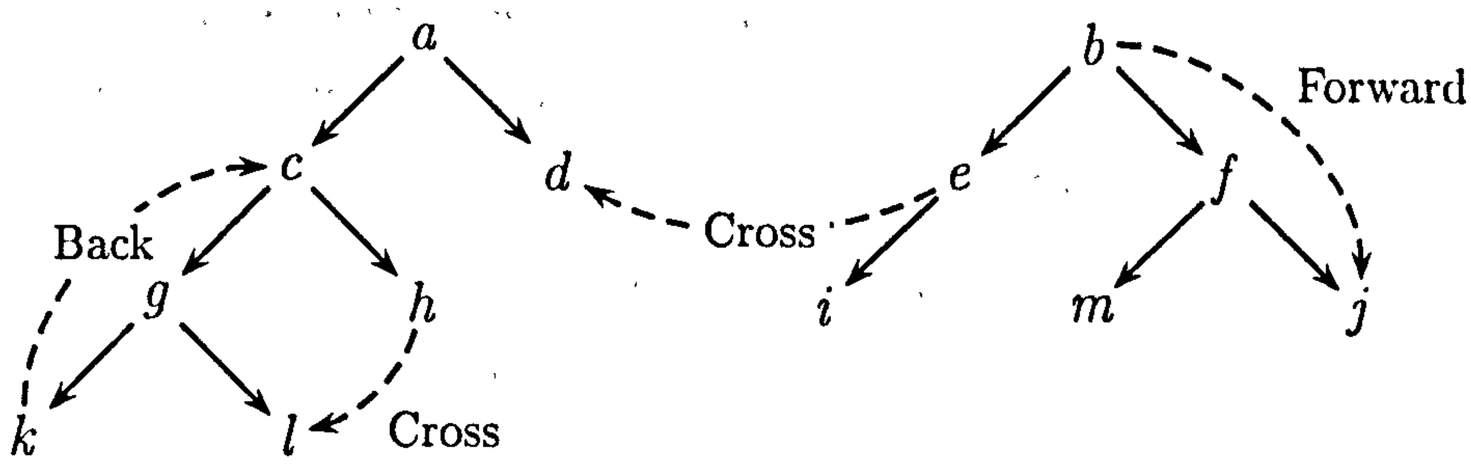


Figure 6.2 A depth-first spanning forest. The dashed lines represent graph edges that are not included in the forest.

Since the approach explored here 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 an appropriate definition of trees and forests is needed.

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 key component to all the DFS algorithms that are expressed here. For now we restrict ourselves to considering its properties, and leave its efficient Haskell implementation until Section 6.5.

Sometimes the initial ordering of vertices is not important. When this is the case the following related function is used:

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

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

Property 6.1 (Spanning subgraph)
The depth-first forest of a graph is a spanning subgraph, that is, it has the same vertex set, and the edge set is a subset of the graph edge set. The subgraph does not contain multi vertices or multi edges.

Property 6.2 (No left-right cross-edges)
The graph contains no left-right cross-edges with respect to the forest.

These two properties are satisfied by every depth-first forest, consequently several functions would satisfy these properties. The next property describes, together with the above two properties, one implementation of depth-first search.

Property 6.3 (Initial ordering)
Given the depth-first spanning forest, every descendant of a root or later node, appears later in the initial ordering than the root.

6.3 The generate–prune paradigm

In order to translate a graph into a depth-first spanning forest 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.

Definition ($\longrightarrow$)
For reasoning purposes, it is convenient to use a notion of *paths* rather than single edges: a path being made up of zero or more edges joined end-to-end. The notation

$v \longrightarrow_g w$ will be used to mean that there is a path from v to w in the graph g. Where there is no confusion the graph subscript will be dropped.

6.3.1 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 (generates g (g!v))

generates :: Graph -> [Vertex] -> [Tree Vertex]
generates g vs = map (generate g) vs
```

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.

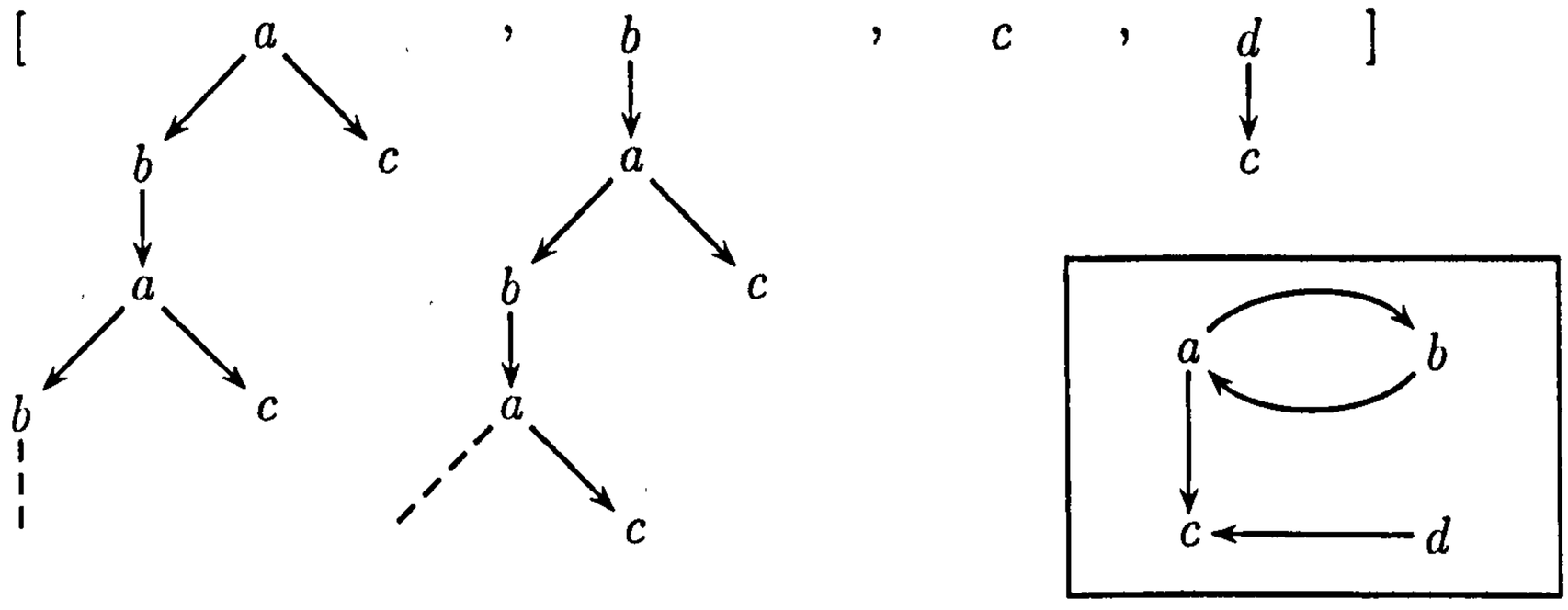


Figure 6.3 A generated forest, for the graph shown in the box.

6.3.2 Pruning

The goal of pruning the (infinite) forest is to discard subtrees whose roots have occurred previously. Thus we need to maintain a set of vertices (traditionally called

"marks") of those vertices to be discarded. The depth-first pruning may be defined as follows where P represents the set of vertices previously visited (or marked). This specification is convenient for reasoning. Specifications will be distinguished from programs by an italic font. Section 6.5 gives an efficient implementation of `prune`.

$$
\begin{array}{lll}
prune :: Set\ Vertex \to Forest\ Vertex \to Forest\ Vertex & & \\
prune\ P\ [\,] & = & [\,] \\
prune\ P\ [Node\ x\ ts] \mid x \in P & = & [\,] \\
\qquad\qquad\qquad\qquad \mid x \notin P & = & [\,Node\ x\ (prune\ (\{x\} \cup P)\ ts)\,] \\
prune\ P\ (ts +\!\!+ us) & = & prune\ P\ ts \\
 & & \quad +\!\!+ prune\ (P \cup \mathcal{F}\,(prune\ P\ ts))\ us
\end{array}
$$

Flatten ($\mathcal{F}$) maps forests to sets and may be defined:

Definition (Flatten)
Flattening transforms a tree to the set of all nodes contained in the tree.

$$\mathcal{F}\ ts = \bigcup_{t \in_{\mathcal{T}} ts} \mathcal{T}\ t, \qquad \mathcal{T}\,(Node\ x\ ts) = \{x\} \cup \mathcal{F}\ ts$$

The set definitions of ($\mathcal{F}$) and ($\mathcal{T}$) are only applied to finite objects, and are therefore computable in those cases. Generally, ($\mathcal{F}$) is applied to expressions of the form *prune* P *ts* which always terminates with a finite tree when *ts* is a rational tree, that is, it has been generated with *generates*, and P is a finite set.

Note that in this chapter the symbol $\in$ is heavily overloaded. In the expression $x \in y$, y may be a set, list, tree, or forest, but x will always be a unitary object. For example, if y is a tree of integers then x is an integer. The notation $x \in_{\mathcal{T}} y$ is used to signify that x is of type tree, and $x \in_{\mathcal{F}} y$ to signify that x is of type forest.

Now *dfs* can be defined in terms of *generates* and *prune*:

Definition (Depth-first search)

$$dfs\ g\ vs = prune\ \emptyset\ (generates\ g\ vs)$$

This definition, although more verbose, is superior to the implementation of DFS given in Chapter 1 because of its modularity (Hughes 1989). It is not only easier to understand, but allows the proofs to be modular. Instead of having properties about one function *dfs*, separate properties can be stated for *generates* and *prune*. This makes inventing properties and proving them easier than would be the case with just *dfs*.

Deforestation of a flattened *dfs*

Frequently a flattened version of *dfs* is useful, this may be deforested into a recursive definition *reaches*.

$$reaches\ g\ P\ xs = \mathcal{F}\,(prune\ P\ (generates\ g\ xs))$$

Case $[\,]$.

$$\begin{aligned}
&\mathcal{F}\,(prune\ P\ (generates\ g\ [\,])) \\
&\quad = \quad \left\{ \text{Definitions of } \mathcal{F}\text{, } prune\text{, and } generates \right\} \\
&\quad\quad \emptyset
\end{aligned}$$

Case $[x]$, where $x \in P$.

$$\begin{aligned}
&\mathcal{F}\,(prune\ P\ (generates\ g\ [x])) \\
&\quad = \mathcal{F}\,(prune\ P\ [\,Node\ x\ (generates\ g\ (g!x))\,]) \\
&\quad = \emptyset
\end{aligned}$$

Case $[x]$, where $x \notin P$.

$$\begin{aligned}
&\mathcal{F}\,(prune\ P\ (generates\ g\ [x])) \\
&\quad = \mathcal{F}\,(prune\ P\ [\,Node\ x\ (generates\ g\ (g!x))\,]) \\
&\quad = \mathcal{F}\,[\,Node\ x\ (prune\ (\{x\} \cup P)\ (generates\ g\ (g!x)))\,] \\
&\quad = \{x\} \cup \mathcal{F}\,(prune\ (\{x\} \cup P)\ (generates\ g\ (g!x))) \\
&\quad = \{x\} \cup reaches\ g\ (\{x\} \cup P)\ (g!x)
\end{aligned}$$

Case $xs +\!\!+ ys$.

$$\begin{aligned}
&\mathcal{F}\,(prune\ P\ (generates\ g\ (xs +\!\!+ ys))) \\
&\quad = \mathcal{F}\,(prune\ P\ (generates\ g\ xs)) \\
&\quad\quad \cup \mathcal{F}\,(prune\ (P \cup \mathcal{F}\,(prune\ P\ (generates\ g\ xs)))\ (generates\ g\ ys)) \\
&\quad = reaches\ g\ P\ xs \cup reaches\ g\ (P \cup reaches\ g\ P\ xs)\ ys
\end{aligned}$$

Hence, this yields the following recursive definition for *reaches*:

$$
\begin{array}{lll}
\mathit{reaches} :: \mathit{Graph} \to \mathit{Set}\ \mathit{Vertex} \to [\mathit{Vertex}] \to \mathit{Set}\ \mathit{Vertex} & & \\
\mathit{reaches}\ g\ P\ [\,] & = & \emptyset \\
\mathit{reaches}\ g\ P\ [x] \mid x \in P & = & \emptyset \\
\qquad\qquad \mid x \notin P & = & \{x\} \cup \mathit{reaches}\ g\ (\{x\} \cup P)\ (g!x) \\
\mathit{reaches}\ g\ P\ (xs +\!\!+ ys) & = & \mathit{reaches}\ g\ P\ xs \\
 & & \cup\ \mathit{reaches}\ g\ (P \cup \mathit{reaches}\ g\ P\ xs)\ ys
\end{array}
$$

Definition (Reaches)
The notation $v{\downarrow_g}$ will be used to denote the set of all vertices in the graph g that can be reached by traversing paths from vertex v. Similarly all the vertices that can be reached from the list of vertices vs are denoted by $vs{\Downarrow_g}$. Formally,

$$
\begin{array}{lcl}
v{\downarrow_g} & = & \{\, w \mid v \longrightarrow_g w \,\} \\
vs{\Downarrow_g} & = & \bigcup_{v \in vs} v{\downarrow_g}
\end{array}
$$

Theorem 6.4 (Reaches)
The recursive function *reaches* terminates when given well-defined arguments, and the call *reaches* $g\ \emptyset\ vs$ for some graph g and list of vertices vs will return a set of all the vertices in g that are reached by paths from elements in vs.

$$\forall x\ .\ x \in \mathit{reaches}\ g\ \emptyset\ vs \iff x \in vs{\Downarrow}$$

Proof A proof is given in Möller (1993*b*) and in Clenaghan (1995). □

Lemma 6.5 (Prune)
The function call *prune* P *ts* for a finite set of vertices P and rational forest *ts* returns a subforest ($\subseteq_{\mathcal{F}}$) of *ts*. Formally,

$$\forall P, ts\ .\ \mathit{prune}\ P\ ts \subseteq_{\mathcal{F}} ts$$

Sketch Proof By well-founded induction. The well-ordering is defined on the visited set and the forest:

$$(Q, us) < (P, ts) \;=\; |Q| < |P| \;\vee\; (\mathit{size}\ us < \mathit{size}\ ts \;\wedge\; |Q| = |P|)$$

where *size* is the number of nodes in a forest. The ordering says that either the subsequent visited sets are becoming larger or subsequent trees are becoming smaller.

Then the following property is shown:

$$\forall Q, us, P, ts \;.\; ((Q, us) < (P, ts) \;\Rightarrow\; prune\ Q\ us \subseteq_{\mathcal{F}} us) \;\Rightarrow\; prune\ P\ ts \subseteq_{\mathcal{F}} ts$$

When the visited set P contains all nodes from ts then $prune\ P\ ts = [\,]$, this is easily shown. □

Lemma 6.6
If a node is in the result of a *prune* for a rational forest ts then it cannot have been in the finite set P passed to *prune*.

$$\forall x, P \;.\; x \in prune\ P\ ts \Rightarrow x \notin P$$

Proof By well founded induction. The well-ordering is defined on the size of forests and visited sets as before, and so the inductive formula is:

$$\forall Q, us, P, ts, x \;.\; ((Q, us) < (P, ts) \;\Rightarrow\; (x \in prune\ Q\ us \Rightarrow x \in us)) \Rightarrow\; (x \in prune\ P\ ts \Rightarrow x \in ts)$$

Case $[\,]$.

$$\begin{array}{ll} & x \in prune\ P\ [\,] \\ \iff & x \in [\,] \\ \iff & False \\ \Rightarrow & x \notin P \end{array}$$

Case $[\,t_0, \ldots, t_n\,]$ for $n \geq 0$.

$$\begin{array}{ll} & x \in prune\ P\ [\,t_0, \ldots, t_n\,] \\ \iff & \left\{ \text{Definition of } prune \right\} \\ & x \in prune\ P_0\ [t_0] +\!\!+ prune\ P_1\ [t_1] +\!\!+ \cdots +\!\!+ prune\ P_n\ [t_n] \end{array}$$

where $P_0 = P$ and $P_i = P_{i-1} \cup \mathcal{F}\,(prune\ P_0\ [\,t_0, \ldots, t_{i-1}\,])$ for $i > 0$. Without loss of generality there will exist some $0 \leq i \leq n$ such that:

$$\begin{array}{ll} \Rightarrow & x \in prune\ P_i\ [t_i] \\ \Rightarrow & \left\{ \text{Consider } t_i = Node\ v\ ts \right\} \\ & x \in prune\ P_i\ [\,Node\ v\ ts\,] \end{array}$$

$$
\begin{array}{ll}
\Rightarrow & \left\{ \text{By definition of } \mathit{prune} \right\} \\
& (v \in P_i \ \wedge \ x \in [\,]) \\
& \vee \ (v \notin P_i \ \wedge \ x \in \mathit{Node}\ v\ (\mathit{prune}\ (\{v\} \cup P_{i-1})\ ts)) \\
\Rightarrow & \left\{ \text{Since } x \notin [\,] \right\} \\
& v \notin P_i \ \wedge \ x \in \mathit{Node}\ v\ (\mathit{prune}\ (\{v\} \cup P_{i-1})\ ts) \\
\Rightarrow & v \notin P_i \ \wedge \ (x = v \ \vee \ x \in \mathit{prune}\ (\{v\} \cup P_{i-1})\ ts) \\
\Rightarrow & \left\{ \text{By induction hypothesis} \right\} \\
& x \notin P_i \ \wedge \ x \notin \{v\} \cup P_{i-1} \\
\Rightarrow & \left\{ \text{Since } x \notin \{\,v\,\} \cup P_{i-1} \cup P_i \text{ and } P \subseteq P_i \right\} \\
& x \notin P
\end{array}
$$

□

The following lemma says that if the result of a *dfs* can be partitioned then each partitioning can be defined in terms of a *prune*.

Lemma 6.7
For some graph g and vertices vs then the following holds,

$$
\begin{array}{l}
\forall ts.\, us\ .\ ts +\!\!+ us = \mathit{dfs}\ g\ vs \ \Rightarrow \ \exists xs, ys\ .\ vs = xs +\!\!+ ys \\
\qquad \wedge\ ts = \mathit{prune}\ \emptyset\ (\mathit{generates}\ g\ xs) \\
\qquad \wedge\ us = \mathit{prune}\ (xs\Downarrow)\ (\mathit{generates}\ g\ ys)
\end{array}
$$

Proof

$$
\begin{array}{ll}
& ts +\!\!+ us = \mathit{dfs}\ g\ vs \\
\Rightarrow & \left\{ \text{By unfolding } \mathit{dfs} \text{ and letting } vs = [\,v_0, \ldots, v_n\,] \text{ for some } n \geq 0 \right\} \\
& ts +\!\!+ us = \mathit{prune}\ \emptyset\ (\mathit{generates}\ g\ [\,v_0, \ldots, v_n\,]) \\
\Rightarrow & \left\{ \text{By } \mathit{prune} \text{ and } \mathit{generates} \right\} \\
& ts +\!\!+ us = \mathit{prune}\ P_0\ (\mathit{generates}\ g\ [v_0]) \\
& \quad +\!\!+ \mathit{prune}\ P_1\ (\mathit{generates}\ g\ [v_1]) \\
& \qquad \vdots \\
& \quad +\!\!+ \mathit{prune}\ P_n\ (\mathit{generates}\ g\ [v_n])
\end{array}
$$

where $P_0 = \emptyset$ and $P_i = \mathcal{F}\,(\mathit{prune}\ (\mathit{generates}\ g\ [\,v_0, \ldots, v_{i-1}\,]))$ for some $0 \leq i \leq n$. Furthermore, by using the definition of *reaches* and by using Theorem 6.4 we have $P_i = [\,v_0, \ldots, v_{i-1}\,]\Downarrow$. Now each element $\mathit{prune}\ P_i\ (\mathit{generates}\ g\ [v_i])$ is either a singleton list holding one tree or it is the empty list. Whence, we are able to choose as

much of *vs* as is necessary (call this amount *xs*) in order to construct *ts*. From this, and using the name *ys* for the remaining segment, then we have

$$\begin{aligned} \Rightarrow \quad & \exists xs, ys \;.\; vs = xs +\!\!+ ys \\ & \wedge\; ts = prune\ \emptyset\ (generates\ g\ xs) \\ & \wedge\; us = prune\ (xs{\Downarrow})\ (generates\ g\ ys) \end{aligned}$$

□

6.4 Correctness of DFS

Now the correctness of DFS may be shown by using the above properties of *prune* and *generates*.

Theorem 6.8 (The function *dff* satisfies Property 6.1)
The function call *dff g* returns a spanning subgraph of the graph *g*.

Proof There are two parts to this proof, first all the vertices in the graph *g* must be shown to be in *dff g*, that is:

(i) $\quad \forall v \;.\; v \in \mathcal{F}\,(dff\ g) \iff v \in vertices\ g$

and second it must be shown that all tree edges in the *dff g* are graph edges in *g*, that is:

(ii) $\quad \forall e \;.\; e \in edgesF\ (dff\ g) \Rightarrow e \in edges\ g$

Since sets have been used throughout, parts (i) and (ii) are not strong enough to show that multiple vertices or multiple edges do not appear in the depth-first forest. Nevertheless, it is straightforward to reformulate everything with lists and verify this.

□

Proof (i)

$$\begin{aligned} \forall v \;.\; v \in \mathcal{F}\,(dff\ g) \quad & \iff \quad v \in \mathcal{F}\,(prune\ \emptyset\ (generates\ g\ (vertices\ g))) \\ & \iff \quad v \in reaches\ g\ \emptyset\ (vertices\ g) \\ & \iff \quad v \in vertices\ g \end{aligned}$$

□

Before verifying part (ii) we introduce set definitions for *edgesT* and *edgesF*.

Definition (Edges of trees and forests)
The expression *edgesT t* is the set of all edges in the tree t, and *edgesF ts* is the set of all edges contained in the forest *ts*. Formally,

$$\begin{aligned} edgesT\ (Node\ x\ ts) &= \{\,(x,y) \mid Node\ y\ ts' \leftarrow ts\,\} \\ &\quad \cup\ edgesF\ ts \\ edgesF\ ts &= \bigcup_{t \in_{\mathcal{T}} ts} edgesT\ t \end{aligned}$$

Proof (ii)

Since,

$$\begin{aligned} dff\ g &= prune\ \emptyset\ (generates\ g\ (vertices\ g)) \\ &\subseteq_{\mathcal{F}} generates\ g\ (vertices\ g) \end{aligned}$$

the theorem is proved by showing the following

$$\forall e\ .\ e \in edgesF\ (generates\ g\ (vertices\ g)) \Rightarrow e \in edges\ g$$

this is shown by straightforward equational reasoning. □

Proposition 6.9 (The function *dff* satisfies Property 6.2)
The function call *dff g* returns a forest where there are no graph edges between left and right subtrees. In the following $ts +\!\!+ us$ is a subforest occurring anywhere within *dff g*.

$$\forall ts +\!\!+ us \in_{\mathcal{F}} dff\ g\ \wedge\ v \in ts\ \wedge\ w \in us\ \Rightarrow\ (v,w) \notin edges\ g$$

The ban on left-right cross edges translates into paths, and is expressed with the following two lemmas. 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.

Lemma 6.10 (No left-right paths between top-level trees)
If $(ts +\!\!+ us = dfs\ g\ vs)$, then $\forall v \in ts\ .\ \forall w \in us\ .\ v \not\rightarrow w$

Proof

$$
\begin{aligned}
& ts +\!\!+\ us = dfs\ g\ vs\ \wedge\ v \in ts\ \wedge\ w \in us \\
\Rightarrow\quad & \left\{ \text{Definition of } dfs \right\} \\
& ts +\!\!+\ us = prune\ \emptyset\ (generates\ g\ vs) \wedge\ v \in ts\ \wedge\ w \in us \\
\Rightarrow\quad & \left\{ \text{Using Lemma 6.7 partition } vs \text{ into two lists } vs = xs +\!\!+\ ys \right. \\
& \quad \left. \text{choosing } xs \text{ such that } ts = prune\ \emptyset\ (generates\ g\ xs) \right\} \\
& ts = prune\ \emptyset\ (generates\ g\ xs) \\
& \wedge\ us = prune\ (xs\Downarrow)\ (generates\ g\ ys) \\
& \wedge\ v \in ts\ \wedge\ w \in us \\
\Rightarrow\quad & v \in prune\ \emptyset\ (generates\ g\ xs)\ \wedge\ w \in prune\ (xs\Downarrow)\ (generates\ g\ ys) \\
\Rightarrow\quad & \left\{ \text{By Theorem 6.4 and Lemma 6.6} \right\} \\
& v \in xs\Downarrow\ \wedge\ w \notin xs\Downarrow \\
\Rightarrow\quad & \left\{ \text{Definition of } \Downarrow \right\} \\
& v \not\twoheadrightarrow w
\end{aligned}
$$

□

Deeper within each tree of the forest, there *can* be paths that 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. In other words 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. This may be formally expressed:

Lemma 6.11
If the tree ($Node\ x\ (ts +\!\!+\ us)$) is a subtree occurring anywhere within $dff\ g$, then

$$\forall v \in ts\ .\ \forall w \in us\ .\ v \longrightarrow w \Rightarrow v \longrightarrow x$$

Unfortunately we don't have a calculational style proof of this lemma. It turns out to be difficult because the proof requires knowledge of the depth-first forest creation process. Nevertheless, the lemma may be shown by reasoning about a process, which is a common style of proof given in traditional texts. An informal argument in this style is now given.

Since w is in us and only one w can exist in $dff\ g$, w is not in ts. If there is a path from v to w, w would become a descendant of v unless the path from v to w contains

a vertex that has been visited before (call it p). Vertex p will either be an ancestor of v or in a previously visited tree. If p is in a previously visited tree then w would also be in a previously visited tree, since $p \longrightarrow w$. But w is in us which occurs to the right of ts. On the other hand, it is possible for p to be an ancestor of v, and w to be a descendant of p. Hence $v \longrightarrow p$ where $(Node\ p\ (ts +\!\!+ us))$ is a subtree occurring anywhere within $dff\ g$.

6.4.1 Ordering properties of DFS

Now two ordering properties are given that show the relationship between the initial order of vertices given to *dfs*, and the structure of the forest.

Lemma 6.12 (Initial ordering property)
The function *dfs* satisfies Property 6.3.

$$as +\!\!+ [\,Node\ a\ bs\,] +\!\!+ cs = dfs\ g\ vs \;\Rightarrow\; \forall b \in [\,Node\ a\ bs\,] +\!\!+ cs\ .\ a \leq_{vs} b$$

for a and b in vs. The notation $\leq_{vs}$ is used for the ordering induced by the list of vertices vs. that is, $v \leq_{vs} w$ if $v = w$ or if v occurs earlier in vs than w.

Proof First the left part of the implication is transformed:

$$\begin{aligned} & as +\!\!+ [\,Node\ a\ bs\,] +\!\!+ cs = dfs\ g\ vs \\ \Longleftrightarrow \quad & \{\ \text{Definition of } dfs\ \} \\ & as +\!\!+ [\,Node\ a\ bs\,] +\!\!+ cs = prune\ \emptyset\ (generates\ g\ vs) \end{aligned}$$

Now we use Lemma 6.7 twice. First partitioning vs into $vs = (xs +\!\!+ [z]) +\!\!+ ys$ such that $as +\!\!+ [\,Node\ a\ bs\,] = prune\ \emptyset\ (generates\ g\ (xs +\!\!+ [z]))$. Then partitioning $xs +\!\!+ [z]$ such that $as = prune\ \emptyset\ (generates\ g\ xs)$, clearly $z = a$ and we have,

$$\begin{aligned} \Rightarrow \quad & as = prune\ \emptyset\ (generates\ g\ xs) \\ & \wedge\ [\,Node\ a\ bs\,] = prune\ P_1\ (generates\ g\ [a]) \\ & \wedge\ cs = prune\ P_2\ (generates\ g\ ys) \end{aligned}$$

where $P_1 = reaches\ g\ \emptyset\ xs$ and $P_2 = P_1 \cup reaches\ g\ P_1\ [a]$.

Now,

$$
\begin{array}{ll}
& b \in [\mathit{Node}\ a\ bs] +\!\!+ cs \\
\Rightarrow & b \in \mathit{prune}\ P_1\ (\mathit{generates}\ g\ [a]) +\!\!+ \mathit{prune}\ P_2\ (\mathit{generates}\ g\ ys) \\
\Rightarrow & \quad \left\{ \text{By Lemma 6.6} \right\} \\
& b \notin P_1 \cup P_2 \\
\Rightarrow & \quad \left\{ \text{Theorem 6.4} \right\} \\
& b \notin xs\Downarrow \\
\Rightarrow & \quad \left\{ \text{As } b \notin xs \text{ and } b \in xs +\!\!+ [a] +\!\!+ ys \right\} \\
& b \in [a] +\!\!+ ys \\
\Rightarrow & a \leq_{vs} b
\end{array}
$$

□

This second property is used in a proof of the strongly connected components algorithm given later.

Lemma 6.13
Let a and b be any two vertices. Write $\longrightarrow$ for paths in the graph g, and $\leq$ for the ordering induced by the list of vertices vs. Then

$$
\begin{array}{l}
\exists t \in_T \mathit{dfs}\ g\ vs\ .\ a \in t\ \wedge\ b \in t \\
\quad \Longleftrightarrow\ \ \exists c\ .\ c \longrightarrow a\ \wedge\ c \longrightarrow b \\
\qquad\qquad \wedge\ (\forall d\ .\ d \longrightarrow a\ \vee\ d \longrightarrow b\ \Rightarrow\ c \leq d)
\end{array}
$$

This Lemma says that:

(⇒) given two vertices that occur within a single depth-first tree (taken from the forest), then there is a predecessor of both (with respect to $\longrightarrow$) that 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).

(⇐) 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).

Proof

$(\Rightarrow)$

$$\exists t \in_T dfs\ g\ vs\ .\ a, b \in t$$
$$\Rightarrow\ \exists ts, us, xs, c\ .\ ts +\!\!+ [\,Node\ c\ xs\,] +\!\!+ us = dfs\ g\ vs\ \wedge\ a, b \in Node\ c\ xs$$
$$\Rightarrow\ \left\{ \text{Theorem 6.8, and excluded middle} \right\}$$
$$\ldots \wedge (c \longrightarrow a\ \wedge\ c \longrightarrow b)\ \wedge\ (\forall d\ .\ d \not\longrightarrow a\ \vee\ d \longrightarrow a)$$
$$\Rightarrow\ \left\{ \text{Lemma 6.10, } a \in \{c\} \cup \mathcal{F}\, xs \cup \mathcal{F}\, us \right\}$$
$$\ldots \wedge (c \longrightarrow a\ \wedge\ c \longrightarrow b)$$
$$\wedge (\forall d\ .\ d \not\longrightarrow a\ \vee\ d \in \{\,c\,\} \cup \mathcal{F}\, xs \cup \mathcal{F}\, us)$$
$$\Rightarrow\ \left\{ \text{Lemma 6.12} \right\}$$
$$\ldots \wedge (c \longrightarrow a\ \wedge\ c \longrightarrow b)\ \wedge\ (\forall d\ .\ d \not\longrightarrow a\ \vee\ c \leq d)$$
$$\Rightarrow\ \left\{ \text{Similarly for } b \text{ as for } a \right\}$$
$$\ldots \wedge (c \longrightarrow a\ \wedge\ c \longrightarrow b)$$
$$\wedge (\forall d\ .\ (d \not\longrightarrow a\ \vee\ c \leq d)\ \wedge\ (d \not\longrightarrow b\ \vee\ c \leq d))$$
$$\Rightarrow\ \exists c\ .\ c \longrightarrow a\ \wedge\ c \longrightarrow b\ \wedge\ (\forall d\ .\ d \longrightarrow a\ \vee\ d \longrightarrow b\ \Rightarrow\ c \leq d)$$

$(\Leftarrow)$

$$\exists c\ .\ c \longrightarrow a\ \wedge\ c \longrightarrow b\ \wedge\ (\forall d\ .\ d \longrightarrow a\ \vee\ d \longrightarrow b \Rightarrow c \leq d)$$
$$\Rightarrow\ \left\{ \text{By spanning property, } a, b, c \in dfs\ g\ vs, \text{ consider } c \in t \right\}$$
$$\ldots \wedge\ ts +\!\!+ [t] +\!\!+ us = dfs\ g\ vs\ \wedge\ c \in t$$
$$\wedge (a \in ts\ \vee\ a \in t\ \vee\ a \in us)$$
$$\Rightarrow\ \left\{ \text{By no left-right edges (Lemma 6.10) } c \longrightarrow a\ \wedge\ c \in t \Rightarrow a \notin us \right\}$$
$$\ldots \wedge\ ts +\!\!+ [t] +\!\!+ us = dfs\ g\ vs\ \wedge\ c \in t\ \wedge\ (a \in ts\ \vee\ a \in t)$$
$$\Rightarrow\ \left\{ \text{Assume } a \in ts, \text{ and consider } a \in Node\ e\ bs \right\}$$
$$\ldots \wedge\ as +\!\!+ [\,Node\ e\ bs\,] +\!\!+ cs +\!\!+ [t] +\!\!+ us = dfs\ g\ vs\ \wedge\ c \in t$$
$$\wedge\ a \in Node\ e\ bs$$
$$\Rightarrow\ \left\{ \text{By initial ordering (Lemma 6.12)} \right\}$$
$$\ldots \wedge ((a \in ts\ \wedge\ (\exists e\ .\ e \longrightarrow a\ \wedge\ e \leq c))\ \vee\ a \in t)$$
$$\Rightarrow\ \ldots\ (\forall d\ .\ d \longrightarrow a \Rightarrow c \leq d)$$
$$\wedge ((a \in ts\ \wedge\ (\neg\forall e\ .\ e \longrightarrow a \Rightarrow c < e))\ \vee\ a \in t)$$
$$\Rightarrow\ \ldots\ \wedge\ a \in t$$
$$\Rightarrow\ \left\{ \text{Similarly for } b \text{ as for } a \right\}$$
$$\exists t, ts, us\ .\ a, b \in t\ \wedge\ ts +\!\!+ [t] +\!\!+ us = dfs\ g\ vs$$

□

6.5 Efficient implementation of prune

The easiest way to achieve an efficient implementation of `prune` 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 is done here.

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.

The set-operations required are initialisation (the empty set), membership test, and addition of a singleton. While it is acceptable to spend linear time in generating the empty set (as it is only done once), it is essential that the other operations are performed in constant time.

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
```

Using these, `prune` is therefore defined as:

```
prune :: Bounds -> Forest Vertex -> Forest Vertex
prune bnds ts = runST (do { m <- mkEmpty bnds;
                            chop m ts
                          })
```

The `prune` function begins by introducing a fresh state thread, then generates an empty set within that thread and calls `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 []                    = return []
chop m (Node v ts : us) = do { visited <- contains m v;
                               if visited then
                                 chop m us
                               else do { include m v;
                                         as <- chop m ts;
                                         bs <- chop m us;
                                         return (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 to the set of marked vertices. Once this is complete, the pruned subforest is named `as`, and the remainder of the original forest is chopped. The result of this is, in turn, named `bs`, and the resulting 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. This is not as restrictive as it may at first seem, however, since all the trees must be computed by DFS, anyway, in order to produce the last one.

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. The flexibility is there to gain the best of both worlds: destructive update is only used where it is vital, everywhere else we may use the powerful modularity options provided by lazy functional languages.

6.5.1 Equivalence of stateful prune with purely functional prune

An equivalence is now shown of the specification of *prune* (p. 88) with the imperative implementation of `prune` given in the last section. Equivalent in the sense that, if the two functions are given the same arguments, they will return the same value. First another version of prune is derived from the specification:

$$
\begin{array}{lll}
sprune :: Forest\ Vertex \rightarrow Set\ Vertex \rightarrow (Forest\ Vertex, Set\ Vertex) & & \\
sprune\ [\,]\ P & = & ([\,], P) \\
sprune\ (Node\ x\ ts : us)\ P \mid x \in P & = & sprune\ us\ P \\
\quad \mid x \notin P & = & (Node\ x\ as : bs, R) \\
& & \textbf{where} \\
& & \quad (as, Q) = sprune\ ts\ (\{x\} \cup P) \\
& & \quad (bs, R) = sprune\ us\ Q
\end{array}
$$

Theorem 6.14

For a list of trees ts and a set of vertices P:

$$prune\ P\ ts = fst(sprune\ ts\ P)$$

Sketch Proof

The proof uses the transformation technique known as *tupling* (Burstall and Darlington 1977). The function *sprune* is derived from *prune* by using the following tuple structure:

$$sprune\ ts\ P = (prune\ P\ ts, P \cup \mathcal{F}\,(prune\ P\ ts))$$

Using case analysis *prune* is unfolded until we have an instance of the above property. When an instance occurs we fold back, giving the above recursive definition of *sprune*. □

The function *sprune*, although purely functional, is a state manipulator. The state in *sprune* being the set of visited vertices. By using the definitions of (;) and *return*,

sprune may be rewritten as follows:

```
sprune :: Forest Vertex → ST s (Forest Vertex)
sprune []                 =  return []
sprune (Node v ts : us)   =  do { visited ← contains v;
                                  if visited then
                                      sprune us
                               else  do { include v;
                                          as ← sprune ts;
                                          bs ← sprune us;
                                          return (Node v as : bs)
                                          }
                               }
```

where

```
contains :: Vertex → ST s Bool
contains v  =  \P → (v ∈ P, P)
```

```
include :: Vertex → ST s ()
include v   =  \P → ((), {v} ∪ P)
```

Theorem 6.15
For a list of trees *ts*:

$$sprune\ ts\ \emptyset = prune\ bnds\ ts$$

where *bnds* defines the range of vertices used. This version of *prune* refers to the implementation given on page 99.

Sketch Proof The definition of `chop` is visibly the same as *sprune* modulo `chop` taking a reference argument. The chief difference is in the way sets are represented, i.e. the definitions of *contains* and *include*. The formal proof relies on showing that arrays can be used to represent sets, which is well-known (Aho et al. 1983). The details of this are left out here. Proposition 6.16 is the critical transformation, that converts between a functional and an imperative program. □

Proposition 6.16 (*runST*-introduction)
Given a functional expression e, the following holds:

$$e = runST\ (return\ e)$$

6.6 Depth-first search algorithms

6.6.1 Depth-first search numbering

Having specified and implemented DFS we turn to consider how it may be used. 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.

Depth-first ordering of a graph is expressed 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. The use of repeated appends (`++`) caused by `concat` introduces an extra logarithmic factor here for the average case, but this is easily removed using standard transformations.

```
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)
```

It is often convenient, however, to translate such an ordered list into actual numbers. For this the function `tabulate` could be used:

```
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.

These can be packaged up into a function as follows:

```
preArr :: Bounds -> Forest Vertex -> Table Int
preArr bnds ts = tabulate bnds (preorderF ts)
```

(it turns out to be convenient for later algorithms if such functions take the depth-first forest as an argument, rather than construct the forest themselves.)

6.6.2 Topological sorting

The converse 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, postorder is define

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

The lack of left-right cross edges in DFS forests leads to a pleasant property when a DFS forest is flattened in postorder. This is expressed with the following definition.

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

$$v \leq w \ \wedge \ v \longrightarrow w \ \Rightarrow \ \exists u \ . \ v \longleftrightarrow u \ \wedge \ w \leq u$$

(where $v \longleftrightarrow u$ means $v \longrightarrow u$ and $u \longrightarrow v$). In words, this definition states that, if from some vertex v there is a path to a vertex later in the ordering, then there is

also a vertex u which occurs no earlier than w and which, like w is also reachable by a path from v. In addition, however, there is also a path from u to v.

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

Theorem 6.17
If $vs = postOrd\ g$, then the order in which the vertices appear in vs 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 $v \longrightarrow w$, by Lemma 6.10, v and w cannot be in different trees of the forest. Then by Lemma 6.11, the lowest common ancestor of v and w will do. □

All this can be applied 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 from v_j to v_i where $i < j$. This problem arises quite frequently, where a set of tasks need to be scheduled, such that every task can only be performed after the tasks it depends on are performed.

We define,

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

Theorem 6.18 (Topological sort)
Given an acyclic directed graph g,

$$\forall a, b \in topSort\ g\ .\ a \longrightarrow b \Rightarrow a \leq b$$

Proof

$$
\begin{array}{ll}
& \forall a, b \in \mathit{topSort}\ g\ .\ a \longrightarrow b \\
\Rightarrow & \quad \left\{ \text{Excluded middle, } (\leq_p) \text{ is defined by } \mathit{postOrd} \right\} \\
& a \leq_p b\ \vee\ b \leq_p a \\
\Rightarrow & \quad \left\{ (\leq_p) \text{ is a post-ordering, Theorem 6.17} \right\} \\
& (\exists c.\ a \longleftrightarrow c\ \wedge\ b \leq_p c)\ \vee\ b \leq_p a \\
\Rightarrow & \quad \left\{ \text{As } g \text{ is acyclic, the first disjunct is false when } a \neq b \right\} \\
& a = b\ \vee\ b \leq_p a \\
\Rightarrow & \quad \left\{ (\leq_p)\ \Longleftrightarrow\ (\geq) \right\} \\
& a \leq b
\end{array}
$$

□

6.6.3 Weakly 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 *weakly 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.

This is translated 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)
```

A graph is connected if there is exactly one component:

```
isConnected :: Graph -> Bool
isConnected g = length (components g) == 1
```

Theorem 6.19 (Connected components)
Given a directed graph g,

$$\exists t \in_{\mathcal{T}} \mathit{components}\ g\ .\ a, b \in t\ \Longleftrightarrow\ a \longleftrightarrow_g v\ b$$

The notation g^U is the undirected graph such that all directed edges in g are undirected edges in g^U.

Proof

($\Rightarrow$)

$$
\begin{array}{ll}
& \exists t \in_T \mathit{components}\ g\ .\ a, b \in t \\
\iff & \quad \{\ \text{Definition of components}\ \} \\
& \exists t \in_T \mathit{dff}\ (\mathit{undirected}\ g)\ .\ a, b \in t \\
\Rightarrow & \quad \{\ \text{Take a common ancestor } x \text{ of } a \text{ and } b\ \} \\
& x \longrightarrow_t a\ \wedge\ x \longrightarrow_t b \\
\Rightarrow & \quad \{\ \text{Lemma tree edges} \longrightarrow_t \text{are graph edges} \longleftrightarrow_g U\ \} \\
& x \longleftrightarrow_g U\ a\ \wedge\ x \longleftrightarrow_g U\ b \\
\Rightarrow & \quad \{\ \text{Transitivity}\ \} \\
& a \longleftrightarrow_g U\ b
\end{array}
$$

($\Leftarrow$)

$$
\begin{array}{ll}
& a \longleftrightarrow_g U\ b \\
\Rightarrow & \quad \{\ \text{By spanning Property 6.1 } a, b \in \mathit{dff}\ g^U\ \} \\
& a,\ b \in \mathit{dff}\ (\mathit{undirected}\ g) \\
\Rightarrow & \quad \{\ \text{Choose } t \in_T \mathit{dff}\ (\mathit{undirected}\ g) \text{ such that } a \in t\ \} \\
& as +\!\!+ [t] +\!\!+ bs = \mathit{dff}\ (\mathit{undirected}\ g)\ \wedge\ a \in t \\
\Rightarrow & \quad \{\ \text{By excluded middle}\ \} \\
& (b \in as\ \vee\ b \in t\ \vee\ b \in bs)\ \wedge\ a \in t \\
\Rightarrow & \quad \{\ \text{By no left-right cross edges (Lemma 6.10) } b \notin bs\ \} \\
& (b \in as\ \vee\ b \in t)\ \wedge\ a \in t \\
& \quad \{\ \text{Contradiction if } b \in as, \text{ as } b \longrightarrow_g U\ a, \text{ by Lemma 6.10, } a \in as\ \} \\
\Rightarrow & \ a \in t\ \wedge\ b \in t
\end{array}
$$

□

6.6.4 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. This problem 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 Sharir (1981).

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

The vertices of a graph are ordered using `postOrd`. 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. Nonetheless, it may be proved as follows.

Theorem 6.20 (Strongly connected components)
Let a and b be any two vertices of g. Then

$$(\exists t \in_T scc\ g\ .\ a \in t\ \wedge\ b \in t) \iff a \longleftrightarrow b$$

Proof

The proof proceeds by calculation. The notation g^T will be used for the transpose of g. Edges $\longrightarrow$ in g^T will be edges $\longleftarrow$ in g. Further, let $\leq$ be the post-ordering defined by $postOrd\ g$. Then its reversal induces the ordering $\geq$. Now,

$$\begin{array}{ll}
& \exists t \in_T scc\ g\ .\ a \in t\ \wedge\ b \in t \\
\iff & \quad \left\{ \text{Definition of } scc \right\} \\
& \quad \exists t \in_T dfs\ g^T\ (reverse\ (postOrd\ g))\ .\ a, b \in t \\
\iff & \quad \left\{ \text{By Lemma 6.13} \right\} \\
& \quad \exists c\ ,\ c \longleftarrow a\ \wedge\ c \longleftarrow b \\
& \quad \wedge\ (\forall d\ .\ d \longleftarrow a\ \vee\ d \longleftarrow b\ \Rightarrow\ c \geq d) \\
\iff & \quad \exists c\ .\ a \longrightarrow c\ \wedge\ b \longrightarrow c \\
& \quad \wedge\ (\forall d\ .\ a \longrightarrow d\ \vee\ b \longrightarrow d\ \Rightarrow\ d \leq c)
\end{array}$$

From here on are loop of implications is constructed.

$$
\begin{array}{l}
\exists c \,.\, a \longrightarrow c \;\wedge\; b \longrightarrow c \\
\quad \wedge\, (\forall d \,.\, a \longrightarrow d \;\vee\; b \longrightarrow d \;\Rightarrow\; d \leq c) \\
\quad \Rightarrow \qquad \left\{ \text{Consider } d = a \text{ and } d = b \right\} \\
\qquad \exists c \,.\, a \longrightarrow c \;\wedge\; a \leq c \;\wedge\; b \longrightarrow c \;\wedge\; b \leq c \\
\qquad\quad \wedge\, (\forall d \,.\, a \longrightarrow d \;\vee\; b \longrightarrow d \;\Rightarrow\; d \leq c) \\
\qquad \Rightarrow \qquad \left\{ \leq \text{ is a post-ordering} \right\} \\
\qquad \exists c \,.\, (\exists e \,.\, a \longleftrightarrow e \;\wedge\; c \leq e) \;\wedge\; (\exists f \,.\, b \longleftrightarrow f \;\wedge\; c \leq f) \\
\qquad\quad \wedge\, (\forall d \,.\, a \longrightarrow d \;\vee\; b \longrightarrow d \;\Rightarrow\; d \leq c) \\
\qquad \Rightarrow \qquad \left\{ e = c \text{ and } f = c \text{ using } (\forall d \ldots) \right\} \\
\qquad \exists c \,.\, a \longleftrightarrow c \;\wedge\; b \longleftrightarrow c \\
\qquad \Rightarrow \qquad \left\{ \text{Transitivity} \right\} \\
\qquad a \longleftrightarrow b
\end{array}
$$

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

$$
\begin{array}{l}
a \longleftrightarrow b \\
\quad \Rightarrow \qquad \left\{ \text{There is a latest vertex reachable from } a \text{ or } b \right\} \\
\qquad a \longleftrightarrow b \;\wedge\; \exists c \,.\, (a \longrightarrow c \;\vee\; b \longrightarrow c) \\
\qquad\quad \wedge\, (\forall d \,.\, a \longrightarrow d \;\vee\; b \longrightarrow d \;\Rightarrow\; d \leq c) \\
\quad \Rightarrow \qquad \left\{ \text{Transitivity of } \longrightarrow \right\} \\
\qquad \exists c \,.\, a \longrightarrow c \;\wedge\; b \longrightarrow c \\
\qquad\quad \wedge\, (\forall d \,.\, a \longrightarrow d \;\vee\; b \longrightarrow d \;\Rightarrow\; d \leq c)
\end{array}
$$

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 Corman et al. (1990), for example) typically take many pages of wordy argument. In contrast, because an earlier algorithm is reused, its properties can also be reused, giving a compact proof. Similarly, we believe that it is because we are using the DFS 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.

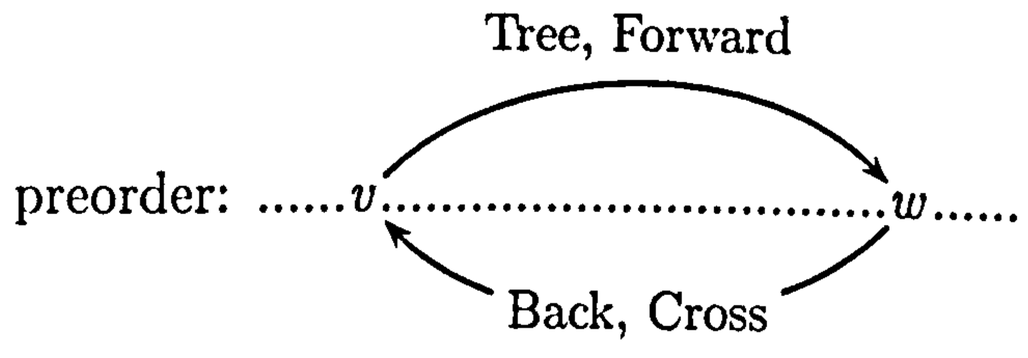
To determine if a graph is strongly-connected, the function `scc` is used to check if a single component is returned:

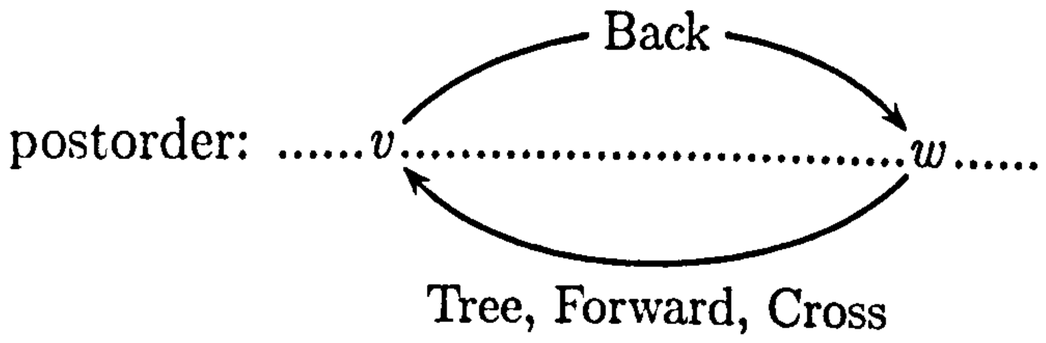
```
isScc :: Graph -> Bool
isScc g = length (scc g) == 1
```

6.6.5 Classifying edges

We have already seen the value of classifying the graph edges with respect to a given depth-first search. This idea is coded 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. The situation is summarised in the following diagram:





The above diagram expresses the relationship between the four types of edge (tree edges, forward edges, back edges, and cross edges) and the preorder and postorder numbers. 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 since we can already determine which are tree edges there is no problem. The implementation of these principles is now immediate and presented in Figure 6.4.

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

back :: Graph -> Table Int -> Graph  --  O(V+E)
back g post = mapA select g
  where  select v ws = [ w | w<-ws, post!v<post!w]

cross :: Graph -> Table Int -> Table Int -> Graph  --  O(V+E)
cross g pre post = mapA 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 = mapA select g
  where  select v ws = [ w | w<-ws, pre!v<pre!w] \\ tree!v
```

Figure 6.4 Classification of graph edges.

To classify an edge the depth-first spanning forest is generated, and used to produce preorder and postorder numbers. These numbers give all the information required to construct the appropriate subgraph. We have been slack with the implementations of `tree`, and `forward`. Neither of these implementations is linear-time. The function `tree` can be made to run in linear-time by making `edgesF` linear, this is achieved by using standard transformation techniques (Section 8.2). The function `forward` is not linear-time because of the quadratic list difference function. This inefficiency can be removed by ordering both lists, and using another list difference operator which takes

advantage of the ordering.

6.6.6 Detecting rooted graphs

A *root* of a graph is a vertex r such that every other vertex in the graph can be reached by a path from r. Hence,

$$\exists r \,.\, \forall v \in g \,.\, r \longrightarrow v$$

If we perform a DFS of a graph, and if a root exists it will clearly be in the final tree constructed. Otherwise there would be a left to right edge from the root. Furthermore, if the graph is rooted then the root of the last DFS tree will be a root of the graph. If performing a second DFS starting from the root of the last tree produces just one tree, then the graph is rooted, otherwise the graph has no root. So the algorithm is simply expressed as:

```
rooted :: Graph -> Bool  --  O(V+E)
rooted g = length ts == 1 ||
           length (dfs g (preorderF (reverse ts))) == 1
  where  ts = dff g
```

6.6.7 Finding reachable vertices

Finding all the vertices that are reachable from a single vertex `v` demonstrates that `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. This is then flattened with preorder to produce the desired list.

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

Lemma 6.21
Flattening the finite and well-defined forest `ts` with `preorderF ts` returns all the nodes that are contained in `ts`.

$$\forall x, ts \,.\, x \in \mathcal{F}\, ts \iff x \in preorderF\ ts$$

Proof By induction on tree depths, that is the following is shown,

$$\forall n \geq 0, x, ts \;.\; x \in \mathcal{F}\,(depthPruneF\ n\ ts) \iff x \in preorderF\ (depthPruneF\ n\ ts)$$

where `depthPrune` has the following definition:

```
depthPruneF :: Int -> [Tree a] -> [Tree a]
depthPruneF 0 ts = []
depthPruneF d ts = map (depthPrune d) ts

depthPrune :: Int -> Tree a -> Tree a
depthPrune d (Node x ts) = Node x (depthPruneF (d-1) ts)
```

□

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

```
path :: Graph -> Vertex -> Vertex -> Bool  -- O(V+E)
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 DFS forest ceases. Thus `dfs` implements a true *search* and not merely a complete *traversal.*

Theorem 6.22 (Paths)

$$path\ g\ v\ w \iff v \longrightarrow w$$

Proof

$$
\begin{array}{ll}
& path\ g\ v\ w \\
\Longleftrightarrow & \quad \{ \text{Definition of } path \} \\
& w \text{ `}elem\text{`} (reachable\ g\ v) \\
\Longleftrightarrow & \quad \{ \text{Unfolding definitions of } reachable \text{ and } dff \} \\
& w \text{ `}elem\text{`} (preorderF\ (prune\ \emptyset\ (generates\ g\ [v]))) \\
\Longleftrightarrow & \quad \{ \text{Lemma 6.21} \} \\
& w \in \mathcal{F}\ (prune\ \emptyset\ (generates\ g\ [v])) \\
\Longleftrightarrow & \quad \{ \text{Definition of } reaches \} \\
& w \in reaches\ g\ \emptyset\ [v] \\
\Longleftrightarrow & \quad \{ \text{By Theorem 6.4} \} \\
& w \in v{\downarrow} \\
\Longleftrightarrow & \quad \{ \text{Definition of } \downarrow \} \\
& w \in \{\, x \mid v \longrightarrow x \,\} \\
\Rightarrow & \quad v \longrightarrow w
\end{array}
$$

□

6.6.8 Biconnected components

This section looks at programming a more complex algorithm — finding *biconnected components*. An undirected graph is biconnected if the removal of any vertex leaves the remaining subgraph connected. A biconnected component is a maximal subgraph that is biconnected. This has a bearing in the problem of *reliability* in communication networks. For example, if you want to avoid driving through a particular town, is there an alternative route?

If a graph is not biconnected the vertices whose removal disconnects the graph are known as *articulation points*. Locating articulation points allows a graph to be partitioned into biconnected components (actually a partition of the *edges*). In Figure 6.5 vertices that are articulation points are marked with an asterisk. The naïve, brute force method is to remove each vertex in turn and check whether the remaining subgraph is connected. However, this would require $O(V(V+E))$ time, since a connectedness check takes $O(V+E)$ time. A more efficient algorithm is described

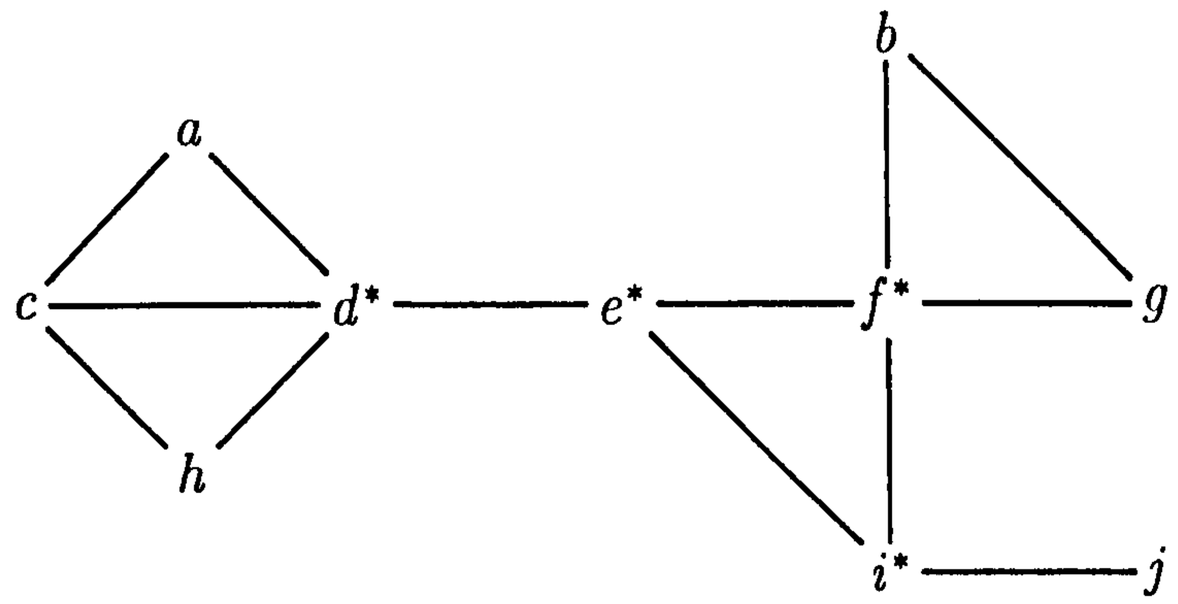


Figure 6.5 An undirected graph.

by Tarjan (1972), where biconnected components are found during the course of a depth-first search in $O(V + E)$ time. Here we apply the same theory as Tarjan, but express it via explicit intermediate values.

Tarjan's method is based on the following theorem:

Theorem 6.23
Given a depth-first spanning forest of a graph, v is an articulation point in the graph if and only if: (i) v is a root with more than one child; or (ii) v is not a root, and for all proper descendants w of v there are no edges to any proper ancestors of v.

This theorem is applied by associating a *low point* number with every vertex. The low point number of v is the smallest DFS numbered vertex that can be reached by following zero or more tree edges, and then along a single graph edge.

Low point numbers are calculated by traversing the DFS trees bottom-up, and associating each vertex with its low point number. The function `label` (see Figure 6.7) annotates a tree with both depth-first numbers and low-point numbers. At any vertex, the low point number is the minimum of:

(i) the DFS number of the vertex;

(ii) the DFS numbers of the vertices reached by a single edge; and

(iii) the low point numbers of the vertex's descendants in the tree.

For example, the result of running `label` on the DFS spanning tree produced from the graph in Figure 6.5, gives the annotated tree depicted in Figure 6.6.

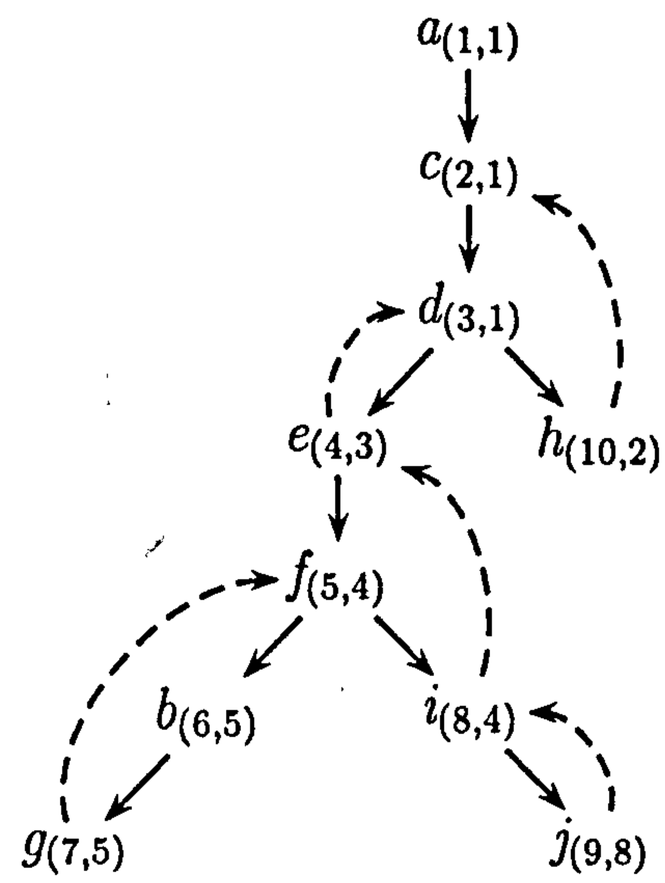


Figure 6.6 The depth-first forest for the undirected graph.

Dashed lines are the important back edges used for calculating low points. Tree nodes are triples, for instance, $e_{(4,3)}$, represents the triple $(e, 4, 3)$, where 4 is the depth-first number and 3 the low point number of vertex e.

From the low points for vertices, articulation points can be calculated. By part (ii) of Theorem 6.23 if the depth-first number of v is less than or equal to the low-point of all proper descendants w of v then v is an articulation point. But since the low-point numbers of descendants of v are always greater than or equal to the low-point for v, we can determine if v is an articulation point by checking the low-point numbers of its immediate children.

The function `collect` coalesces each DFS tree into a *biconnected tree*, that is, a tree where the node elements are biconnected components. At each node the DFS number is compared with the low-point number of all the children. If the child's low-point number is strictly less than the node's DFS number, then the component involving that vertex is not completed. On the other hand, if the node's DFS number is less than or equal to the child's low-point number, then that component is completed once the node is included. The function `bicomps` handles the special case of the root. Finally, `bcc` ties all the other functions together.

Coalescing the tree from Figure 6.6 will produce the following forest containing two trees.

While this algorithm is complex, again it is made up of individual components whose

```
bcc :: Graph -> Forest [Vertex]  --  O(V+E)
bcc g = (concat . map bicomps . map (label g dnum)) forest
  where  forest = dff g
         dnum = preArr (bounds g) forest

label :: Graph -> Table Int -> Tree Vertex -> Tree (Vertex,Int,Int)
label g dnum (Node v ts) = Node (v,dnum!v,lv) us
   where  us = map (label g dnum) ts
          lv = minimum ([dnum!v]++[ dnum!w | w<-g!v]
                                ++[ lu | Node (u,dw,lu) xs<-us])

bicomps :: Tree (Vertex,Int,Int) -> Forest [Vertex]
bicomps (Node (v,dv,lv) ts)
      = [ Node (v:vs) us | (l, Node vs us)<-map collect ts]

collect :: Tree (Vertex,Int,Int) -> (Int, Tree [Vertex])
collect (Node (v,dv,lv) ts) = (lv, Node (v:vs) cs)
  where  collected = map collect ts
         vs = concat [ ws | (lw, Node ws us)<-collected, lw<dv]
         cs = concat [ if lw<dv then us else [Node (v:ws) us]
                     | (lw, Node ws us)<-collected]
```

Figure 6.7 Biconnected components algorithm.

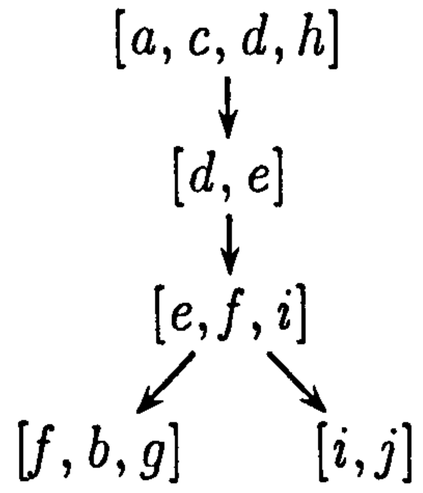


Figure 6.8 The biconnected trees.

correctness may (potentially at least) be established independently of the other components. This is quite unlike typical imperative presentations where the bones of the recursive DFS procedure are filled out with the other components of the algorithm, resulting in a single monolithic procedure.

A graph is biconnected when the number of biconnected components is 1, hence the following function:

```
isBcc :: Graph -> Bool  --  O(V+E)
isBcc g = length (bcc g) == 1
```

6.6.9 Finding bridges

A bridge is an edge whose deletion disconnects an undirected graph, and an edge is a bridge if and only if it does not lie on a cycle. Hence, a bridge is a biconnected component with exactly one edge. Therefore, all the bridges can be found in an undirected graph by returning all the components with two vertices.

```
bridges :: Graph -> [[Vertex]]  -  O(V+E)
bridges g = filter ((2==).length) (preorderF (bcc g))
```

Chapter 7

Graph algorithms

In this chapter several traditional graph algorithms are implemented. As much as possible purely functional implementations will be given. We will look at *weighted* graph problems, and some *dynamic* graph algorithms. Weighted problems are ones where the edges are labelled with some cost. The term dynamic graph algorithm is used here to classify the algorithms where it is necessary to change the graph during the course of the algorithm. These algorithms require state to be used throughout for their efficiency. Although a more functional solution to these problems is not ruled out, if one existed it would probably be more verbose than the imperative solution. Breadth-first search based algorithms will also be covered in this chapter. The breadth-first algorithm itself will be expressed purely functionally using the `lazyArray` combinator.

7.1 Kruskal's minimum spanning forest algorithm

Kruskal's (1956) algorithm takes an undirected graph, with the edges labelled with costs, and returns a spanning forest of minimum cost. The algorithm is expressed quite simply: repeatedly choose a new edge of minimum cost; add this edge to the spanning forest if and only if it does not form a cycle. This process is complete when all edges have been considered.

With the example graph in Figure 7.1, first the edge (d, c) is chosen, then (h, g), (f, c), (s, a), (d, e), (b, e), and (s, d). Next the edge (s, b) is chosen, but this forms the cycle b, e, d, s, so (s, b) is rejected. Finally (f, g) is chosen which completes the

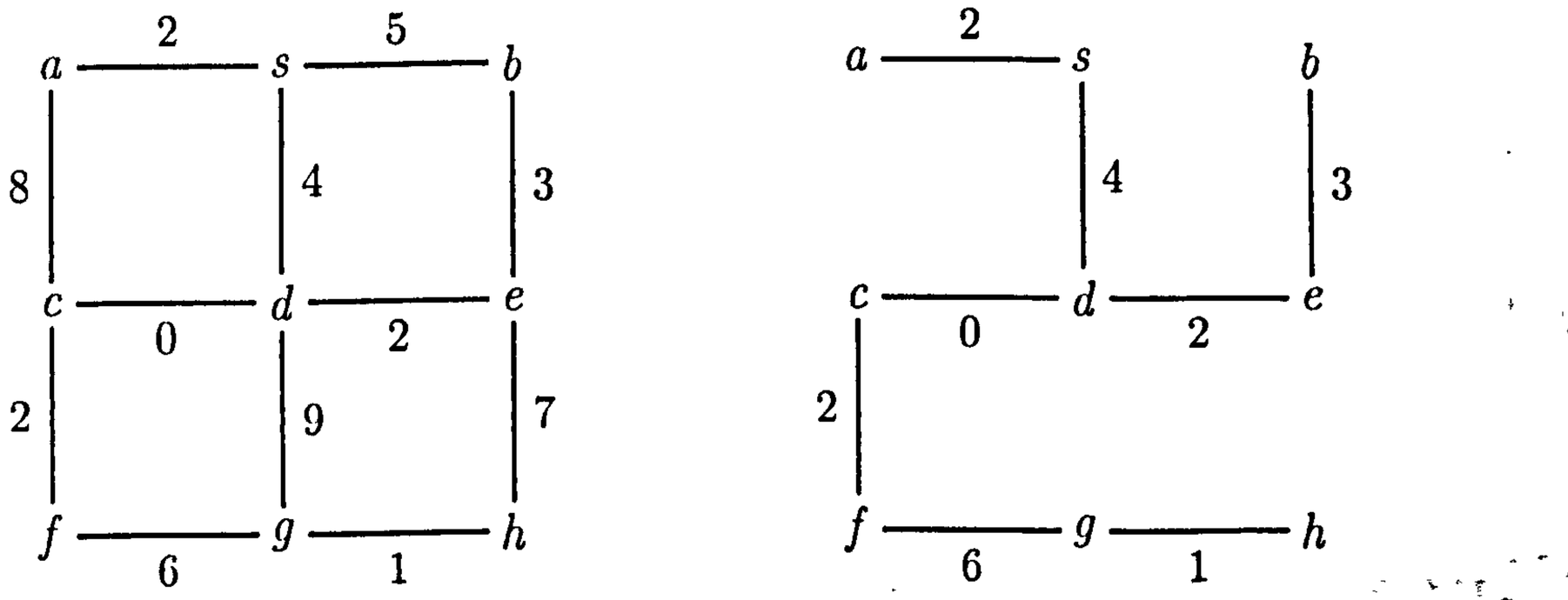


Figure 7.1 An undirected labelled graph, and minimum cost spanning tree.

tree, also shown in Figure 7.1.

The crux of an efficient implementation requires fast cycle detection. Cycles can be detected in almost constant time by using a good implementation of disjoint sets. Initially, each vertex is in a set of its own. When an edge is chosen the two disjoint sets that contain the endpoints of the edge are combined, thus graph components are being built-up. If two endpoints of a chosen edge are in the same component, then there is a cycle. In this case the edge is elided and another one is chosen. The edges are best stored in a priority queue, with their cost as the keys. So that at each stage in the process the item with minimum key is the next edge considered.

The disjoint set operations union/find used here have an almost constant running time, and were described in Section 4.8. The priority queue operations used here were implemented with a binomial queue (Section 3.4) which has an $O(\log E)$ worst case running time for `deleteMin` and `insertQ`. With these running times this implementation of Kruskal's algorithm should run in O(E log E) time.

```
initSet :: LGraph -> ST s (Set s Vertex)
initSet g = do { set <- newArr (limitsL g) Empty;
                 applyST (insSet set) (verticesL g);
                 return set
               }
        where  insSet set x = insElem set x [x]
```

Initialisation of the priority queue runs in O(E log E) time.

```
kruskal :: LGraph -> [Edge]
kruskal g = runST (do { set <- initSet g;
                        loop [] set (initQ g);
                      })

loop :: [Edge] -> Set s Vertex -> BinQ -> ST s [Edge]
loop es set q | q==emptyQ = return es
              | True      = do { (pu,nu) <- find set u;
                                 (pv,nv) <- find set v;
                                 if nu==nv then loop es set q'
                                   else do { union set pu pv nu;
                                             loop ((u,v):es) set q'
                                           }
                               }
      where q' = deleteMin q
            (u,v) = entry (findMin q)
```

Figure 7.2 Kruskal's minimum spanning forest algorithm.

```
initQ :: LGraph -> BinQ
initQ g = insertMany [ (e, weight g e) | e<-edgesL g]
```

7.2 Dijkstra's single-source shortest paths algorithm

Dijkstra (1959) presented two algorithms on undirected graphs, one of which is to find the shortest path between two given vertices. This is extended here to find the shortest path from a source vertex to every other vertex in the graph. Each vertex is labelled with its distance from the source, initially all vertices are marked with a sentinel value larger than any other, except the source which will have a label of 0. Then, we repeatedly choose the vertex with minimum distance and update all of its neighbours' distances.

In the example (Figure 7.3), vertex s is the source vertex. Initially, vertex s is chosen as it has the smallest distance from itself, and its neighbours a, b, and d have their distances updated. Next, a new vertex is chosen with minimum distance, in this case the vertex a is chosen, then its neighbours are updated. A vertex's distance from the source is only updated if the new path is of less cost than the old path, that is, the

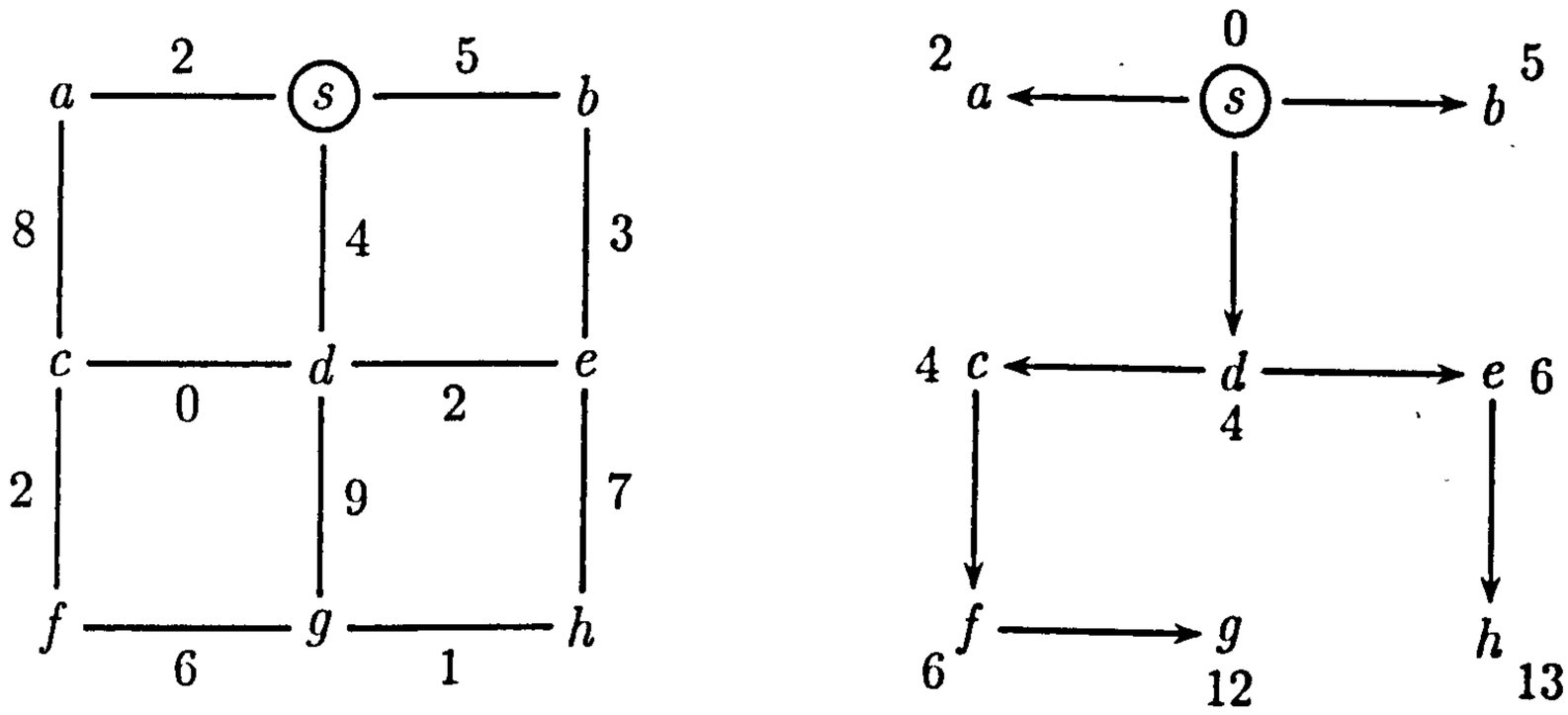


Figure 7.3 An undirected labelled graph, and shortest paths spanning tree.

new *distance* w is updated with:

$$min\ (distance\ w)\ (distance\ v + weight\ (v, w))$$

for neighbour w of v.

The implementation of Dijkstra's algorithm (Figure 7.4) is quite traditional. All the vertices are placed into a priority queue with their initial distances from the source as key. This is appropriate, since the required vertex is retrieved with `deleteMinPQ`, and `decreaseKey` is used when a vertex's distance is updated. The `Maybe` datatype is used for keys so that the sentinel distances are `Nothing` and known distances are `Just distance`. The priority queue is initialised with:

```
initQ :: LGraph -> Vertex -> BinQ
initQ g s = insertMany ((s,Just 0):[(v,Nothing) | v<-verticesL g, s/=v])
```

where `s` is the source vertex. Ordering is defined on labels so that the sentinel `Nothing` is larger than all defined distances:

```
instance Ord Label where
  Just a  <= Just b  = a<=b
  Nothing <= Just b  = False
  _       <= Nothing = True
```

An updatable array of distances is maintained throughout the algorithm, which forces us to remain inside the monadic code.

```
type Entry = Vertex
type key   = Label

dijkstra :: LGraph -> Array Vertex Label
dijkstra g s = runST (do { dist <- newArr (limitsL g) Nothing;
                           writeArr dist s (Just 0);
                           loop dist g (initQ g s);
                           freezeArr dist
                         })


loop :: MutArr s Vertex Label -> LGraph -> BinQ -> ST s ()
loop dist g q | isEmptyPQ q = return ()
              | otherwise   = do { us <- mapST getUps (succL g v);
                                   loop dist g (foldr decreaseKey q' us)
                                 }
  where (v,dv) = findMinPQ q
        q'     = deleteMinPQ q

      getUps w = do { dw <- readArr dist w;
                      let dw' = min dw (dv + weight g (v,w)) in
                      writeArr dist w dw';
                      return (w,dw')
                    }
```

Figure 7.4 Dijkstra's single-source shortest paths algorithm.

7.3 Floyd's all shortest paths algorithm

The *all-pairs shortest paths* problem is to compute the shortest paths from every vertex to every other vertex. For simplicity the problem is reduced to an algorithm to find just the lengths of the shortest paths. Our purely functional implementation is based on Floyd's (1962) algorithm, which is best described in terms of induction (Manber 1989). Vertices must be ordered. A path from v to w is a *k-path* if the highest vertex on the path, excluding v and w is k.

Inductive hypothesis: We know the lengths of the shortest paths between all pairs of vertices, considering all paths up to k-paths, for some $k < m$.

In the base case only directed edges are considered. As usual with induction we need to work out how to extend a solution for m to a solution for $m+1$. We now consider all k-paths such that $k < m+1$. So the only new paths we need to consider are m-paths. The shortest m-path between x and y, must contain m exactly once. It can be calculated by taking the shortest i-path (for some $i < m$) from $x \longrightarrow m$ and adding the shortest j-path (for some $j < m$) from $m \longrightarrow y$. By induction we already know all the shortest paths up to k-paths, hence only two lengths need to be summed.

```
allShortPaths :: LGraph -> LGraph
allShortPaths g = foldr induct g (verticesL g)

induct :: Vertex -> LGraph -> LGraph
induct m g = short
    where
      short = mapA (const.update) g
      update (x,y) | wgt(x,m)+wgt(m,y)<wgt(x,y) = wgt(x,m)+wgt(m,y)
                   | otherwise                  = wgt(x,y)
              where
                wgt (v,w) | v<x && w<y = weight short (v,w)
                          | otherwise  = weight g (v,w)
```

Figure 7.5 All-pairs shortest-paths problem.

Figure 7.5 gives a functional implementation of Floyd's algorithm. If there is no edge between two vertices then its length is ∞, and self-loops have length 0. The functional implementation runs in $O(V^3)$ time, since all-pairs of vertices are considered for each vertex. The difference between this implementation and traditional presentations is that a new array is created each time `induct` is called. This avoids using destructive update. The function `wgt` is need to determine if the length between two vertices should come from the new array being constructed, or from the old array.

7.3.1 Transitive closure

The *transitive closure* of a graph g is a graph h such that edge (v, w) is in h if and only if $v \longrightarrow_g w$. If there is a path between v and w then there must be a shortest path. Hence the transitive closure can be found by first using the all-pairs shortest paths

algorithm, and then creating an edge if there exists a shortest path. The algorithm follows:

```
transitive_closure :: LGraph -> LGraph
transitive_closure g = mapA (toEdge.const.isEdge short) short
      where
        short = allShortPaths g
        toEdge False = Nothing
        toEdge True  = Just 1
```

Figure 7.6 Transitive closure.

The implementation will not have the best performance, since the constant factor overhead is quite large. But nonetheless, its asymptotic complexity is $O(V^3)$, since we are mapping over the graph created by the all-pairs shortest paths algorithm.

7.4 Vertex colouring

The vertices of a graph can be coloured by ordering them and then colouring each vertex with the first available colour, taking account of the vertices already coloured. One way of ordering the vertices — which works quite well in practice, although it doesn't necessarily give the best colouring — is to order by vertex degrees. This heuristic was first recommended by Brelaz (1979).

```
colour :: Graph -> Table Vertex
colour g = col
    where
      col = array (bounds g) [ (v,paint v) | v<-vs]
      vertexOrd = array (bounds g) (zip vs [1..])
      vs = degreeOrd g
      paint v = head ([1..]\\[ col!w | w<-g!v,vertexOrd!w<vertexOrd!v])
```

Figure 7.7 A graph colouring algorithm.

Figure 7.7 gives a purely functional implementation of the vertex colouring algorithm. (This was written by Simon Peyton Jones after seeing my stateful version.) Colours

are represented by positive integers, which gives the ordering on them. Vertices are ordered in descending degree order by `degreeOrd`. A table `vertexOrd` is created by mapping vertices in this ordering with successive positive integers. Thus giving a total ordering of vertices. The colour table `col` is created by applying `paint` to each vertex in descending degree order. The function `paint` takes a vertex and looks at all of its coloured neighbours choosing the smallest colour (i.e. positive integer) that doesn't match.

The algorithm is linear $O(V + E)$ if the function `paint` is linear. It is linear because all graph vertices are considered, and for each one all its edges are considered. In the implementation, however, `paint` runs in $O(n^2)$ time, where n is the length of the list of neighbour's colours. Nevertheless, if the list of neighbours is sorted, and a list difference function is used to take this into account, then paint would run in $O(n)$ time.

7.5 Breadth-first search based algorithms

Breadth-first search is a graph traversal strategy that is important for a host of algorithms. The dual of breath-first search is depth-first search which was covered extensively in Chapter 6. A breadth-first search of a graph fans out exploring the adjacent vertices before penetrating deep into the graph. In the example shown in Figure 7.8 a breadth-first traversal commences from vertex a, and bold edges highlight the path taken.

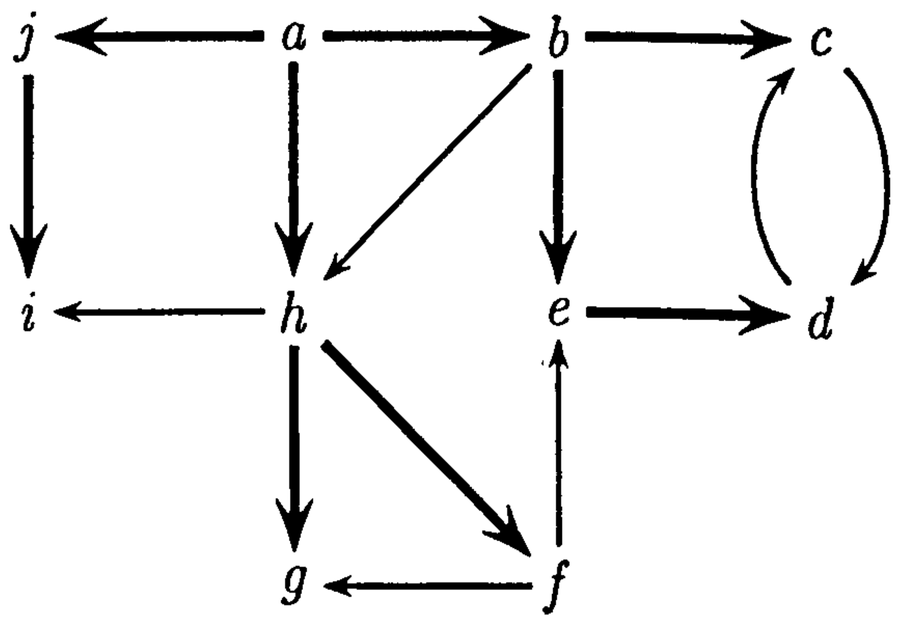


Figure 7.8 A directed graph: bold edges give breadth-first traversal.

7.5.1 Implementing BFS

Just like depth-first search, breadth-first search can be expressed as the composition of prune and generate. The only difference is that pruning is done in a breadth-first order.

```
bfs :: Graph -> [Vertex] -> [Tree Vertex]
bfs g vs = bfsPrune (generates v vs)
```

The implementation of breadth-first prune on graphs presented here is purely functional, and runs in linear time with respect to the size of the graph. It is based on two separate functional programming *tricks*. The first trick is a neat breadth-first labelling algorithm described by Jones and Gibbons (1992); and the second is based on a neat way of using the function `lazyArray` (Johnsson 1995) see Section 4.9. First we start with a breadth-first pruning algorithm, albeit an inefficient one:

```
bfsPrune :: [Tree Vertex] -> [Tree Vertex]
bfsPrune ts = us      where (us,ss) = traverse ts ([]:ss)

traverse :: [Tree Vertex] -> [[Vertex]] -> ([Tree Vertex], [[Vertex]])
traverse [] ss                    = ([],ss)
traverse (Node x ts:us) (s:ss) = if x `elem` s then (us',sn)
                                                else (Node x ts':us',sn)
      where   (ts',s1) = traverse ts ss
              (us',sn) = traverse us ((x:s):s1)
```

The cleverness lies in the way the second argument to `traverse` is demanded. This argument holds a list of states, where each state contains a list of the vertices currently visited. The first state is empty, and the second state contains the first root node, and so on. The subtrees of the first root node depend on later states, which in turn depend on later trees, hence the demand driven basis of the algorithm. The inefficiency here lies in the use of `elem`, which we now seek to remove.

First a table is constructed of breadth-first numbers for vertices. The function `bfsOrd` does a breadth-first traversal returning a list of vertex/breadth-first number pairs. The subtleness here lies the condition `bfsNo!x==n`, which will be true if this is the first time `x` has been visited. If `x` has been visited before, there will already be a

vertex/breadth-first number pair created by `bfsOrd`, and hence this will be contained in the `bfsNo` array.

```
bfsNum :: Bounds -> [Tree Vertex] -> Table Int
bfsNum bnds ts = bfsNo
  where
      bfsNo :: Table Int
      bfsNo = amap (\xs -> if null xs then 0 else head xs)
                   (lazyArray bnds (bfsOrd ts 1))

bfsOrd :: [Tree Vertex] -> Int -> [(Vertex,Int)]
bfsOrd [] n                = []
bfsOrd (Node x ts: us) n = (x,n):
        if bfsNo!x==n then bfsOrd (us++ts) (n+1)
                      else bfsOrd (us++ts) n
```

Note `bfsOrd` is not efficient because of repeated appends, but this can be removed by standard methods. Now that the table of breadth-first numbers is known, `bfsPrune` can be re-written efficiently. Instead of doing an `elem` test to check if a vertex has been visited before, `bfsNo` is used to check for previously visited vertices. The list of states now contains the current breadth-first number. If when visiting `x` its state number is the same as `bfsNum!x` then `x` has not been visited before.

7.5.2 Bfs numbering

Breadth-first numbers where used in the above implementation of BFS, so the above algorithm may be reused to produce a table of BFS numbers.

```
bfsNums :: Graph -> [Vertex] -> Table Int  --  O(V+E)
bfsNums g vs = bfsNum (bounds g) (bfs g vs)
```

7.5.3 Finding the diameter of a graph

The *diameter* of a graph is the longest of all the shortest paths between any two vertices. Where a path's length is considered to be the number of edges it contains.

```
bfsPrune :: (Vertex,Vertex) -> [Tree Vertex] -> [Tree Vertex]
bfsPrune b ts = us
  where
      (us,ns) = traverse ts (1:ns)
      bfsNo = bfsNum b ts

      traverse [] ns                    = ([],ns)
      traverse (Node x ts:us) (n:ns) = if b then (qs,ns')
                                            else (Node x ps:qs,ns')
        where
            (b,n')   = if bfsNo!x==n then (False,n+1)
                                     else (True,n)
            (ps,ms)  = traverse ts ns
            (qs,ns') = traverse us (n':ms)
```

Figure 7.9 Efficient BFS pruning.

This can be found by creating a breadth-first search from each vertex, which yields a shortest paths forest. Then it's simply a matter of finding the longest one, which is done by converting all tree paths to lists, and finding the longest list.

```
diameter :: Graph -> Int  --  O(V+E)
diameter g = depthF [ head (bfs g [v]) | v<-vertices g]

diameterPath :: Graph -> [Vertex]
diameterPath g = longestList (concat
                    [ paths (head (bfs g [v]))] | v<-vertices g])

paths :: Tree a -> [[a]]
paths (Node x []) = [[x]]
paths (Node x ts) = map (x:) (concat (map paths ts))
```

This version of paths is not the most efficient because of repeated appends caused by `concat`, but again the inefficiency can be removed by standard techniques (Section 8.2).

The auxiliary functions may be defined as follows:

```
longestList :: [[a]] -> [a]
longestList xss = snd (foldr f (0,[]) xss)
      where f xs (n,ys) = if m>n then (m,xs) else (n,ys)
              where m = length xs

depth :: Tree a -> Int
depth (Node x ts) = 1 + depthF ts

depthF :: Forest a -> Int
depthF []     = 0
depthF (t:ts) = max (depth t) (depthF ts)
```

7.5.4 Shortest path between two vertices

A similar algorithm to the diameter problem is to find the fewest number of edges between two vertices. This may be done by first constructing the breadth-first tree from a given vertex. All the paths are searched for the required vertex, and the paths are built up during the traversal.

```
path :: Graph -> Vertex -> Vertex -> [Vertex]
path g v w = reverse (collect t w [])
      where t = head (bfs g [v])

collect :: Tree Vertex -> Vertex -> [Vertex]
collect (Node x ts) w ps
              | x==w       = w:ps
              | otherwise = extract (map (\t->collect t w (x:ps)) ts)

extract :: [[a]] -> [a]
extract []                    = []
extract (xs:xss) | null xs    = extract xss
                 | otherwise = xs
```

7.5.5 Checking if a graph is bipartite

An undirected graph is *bipartite* if its vertices can be split into two sets, so that every edge contains one vertex in each set. If a component is bipartite, then in a breadth-first traversal, nodes at even numbered levels are in one set, and nodes at odd numbered levels are in another set. If the level numbering between two vertices in a graph edge is from odd to even or even to odd, then the component is bipartite.

```
isBipartite :: Graph -> [Vertex] -> Bool  --  O(V+E)
isBipartite g vs = and [ odd (depth!v - depth!w) | (v,w)<-edges g]
      where
              ts    = bfs g vs
              depth = depthArr (bounds g) ts

depthArr :: Bounds -> Forest Vertex -> Table Int
depthArr bnds ts = array bnds (preorderF (annotateF 1 ts))

annotateF :: Int -> Forest a -> Forest (a,Int)
annotateF n ts = map (ann n) ts
      where
          ann n (Node x ts) = Node (x,n) (annotateF (n+1) ts)
```

Figure 7.10 Checking if an undirected graph is bipartite.

7.6 Discussion

This chapter presented numerous graph algorithms in Haskell with no loss of efficiency. Some algorithms seem intrinsically to require state throughout such as: Kruskal's minimum spanning forest algorithm (Section 7.1); and Dijkstra's shortest paths algorithm (Section 7.2). These were called dynamic algorithms, because the graph changes during the algorithm. The use of state can sometimes be avoided, even although some information about parts of the graph changes during the algorithm. This was demonstrated with algorithms for: graph colouring (Section 7.4); and Floyd's all-pairs shortest paths algorithm (Section 7.3). With these algorithms parts of the graph are being changed, but in a predictable manner. With the all-pairs

shortest paths algorithm edges are repeatedly traversed in a fixed order. The same is the case with graph colouring, except vertices were traversed in a fixed order.

Although our Haskell implementations of Dijkstra's and Kruskal's algorithms had to use state, expressiveness was not completely lost. Purely functional data structures and higher-order functions were used to good effect. Moreover if a purely function solution exists for these algorithms, it will probably involve using a state-encapsulating combinator. This was used in a purely functional solution of breadth-first search. The combinator `lazyArray` was used to encapsulate the state. The resulting algorithm is subtle, and more complex than an imperative implementation. Hence, although it is necessary to experiment with these combinators, they currently do not seem to offer any benefits over an imperative implementation.

Again in this chapter code reuse and modularity was demonstrated. The implementation of transitive closure (Section 7.3.1) was expressed as a mapping over the result of the all-pairs shortest paths algorithm. Furthermore, several algorithms were expressed in terms of breadth-first search (Section 7.5).

Chapter 8

Aspects of complexity, efficiency, and style

Algorithm efficiency has been measured in terms of asymptotic complexity since Knuth (1973*a*). With computers becoming ever faster, the more asymptotic complexity matters. For example, suppose we have an algorithm that is quadratic in the size of its input, that is $O(n^2)$. If computing speed is increased by a factor of 100, how much more input can be handled? Only 10 times as much unfortunately, because in the time it used to take for n^2 it now takes $100n^2 = (10n)^2$. If the algorithm was linear in its input, however, then 100 times as much input could be handled on the faster machine.

Commonly the *worst-case* complexity of an algorithm is given, but this does not always give a reasonable correspondence with running time. For example, a component of an algorithm may be executed many times, each time with a different cost. Taking the sum of the worst case each time can be wildly pessimistic, since some runs may have the best-case time. Tarjan (1985) discusses *amortised* complexity, which is a more precise measure. Instead of taking the worst case every time, he amortises the different costs. Sequences of operations are considered, rather than looking at each operation independently. This is not to be confused with *average-case* analysis, which considers the complexity of an operation with an average input.

Asymptotic complexity has been expounded upon by Tarjan and others. It has now superseded empirical analysis for assessing algorithm efficiency. Asymptotic complexity abstracts away from constant factors which different language implementations may give. This seems the right approach, since it would be difficult to generalise

how many machine cycles an algorithm would take, especially when each language compiler has its own nuances. Nevertheless, constant factors cannot be ignored outright. A price is being payed in constant factors for using a functional language, so we should know what that price is. The easiest way to do this is to take empirical measurements.

Hardly any work has been done to study the complexity of lazy functional languages, though Sands (1990) in his thesis developed a simple calculus for time analysis of strict functional languages; and he later extended this for lazy functional languages (Sands 1995). The complexity of lazy functional languages is troublesome because there isn't a static evaluation order. The complexity of a fragment of code is not fixed; it can change depending on its surrounding components. The complexity of the function composition `f.g` is not the sum of the complexities of `f` and `g`. A well known example of this phenomenon, due to Bird, is set as an exercise in Bird and Wadler (1988, p. 158), and is further explained in Wadler (1988*b*). Given insertion sort, and composing it with the function `head`, yields a function that returns the minimum of a list:

```
minimum = head . insertion-sort
```

Insertion sort runs in $O(n^2)$ time, but the minimum function that uses it runs in $O(n)$ time. This happens as only the head of the list is being demanded; computations such as insertion on the tail are never demanded, hence not performed. In a strict language this definition of minimum would be $O(n^2)$, since the complexity of a strict `insertion-sort` will not change with context. Another more realistic example of this behaviour — that doesn't change the complexity, but has a large constant time improvement — is path finding (Section 6.6.7). Examples of this kind illustrate that lazy languages promote modularity.

8.1 The complexity of functional algorithms

Since functional languages are more amenable to formal manipulation a rigorous formal analysis of a functional program should be easier than for an imperative program. This is usually the case with strict functional languages, but non-strict functional languages pose numerous problems as described above. Let us first look at an example of calculating the complexity of a simple functional program.

The usual approach (and the approach taken by Bjerner and Holmström (1989), and Sands (1990, 1995)) is to derive a step-counting version of a function. The step-counting version takes the same arguments as the original function, but returns the computation cost; hence they are dubbed cost functions. The cost can be measured in any units, the most convenient is the number of non-primitive function calls used in the computation. This corresponds with the number of graph reductions made, which is a more accurate measure. Not including the cost of primitives like (+) is standard, since the goal in calculating cost is to determine an asymptotic time bound, and the amount of time per (+) operation does not increase with larger numerical inputs.

Each non-primitive function call will be counted with a cost of 1, and the notation $\langle\!\langle E \rangle\!\rangle$ will be used to represent the cost of evaluating E. So, for example, given the function definition:

$$f\ x_1 \cdots x_n = e$$

the cost of a call to this function is:

$$\langle\!\langle f\ e_1\ \cdots\ e_n \rangle\!\rangle = 1 + \langle\!\langle e\{e_1/x_1, \ldots, e_n/x_n\} \rangle\!\rangle$$

Cost will be expressed in terms of a functional language, which prevents new notation being introduced. To begin with some cost rules for a strict language will be given. These rules may be used for a function in a lazy language, if everything is fully evaluated. If everything is fully evaluated, the order of evaluation does not change the asymptotic time bound.

$$\begin{array}{lcl}
\langle\!\langle c \rangle\!\rangle & = & 0 \\
\langle\!\langle (e_1, \ldots, e_n) \rangle\!\rangle & = & \langle\!\langle e_1 \rangle\!\rangle + \cdots + \langle\!\langle e_n \rangle\!\rangle \\
\langle\!\langle \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rangle\!\rangle & = & \langle\!\langle e_1 \rangle\!\rangle + \textbf{if } e_1 \textbf{ then } \langle\!\langle e_2 \rangle\!\rangle \textbf{ else } \langle\!\langle e_3 \rangle\!\rangle \\
\langle\!\langle \textbf{let } x = e_1 \textbf{ in } e_2 \rangle\!\rangle & = & \langle\!\langle e_1 \rangle\!\rangle + \langle\!\langle e_2\{e_1/x\} \rangle\!\rangle \\
\langle\!\langle \textbf{case } e \textbf{ of} & = & \langle\!\langle e \rangle\!\rangle + \ \textbf{case } e \textbf{ of} \\
\quad pat_1 \ \rightarrow \ e_1 & & \quad pat_1 \ \rightarrow \ \langle\!\langle e_1 \rangle\!\rangle \\
\quad \vdots & & \quad \vdots \\
\quad pat_n \ \rightarrow \ e_n \rangle\!\rangle & & \quad pat_n \ \rightarrow \ \langle\!\langle e_n \rangle\!\rangle
\end{array}$$

where e's are expressions, c is a constant, x is a variable, and pat are patterns. Using these rules, it is straightforward to derive the cost functions for some basic list

operations:

$$
\begin{aligned}
\langle\langle [\,] \rangle\rangle &= 0 \\
\langle\langle x : xs \rangle\rangle &= 0 \\
\langle\langle xs +\!\!+ ys \rangle\rangle &= 1 + \mathit{length}\ xs \\
\langle\langle \mathit{map}\ f\ xs \rangle\rangle &= 1 + \mathit{sum}\,[\langle\langle f\ x \rangle\rangle \mid x \leftarrow xs\,] + \mathit{length}\ xs \\
\langle\langle \mathit{reverse}\ xs \rangle\rangle &= 1 + 2(\mathit{length}\ xs) \\
\langle\langle \mathit{concat}\ xss \rangle\rangle &= 1 + 2(\mathit{length}\ xss) + \mathit{sum}\,[\mathit{length}\ xs \mid xs \leftarrow xss\,]
\end{aligned}
$$

These cost functions assume that their elements have been evaluated, which is not the case with lazy evaluation. Nevertheless, this naïve approach is powerful enough to calculate the complexity of lazy functions whose results are known to be fully evaluated. An example of this is now given, which is the non-linear version of `preorder` on general trees.

8.1.1 Example: preorder

The function `preorder` is a good example because it is a function on trees, and the asymptote is not immediately obvious.

```
preorder :: Tree a -> [a]
preorder (Node x ts) = x: concat (map preorder ts)
```

To simplify the calculation only trees of the form t_d^b will be considered, where b is the number of branches at each node and d is the depth of the tree. This tree is perfectly balanced, and may be considered the average case. The function *size* will be used, which returns the number of nodes in a tree, thus $\mathit{size}(t_d^b) = \frac{b^d-1}{b-1}$.

Case t_1^b, the singleton tree.

$$\begin{aligned}
\langle\!\langle preorder\ t_1^b \rangle\!\rangle &= 1 + \langle\!\langle concat\ (map\ preorder\ [\,])\rangle\!\rangle \\
&= \left\{ \text{Cost of } concat \right\} \\
&\quad 2 + \langle\!\langle map\ preorder\ [\,]\rangle\!\rangle \\
&\quad\ + \textbf{case}\ (map\ preorder\ [\,])\ \textbf{of} \\
&\qquad [\,] \rightarrow \langle\!\langle [\,]\rangle\!\rangle \\
&\qquad (x : xs) \rightarrow \langle\!\langle x +\!\!+ concat\ xs\rangle\!\rangle \\
&= 2 + \langle\!\langle map\ preorder\ [\,]\rangle\!\rangle + \langle\!\langle [\,]\rangle\!\rangle \\
&= \left\{ \text{Cost of } map \right\} \\
&\quad 2 + 1 + \langle\!\langle [\,]\rangle\!\rangle \\
&= 3
\end{aligned}$$

Case t_d^b.

$$\begin{aligned}
\langle\!\langle preorder\ t_d^b \rangle\!\rangle &= \left\{ \text{Definition of preorder, } ts_d^b \text{ is a list of } b\ t_d^b \text{ trees} \right\} \\
&\quad 1 + \langle\!\langle concat\ (map\ preorder\ ts_{d-1}^b)\rangle\!\rangle \\
&= \left\{ \text{Definition of } concat \right\} \\
&\quad 2 + \langle\!\langle map\ preorder\ ts_{d-1}^b\rangle\!\rangle \\
&\quad\ + \textbf{case}\ (map\ preorder\ ts_{d-1}^b)\ \textbf{of} \\
&\qquad [\,] \rightarrow \langle\!\langle [\,]\rangle\!\rangle \\
&\qquad (x : xs) \rightarrow \langle\!\langle x +\!\!+ concat\ xs\rangle\!\rangle \\
&= \left\{ \text{Cost of } map, \text{ definition of } ts_{d-1}^b \right\} \\
&\quad 2 + (1 + b\langle\!\langle preorder\ t_{d-1}^b\rangle\!\rangle + b) \\
&\quad\ + \langle\!\langle [\,x_1, \ldots, x_{size\ (t_{d-1}^b)}\,] +\!\!+ concat\ [\,xs_1, \ldots, xs_{b-1}\,]\rangle\!\rangle \\
&= \left\{ \text{Cost of } +\!\!+, \text{ and } concat \right\} \\
&\quad 3 + b\langle\!\langle preorder\ t_{d-1}^b\rangle\!\rangle + b \\
&\quad\ + (1 + size\ (t_{d-1}^b)) + (1 + 2(b-1) + (b-1)size\ (t_{d-1}^b)) \\
&= \left\{ \text{By } size\ (t_d^b) = \tfrac{b^d-1}{b-1} \right\} \\
&\quad 3 + b\langle\!\langle preorder\ t_{d-1}^b\rangle\!\rangle + 3b + b\tfrac{b^d-1}{b-1}
\end{aligned}$$

This is a recurrence relation which can be solved by repeated substitution to yield:

$$\langle\!\langle preorder\ t_d^b\rangle\!\rangle = (d+6)b^d + (d+5)b^{d-1} + \cdots + 7b + 3$$

This gives the asymptote $O(db^d)$ or $O(n \log n)$ where n is the size of the tree. Therefore, this is a slow algorithm for preorder, since $O(n)$ is possible. The reason for this behaviour is apparent in the proof, and is known as the *repeated appends* phenomenon. The recursive call to `preorder` causes `concats` to be embedded inside each other, hence the same lists are traversed several times.

Although the algorithm is slow, it is clear, good for equational reasoning, and close to a specification of preorder. In an ideal world this inefficient version would be defined and the compiler would be left to transform it into an efficient version. The next section summarises some of the standard techniques for transforming examples like preorder into efficient functions.

8.2 Standard optimisation techniques

Occasionally throughout the thesis an inefficient function has been given with a comment that, by using *standard transformation techniques*, the inefficiency may be removed. Here the most common techniques are summarised (Table 8.1). With functional language compilers, it's a realistic proposition that an algorithm's complexity may be improved by an automatic transformation. The last two techniques documented in the table are automatic. The foldr/build transformation has been implemented in the Glasgow Haskell compiler, and this has the potential, to transform the above preorder example into the linear-time version.

8.3 The complexity of stateful algorithms

Commands on the state are just function calls, but ultimately they will cause an imperative action. These hidden actions have a cost, so assumptions need to be made about the imperative actions that are being used. The monadic combinators *return* and (;) are purely functional, so we can be precise about their cost. With the other

Technique and author	Description
Tupling Burstall and Darlington (1977)	This is applied to functions that have multiple calls to themselves with different arguments. These are combined to one call and the function returns a tuple of the required results.
Fold/unfold Burstall and Darlington (1977)	This is a system of rules for transforming recursive equations, and is the basis for most other techniques. Function calls are *unfolded* to their definitions, *laws* are applied, and then definitions are *folded* back to function calls.
Novel representation Hughes (1986)	Lists are represented by functions, allowing list append to be performed in $O(1)$ time.
Accumulating parameters Bird (1984*a*)	An extra parameter is added to recursive functions, which serves to accumulate an intermediate result.
Deforestation Wadler (1988*a*)	An automatic algorithm, which fuses functions together, removing intermediate data structures.
foldr/build Gill et al. (1993)	An automatic algorithm for the removal of intermediate data structures. Functions need to be re-expressed in terms of special combinators. Then rules for reducing these combinators are applied.

Table 8.1 Summary of some standard transformation techniques, for improving the efficiency of functional programs.

operations, however, some reasonable assumptions need to be made:

$$
\begin{array}{lcl}
\langle\!\langle return\ x \rangle\!\rangle & = & 1 + \langle\!\langle x \rangle\!\rangle \\
\langle\!\langle m; n \rangle\!\rangle & = & 2 + \langle\!\langle m \rangle\!\rangle + \langle\!\langle n \rangle\!\rangle \\
\langle\!\langle runST\ m \rangle\!\rangle & = & 2 + \langle\!\langle m \rangle\!\rangle \\
\langle\!\langle a \leftarrow newArr\ (l,u)\ v \rangle\!\rangle & = & 2 + rangeSize\ (l,u) + \langle\!\langle v \rangle\!\rangle \\
\langle\!\langle v \leftarrow readArr\ a\ x \rangle\!\rangle & = & 2 + \langle\!\langle x \rangle\!\rangle \\
\langle\!\langle writeArr\ a\ x\ v \rangle\!\rangle & = & 2 + \langle\!\langle x \rangle\!\rangle + \langle\!\langle v \rangle\!\rangle
\end{array}
$$

We assume that l, u, and a are fully evaluated. Again, this approach is only useful if functions are being fully evaluated, but this is still of use for many examples.

8.3.1 Example: `binsort`

This example is taken from Section 4.7 and is the imperative functional version of binsort. First consider the function `insert`:

```
insert :: Ix i => MutArr s i [a] -> (a -> i) -> [a] -> ST s ()
insert bin key []     = return ()
insert bin key (x:xs) = do { let i = key x in
                             ys <- readArr bin i;
                             writeArr bin i (x:ys)
                           }
```

Case `[]`.

$$\begin{aligned} \langle\!\langle \textit{insert bin key } [] \rangle\!\rangle &= 1 + \langle\!\langle \textit{return } () \rangle\!\rangle \\ &= 2 \end{aligned}$$

Case $(x : xs)$. Assume that the list xs is finite and well-defined.

$$\begin{aligned} &\langle\!\langle \textit{insert bin key } (x : xs) \rangle\!\rangle \\ &= 1 + \langle\!\langle \textbf{do } \{ \\ &\qquad \textbf{let } i = \textit{key } x \textbf{ in} \\ &\qquad \textit{ys} \leftarrow \textit{readArr bin } i; \\ &\qquad \textit{writeArr bin } i \ (x : ys); \\ &\qquad \textit{insert bin xs} \\ &\quad \} \rangle\!\rangle \\ &= \left\{ \text{Assume that } \textit{key } x \text{ will be demanded} \right\} \\ &\quad 1 + \langle\!\langle \textit{key } x \rangle\!\rangle + 2 \\ &\qquad + \langle\!\langle \textit{ys} \leftarrow \textit{readArr bin } i \rangle\!\rangle + 2 \\ &\qquad + \langle\!\langle \textit{writeArr bin } i \ (x : ys) \rangle\!\rangle + 2 \\ &\qquad + \langle\!\langle \textit{insert bin key xs} \rangle\!\rangle \\ &= 1 + \langle\!\langle \textit{key } x \rangle\!\rangle + 2 \\ &\qquad + 2 + \langle\!\langle i \rangle\!\rangle + 2 \\ &\qquad + 2\langle\!\langle i \rangle\!\rangle + \langle\!\langle x : ys \rangle\!\rangle + 2 \\ &\qquad + \langle\!\langle \textit{insert bin key xs} \rangle\!\rangle \\ &= 11 + \langle\!\langle \textit{key } x \rangle\!\rangle + \langle\!\langle \textit{insert bin key xs} \rangle\!\rangle \end{aligned}$$

Hence,

$$\langle\!\langle insert\ bin\ key\ xs \rangle\!\rangle = 2 + 11(length\ xs) + sum\ [\, \langle\!\langle key\ x \rangle\!\rangle \mid x \leftarrow xs\,]$$

Deriving the cost of `extract` is similar to the above calculation, and reveals the cost function:

$$\begin{aligned}\langle\!\langle extract\ bin\ is \rangle\!\rangle \;\; &= \;\; 2 + 8(length\ xs) \\ &\qquad + sum\ [\, length\ xs \mid i \leftarrow is, xs \leftarrow immutableBin!i\,] \\ \textbf{where} \quad immutableBin &= runST\ (\textbf{do}\ \{\ freezeArr\ bin\ \})\end{aligned}$$

Now this cost is used together with the cost of `insert` for deriving the cost of `binsort`, which has the definition:

```
binsort :: Ix i => (i,i) -> (a -> i) -> [a] -> [a]
binsort bnds key xs = runST (do { bin <- newArr (l,u) [];
                                  insert bin key xs;
                                  extract bin (range bnds)
                                })
```

The calculation proceeds as follows.

$$\begin{aligned}&\langle\!\langle binsort\ bnds\ key\ xs \rangle\!\rangle \\ &= \;\; 1 + \langle\!\langle\ \textbf{do}\ \{ \\ &\qquad\qquad bin \leftarrow newArr\ bnds\ [\,]; \\ &\qquad\qquad insert\ bin\ key\ xs; \\ &\qquad\qquad extract\ bin\ (range\ bnds) \\ &\qquad \}\ \rangle\!\rangle \\ &= \;\; 1 + 2 \\ &\qquad + \langle\!\langle bin \leftarrow newArr\ bnds\ [\,] \rangle\!\rangle + 2 \\ &\qquad + \langle\!\langle insert\ bin\ key\ xs \rangle\!\rangle + 2 \\ &\qquad + \langle\!\langle extract\ bin\ (range\ bnds) \rangle\!\rangle \\ &= \;\; 7 + 3 + rangeSize\ bnds + \langle\!\langle [\,] \rangle\!\rangle \\ &\qquad + 2 + 11(length\ xs) + sum\ [\, \langle\!\langle key\ x \rangle\!\rangle \mid x \leftarrow xs\,] \\ &\qquad + sum\ [\, length\ xs \mid xs \leftarrow readArr\ bin\ i,\ i \leftarrow range\ bnds\,] \\ &\qquad + \langle\!\langle range\ bnds \rangle\!\rangle \\ &= \;\; 14 + 9(rangeSize\ bnds) + 12(length\ xs) + sum\ [\, \langle\!\langle key\ x \rangle\!\rangle \mid x \leftarrow xs\,] \\ &\qquad + \langle\!\langle range\ bnds \rangle\!\rangle\end{aligned}$$

Assuming that $sum\ [\langle\langle key\ x\rangle\rangle \mid x \leftarrow xs\,]$ has asymptote $O(length\ xs)$ and $\langle\langle range\ bnds\rangle\rangle$ has asymptote $O(rangeSize\ bnds)$. Then,

$$\langle\langle binsort\ bnds\ key\ xs\rangle\rangle$$

has asymptote $O(length\ xs + rangeSize\ bnds)$.

8.4 The complexity of lazy functions

In this section the cost of a lazy example is calculated to illustrate the difficulties involved. This will be done in a fashion similar to Bjerner and Holmström (1989). The difference is in notation, and some details that they would include, will be left out here. The notation $\langle\langle\rangle\rangle$ for cost is as before, except this time it is augmented with two other arguments. The first is a variable environment, and the second describes how much of the expression inside $\langle\langle\rangle\rangle$ is demanded. So for example,

$$\langle\langle map\ f\ [2,7,6]\rangle\rangle\ \langle f = (*)3\rangle\ (\phi : 21 : \phi)$$

describes the cost of evaluating $map\ f\ [2,7,6]$ where f is defined as $(*)3$, and only the second element of the resulting list is demanded. The most modest demand is ϕ which describes $\perp$ and says that no output at all should be produced. An unknown demand will be denoted with δ.

The chosen example comes from the graph colouring algorithm (Section 7.4) the function `paint` was used to determine an unused colour for the latest vertex, the definition is changed slightly here to a more general function:

```
paint xs = head ([1..]\\xs)
```

this function is lazy by virtue of using the infinite list `[1..]`. The function `(\\)` is defined in the Haskell Prelude as:

```
(\\) :: Eq a => [a] -> [a] -> [a]
xs \\ ys = foldl del xs ys
      where   del []     _                = []
              del (x:xs) y | x==y         = del xs y
                           | otherwise    = x: del xs y
```

The calculation commences as follows, where v is the fully evaluated value of the required result.

$$
\begin{aligned}
&\langle\!\langle paint\ xs \rangle\!\rangle\ \langle\rangle\ (v) \\
&\quad = \quad 2 + \langle\!\langle [1..] \setminus\setminus xs \rangle\!\rangle\ \langle\rangle\ (v : \phi) \\
&\quad = \quad \left\{ \text{Let } xs = [x_1, \ldots, x_n] \right\} \\
&\qquad 3 + \langle\!\langle foldl\ del\ [1..]\ [x_1, \ldots, x_n] \rangle\!\rangle\ \langle\rangle\ (v : \phi) \\
&\quad = \quad 3 + n + \langle\!\langle del\ (del\ (\cdots (del\ [1..]\ x_1) \cdots))\ x_n \rangle\!\rangle\ \langle\rangle\ (v : \phi) \\
&\quad = \quad 3 + n \\
&\qquad + \langle\!\langle del\ ys_1\ x_n \rangle\!\rangle\ \langle ys_1 = del\ ys_2\ x_{n-1} \rangle\ (v : \phi) \\
&\qquad + \langle\!\langle del\ ys_2\ x_{n-1} \rangle\!\rangle\ \langle ys_2 = del\ ys_3\ x_{n-2} \rangle\ \delta_2 \\
&\qquad + \langle\!\langle del\ ys_3\ x_{n-2} \rangle\!\rangle\ \langle ys_3 = del\ ys_4\ x_{n-3} \rangle\ \delta_3 \\
&\qquad \vdots \\
&\qquad + \langle\!\langle del\ [1..]\ x_1 \rangle\!\rangle\ \langle\rangle\ \delta_n
\end{aligned}
$$

Now we need to calculate the cost of $del\ [1..]\ x$, when a list of length n is demanded from it:

$$
\begin{aligned}
&\langle\!\langle del\ [1..]\ x \rangle\!\rangle\ \langle\rangle\ (v_1 : v_2 : \ldots : v_n : \phi) \\
&\quad = \quad 1 + \langle\!\langle \textbf{if } x == 1 \textbf{ then } [2..] \textbf{ else } x : del\ [2..]\ x \rangle\!\rangle\ \langle\rangle\ (v_1 : v_2 : \ldots : v_n : \phi) \\
&\quad = \quad \left\{ \text{Taking the worst-case, that is } x > n \right\} \\
&\qquad 1 + \langle\!\langle 1 : del\ [2..]\ x \rangle\!\rangle\ \langle\rangle\ (v_1 : v_2 : \ldots : v_n : \phi) \\
&\quad = \quad 1 + n + \langle\!\langle 1 : 2 : \ldots : n : del\ [n+1..]\ x \rangle\!\rangle\ \langle\rangle\ (v_1 : v_2 : \ldots : v_n : \phi) \\
&\quad = \quad 1 + n
\end{aligned}
$$

Going back to our calculation of *paint*, the difficulty now is calculating the δs.. The details are left out here, since they require a demand analysis (Bjerner and Holmström 1989), but an informal justification is given. Since we require a list with the head defined from $del\ ys_1\ x_n$, then ys_1 should be a list with at least the first two elements defined. because one of them may match and be deleted. This requires ys_2 having at least three elements defined, and ys_3 at least four elements, and so on until $del\ [1..]\ x_1$ requires at least the first n elements to be defined.

Continuing,

$$
\begin{aligned}
&= 3+n \\
&\quad +\langle\langle del\ ys_1\ x_n\rangle\rangle\ \langle ys_1 = del\ ys_2\ x_{n-1}\rangle\ (v:\phi) \\
&\quad +\langle\langle del\ ys_2\ x_{n-1}\rangle\rangle\ \langle ys_2 = del\ ys_3\ x_{n-2}\rangle\ (v_1^2 : v_2^2 : \phi) \\
&\quad +\langle\langle del\ ys_3\ x_{n-2}\rangle\rangle\ \langle ys_3 = del\ ys_4\ x_{n-3}\rangle\ (v_1^3 : v_2^3 : v_3^3 : \phi) \\
&\quad \vdots \\
&\quad +\langle\langle del\ [1..]\ x_1\rangle\rangle\ \langle\rangle\ (v_1^n : v_2^n : \ldots : v_n^n : \phi) \\
&= \quad \left\{ \text{By the above cost of } del\ [1..]\ x \right\} \\
&\quad 3+n+2+3+\cdots+(n+1) \\
&= 3+n+\tfrac{n(n+1)}{2} \\
&= \tfrac{1}{2}n^2+\tfrac{3}{2}n+3
\end{aligned}
$$

Giving the asymptote $O(n^2)$ for *paint*.

8.5 Empirical measurements of some functional algorithms

There are two main reasons to carry out empirical measurements here: (i) to demonstrate that some algorithms have the expected complexity; and (ii) to discover what the constant factor is between our functional algorithms in Haskell and imperative algorithms in a conventional language. Although analytical complexity has superseded empirical measurements; complexity analysis of functional programs is still a research topic. so some hard evidence is needed. Constant factors are also widely regarded with disdain. Nevertheless, a certain magnitude of time difference is considered unacceptable. Clearly if Haskell programs run in days, whereas C runs in seconds, this is unacceptable. Everybody has their own opinion as to what is an acceptable speed difference between functional and imperative. No judgement is made here, but the question is answered by comparing a functional algorithm in Haskell running on the Glasgow Haskell compiler with the same algorithm in C running on the Gnu C compiler.

8.5.1 Evidence that we have the right asymptotic complexity

Some care is required when taking measurements. All the measurements reported here where done on a large machine, which was not running any other major processes. The amount of swapping, caching etc. was low. Each measurement is the mean user time over three runs. The input data was generated by a random graph generator not unlike that presented in Section 5.4.3. The graph was placed in a file and read in, so the overheads of graph generation wasn't included.

The first measurements were taken on the strongly connected components algorithm (Section 6.6.4) which should run in $O(V + E)$ time. To test for linearity timings were taken over many inputs (Figure 8.1).

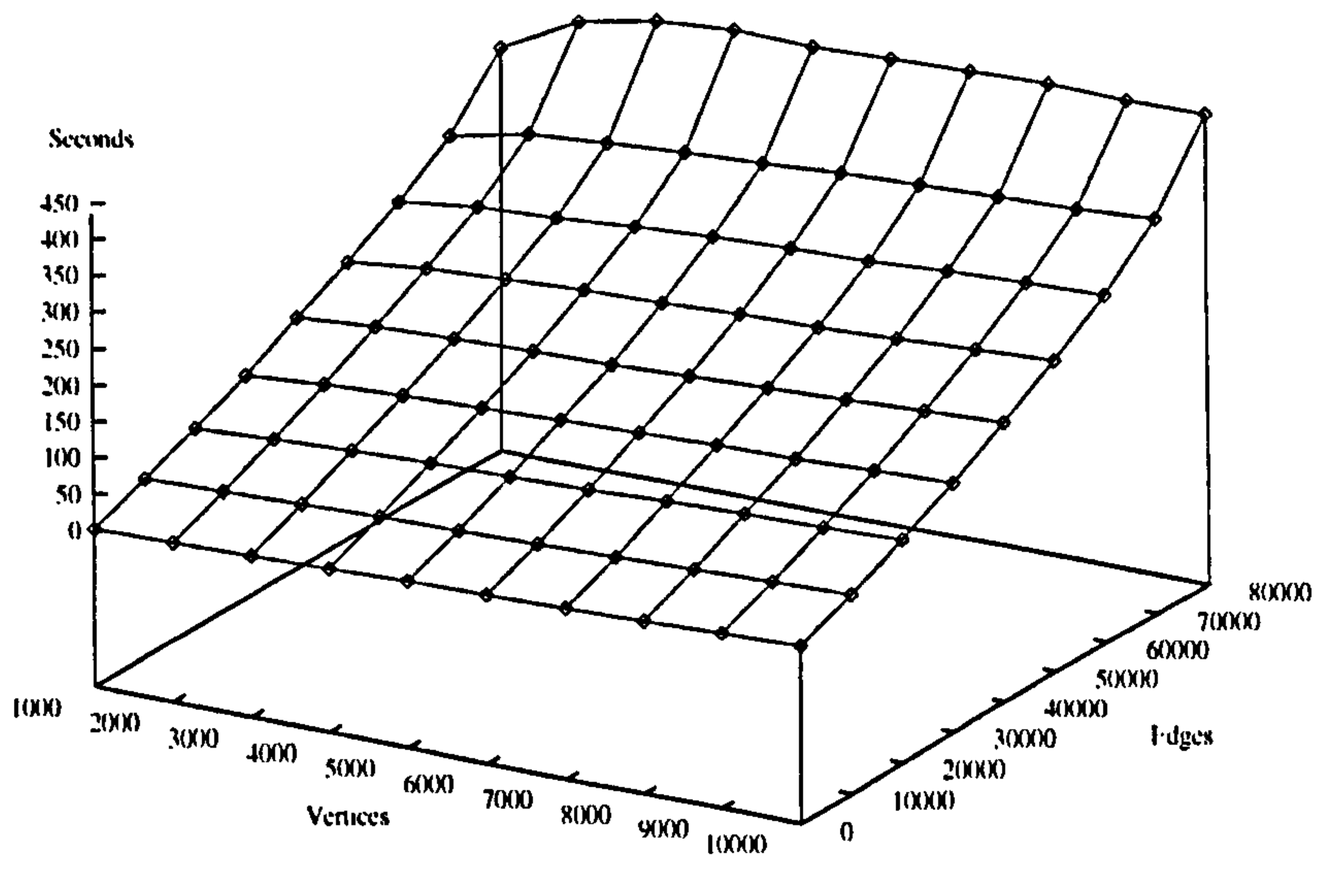


Figure 8.1 Measurements of the Haskell strongly connected components algorithm.

This graph shows that the timings are *not* linear. If they were linear the diagram would show a plane. At first this is surprising, but what is probably happening is that when the input size becomes larger more of the heap is being used, and so more garbage collection is taking place. This claim can be justified by removing the time for garbage collection from the timings (Figure 8.2).

This graph (Figure 8.2) shows a plane demonstrating that the strongly connected components algorithm runs in $O(V + E)$ time. The next measurements were taken on the same algorithm to determine its space usage, which should be $O(V + E)$ space. This was done by looking at how many bytes were allocated in the heap for many input data sets. The results shown in Figure 8.3, show a plane, giving strong evidence

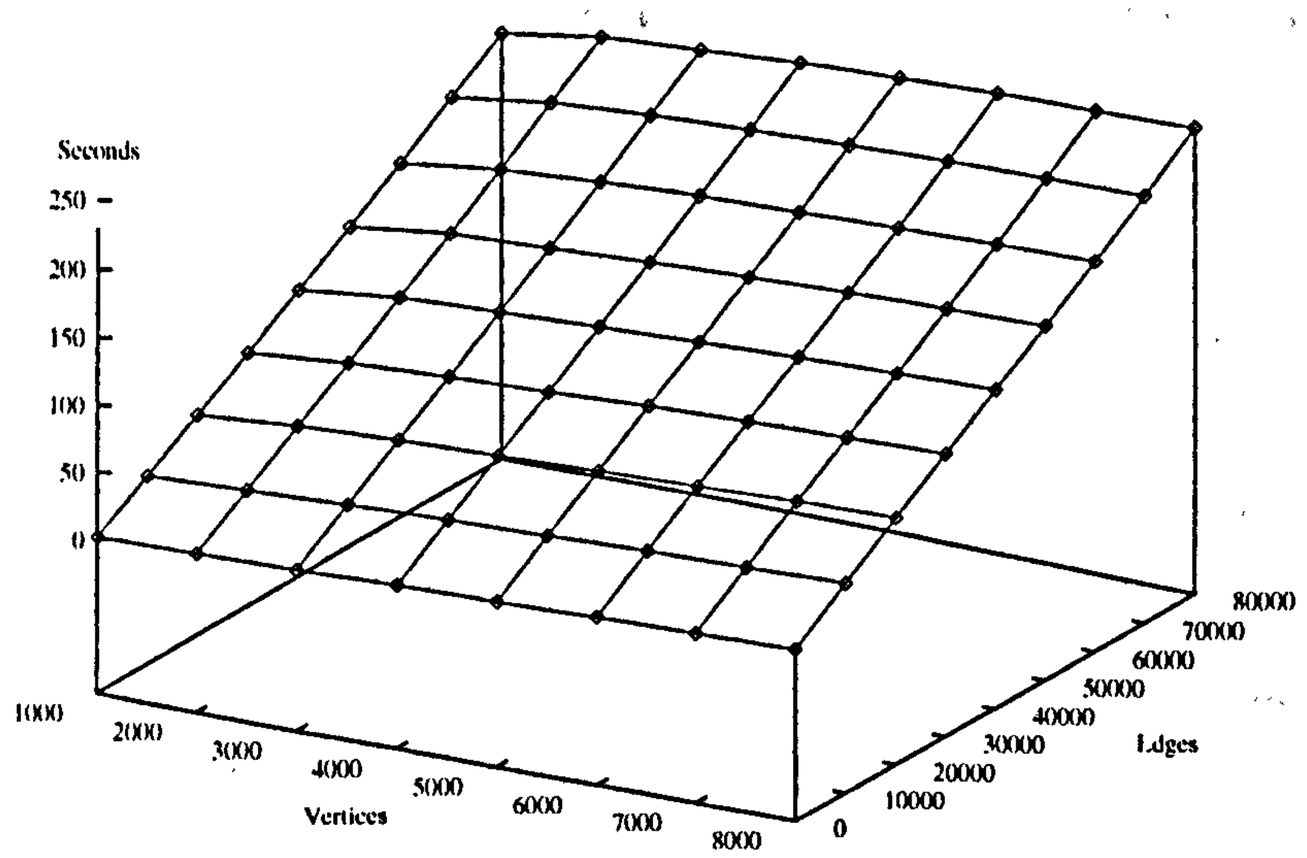


Figure 8.2 Measurements of the Haskell strongly connected components algorithm without garbage collection time.

that the strongly connected components algorithm runs in linear space.

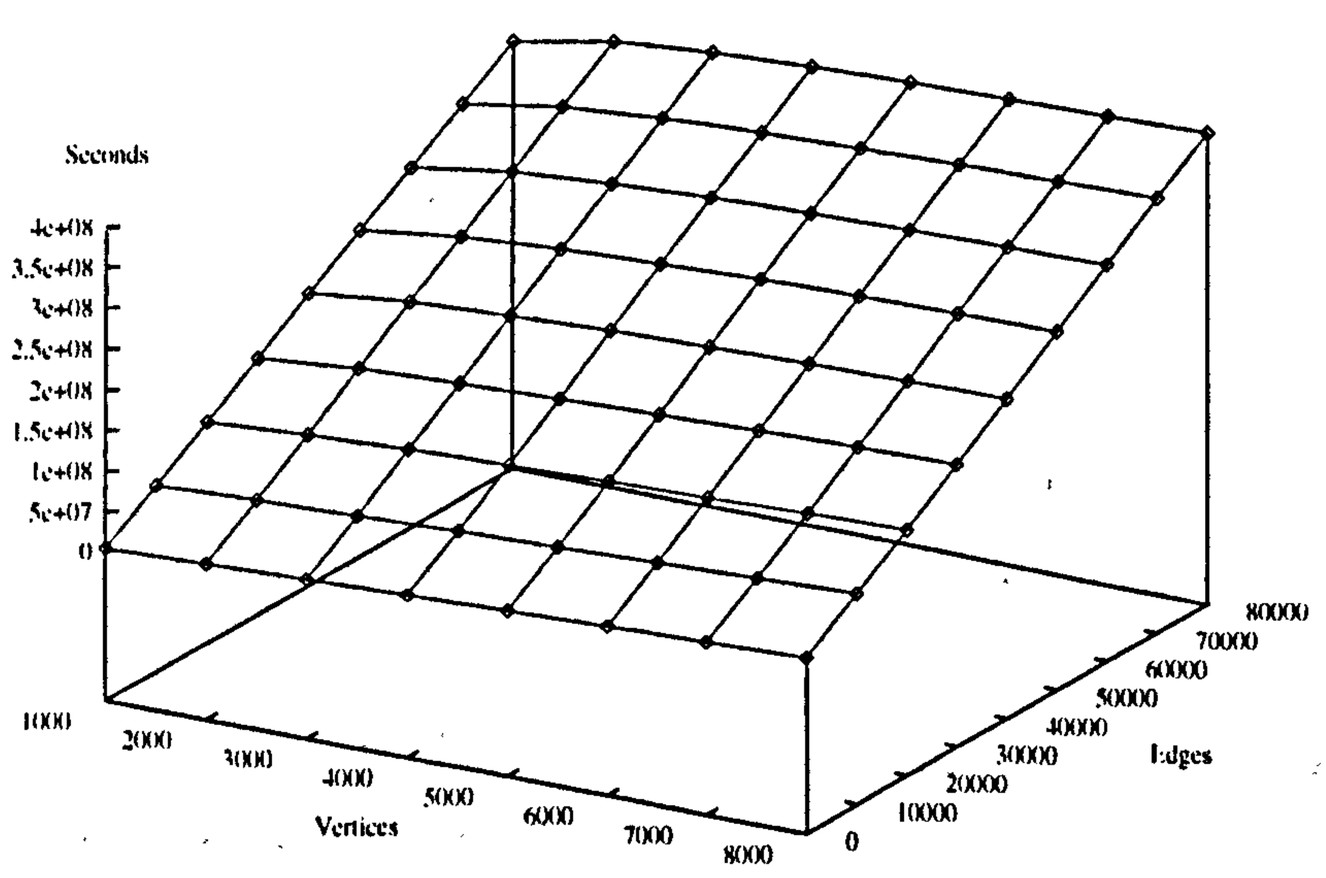


Figure 8.3 Measurements of the space usage used by the Haskell version of the strongly connected components algorithm.

Finally measurements were taken on the strongly connected components algorithm of Tarjan (1972) over the same input sets. This algorithm is entirely different from the Haskell one used here, so it is unfair to use it as a comparison between C and Haskell. Nevertheless, the graph (Figure 8.4) is given as a control, and as expected is a plane.

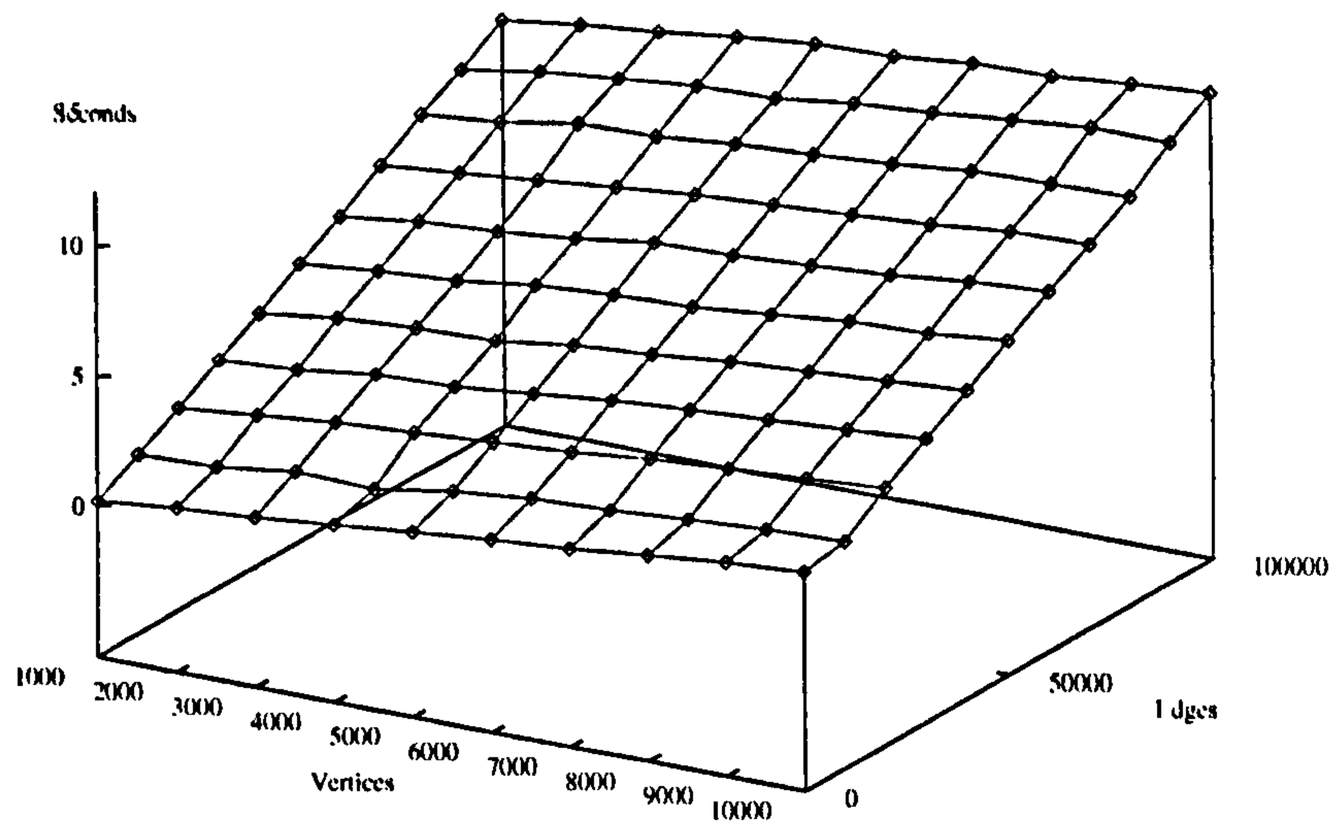


Figure 8.4 Measurements of the C version of the strongly connected components algorithm.

8.5.2 The constant factor between Haskell and C

There is no precise figure that can be said to be the constant factor between Haskell and C. There will be a different factor for different algorithms, and a different factor for the same algorithm running on different data. Hence, our goal is merely to discover the difference in terms of order of magnitude. The biconnected components algorithm was used as the example, since the Haskell implementation is a variation the imperative implementation by Hopcroft and Tarjan (1973). The imperative version of the algorithm was implemented in C, keeping as close as possible to the original version. Measurements for these Haskell and C implementations of the biconnected components algorithm are presented in Table 8.2.

Hopcroft and Tarjan (1973) wrote their biconnected components algorithm some twenty years ago. It only seems fair to compare the latest imperative implementation of this algorithm, with our Haskell implementation. GraphBase (Knuth 1993) has a highly efficient implementation of the algorithm, and also gives data sets which can be used as benchmarks. The examples used here were taken from classic literature. where characters are vertices and encounters between characters are edges. The biconnected components algorithm applied to this data separates characters into acquaintance groups, so that if someone is removed from a group, every remaining person will know at least one other person in the group.

	Time (seconds)	C	Haskell	Difference
Sparse graph	Total	2.0	6.76	×3
(50000V,5000E)	Algorithm	0.24	1.4	×6
Medium graph	Total	0.9	8.1	×9
(2000V,10000E)	Algorithm	0.24	3.7	×15
Dense graph	Total	9.7	25.12	×3
(500V,124750E)	Algorithm	2.54	4.4	×2
GraphBase	Total	0.8	46.39	×58
benchmark	Algorithm	0.09	5.98	×66

Table 8.2 Comparisons of the biconnected components algorithm.

These timings were done on a Sun SPARCstation-10 when no one else was using it. Each measurement is the mean of three runs, taking the user time. The Haskell binaries were given 80M of heap and a 1M stack. Garbage collection time was not included in these measurement, because of the irregularities it may cause (Section 8.5). To be fair we should perhaps add 10% (the average garbage collection time) to the Haskell timings. There were five GraphBase examples, and the C and Haskell ran on all the benchmarks 20 times. Profiling was used with Haskell to find out the percentage of time spent in just the algorithm. Then a separate run was done without profiling, so that the overheads of profiling do not have a bearing on the results. To measure C time stamps were placed in the code.

Table 8.2 gives the results of our measurements. Of the four programs considered, the GraphBase CWEB code is the only one that has been optimised. Perhaps it is not surprising then that it runs 60 times faster than our Haskell code. There is plenty of scope for optimisation of the Haskell code. For instance, code fusion to transform the algorithm into a single pass algorithm.

8.6 The style factor between functional and imperative

A case in point is the following algorithm presented in Figure 8.5 which calculates biconnected components of a graph. (The algorithm is taken from Tarjan (1972)). The syntax has been updated so that it is more like C. The algorithm calculates two

pieces of information for each vertex, namely LOWPT and NUMBER. Stacking operations are performed as well as recording the components. All of this is carried out during the course of a graph traversal. So it isn't surprising that the algorithm is non-trivial to follow.

```
biconnect(v,u)
{

NUMBER(v) = ++dfs_number;
LOWPT(v) = NUMBER(v);
for w in the adjacency list of v {
  if (!NUMBER(w)) {
    push (v,w) onto edge stack
    biconnect(w,v);
    LOWPT(v) = min(LOWPT(v),LOWPT(w));
    if (LOWPT(w) >= NUMBER(v)) {
      start of new component with articulation point v;
      pop (u1,u2) from edge stack;
      while (NUMBER(u1) >= NUMBER(w)) {
        add (u1,u2) to current component;
        pop (u1,u2) from edge stack;
      }
      add (v,w) to current component;
    }
  }
  else if ((NUMBER(w) < NUMBER(v)) && (w!=u)) {
    push (v,w) onto edge stack;
    LOWPT(v) = min(LOWPT(v),NUMBER(w));
  }
}
}
```

Figure 8.5 Tarjan's biconnected components algorithm.

The functional version of the biconnected components algorithm (Figure 8.6) is essentially the same as the imperative algorithm (Figure 8.5). The difference is that the functional version separates parts of the algorithm into different phases. First the graph is decomposed into a depth-first spanning forest; from this a depth-first number table is calculated for every vertex; then the spanning forest is traversed using

the table of depth-first numbers to calculate the low point numbers for each vertex. The spanning forest is annotated with these two pieces of information (depth-first number and low point number). This annotated tree is then traversed to return the biconnected components.

```
bcc :: Graph -> Forest [Vertex]
bcc g = (concat . map bicomps . map (label g dnum)) forest
  where  forest = dff g
         dnum = preArr (bounds g) forest

label :: Graph -> Table Int -> Tree Vertex -> Tree (Vertex,Int,Int)
label g dnum (Node v ts) = Node (v,dnum!v,lv) us
   where  us = map (label g dnum) ts
          lv = minimum ([dnum!v]++[ dnum!w | w<-g!v]
                                ++[ lu | Node (u,dw,lu) xs<-us])

bicomps :: Tree (Vertex,Int,Int) -> Forest [Vertex]
bicomps (Node (v,dv,lv) ts)
      = [ Node (v:vs) us | (l, Node vs us)<-map collect ts]

collect :: Tree (Vertex,Int,Int) -> (Int, Tree [Vertex])
collect (Node (v,dv,lv) ts) = (lv, Node (v:vs) cs)
  where  collected = map collect ts
         vs = concat [ ws | (lw, Node ws us)<-collected, lw<dv]
         cs = concat [ if lw<dv then us else [Node (v:ws) us]
                              | (lw, Node ws us)<-collected]
```

Figure 8.6 Tarjan's biconnected components algorithm (functional version).

The fundamental difference between the two versions of biconnected components (Figure 8.5 and Figure 8.6) is the modularity of the functional version. In this example there is no reason why the imperative version couldn't be written in this way. Functional languages encourage this style of programming with data transformations, whereas conventional languages make it tedious to introduce new data structures.

8.7 Comparing lazy with strict

It is a reasonable question to ask, how useful is laziness? Could all the algorithms presented be expressed in Standard ML or a similar non-lazy functional language? Most of the algorithms presented do not require laziness; however, there where occasions when it proved to be extremely useful. For example, the prune/generate paradigm (Section 6.3) was a useful way of breaking depth-first search into two distinct phases, which was also helpful for proof. The algorithm to detect paths (Section 6.6.7), was expressed with a potential full graph traversal, but with laziness would stop as soon as the required result was found. The implementation of path compression in up-trees (Section 4.8) was expressed as a one pass algorithm using a cyclic combinator. In summary the two places where laziness proved useful was in modularisation (Hughes (1989) makes this case well) and cyclic programming (Bird 1984*b*).

There is perhaps a side issue as well, that with strict languages it is common practice to resort to using side effects in places where efficiency is required. Lazy functional languages, on the other hand, cannot include side effects so easily because it would be unclear when they would be evaluated. In a sense lazy functional language designers were driven to the monadic model, which allows actions on the state whilst retaining referential transparency.

In a strict language lazy functions can be expressed by using special data structures (Reade 1989, Appendix 3). This is clumsy and in practice discourages the use of laziness except where it is essential. Most strict languages are not purely functional having side effects, this is useful for debugging purposes, but makes reasoning more difficult. Having a fixed evaluation order, it is easier to analyse formally the complexity of functions. More often than not, strict language compilers are more efficient than their lazy counterparts.

8.8 Discussion

In this chapter Haskell has been compared with more conventional languages. The aspect of efficiency is always brought into a discussion of this kind. This chapter showed that algorithms can be written in Haskell with no loss of asymptotic complexity, however, there is still an order of magnitude time difference between Haskell

and C. However, the cost of developing and maintaining a correct implementation is often smaller with Haskell than with C. This claim has been justified by an official experiment by the US Navy (Hudak and Jones 1994). Several imperative languages including Ada, C++, Awk, and the function language Haskell were used to prototype a Naval Surface Warfare Center. The results showed that the Haskell prototype took significantly less time to develop, and was considerably more concise and easier to understand than the corresponding prototypes written in imperative languages. This was again demonstrated in this chapter with the biconnected components algorithm. The Haskell algorithm is more modular, making the implementation easier to understand.

This chapter also looked at the problems of calculating the complexity of functional programs. It was found that with strict programs, it is relatively straightforward to derive a cost estimate for a program. Frequently, although working in a lazy language, functions do not require lazy evaluation, and the order of evaluation has no effect on the function's time complexity. In these cases, we are at liberty to analyse the complexity using cost rules for a strict language. This was demonstrated by showing the complexity of a purely functional example (Section 8.1.1), and then of a stateful algorithm (Section 8.3.1). The last example of deriving cost functions for a program was one which required laziness for its termination (Section 8.4). The approach taken here was based on the work by Bjerner and Holmström (1989). The main difference was that the details were left out (which were the difficult part) of deriving the parts of an expression that are demanded to return the result. This requires a sophisticated form of strictness analysis, and is almost akin to computing the program. This cost of lazy programs, is therefore, still an open problem. Although solutions exist (Sands 1995), they become cumbersome and tedious for working out simple examples.

So far little has been said here about the space behaviour of functional programs, apart from demonstrating that the linear-time strongly connected components algorithm runs in a linear amount of space (Section 8.5.1). The space behaviour of lazy functional programs is difficult to predict; hence complex to calculate. The space complexity is the residency of the computation, which is the maximum amount of live data in the heap at any point during a computation.

Chapter 9

Conclusion

This chapter summarises the previous chapters, and gives a discussion of future research.

9.1 Original objective

The original objective of this dissertation was to determine the advantages of expressing graph algorithms in a functional language. Graph algorithms have been notoriously difficult, and have not been given a good treatment in functional languages, predominantly because they do not have an inductive structure. Since functional languages are renowned for expressing other mathematical structures elegantly, it has been a failing that graph algorithms have not been fully explored. My objective was therefore to apply all the benefits that functional languages provide to the algorithms of graph theory.

With the advent of monads, all the previously intractable problems for purely functional languages became solvable. With monads purely functional languages could have mutable data structures. For example, arrays that can be updated in $O(1)$ time. This is a young but powerful tool, with undesirable as well as desirable effects. One undesirable effect is that the programmer is at liberty to express all his programs in this style. The resulting code can be more unwieldy, and just as troublesome as conventional imperative code. Therefore, it was necessary to study algorithms in this style, to determine if the usual expressiveness remained, or was totally lost.

9.2 Appraisal

In Chapter 1, the difficulties of implementing graph algorithms efficiently in a purely functional language were described. This was done by comparing and contrasting an algorithm for connected components in three styles: (i) with conventional pseudo-code; (ii) with inefficient functional code; and (iii) with efficient imperative functional code. The different presentations served to illustrate some of the advantages of functional languages, such as: expressiveness, code reuse, modularisation, and provability. It was explained that one of the main reasons for these advantages, namely not having side effects, was the very thing that made it difficult in the past to express graph algorithms efficiently.

After this, in Chapter 2, related work was reviewed. There is an abundance of work on the design of graph algorithms. Perhaps unsurprisingly, there is no universal approach that is good for all algorithms. Approaches that seem destined to win through are: GraphBase, Knuth's literate style of expressing graph algorithms in a conventional language; and languages that provide good settings for dealing with mathematics, like Mathematica. Algebraic approaches are scarce, and so far have not offered new insights to graph algorithms, although they do provide a framework for demonstrating correctness. Previous functional language approaches are also discussed, all of which lose out in asymptotic complexity as compared to a conventional implementation.

Many claims are made of functional languages, and often they are not substantiated. One is that they provide high-level abstraction powers. This claim is justified in Chapter 3, by demonstrating the equivalence of two algorithms: treesort and functional quicksort. These two algorithms, at first, seem strikingly unalike. Since the algorithm for treesort has the functionality: $List \rightarrow Tree \rightarrow List$, it is natural to consider if the functionality can be optimised to: $List \rightarrow List$. This is done quite simply with standard program transformations. In a conventional language one would not consider the optimisation in the first place, let alone do the program transformations. The expressive powers are further justified by giving a functional implementation of binomial queues. Priority queues are used by several graph algorithms, so it is important to have an efficient implementation. They were shown to have a clear implementation, and their formal verification was shown to be possible in all detail.

After motivating the need for state, in Chapter 4, the monadic model was introduced. With examples it was demonstrated how dynamic data structures can be expressed.

Some examples were shown only for illustrative purposes, but others like union/find and binsort seem intrinsically to require state for an efficient implementation. Combinators on the state were provided as an aid to expressing stateful algorithms, thus providing some benefit of expressing algorithms in a stateful way.

Different models for graphs were discussed in Chapter 5. Several were considered, but the ones chosen were the traditional adjacency list and adjacency matrix. These were chosen because of their efficiency, and because they could be expressed with Haskell immutable arrays. Adjacency matrices were used for expressing weighted graphs. Several examples of simple graph functions were presented, giving testimony to how concise and expressive the language can be.

In Chapter 6, algorithms that use depth-first search were studied in detail. Depth-first search turned out to be a good example of state being encapsulated — the function `dff` had type `Graph -> Forest`, where `Graph` and `Forest` are both purely functional values — although state is used within the definition of `dff`. Moreover, this example epitomised the concepts of code reuse, modularity, clarity of expression, laziness, and straightforwardness of correctness proofs. The value returned by `dff` is a depth-first spanning forest, and graph algorithms were expressed in terms of this. Therefore, the `dff` component was repeatedly reused. The algorithms were expressed clearly, typically as one or two line functions, as the composition of simpler components. With this modularisation correctness proofs were shown in all detail.

Several traditional graph algorithms were given in Chapter 7, including dynamic graph algorithms. These were graph algorithms where an encapsulation of state is not possible. State has to be threaded throughout the entire algorithm. It was worth looking at these to see if anything at all could be gained by expressing them in an imperative functional style. There were two important outcomes from looking at these algorithms: first, that any conventional graph algorithm can be expressed without loss of efficiency; and second, that the imperative functional approach benefits from using stateful combinators, so in some ways it is superior to a conventional imperative approach. This chapter also looked at algorithms that can be implemented purely functionally. With examples such as graph colouring, all-pairs shortest paths, and transitive closure, reasonable solutions were found. With breadth-first search devious means had to be used, namely `lazyArray`, to achieve a purely functional solution. Although the solution is purely functional, it is more difficult to understand than an imperative solution. Hence, we should not be happy with a purely functional imple-

mentation that has the best complexity, if it is too obscure to be easily understood. The implementation should be as clear to understand as possible.

Whenever algorithms of any sort are studied, their efficiency should always be considered. This was explored for functional and stateful algorithms in Chapter 8. All aspects of efficiency were explored, including calculating the complexity of functional and stateful algorithms, looking at empirical measurements of the Haskell implementations compared with C, and looking at space usage. Because of laziness, an analytical measurement of the time and space behaviour of lazy programs is much harder than with strict programs. This point was illustrated by calculating the time complexity of a lazy program. Empirical measurements were made, and the two main results were: (i) evidence that the time and space of a Haskell graph algorithm were as expected; and (ii) that for one algorithm, an order of magnitude time difference was shown between Haskell and C. This last result was not ideal, but some may consider it an acceptable price to pay for the benefits that Haskell provides over languages like C.

The benefits of using a functional language have been substantiated throughout the thesis. In Chapter 8 a comparison was made of a functional implementation, with an imperative implementation. The size of the two programs is about the same, and they are both based on the same algorithm. The main difference between the two is that the Haskell version is modular, which is a style promoted by the nature of functional languages. This makes the algorithms far easier to understand, often providing more insight. No claim is made that Haskell should replace all imperative languages, but although less efficient, there are several other benefits. A language like Haskell is therefore ideal for prototyping, where getting a good understanding of the problem is essential. This may then be transcribed into a more efficient language if performance is critical.

9.3 Further work

This section summarises some of the possibilities for further work.

There were many aspects that were not considered and that would make the work more complete. Parallelism was not considered at all. Purely functional languages have a good potential for parallelism, since programs are not evaluated in a fixed

sequential order. However, the monadic model for including actions on the state sequentialises actions, and so prevents parallelism. This tension between parallelism and the monadic model needs to be explored.

Another topic of further work is to look at larger problems, and NP-complete problems, to see if the same ideas are applicable.

Our comparison between lazy and strict languages was very brief. Empirical measurements could be taken to discover the performance cost for laziness. Also, a more extensive comparison between the two evaluation models needs to be made. Several people in the Standard ML community believe that lazy languages have virtually no benefits over strict languages. Throughout the thesis, advantages afforded by lazy languages have been discussed fully, but there was only a cursory mention of the many benefits of strict languages.

Another topic of research is to try and remove all actions on state from our programs. This is often achieved by using a special combinator to encapsulate state. This was demonstrated in Section 7.5 with an algorithm for breadth-first search. Solutions of this kind are often more complex, than an imperative solution. Work therefore needs to be done to explore such combinators in the context of reasoning and programming.

BLANK IN ORIGINAL

Bibliography

Adams, S. (1993), Efficient sets: a balancing act, *Journal of Functional Programming* **3**(4), 553–561. (p 46)

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1983), *Data Structures and Algorithms*, Addison-Wesley. (pp. 21, 29, 45, 102)

Backhouse, R. C. (1989), An exploration of the Bird-Meertens formalism, *in* International Summer School on Constructive Algorithmics. (p 14)

Backhouse, R. C. and Carré, B. A. (1975), Regular algebra applied to path-finding problems, *Journal of the Institute of Mathematics and its Applications* **15**(2), 161–186. (p 18)

Backus, J. (1978), Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Communications of the ACM* **21**(8), 613–641. (pp. 1, 8)

Barth, P. S., Nikhil, R. S. and Arvind (1991), M-structures: Extending a parallel, non-strict, functional language with state, *in* J. Hughes, ed., Conference on Functional Programming Languages and Computer Architecture, LNCS 523, Springer-Verlag, Cambridge, Massachusetts, pp. 538–568. (pp. 19, 67, 70)

Bauderon, M. and Courcelle, B. (1986), An algebraic formalism for graphs, *in* P. Franchi-Zannettacci, ed., 11'th Colloquium on Trees in Algebra and Programming, LNCS 214, Springer-Verlag, Nice, France. (p 16)

Bird, R. S. (1984*a*), The promotion and accumulation strategies in transformational programming, *ACM Transactions on Programming Languages and Systems* **6**(4), 487–504. See also Bird (1985). (pp. 14, 139)

Bird, R. S. (1984*b*), Using circular programs to eliminate multiple traversals of data, *Acta Informatica* **21**(3), 239–250. (pp. 35, 151)

Bird, R. S. (1985), Addendum to the "The promotion and accumulation strategies in transformational programming", *ACM Transactions on Programming Languages and Systems* **7**(3), 490–492. (p 159)

Bird, R. S. (1987), An introduction to the theory of lists, *in* M. Broy, ed., Logic of Programming and Calculi of Discrete Design, Springer-Verlag, pp. 3–42. Also available as Technical Monograph PRG-56, Oxford University. (p 14)

Bird, R. S. (1988), Lectures on constructive functional programming, *in* M. Broy, ed., Constructive Methods in Computer Science, Vol. 55, Springer-Verlag, pp. 151–218. Also available as Technical Monograph PRG-69, Oxford University. (p 14)

Bird, R. and Wadler, P. (1988), *Introduction to Functional Programming*, Prentice Hall. (pp. 21, 134)

Bjerner, B. and Holmström, S. (1989), A compositional approach to time analysis of first-order lazy functional programs, *in* Functional Programming Languages and Computer Architecture, ACM, London, pp. 157–165. (pp. 21, 135, 142, 143, 152)

Brelaz, D. (1979), New methods to color the vertices of a graph, *Communications of the ACM* **22**, 251–256. (p 125)

Brodal, G. S. and Okasaki, C. (1995), Optimal purely functional priority queues. Unpublished manuscript. (pp. 45, 46)

Brown, M. R. (1978), Implementation and analysis of binomial queue algorithms, *SIAM Journal of Computing* **7**(3), 298–319. (pp. 31, 34)

Burstall, R. M. and Darlington, J. (1977), A transformation system for developing recursive programs, *Journal of the ACM* **24**(1), 44–67. (pp. 25, 101, 139)

Burton, F. W. (1982), An efficient functional implementation of FIFO queues, *Information Processing Letters* **14**(5), 205–206. (p 46)

Burton, F. W. and Yang, H.-K. (1990), Manipulating multilinked data structures in a pure functional language, *Software — Practice and Experience* **20**, 1167–1185. (pp. 19, 66)

Carré, B. (1979). *Graphs and Networks*, Oxford Applied Mathematics and Computing Science Series, Oxford University Press, Clarendon Press, Oxford. (p 18)

Chuang, T.-R. and Goldberg, B. (1993), Real-time deques, multihead turing machines, and purely functional programming, *in* Conference on Functional Programming Languages and Computer Architecture, ACM SIGPLAN/SIGARCH, Copenhagen, Denmark, pp. 289–298. (pp. 46, 47)

Clack, C., Clayman, S. and Parrott, D. (1995), Dynamic cyclic data structures in lazy functional languages, Technical report, University College London, Department of Computer Science. (p 71)
URL: `http://www.cs.ucl.ac.uk/staff/clack/papers/guide.html`

Clenaghan, K. (1995), Calculational graph algorithms: reconciling two approaches with dynamic algebra, Report CS-R9518, CWI, Amsterdam, Computer Science, Department of Algorithmics and Architecture. (p 90)

Corman, T. H., Leiserson, C. E. and Rivest, R. L. (1990), *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts. (pp. 21, 109)

Dijkstra, E. W. (1959), A note on two problems in connexion with graphs, *Numerische Mathematik* **1**, 269–271. (pp. 28, 121)

Dijkstra, E. W. (1976), *A Discipline of Programming*, Academic Press. (p 16)

Erwig, M. (1992), Graph algorithms = iteration + data structures? The structure of graph algorithms and a style of programming, *in* E. Mayr, ed., Graph-Theoretic Concepts in Computer Science, LNCS 657, Springer-Verlag, pp. 277–292. (p 13)

Euler, L. (1736), Solutio problematis ad geometriam situs pertinentis (The solution of a problem relating to the geometry of position), *Commentarii Academiae Scientiarum Imperialis Petropolitanae* 8, 128–140. (p 1)

Field, A. J. and Harrison, P. G. (1988), *Functional Programming*, Addison-Wesley. (p 24)

Floyd, R. (1962), Algorithm 97: Shortest path, *Communications of the ACM* **5**(6), 345. (pp. 18, 123)

Fourman, M. (1994), Notes for a course on Algorithms and Data Structures (using ML), given at the University of Western Australia, Perth. (p 45)

Fredman, M. L. and Tarjan, R. E. (1987), Fibonacci heaps and their uses in improved network optimization algorithms, *Journal of the ACM* **34**(3), 596–615. (p 44)

Frenkel, K. and Milner, R. (1993), An interview with Robin Milner, *Communications of the ACM* **36**(1), 90–97. (p 1)

Galler, B. A. and Fisher, M. J. (1964), An improved equivalence algorithm, *Communications of the ACM* **7**(5), 301–303. (p 59)

Gibbons, J. (1991), Algebras for tree algorithms, PhD thesis, Oxford University. Technical monograph PRG-94. (p 15)

Gibbons, J. (1994), An initial-algebra approach to directed acyclic graphs. Department of Computer Science, University of Auckland. (p 17)
URL: `http://www.cs.auckland.ac.nz/~jeremy/publications.html`

Gifford, D. K. and Lucassen, J. M. (1986), Integrating functional and imperative programming, *in* Proceedings of the ACM Conference on Lisp and Functional Programming, ACM, MIT, pp. 28–38. (p 67)

Gill, A., Launchbury, J. and Peyton Jones, S. L. (1993), A short cut to deforestation, *in* Conference on Functional Programming Languages and Computer Architecture, ACM SIGPLAN/SIGARCH, Copenhagen, Denmark, pp. 223–232. (p 139)
URL: `http://www.dcs.gla.ac.uk/fp/authors/Andy_Gill`

Gries, D. (1981), *The Science of Programming*, Springer-Verlag. (p 46)

Gries, D. and Schneider, F. B. (1993), *A Logical Approach to Discrete Math*, Springer-Verlag. (p 16)

Guzmán, J. C. and Hudak, P. (1990), Single-threaded polymorphic lambda calculus, *in* Proceedings of 5'th Annual IEEE Symposium on Logic in Computer Science, pp. 333–343. (p 67)

Harrison, R. (1993), *Abstract Data Types in Standard ML*, John Wiley and Sons. (p 18)

Hartel, P. H. and Glaser, H. (1994), The resource constrained shortest path problem implemented in a lazy functional language, Technical report 94-05, University of Southampton, Department of Electronics and Computer Science. (p 20)
URL: `http://www.ecs.soton.ac.uk/research/tr/94-05/hg.html`

Hibbard, T. N. (1962), Some combinatorial properties of certain trees with applications to searching and sorting, *Journal of the ACM* **9**, 13–28. (p 28)

Hoare, C. A. R. (1969), An axiomatic basis for computer programming, *Communications of the ACM* **12**, 576–580,583. (p 16)

Holyer, I. (1991), *Functional Programming with Miranda*™, Pitman, London. (pp. 18, 69)

Hood, R. and Melville, R. (1981), Real-time queue operations in pure Lisp, *Information Processing Letters* **13**(2), 50–53. (pp. 46, 53)

Hoogerwoord, R. R. (1989), The design of functional programs: a calculational approach, PhD thesis, Eindhoven University of Technology. (p 15)

Hopcroft, J. E. and Tarjan, R. E. (1973), Algorithm 447: Efficient algorithms for graph manipulation, *Communications of the ACM* **16**(6), 372–378. (pp. 83, 147)

Hudak, P. and Fasel, J. H. (1992), A gentle introduction to Haskell, *ACM SIGPLAN Notices* **27**(5). (p xiii)
URL: `http://haskell.systemsz.cs.yale.edu/haskell/tutorial/tutorial.ps.Z`

Hudak, P. and Jones, M. P. (1994), Haskell vs. Ada vs. C++ vs. Awk vs. ... An experiment in software prototyping and productivity, Technical report, Department of Computer Science, Yale University. (p 152)
URL: `ftp://nebula.systemsz.cs.yale.edu:/pub/yale-fp`

Hudak, P., Peyton Jones, S. L., Wadler, P., Arvind, Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, R., Nikhil, R. S., Partain, W. and Peterson, J. (1992), Report on the functional programming language Haskell, Version 1.2, *ACM SIGPLAN Notices* **27**(5). (p xiii)
URL: `ftp://ftp.dcs.gla.ac.uk/pub/haskell/report`

Hughes, J. (1986), A novel representation of lists and its application to the function "reverse", *Information Processing Letters* **22**(3). Also appeared as a Programming Methodology Group Memo PMG–38, Chalmers Institute, Sweden, (1984). (p 139)

Hughes, J. (1989), Why functional programming matters, *The Computer Journal* **32**(2), 98–107. (pp. 8, 88, 151)

Jeuring, J. (1991), The derivation of hierarchies of algorithms on matrices, *in* B. Möller, ed., IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications, North-Holland, pp. 9–32. (p 15)

Jeuring, J. (1992), Theories for Algorithm Calculation, PhD thesis, CWI, Amsterdam, The Netherlands. (p 15)

Johnsson, T. (1995), Lazy monolithic array algorithms. Chalmers University. (pp. 63, 127)

Jones, D. W. (1986), An empirical comparison of priority-queue and event-set implementations, *Communications of the ACM* **29**(4), 300–311. (pp. 29, 44)

Jones, G. and Gibbons, J. (1992), Linear-time breadth-first tree algorithms: an exercise in the arithmetic of folds and zips, Technical Report TR–31–92, Programming Research Group, Oxford University. (p 127)
URL: `http://www.comlab.ox.ac.uk/oucl/publications/tr/TR-31-92.html`

Jones, M. P. (1994), *Release notes for Gofer 2.30*, Computer Science Department, University of Nottingham. (p 49)
URL: `ftp://ftp.cs.nott.ac.uk/nott-fp/languages/gofer`

Kashiwagi, Y. and Wise, D. S. (1991), Graph algorithms in a lazy functional programming language, *in* Proceedings of the 4'th International Symposium on Lucid and Intensional Programming, pp. 35–46. Also available as Technical Report Number 330. Computer Science Department, Indiana University. (p 19)

King, D. J. (1995), Functional binomial queues, *in* K. Hammond, D. N. Turner and P. M. Sansom, eds, Proceedings of the 1994 Glasgow Workshop on Functional Programming, Springer-Verlag, Ayr, Scotland, pp. 141–150. (p xiii)
URL: `http://www.dcs.gla.ac.uk/fp/authors/David_King`

King, D. J. and Launchbury, J. (1995), Structuring depth-first search algorithms in Haskell, *in* The 22'nd Symposium on Principles of Programming Languages, ACM SIGPLAN-SIGACT, San Francisco, California, pp. 344–354. (p xiii)
URL: `http://www.dcs.gla.ac.uk/fp/authors/David_King`

Kingston, J. H. (1990), *Algorithms and Data Structures*, International Computing Science Series, Addison-Wesley. (pp. 21, 61)

Klarlund, N. and Schwartzbach, M. I. (1993), Graph types, *in* 20'th Symposium on Principles of Programming Languages, ACM, Charleston, North Carolina. (p 16)

Knuth, D. E. (1973*a*), *The Art of Computer Programming: Fundamental Algorithms*, Vol. 1, 2'nd edn, Addison-Wesley, Reading, Massachusetts. (pp. 21, 47, 133)

Knuth, D. E. (1973*b*), *The Art of Computer Programming: Sorting and Searching*, Vol. 3, Addison-Wesley, Reading, Massachusetts. (pp. 28, 29)

Knuth, D. E. (1993), *The Stanford GraphBase: A Platform for Combinatory Computing*, ACM Press and Addison-Wesley. (pp. 12, 147)
URL: `ftp://labrea.stanford.edu/`

Kruskal Jr., J. B. (1956), On the shortest spanning subtree of a graph and the travelling salesman problem, *Proceedings of the American Mathematical Society* **7**, 48–50. (pp. 58, 119)

Launchbury, J. (1989), Functional strongly connected components algorithm. Distributed on the `comp.lang.functional` network newsgroup. (p 19)

Launchbury, J. (1993), Lazy imperative programming, *in* Workshop on State in Programming Languages, ACM SIGPLAN, Copenhagen, Denmark, pp. 46–56. (pp. 48. 49)
URL: `http://www.cse.ogi.edu/~jl/Papers`

Launchbury, J. and Peyton Jones, S. L. (1994), Lazy functional state threads, *in* Conference on Programming Language Design and Implementation, ACM SIGPLAN, Orlando, Florida. (pp. 47, 48, 50, 51)
URL: `http://www.dcs.gla.ac.uk/fp/papers/`

Launchbury, J. and Peyton Jones, S. L. (1996), State in Haskell, *Lisp and Symbolic Computation* . (pp. 47, 48, 50, 51, 63)

Manber, U. (1989), *Introduction to Algorithms — A Creative Approach*, Addison-Wesley, Reading, Massachusetts. (pp. 4, 5, 12, 123)

Meertens, L. (1986), Algorithmics: Towards programming as a mathematical activity, *in* J. W. de Bakker, H. Hazewinkel and J. Lenstra, eds, Proceedings of the CWI Symposium on Mathematics and Computer Science, North-Holland, pp. 289–334. (p 14)

Mehlhorn, K. and Naher, S. (1989), LEDA — A library of efficient data types and algorithms, *in* Mathematical Foundations of Computer Science, LNCS 379, Springer-Verlag, pp. 88–106. (p 12)

Meira, S. L. (1985*a*), A linear applicative solution for the set union problem, *Information Processing Letters* **20**, 43–45. (p 67)

Meira, S. L. (1985*b*), On the Efficiency of Applicative Algorithms, PhD thesis, University of Kent, Canterbury. Report number T1. (p 19)

Moggi, E. (1989), Computational lambda-calculus and monads, *in* Symposium on Logic in Computer Science, IEEE, Asilomar, California. (p 48)

Möller, B. (1993*a*), Algebraic calculation of graph and sorting algorithms, *in* D. Bjørner, M. Broy and I. V. Pottosin, eds, Formal Methods in Programming and their Applications, LNCS 735, Berlin, Germany, pp. 394–413. (p 16)

Möller, B. (1993*b*), Derivation of graph and pointer algorithms, Report Number 280, Institut für Mathematik, University of Augsburg, D-86135 Augsburg, Germany. (pp. 16, 90)

Möller, B. and Russling, M. (1992), Shorter paths to graph algorithms, *in* R. S. Bird, C. Morgan and J. Woodcock, eds, Proceedings of the 2'nd International Conference on the Mathematics of Program Construction, LNCS 669, Springer-Verlag, Oxford, UK. For an extended version see Möller and Russling (1994). (p 16)
URL: `http://www.informatik.uni-augsburg.de/info2/mitarbeiter/Russling/`

Möller, B. and Russling, M. (1994), Shorter paths to graph algorithms, *Science of Computer Programming* **22**, 157–180. (p 165)

Morrisett, J. G. (1993), Generalizing first-class stores, *in* Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark, pp. 73–87. Published as Technical Report YALEU/DCS/RR-968, Department of Computer Science, Yale University. (p 70)
URL: `ftp://vache.venari.cs.cmu.edu/usr0/jgmorris/pub/`

Nagl, M. (1979), GRAPL — A programming language for handling dynamic problems on graphs, *in* Proceedings of 5'th International Workshop on Graph Theoretic Concepts in Computer Science, pp. 25–45. (p 13)

Nikhil, R. S. (1991), Id language reference manual, version 90.1, Computation structures group memo 284-2, MIT, Laboratory for Computer Science. (p 67)

Nikhil, R. S. and Arvind (1990), *Programming in Id: A Parallel Programming Language.* (p 19)

Okasaki, C. (1994), Simple and efficient purely functional queues and deques, *Journal of Functional Programming* 4(4). (pp. 46, 48, 53)
URL: `http://foxnet.cs.cmu.edu/people/cokasaki/papers.html`

Okasaki, C. (1996), The role of lazy evaluation in amortized data structures, *in* International Conference on Functional Programming, ACM, Philadelphia, Pennsylvania. (p 45)

Pape, U. (1979), GRAMAS — A graph manipulation system, *in* Proceedings of 5'th International Workshop on Graph Theoretic Concepts in Computer Science, pp. 47–63. (p 13)

Paulson, L. C. (1991), *ML for the Working Programmer*, Cambridge University Press, Cambridge. (pp. 18, 29, 44, 69)

Peyton Jones, S. L., Hall, C., Hammond, K., Partain, W. and Wadler, P. (1993), The Glasgow Haskell compiler: A technical overview, *in* Proceedings of the UK Joint Framework for Information Technology, Technical Conference, Keele. (p xiii)
URL: `http://www.dcs.gla.ac.uk/fp/papers/grasp-jfit.ps.Z`

Peyton Jones, S. L. and Wadler, P. (1993), Imperative functional programming, *in* 20'th Symposium on Principles of Programming Languages, ACM, Charleston, North Carolina. (p 48)
URL: `http://www.dcs.gla.ac.uk/fp/authors/Philip_Wadler`

Ponder, C. G., McGeer, P. C. and Ng, A. P.-C. (1988), Are applicative languages inefficient?, *ACM SIGPLAN Notices* **23**(6), 135–139. (pp. 48, 78)

Pountain, D. (1994), Functional programming comes of age, *Byte* **19**(8), 183–184. (p 8)

Reade, C. (1989), *Elements of Functional Programming*, Addison-Wesley. (pp. 18, 19, 24. 69, 151)

Reade, C. (1992), Balanced trees with removals: an exercise in rewriting and proof, *Science of Computer Programming* **18**, 181–204. (pp. 45, 46)

Reif, J. H. and Scherlis, W. L. (1984), Deriving efficient graph algorithms (summary), *in* E. Clarke and D. Kozen, eds, Logics of Programs, LNCS 164, Springer-Verlag, pp. 421–441. (p 15)

Russling, M. (1994), An algebraic treatment of graph and sorting algorithms, *in* Proceedings of the 14'th International SCCC Conference, Concepción, Chile. Extended version available from Institut für Mathematik der Universität Augsburg, Germany, Report Number 324, 1995. (p 16)
URL: `http://www.informatik.uni-augsburg.de/info2/mitarbeiter/Russling/`

Russling, M. (1995), A general scheme for breadth-first graph traversal, *in* Proceedings of the 3'rd International Conference on the Mathematics of Program Construction, LNCS 947, Springer-Verlag, Kloster Irsee, Germany, pp. 380–398. (p 16)
URL: `http://www.informatik.uni-augsburg.de/info2/mitarbeiter/Russling/`

Sands, D. (1990), Calculi for time analysis of functional programs, PhD thesis, Imperial College, University of London. (pp. 21, 134, 135)
URL: `ftp://theory.doc.ic.ac.uk/theory/papers/Sands`

Sands, D. (1995), A naïve time analysis and its theory of cost equivalence, *The Journal of Logic and Computation* **5**(4), 495–541. Preliminary version available as TOPPS report D-173, DIKU, University of Copenhagen, 1993. (pp. 21, 134, 135, 152)
URL: `ftp://ftp.diku.dk/diku/semantics/papers`

Schoenmakers, B. (1992), Data Structures and Amortized Complexity in a Functional Setting, PhD thesis, Eindhoven University of Technology. (p 20)
URL: `ftp://ftp.cwi.nl/pub/berry`

Sedgewick, R. (1988), *Algorithms*, 2'nd edn, Addison-Wesley, Reading, Massachusetts. (p 21)

Sharir, M. (1981), A strong-connectivity algorithm and its applications in data flow analysis, *Computers and mathematics with applications* **7**(1), 67–72. (pp. 19, 108)

Skiena, S. (1990), *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*, Addison-Wesley. (p 14)

Sleator, D. D. and Tarjan, R. E. (1983), Self-adjusting binary trees, *in* Proceedings of the 15'th Annual ACM Symposium on Theory of Computing, ACM, Boston, Massachusetts, pp. 235–245. (p 29)

Smetsers, S., Barendsen, E., van Eekelen, M. and Plasmeijer, R. (1993), Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. University of Nijmegen. (pp. 67, 70)
URL: `ftp://ftp.cs.kun.nl/`

Swarup, V., Reddy, U. S. and Ireland, E. (1991), Assignments for applicative languages, *in* Functional Programming Languages and Computer Architecture, LNCS 523, Springer-Verlag, pp. 192–214. (p 67)

Tarjan, R. E. (1972), Depth-first search and linear graph algorithms, *SIAM Journal of Computing* **1**(2), 146–160. (pp. 15, 83, 115, 146, 148)

Tarjan, R. E. (1974), Finding dominators in directed graphs, *SIAM Journal of Computing* **3**(1), 62–89. (p 58)

Tarjan, R. E. (1981), A unified approach to path problems, *Journal of the ACM* **28**, 577–595. (p 18)

Tarjan, R. E. (1983), *Data Structures and Network Algorithms*, SIAM. (p 29)

Tarjan, R. E. (1985), Amortized computational complexity, *SIAM Journal on Algebraic and Discrete Methods* **6**, 306–315. (p 133)

Temperley, H. N. V. (1981), *Graph Theory and Applications*, John Wiley and Sons, New York. (p 1)

Thompson, S. (1995), *Miranda*^{TM}*: The craft of Functional Programming*, Addison-Wesley. (p 18)

Vuillemin, J. (1978), A data structure for manipulating priority queues, *Communications of the ACM* **21**(4), 309–315. (pp. 30, 31, 32)

Wadler, P. (1988*a*), Deforestation: Transforming programs to eliminate trees, *in* European Symposium on Programming, LNCS 300, Springer-Verlag, Nancy, France, pp. 344–358. See also Wadler (1990*b*). (pp. 25, 139, 169)

Wadler, P. (1988*b*), Strictness analysis aids time analysis, *in* 15'th Symposium on Principles of Programming Languages, ACM. (p 134)
URL: `http://www.dcs.gla.ac.uk/fp/authors/Philip_Wadler`

Wadler, P. (1990*a*), Comprehending monads, *in* Conference on Lisp and Functional Programming, ACM, Nice, France, pp. 61–78. (p 48)
URL: `http://www.dcs.gla.ac.uk/fp/authors/Philip_Wadler`

Wadler, P. (1990*b*), Deforestation: Transforming programs to eliminate trees, *in* Theoretical Computer Science, LNCS 73, Springer-Verlag, pp. 231–248. See also Wadler (1988*a*). (p 168)
URL: `http://www.dcs.gla.ac.uk/fp/authors/Philip_Wadler`

Wadler, P. (1990*c*), Linear types can change the world!, *in* M. Broy and C. Jones, eds, Programming Concepts and Methods, North Holland. (p 67)

Wadler, P. (1992), The essence of functional programming (invited talk), *in* 19'th Symposium on Principles of Programming Languages, ACM, Santa Fe, New Mexico. (p 48)
URL: `http://www.dcs.gla.ac.uk/fp/authors/Philip_Wadler`

Walther, H. (1984), *Ten Applications of Graph Theory*, D. Reidel, Dordrecht. (p 1)

Wikström, Å. (1987), *Functional Programming using Standard ML*, Prentice Hall. (p 18)

Wilson, R. J. and Beineke, L. W., eds (1979), *Applications of Graph Theory*, Academic Press, London. (p 1)

Wirth, N. (1971), Program development by stepwise refinement, *Communications of the ACM* **14**, 221–227. (p 16)

Wolfram, S. (1991), *Mathematica: A System for Doing Mathematics by Computer*, 2'nd edn, Addison-Wesley. (p 14)

Wright, C. J. (1988), A theory of arrays for program derivation, Transferral dissertation, Oxford University. (p 15)

Zimmermann, W. (1990), *Automatische Komplexitätsanalyse Funktionaler Programme*, Informartik-Facherichte, Springer, Berlin. (p 3)