Inductive Graphs and Functional Graph Algorithms

Martin Erwig
Oregon State University
Department of Computer Science
Corvallis, Oregon 97331, USA
(*e-mail:* erwig@cs.orst.edu)

Abstract

We propose a new style of writing graph algorithms in functional languages which is based on an alternative view of graphs as inductively defined data types. We show how this graph model can be implemented efficiently, and then we demonstrate how graph algorithms can be succinctly given by recursive function definitions based on the inductive graph view. We also regard this as a contribution to the teaching of algorithms and data structures in functional languages since we can use the functional-style graph algorithms instead of the imperative algorithms that are dominant today.

Keywords: Graphs in Functional Languages, Recursive Graph Algorithms, Teaching Graph Algorithms in Functional Languages

1 Introduction

How should I implement a graph algorithm in a functional programming language? This seemingly simple question has attracted attention for quite a long time, and there are many different proposals for how to do so. Of course, it is not really difficult to *somehow* realize graph algorithms in functional languages. The real challenge is to obtain *clear* and *efficient* programs, that is, functional programs that do not lose their elegance and simplicity and that have the same asymptotic complexity as imperative ones.

The main difficulties that arise when dealing, for example, with depth-first search in functional languages are caused by the fact that a node might be reachable via different edges, whereas the algorithm requires that it must be visited at most once. In traditional descriptions of graph algorithms and in imperative languages this behavior is achieved by simply marking a node as being visited after it has been encountered the first time. When a node is reached again, checking its mark prevents the algorithm from re-processing it (and also from possibly running into an infinite loop). This node-marking strategy can easily be mimicked in functional languages: remember visited nodes in a data structure and pass this data structure through all function calls that occur in the context of the algorithm. In this way a local state of node marks is maintained by the algorithm. However, this approach bears two problems with regard to efficiency and clarity. First, whereas in the imperative setting marking a node and testing for a node mark can be performed in constant time by using an array, functional set data structures generally cannot meet this time bound: for example, using balanced binary search trees, set insertion and membership test take $O(\log n)$ time (where n denotes the number of elements in the set). Second, the threading of data structures requires all participating functions to have an additional parameter for passing the state around, and this affects the readability of the algorithms and, what is worse, the ease of manipulating programs and proving program properties, which is a key attribute of functional programming.

One solution to the first problem is to use monads to thread arrays with constant-time access through function calls (King & Launchbury, 1995; King, 1996). However, this complicates the function definitions, making them less readable and more difficult to understand, as well as forcing the algorithms to be written in an imperative style. Another answer to the first problem is the use of uniqueness types (offered, for example, by the Clean language (Barendsen & Smetsers, 1996)) or to rely on automatic discovery of single-threadedness and generate in these cases safe imperative code (Hudak & Bloss, 1985; Sastry *et al.*, 1993). Despite the facts that uniqueness types are not widespread (for example, neither Standard ML nor Haskell has them) and that single-threaded analysis cannot discover all cases, both approaches adhere to the imperative node marking view of graph algorithms and thus provide no answer to the question of functional style.

Another aspect is that of teaching graph algorithms in functional languages. Why, the reader might ask, is this an issue at all? It can be seen from newsgroup discussions that re-appear in rather regular intervals that functional languages are still in need of defending themselves and demonstrating that they can be used as well as mainstream imperative or object-oriented languages.

Now the proof of usability of a programming language manifests itself not only in applications that are written in that language, but also to a certain degree in the available teaching materials for that language. For example, textbooks can convince programmers and students that a language is really usable. First of all, explanations of the language itself are needed, so that programmers and students are able to learn the language and its programming style. There are quite a few textbooks available introducing functional programming in general and also particular functional languages, but only very few that could be used for a (general) course in algorithms and data structures — one example is Rabhi and Lapalme (1999). However, to be really cogent in saying that functional languages are a true alternative to imperative or object-oriented languages, it is indispensable to have also teaching material in functional languages to demonstrate that functional languages are not just toys, but can also address standard topics in algorithms and data structures. Moreover, this teaching material could also be used to implement a typical (undergraduate) curriculum completely in a functional language. If, on the other hand, functional algorithm and data structure textbooks are not available, the fatal impression is conveyed that to implement real data structures, one has resort to C or the like.

We believe that this leaves a very negative impression of the usability of functional languages in general.

Why do so few functional data structure textbooks exist? We believe that one reason is that the treatment of graphs in functional languages has been rather weak so far.

Hence, the goal of this paper is twofold. First, we want to demonstrate that it is possible to define graph algorithms in a distinctive functional style and that these algorithms are at the same time often competitive in terms of efficiency with typical imperative implementations. Second, by giving a collection of algorithms typically found in courses on algorithms and data structures, we try to close a gap in functional algorithms and data structure textbooks. In Section 2 we review related work. The foundations for functional graph algorithms are laid in Section 3 where the inductive graph view is explained. There we also discuss the implementation of these functional graphs. In Section 4 we then define several graph algorithms by recursive function definitions that follow the inductive graph structure. Conclusions given in Section 5 complete this paper.

The source code for all the examples of this paper are available as part of the *Functional Graph Library* (*FGL*) that can be obtained through the World-Wide Web from `www.cs.orst.edu/~erwig/fgl/`. Throughout this paper we use Haskell notation. As an extension we employ *active patterns*, which provide a special kind of pattern matching explained in Section 3.2, because this allows a more succinct description of most algorithms. The use of active patterns is is helpful, but not essential, and the versions of the algorithms in the FGL are actually defined without using them.

2 Related Work

A straightforward approach to implementing graph algorithms in functional languages proposed in Burton and Yang (1990) is to pass the state used by graph algorithms through function calls where the state itself is represented by a functional array. This is certainly a standard way of implementing any imperative algorithm in a functional language. Burton and Yang show how classical algorithms can be translated into a lazy functional language, but no particular use of functional languages is made in the design of the algorithms themselves.

In contrast, Kashiwagi and Wise (1991) describe algorithms as fixed points of recursive equations, which essentially relies on lazy evaluation. The algorithms become quite complex and are rather difficult to comprehend. As with Burton and Yang (1990) this approach does not achieve the asymptotic running time of imperative algorithms.

A kind of combinator approach in Erwig (1992), which identified some classes of graph algorithms and introduced a few corresponding predefined operators. A graph algorithm is realized by selecting an operator and providing it with appropriate parameter functions and data structures. We believe that the approach reflects the structure of graph algorithms very well. However, like in the previous two approaches there is not much potential for formal program manipulation. Another drawback is that the combinator approach is limited in expressiveness.

The proposal of King and Launchbury (1995) is concerned only with depth-first search, and the focus is on a generated data structure, the depth-first spanning forest, instead of the underlying graph algorithm. This facilitates formal reasoning — in particular, the formal development of many algorithms based on depth-first search becomes possible. The depth-first search function itself is realized nicely in a generate-and-prune manner. Monads are used to implement the state maintained during the search (that is, the vertices visited) to achieve linear running time. At this point the approach is stuck with the imperative programming style. Although encapsulated and restricted to a single point, it comes up in the process of program fusion where transformations become quite complex when functions are moved across state transformers. For example, see Launchbury (1995) where it is demonstrated how phase fusion can be applied to eliminate intermediate results of some of these algorithms. King (1996) defines in his thesis many more algorithms, but as with depth-first search, the defined functions are mainly implementations of imperative algorithms.

Fegaras and Sheard (1996) investigate a generalization of fold operations to data types with embedded functions. As one motivating example they show how to model graphs. However, that approach is somewhat limited (it is not clear how to define, for example, a function for reversing all edges in a graph) and it is highly inefficient since direct access to a node requires, in general, traversal of the whole graph.

Also related is the work of Gibbons (1995) who considers the definition of graph fold operations within an algebraic framework. But he deals only with acyclic graphs, and an implementation is not discussed. A categorical definition for fold operators on abstract data types was proposed in Erwig (2000). In that approach the decomposition of ADTs can be controlled by external values. One main application was the definition of graph operations like depth-first search that, in contrast to Gibbon's approach, work on arbitrary graphs.

In contrast to the monolithic view of graphs which is so dominating that it is even adopted by most functional approaches, we suggest to view graphs inductively, as a data type defined by two constructors, much like lists or trees. This view was first presented in Erwig (1997b) where the focus was to define several kinds of graph fold operations and to identify laws for them that can be used for program transformation. Also a first implementation of functional graphs was provided. In Erwig (1997a) we have extended the implementation in several ways and have compared different representation schemes by performing some benchmarks. The inductive graph view has also applications that go beyond the realization of functional graph algorithms. For example, inductive graphs have facilitated the denotational semantics definition of visual languages (Erwig, 1998b). Another application, which has a strong educational component, is the purely functional description of graph reduction (Erwig, 1998a). Still another application that we are currently investigating is the treatment of graph grammars in a functional setting.

Most of the existing textbooks on functional programming concentrate on explaining the fundamental programming and language concepts. To a certain extent data structures are sometimes covered, too. In fact, most books contain examples of list and tree algorithms, for example, Bird (1998) and Ullman (1998), but they do not treat graphs. Some books mention graphs, but do so rather superficially (Paulson, 1996; Reade, 1989).

A dedicated data structure textbook is Rabhi and Lapalme (1999). To some degree the book by Harrison (1989) could also be used. Both books contain material about graphs and graph algorithms, but the representations chosen are those known from the imperative world. This causes a breach in the presentation since the distinctive clear and succinct functional programming style is lost to some degree. Several graph algorithms are also contained in King's PhD thesis (King, 1996). These make heavy use of monads and rely very much on state-based computations. Hence, the algorithms are as well mainly those known from the imperative world. The most comprehensive and most advanced book on data structures in functional languages is that of Chris Okasaki (1998). However, it does not contain material on graphs or graph algorithms.

3 Inductive Graphs

The prevailing view of graphs in programming is that of a large monolithic block: disregarding node and edge labels, a graph is viewed as a pair $G = (V, E)$ where V is a set of nodes and $E \subseteq V \times V$ is a set of edges. The descriptions of algorithms that work incrementally on graphs, that is, algorithms that visit nodes one after the other, then need an additional data structure for remembering the parts of the graph that have already been dealt with. Alternatively, the graph representation is defined to have additional fields that allows for marking nodes and edges directly in the graph structure itself.

This "node marking" strategy reflects an inherently imperative style of algorithms, and this also shows up a bit painfully when one tries to implement these algorithms in a functional language: one has to thread a data structure for the node marks through all the functions that are involved in the implementation of an algorithm. This might be done by passing an additional parameter or by using monads. In any case, a state has to be threaded through the algorithm, and this complicates all aspects of the algorithm. Moreover, it complicates correctness proofs and program transformations considerably.

This has to be seen in contrast to list or tree algorithms that have beautiful and simple definitions not needing additional bookkeeping. The reason is that lists and trees are inductively defined data types, and function definitions, which can follow quite naturally the definition of the argument data type, are inductive in style, too. Finally, the use of pattern matching contributes significantly to the succinctness and elegance of those function definitions.

Now what we are proposing is essentially to regard a graph as an inductive data type. This makes graph algorithms amenable to inductive function definitions with all their advantages.

Graphs will conceptually be represented by two constructors; we will introduce these constructors in Section 3.1. Simple algorithms can be implemented immediately using pattern matching on these two basic constructors. However, more advanced algorithms require the ability to visit nodes in specific order, and this is supported by a particular kind of pattern matching, which is described in Section 3.2. In Section 3.3 we describe and discuss several ways to implement inductive graphs. In the following we denote by n (m) the number of nodes (edges) in a graph.

3.1 Graph Constructors

A graph consists of a set of nodes that are connected by edges. For simplicity we assume that nodes are represented by integers, and for generality we define a single graph type for directed node- and edge-labeled multi-graphs. Other graph types can be obtained as special cases: for example, undirected graphs can be simulated by directed graphs having a symmetric edge structure, where we say that a directed graph g *properly represents* an undirected graph if for each edge (v, w) in g there is also an edge (w, v) in g with the same label. Moreover, unlabeled graphs simply have the node and/or edge label type "()" (unit).

The inductive view of graphs is captured in the following description: a graph is either the empty graph or a graph extended by a new node v together with its label and with edges to those of v's successors and predecessors that are already in the graph. The representation of each edge contains the successor/predecessor node and the label of the edge. This information about a one-step inductive graph extension is contained in a type called the *context*.

$$\begin{array}{lcl}
\textbf{type}\ Node & = & Int \\
\textbf{type}\ Adj\ b & = & [(b, Node)] \\
\textbf{type}\ Context\ a\ b & = & (Adj\ b, Node, a, Adj\ b)
\end{array}$$

The graph type itself is implemented for efficiency reasons as an abstract type (see Section 3.3). However, it is very convenient to think of the graph type being defined as an algebraic type with two constructors *Empty* and & (used in infix notation):

$$\textbf{data}\ Graph\ a\ b\ =\ Empty \mid Context\ a\ b\ \&\ Graph\ a\ b$$

The above definition suggests that graphs are isomorphic to lists, however this is not the case because graphs are not freely generated by *Empty* and &. With these two constructors we can now denote graphs by data type terms. Consider, for instance, the graph shown (on the left) in Figure 1.

We can build this graph, for example, with the following expression (first the solid, then the dashed, and finally the dotted part, see graph in the middle of Figure 1):

$$\begin{array}{ll}
([(\text{"}left\text{"}, 2), (\text{"}up\text{"}, 3)], 1, \text{'}a\text{'}, [(\text{"}right\text{"}, 2)]) & \& \\
\qquad\qquad\qquad ([\,], 2, \text{'}b\text{'}, [(\text{"}down\text{"}, 3)]) & \& \\
\qquad\qquad\qquad ([\,], 3, \text{'}c\text{'}, [\,]) & \&\ Empty
\end{array}$$

The chosen order of inserting node contexts is not the only possible one. For example, we can also reverse the order. Then, however, the contexts have to be changed

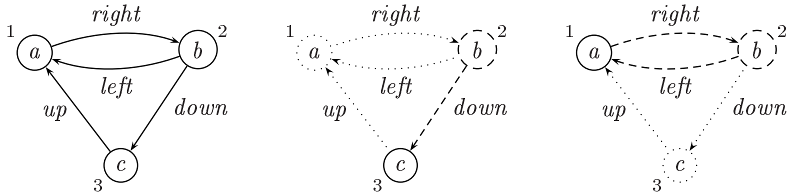

Figure 1. Directed graph with two inductive constructions.

accordingly since we can refer in predecessor and successor lists only to nodes that are already present in the graph to be extended (see graph on the right in Figure 1).

$$\begin{array}{rl}
([(\text{"}\mathit{down}\text{"}, 2)], 3, \text{'}c\text{'}, [(\text{"}\mathit{up}\text{"}, 1)]) & \& \\
([(\text{"}\mathit{right}\text{"}, 1)], 2, \text{'}b\text{'}, [(\text{"}\mathit{left}\text{"}, 1)]) & \& \\
([\ ], 1, \text{'}a\text{'}, [\ ]) & \&\ \mathit{Empty}
\end{array}$$

Since & is defined as a function, consistency checks for graph construction can be integrated. In fact, an error is reported when a context is added for a node that is already present in the graph or when a node mentioned in the successor or predecessor list is missing in the graph.

Actually, we can choose an arbitrary order of node insertion for building a graph. This is a very important property that sets the theoretical foundation for the possibility of a powerful kind of pattern matching on graphs to be described in the next subsection. We express this result by the following two facts:

Fact 1 (*Completeness*)
Each labeled multi-graph can be represented by a graph term.

Fact 2 (*Choice of Representation*)
For each graph g and each node v contained in g there exist p, l, s and g' such that (p, v, l, s) & g' denotes g.

These two observations can also be established more formally: we can define a semantics of the graph constructors and express the relationships based on this semantics, see Erwig (1997b).

The inductive graph view does not mean that one is always forced to invent a proper sequence of contexts to define graphs. In fact, this can become quite tedious, and we have therefore defined a number of operations to insert lists of nodes and edges into a graph. In this connection we also mention the function *newNodes* that yields a list of nodes that are *not* contained in a graph.

$$\begin{array}{l}
\mathit{newNodes} :: \mathit{Int} \to \mathit{Graph}\ a\ b \to [\mathit{Node}] \\
\mathit{newNodes}\ i\ g\ =\ [n+1..n+i]\ \mathbf{where}\ n\ =\ \mathit{foldr}\ \mathit{max}\ 0\ (\mathit{nodes}\ g)
\end{array}$$

This function is particularly useful for extending a graph whose construction history is not known. The function *nodes* extracts the node values from a graph; it is defined in the next subsection.

3.2 Pattern Matching on Graphs

Having introduced and described inductive graphs as terms, we can use pattern matching on this representation. First, we can define elementary functions like:

$$
\begin{array}{l}
\mathit{isEmpty} :: \mathit{Graph}\ a\ b \ \rightarrow \mathit{Bool} \\
\mathit{isEmpty}\ \mathit{Empty} \;=\; \mathit{True} \\
\mathit{isEmpty}\ \_ \qquad\quad = \; \mathit{False}
\end{array}
$$

But we can also realize more interesting operations. For example, we can define a map function for graphs by simple term pattern matching:

$$
\begin{array}{l}
\mathit{gmap} :: (\mathit{Context}\ a\ b \ \rightarrow\ \mathit{Context}\ c\ d) \rightarrow \mathit{Graph}\ a\ b \rightarrow \mathit{Graph}\ c\ d \\
\mathit{gmap}\ f\ \mathit{Empty} \;\; = \; \mathit{Empty} \\
\mathit{gmap}\ f\ (c\ \&\ g) \; = \; f\ c\ \&\ \mathit{gmap}\ f\ g
\end{array}
$$

Note that gmap preserves the structure of the nodes, but not necessarily of the edges. A graph reversal function can be easily defined using *gmap*:

$$
\begin{array}{l}
\mathit{grev} :: \mathit{Graph}\ a\ b \ \rightarrow\ \mathit{Graph}\ a\ b \\
\mathit{grev} \;=\; \mathit{gmap}\ \mathit{swap}\ \textbf{where}\ \mathit{swap}\ (p, v, l, s) \;=\; (s, v, l, p)
\end{array}
$$

The advantages of this programming style are, in particular, very simple proofs of program properties or transformation rules. For example, it needs just a couple of lines to prove by induction, say, a fusion law for *gmap* and an inversion rule for *grev*:

$$
\begin{array}{ll}
\mathit{gmap}\ f\ .\ \mathit{gmap}\ f' \;=\; \mathit{gmap}\ (f\ .\ f') & (\mathit{gmap\ fusion}) \\
\mathit{grev}\ .\ \mathit{grev} \;=\; \mathit{id} & (\mathit{grev\ inversion})
\end{array}
$$

We can prove *gmap fusion* by induction on the graph structure. For $g = \mathit{Empty}$ we have by definition $\mathit{gmap}\, f\ (\mathit{gmap}\, f'\ \mathit{Empty}) \;=\; \mathit{gmap}\, f\ \mathit{Empty} \;=\; \mathit{Empty} \;=\; \mathit{gmap}\ (f\ .\ f')\ \mathit{Empty}$. Otherwise, with $g = c\ \&\ g'$ we conclude by induction:

$$
\begin{array}{rcll}
\mathit{gmap}\ f\ (\mathit{gmap}\ f'\ g) & = & \mathit{gmap}\ f\ (\mathit{gmap}\ f'\ (c\ \&\ g')) & (\text{Def. } g) \\
& = & \mathit{gmap}\ f\ (f'\ c\ \&\ (\mathit{gmap}\ f'\ g')) & (\text{Def. } \mathit{gmap}) \\
& = & f\ (f'\ c)\ \&\ \mathit{gmap}\ f\ (\mathit{gmap}\ f'\ g') & (\text{Def. } \mathit{gmap}) \\
& = & (f\ .\ f')\ c\ \&\ \mathit{gmap}\ (f\ .\ f')\ g' & (\text{Ind. Hyp.}) \\
& = & \mathit{gmap}\ (f\ .\ f')\ (c\ \&\ g') & (\text{Def. } \mathit{gmap}) \\
& = & \mathit{gmap}\ (f\ .\ f')\ g & (\text{Def. } g)
\end{array}
$$

In the proof for the second equation we need the following two obvious facts about *swap* and *gmap*:

$$
\begin{array}{ll}
\mathit{swap}\ .\ \mathit{swap} \;=\; \mathit{id} & (\mathit{swap\ idempotency}) \\
\mathit{gmap}\ \mathit{id} \;=\; \mathit{id} & (\mathit{gmap\ unit})
\end{array}
$$

Now we can prove *grev inversion* with the help of the *gmap fusion* law.

$$
\begin{array}{rcll}
\mathit{grev}\ .\ \mathit{grev} & = & \mathit{gmap}\ \mathit{swap}\ .\ \mathit{gmap}\ \mathit{swap} & (\text{Def. } \mathit{grev}) \\
& = & \mathit{gmap}\ (\mathit{swap}\ .\ \mathit{swap}) & (\mathit{gmap\ fusion}) \\
& = & \mathit{gmap}\ \mathit{id} & (\mathit{swap\ idempotency}) \\
& = & \mathit{id} & (\mathit{gmap\ unit})
\end{array}
$$

To really appreciate the elegance of this proof, the reader might try to prove the same property for the imperative graph reversal algorithm that works by iterating over all adjacency lists.

Another useful basic function on graphs is *ufold*.[1]

$$
\begin{aligned}
&\mathit{ufold} :: (\mathit{Context}\ a\ b \to c \to c) \to c \to \mathit{Graph}\ a\ b \to c\\
&\mathit{ufold}\ f\ u\ \mathit{Empty}\ = u\\
&\mathit{ufold}\ f\ u\ (c\ \&\ g) = f\ c\ (\mathit{ufold}\ f\ u\ g)
\end{aligned}
$$

With *ufold* we can implement *gmap* and a couple of other graph functions:

$$
\mathit{gmap}\ f\ = \mathit{ufold}\ (\backslash c \to (f\ c\ \&))\ \mathit{Empty}
$$

$$
\begin{aligned}
&\mathit{nodes} :: \mathit{Graph}\ a\ b \to [\mathit{Node}]\\
&\mathit{nodes}\ = \mathit{ufold}\ (\backslash(p, v, l, s) \to (v{:}))\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{undir} :: \mathit{Eq}\ b \Rightarrow \mathit{Graph}\ a\ b \to \mathit{Graph}\ a\ b\\
&\mathit{undir}\ = \mathit{gmap}\ (\backslash(p, v, l, s) \to \mathbf{let}\ \mathit{ps} = \mathit{nub}\ (p \mathbin{+\!\!+} s)\ \mathbf{in}\ (\mathit{ps}, v, l, \mathit{ps}))
\end{aligned}
$$

Since graphs are implemented in FGL as an abstract type, the reader should be aware of the fact that & is a function and not a constructor and therefore cannot be used in patterns. Instead, FGL defines the predicate *isEmpty* and a function for extracting an arbitrary context:

$$
\mathit{matchAny} :: \mathit{Graph}\ a\ b \to (\mathit{Context}\ a\ b, \mathit{Graph}\ a\ b)
$$

Note that *matchAny* reports an error when it is applied to empty graphs.

Now a function like *gmap* is implemented in FGL typically as follows.

$$
\begin{aligned}
\mathit{gmap}\ f\ g\ &|\ \mathit{isEmpty}\ g\ = g\\
&|\ \mathit{otherwise}\ = f\ c\ \&\ (\mathit{gmap}\ f\ g')\\
&\mathbf{where}\ (c, g')\ = \mathit{matchAny}\ g
\end{aligned}
$$

In general, contexts are *not* encountered in the reverse order in which they were inserted into a graph. However, this does not affect reasoning as demonstrated above since preconditions like $g = c\ \&\ g'$ refer to *arbitrary* representations. As far as practical work with FGL is concerned, not keeping the term representation is not a problem either since equality of graphs is defined on the basis of the set of nodes and edges contained in graphs. Thus, like we have in our model $g = \mathit{gmap}\ \mathit{id}\ g$, we also find in FGL that $g == \mathit{gmap}\ \mathit{id}\ g$ evaluates to *True*. Another question that arises with different term representations for one graph is whether graph algorithms defined using *ufold* or *gmap* are correct at all because they might encounter an arbitrary representation. Correctness of an algorithm requires that it is in a certain sense robust with respect to the term representation of a graph, that is, it works for an arbitrary (valid) term representation. Indeed, this question has to be considered for each definition individually. For example, the definition of *grev* works correctly

[1] The "*u*" stands for *unordered* and emphasizes that the order of encountering nodes is not important. Other fold operations, in particular, using ordered graph decomposition, are defined in Erwig (1997b).

since it does not change the order of contexts and since the dependency of the successor/predecessor nodes on being already inserted by previous contexts is not affected by exchanging the successor and predecessor lists. In contrast, for functions f that change the contexts in arbitrary ways, *gmap* may succeed for some orders and fail for others, because of the requirement that successor/predecessor nodes already exist in the graph.

Many other algorithms require contexts to be matched in a very specific order. At this point, the multiple representations for graphs offer a great opportunity to define a special kind of pattern matching that just allows the selection of contexts for specific nodes.

In Erwig (1996) we introduced *active patterns*, which essentially extend patterns by a function component that is applied to the argument value before it is matched against the pattern. This function can be used to transform the argument into the desired form so that matching can extract afterwards just the parts in their desired form. In this respect active patterns are similar to views (Wadler, 1987; Burton & Cameron, 1993). However, an active pattern's function has access to external values other than the argument, which facilitates the external control of the argument reorganization. This is not possible with views, but it is possible with the approach presented in Erwig and Peyton Jones (2000) and, to a limited degree, also with the proposal of Palao Gostanza *et al.* (1996). (For a more detailed comparison with several other pattern-matching extensions, see Erwig and Peyton Jones (2000).)

We are not going to explain active patterns in full detail here; for our purpose it is sufficient just to define an *active graph pattern*: Fact 2 tells us that for each node v contained in a graph there is a term representation (p, v, l, s) & g for some suitable p, l, s, and g. Now the active pattern $(c \overset{v}{\&} g)$ is matched against a graph g' by searching for node v in g' and transforming, at least conceptually, g' into a term representation in which v's context is inserted last, so that it is the argument of the outermost application of &. Of course, this can be done only if v is contained in g'. In that case the pattern is said to match and v's context is bound to c, and the graph without the context, that is, without v and its incident edges, is bound to g. On the other hand, if v is not contained in g', the pattern fails, no bindings are produced, and pattern matching continues as after a normal pattern-match failure. Note that the v in $\overset{v}{\&}$ is an expression, not a pattern. If it is a variable, then it must already be bound to a value when the active graph pattern is evaluated. This typically happens by using v as a parameter preceding the graph pattern.

Some examples of the use of active graph patterns are determining a node's successors, computing the degree of a node, or deleting a node from a graph:

$$
\begin{aligned}
&gsuc :: Node \rightarrow Graph\ a\ b \rightarrow [Node]\\
&gsuc\ v\ ((\_,\_,\_,s) \overset{v}{\&}\ g) \ = \ map\ snd\ s\\
\\
&deg :: Node \rightarrow Graph\ a\ b \rightarrow Int\\
&deg\ v\ ((p,\_,\_,s) \overset{v}{\&}\ g) \ = \ length\ p + length\ s\\
\\
&del :: Node \rightarrow Graph\ a\ b \rightarrow Graph\ a\ b\\
&del\ v\ (\_ \overset{v}{\&}\ g) \ = \ g
\end{aligned}
$$

Table 1. *Basic Graph Operations*

Construction		Decomposition	
Empty graph	(*Empty*)	Test for empty graph	(*Empty*-match)
Add context	(&)	Extract arbitrary context	(&-match)
		Extract specific context	($\overset{v}{\&}$)

Note that the use of active patterns is not essential; we can always rewrite functions by explicitly calling a function *match* that searches for the context of a given node v in a graph g and returns this context (if found) together with the remaining graph:

$$match :: Node \to Graph\ a\ b \to (Maybe\ (Context\ a\ b), Graph\ a\ b)$$

Then a function f that uses an active pattern in the following way:

$$\begin{aligned} f\ p\ (c \overset{v}{\&}\ g) &= e \\ f\ p\ g \quad\quad &= e' \end{aligned}$$

can be rewritten with a **case** expression as

$$\begin{aligned} f\ p\ g' \ = \ \textbf{case}\ match\ v\ g'\ \textbf{of} \\ (Just\ c, g) \quad &\to \quad e \\ (Nothing, g) \quad &\to \quad e' \end{aligned}$$

(Note that other equations for f that precede the one with the active pattern can be kept unchanged. See, for instance, *dfs* in Section 4.1.)

Now the function *gsuc* can also be implemented as:

$$\begin{aligned} gsuc\ v\ g' \ = \ \textbf{case}\ match\ v\ g'\ \textbf{of} \\ (Just\ (\_, \_, \_, s), g) \quad \to \quad map\ snd\ s \end{aligned}$$

In fact, this is the way functions are implemented in FGL because active patterns are not available in Haskell.

In addition to *gsuc* we will frequently need a function that selects the successors from a known context. Therefore, we also define:

$$\begin{aligned} &suc :: Context\ a\ b \to [Node] \\ &suc\ (\_, \_, \_, s) \ = \ map\ snd\ s \end{aligned}$$

3.3 Implementation and Complexity

The implementation of inductive graphs has to support the operations for constructing and decomposing graphs shown in Table 1.

In particular, graphs have to be fully persistent, that is, updates on a graph must leave previous versions intact.

3.3.1 *Graph Representations and Persistence*

One idea is to use a plain term representation, which offers persistence for free. However, a closer look rules out this option because the implementation of the $\&^v$ pattern is hopelessly inefficient, and even the implementation of the & function is inefficient since it has to ensure the existence of the predecessors and successors and the non-existence of the newly inserted node, and testing this takes at least linear time with respect to the size of the graph.

Considering the imperative world, there are two main representations: adjacency lists and incidence matrices. Each has strengths and weaknesses. Except for special applications, adjacency lists are generally favored over incidence matrices because

1. adjacency lists require less space for all but very dense graphs and
2. adjacency lists offer $O(1)$ access time to the successors of an arbitrary node in contrast to $\Omega(n)$ time needed to scan a complete row in an incidence matrix.

We have therefore concentrated on two alternatives for making adjacency lists persistent: the first representation uses a variant of the version-tree implementation of functional arrays, and the second representation stores successor and predecessor lists in a balanced binary search tree. The version-tree implementation is based on Aasa *et al.* (1988) and records changes to the original array in an inward directed tree of (index, value) pairs that has the original array at its root.[2] Each different array version is represented by a pointer to a node in the version tree, and the nodes along the path to the root mask older definitions in the original array (and the tree). Adding a new node to the version tree can be done in constant time, but index access might take up to u steps where u denotes the number of updates to the array. This basic structure can be extended by an additional cache array, and we add a further array carrying time stamps for nodes. Moreover, to support some specialized operations efficiently, this structure is supplemented by a two-array implementation of node partitions to keep track of inserted and deleted nodes.

3.3.2 *Optimizing the Version-Tree Array Representation*

In the version-tree representation, the implementation of $\&^v$ becomes quite inefficient since the deletion of a context (p, v, l, s) requires the removal of v from each predecessor's successor list and from each successor's predecessor list. When c denotes the size of the context ($c \approx$ *lengthp* + *length s*), this means a running time of $O(uc^2)$ (recall that u gives the total number of previous updates to the graph). By keeping the predecessors and successor in balanced binary search trees, the effort can be reduced to $O(uc \log c)$.

[2] In fact, there are more sophisticated functional array implementations available, for example, Dietz (1989) and O'Neill and Burton (1997). However, the implementation of these data structures requires considerable effort, and benchmarks have shown that even the simpler version-tree implementation does not deliver in practice what its asymptotic complexity promises (Erwig, 1997a).

Avoiding Node Deletion. To avoid these costly deletion activities we equip each node in the graph with a positive integer, and this integer is negated once the node is deleted. Positive node stamps are also put into successor/predecessor lists. Now when a node context is deleted, we need not remove v from all referencing successor and predecessor lists because when a successor list l (of a node w) is accessed that contains v, all elements that have non-matching stamps are ignored, that is, v will not be returned as a successor because it has a negative node stamp whereas l contains v with a positive stamp. When v is re-inserted into the graph later, we make the stamp of v positive again and increase it by 1, and we take this new stamp over to all newly added predecessors and successors. Now if w is not among the new predecessors, the old entry in l is still correctly ignored when l is accessed because its value is smaller than v's current stamp.

In practice, the garbage nodes in successor and predecessor lists (that is, invalid and unused references to deleted nodes) do not seem to be a significant source of inefficiency for most applications. For example, in the case of graph reduction, where graphs are heavily updated, only 25-30% of nodes in successor and predecessor lists are filtered out due to invalid stamps.

Avoiding Version-Tree Lookups. We can define the "leftmost" node of a version tree as the leaf that is reached by repeatedly following the first added version of each node starting at the root. Now we add an imperative cache array to the leftmost node of the version tree. This means that the array represented by that node is, in fact, duplicated. Since index access within this array is possible in constant time, algorithms that use the functional array in a single-threaded way have the same complexity as in the imperative case because the version tree degenerates to a left spine path with the leaf node offering constant-time access during the whole algorithm.

There is a subtlety in this implementation involving having just one cache array: if a functional array is used a second time, the cache has already been consumed for the previous computation and cannot be used again. This gives a surprising time behavior: the user executes a program on a functional array, and it runs quite fast. However, running the same program again results, in general, in much larger execution times since all access now goes through the version tree. Therefore, we create in our implementation a new cache for each new version derived from the root of the version tree.

Support for Special Operations. The version-tree implementation described so far is surprisingly inefficient for the operations *Empty*-match and &-match. Testing for the empty graph can be easily supported by extending the graph representation to include the number of nodes in the graph. The operation &-match is more difficult to realize because we must, in general, scan the whole stamp array to find a valid (that is, non-deleted) node. Note that even a simple imperative array implementation requires, in general, linear time for this operation because it scans the whole array. (This is not surprising since the question of graph updates is completely ignored anyway in almost all descriptions of imperative graph representations.)

To account for &-match we keep for each graph a partition of inserted nodes (that is, nodes existent in the graph) and deleted nodes: when a node is deleted (decomposed), it is moved from the inserted-set into the deleted-set, when a node is inserted into the graph, it is moved the other way. The node partition is realized by two arrays, *index* and *elem*, and an integer k giving the number of existent nodes, or, equivalently, pointing to the last existing node. The array *elem* stores all existent nodes in its left part and all deleted nodes in its right part, and *index* gives for each node its position in the *elem* array. A node v is existent if $index[v] \leq k$, and it is deleted if $index[v] > k$. Inserting a new node v means to move it from the deleted-set into the inserted-set. This is done by exchanging v's position in *elem* with the node stored at $elem[k+1]$ (that is, the first deleted node) followed by increasing k by 1. The entries in *index* must be updated accordingly. To delete node v, first swap v and $elem[k]$, and then decrease k by 1. All this is possible in constant time.

Now all the above mentioned graph operations can be implemented to work in constant time: &-match can be realized by calling $\&^v$ with $elem[1]$, and *Empty*-match is true if $k = 0$. Moreover, some other useful graph operations are efficiently supported by the node partition: a list of i fresh nodes as required for the implementation of *newNodes* is simply given by $[elem[k+1], \ldots, elem[k+i]]$, k gives the number of nodes in the graph, and all nodes can be reported in time $O(k)$ which might be much less than the size of the array. The described implementation of node partitions is an extension of the sparse set technique proposed in Briggs and Torczon (1993). The drawback of this extension is that keeping the partition information requires additional space and causes some overhead. Moreover, graphs still are not truly dynamic since arrays can neither grow nor shrink.

3.3.3 Binary Search Tree Representation

A binary search tree can also be used as a functional array implementation, and this offers an immediate realization of functional graphs: a graph is represented by a pair (t, m) where t is a tree of pairs (node, (predecessors, label, successors)) and m is the largest node occurring in t. Note that m is used to support the creation of new nodes, which is possible in $O(1)$ time.

However, inserting and deleting a node context (p, v, l, s) requires considerable effort. For insertion we have to insert the context itself, which takes $O(\log n)$ steps, and we have to insert v as a successor (predecessor) for each node in p (s), which requires $O(c \log n)$ steps. Hence, insertion runs in $O(c \log n)$ time which can be as large as $O(n \log n)$ for dense graphs. Context deletion takes even more time since we have to remove v from the successor (predecessor) list for each element of p (s), which requires searching these lists for v. Altogether deletion runs in $O(c^2 \log n)$ time or $O(c \log c \log n)$ if predecessors/successors are stored as search trees. In dense graphs, this gives a complexity of $O(n^2 \log n)$ or $O(n \log^2 n)$.

Although the asymptotic behavior of the search tree representation is clearly worse (at least for single-threaded graph uses) than the array implementation, it performs very well in practice (see Erwig (1997a)), maybe because it is much simpler and does not require so much tuning. It also has the great advantage that it is a truly dynamic structure that supports unbounded growth of graphs. A further problem with the array implementation is that it is difficult to realize in Haskell. Either unsafe features have to be exploited, or operations like constant time array updates have to be encapsulated in a monad. Since such a monad has to extend as far as access to the array is made, the monad would eventually show up in the algorithms and cannot be hidden in the graph implementation. This is very bad and would completely destroy the functional flavor of the algorithms using inductive graphs. The version-tree implementation is therefore contained only in the ML version of FGL, the Haskell version currently provides only the search-tree representation.

4 Functional Graph Algorithms

We are now ready to present a collection of graph algorithms based on the inductive view of graphs. We describe a selection of algorithms that are often used in courses on algorithms and data structures. This should serve as evidence that it is possible to teach graph algorithms using functional languages.

We will, however, not provide all the additional explanations that are required for a potential graph chapter of a textbook. Rather we shall discuss the different flavor of the functional algorithms compared with the imperative versions. We also comment on complexity, and we assume that & and $\&^{v}$ are $O(1)$, even though that may not be true for some implementations. More example programs can be found in Erwig (1997b) where we have concentrated on the definition of several kinds of graph fold operations and on optimization rules for these and in Erwig (1997a) where we have used example programs to benchmark different inductive graph implementations. One of these examples is a completely functional graph reducer that is also described in more detail in Erwig (1998a).

4.1 Depth-First Search

Depth-first search is one of the most basic and most important graph algorithms. It can reveal a lot about the internal structure of a graph, and this information can be used to implement several other algorithms, such as topological sorting or computing strongly connected components.

A depth-first walk through a graph essentially means to visit each node in the graph once by visiting successors before siblings. The parameters of depth-first search are, of course, the graph to be searched and a list of nodes saying which nodes might be left to visit. This list is needed for unconnected graphs where, after having completely explored one component, a node of another component is needed to continue the search. The result of depth-first search can be, for example, the list of nodes in the order visited (this list is said to be in depth-first order) or a depth-first spanning forest, which keeps the edges that have been traversed to reach all the nodes.

We begin by giving an algorithm that yields a list of nodes in depth-first order:

$$
\begin{array}{lll}
\mathit{dfs} :: [\mathit{Node}] \rightarrow \mathit{Graph}\ a\ b \rightarrow [\mathit{Node}] & & \\
\mathit{dfs}\ [\,]\quad g & = & [\,] \\
\mathit{dfs}\ (v{:}vs)\ (c \overset{v}{\&}\ g) & = & v{:}\mathit{dfs}\ (\mathit{suc}\ c{+\!\!+}vs)\ g \\
\mathit{dfs}\ (v{:}vs)\ g & = & \mathit{dfs}\ vs\ g
\end{array}
$$

The algorithm works as follows. If there are no nodes left to be visited (first case), *dfs* stops without returning any nodes. In contrast, if there are still nodes that must be visited, *dfs* tries to locate the context of the first of these nodes (v) in the argument graph. If this is possible (second equation), which is the case whenever v is contained in the argument graph, v is the next node on the resulting node list, and the search continues on the remaining graph g with the successors of v to be visited before the remaining list of nodes vs. The fact that the successors are put in front of all other nodes causes *dfs* to favor searching in the depth and not in the breadth. Finally, if v cannot be matched (last line), *dfs* continues the search with the remaining list of nodes vs. Note that the last case can only occur if v is not contained in the graph because otherwise the pattern in the second equation would have matched.

The reader might have noticed a source of optimization in the above definition: in fact, we can immediately terminate *dfs* and return an empty node list when nodes to be visited are still left but the graph is empty. This can be achieved by either adding as a first or second equation

$$\mathit{dfs}\ vs\ \mathit{Empty} = [\,]$$

or changing the first equation to something like:

$$\mathit{dfs}\ vs\ g \mid \mathit{null}\ vs \mathbin{||} \mathit{isEmpty}\ g = [\,]$$

This does not affect the computed results, but might improve efficiency, in particular, in dense graphs, where all the expanded edges can cause up to $\Omega(n^2)$ nodes to be checked even after the graph has been already completely traversed.

It is interesting to note how the single-visit constraint of depth-first search is realized through active patterns: once a node v is visited, that is, once v has been successfully matched in the second equation, the algorithm continues with a graph that does not contain v anymore. So instead of remembering visited nodes, be it imperatively by node marks or functionally by threading a set structure, in our approach visited nodes are simply forgotten! Also consider how easy it is to realize breadth-first search: simply replace $\mathit{suc}\ c{+\!\!+}vs$ by $vs{+\!\!+}\mathit{suc}\ c$ in the second equation — the new nodes (of the deeper level) are not visited until all other nodes (of the current level) have been visited. Of course, this treatment of lists as queues is inefficient — Section 4.2 discusses how to replace lists with a more efficient implementation of queues.

As mentioned above, in its general form *dfs* needs as a parameter the list of possibly unvisited nodes. To run *dfs* on a graph g without mentioning a start node, it is sufficient to call $\mathit{dfs}\ g\ (\mathit{nodes}\ g)$. Note, however, that this works for *bfs* only if we use a list of queues as a parameter. Otherwise, we would end up in visiting all nodes in the same order as the list provided. Therefore, *bfs* defined to work on just one queue always needs a single start node as a parameter.

Once again we note that the use of active patterns is just for notational convenience, and we could implement *dfs* by using the *match* function:

$$
\begin{array}{lll}
\mathit{dfs}\ [\,]\quad\ g' & = [\,] & \\
\mathit{dfs}\ (v{:}vs)\ g' & = \textbf{case}\ \mathit{match}\ v\ g'\ \mathit{of} & \\
 & \quad (\mathit{Just}\ c, g) \ \ \rightarrow & v{:}\mathit{dfs}\ (\mathit{suc}\ c{+\!\!+}vs)\ g \\
 & \quad (\mathit{Nothing}, g) \rightarrow & \mathit{dfs}\ vs\ g
\end{array}
$$

Computing a depth-first spanning forest is slightly more complex because we have to distinguish between the relationship of a node to its successors and that to its siblings to obtain the spanning tree structure. This was not needed in *dfs* since all nodes were just put into one list, that is, successors as well as siblings were concatenated. In contrast, to compute a tree structure, the spanning trees for siblings have to be concatenated whereas otherwise a node makes up a new branch of the tree with the spanning trees of its successors as subtrees.

First, we need a definition of multi-way trees together with a postorder traversal function that visits the nodes of all subtrees before the root.

$$\textbf{data}\ \mathit{Tree}\ a\ =\ \mathit{Br}\ a\ [\mathit{Tree}\ a]$$

$$
\begin{array}{l}
\mathit{postorder} :: \mathit{Tree}\ a \rightarrow [a] \\
\mathit{postorder}\ (\mathit{Br}\ v\ ts)\ =\ \mathit{concatMap}\ \mathit{postorder}\ ts{+\!\!+}[v]
\end{array}
$$

Note that *concatMap* is a function defined in the Haskell prelude:

$$
\begin{array}{l}
\mathit{concatMap} :: (a \rightarrow [b]) \rightarrow [a] \rightarrow [b] \\
\mathit{concatMap}\ f\ =\ \mathit{concat}\ .\ \mathit{map}\ f
\end{array}
$$

Of course, the given simple implementation for *postorder* has quadratic running time, but by using an accumulating parameter or an O(1) queue data structure we can obtain linear complexity.

Now we can define a function to compute spanning forests. The function is very similar to *dfs*; it mainly differs in the second equation which applies when the next node to be visited v can be found in the graph. In that case we have to create independently two spanning forests: one forest f for all successors of v; these trees become the subtrees of the newly created tree with root v, and this tree is added to the second forest f' that is computed for the remaining nodes to be visited. To ensure that each node is used only once we have to remember the unused graph parts in a second result and thread this graph through successive function calls.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{df} :: [\mathit{Node}] \rightarrow \mathit{Graph}\ a\ b \rightarrow ([\mathit{Tree}\ \mathit{Node}], \mathit{Graph}\ a\ b)} \\
\mathit{df}\ [\,]\quad\ g & = ([\,], g) & \\
\mathit{df}\ (v{:}vs)\ (c \overset{v}{\&}\ g) & = (\mathit{Br}\ v\ f{:}f', g_2)\ \textbf{where} & (f, g_1)\ =\ \mathit{df}\ (\mathit{suc}\ c)\ g \\
 & & (f', g_2)\ =\ \mathit{df}\ vs\ g_1 \\
\mathit{df}\ (v{:}vs)\ g & = \mathit{df}\ vs\ g &
\end{array}
$$

Since the graph result from *df* is only used internally, we define an additional function *dff* that returns just the computed forest.

$$
\begin{array}{l}
\mathit{dff} :: [\mathit{Node}] \rightarrow \mathit{Graph}\ a\ b \rightarrow [\mathit{Tree}\ \mathit{Node}] \\
\mathit{dff}\ vs\ g\ =\ \mathit{fst}\ (\mathit{df}\ vs\ g)
\end{array}
$$

Being able to compute depth-first spanning forests we can now implement quite easily functions for topologically sorting a graph and computing strongly connected components (Erwig, 1992; King & Launchbury, 1995; King, 1996): a topological sorted list of nodes can be obtained by a reversed postorder list of nodes of the depth-first spanning tree, and Sharir's algorithm for computing strongly connected components works by computing a depth-first spanning forest on the reversed graph starting with a topologically sorted list of nodes.

$$\begin{array}{l} \mathit{topsort} :: \mathit{Graph}\ a\ b\ \rightarrow [\mathit{Node}] \\ \mathit{topsort}\ =\ \mathit{reverse}\ .\ \mathit{concatMap}\ \mathit{postorder}\ .\ \mathit{dff} \end{array}$$

$$\begin{array}{l} \mathit{scc} :: \mathit{Graph}\ a\ b\ \rightarrow [\mathit{Tree}\ \mathit{Node}] \\ \mathit{scc}\ g\ =\ \mathit{dff}\ (\mathit{topsort}\ g)\ (\mathit{grev}\ g) \end{array}$$

These two examples demonstrate nicely how the compositional style of functional languages can be used to give succinct descriptions of graph algorithms. In Launchbury (1995) and Erwig (1997b) it is shown how these definitions can be optimized further.

4.2 Breadth-First Search

Breadth-first search essentially means visiting siblings before successors. This has the effect of first visiting all nodes of a certain distance (measured in number of edges) from the start node before visiting nodes that are further away. This property is exploited by the shortest-path algorithm *esp* given below that is based on breadth-first search.

$$\begin{array}{lcl} \mathit{bfs} :: [\mathit{Node}] \rightarrow \mathit{Graph}\ a\ b \rightarrow [\mathit{Node}] & & \\ \mathit{bfs}\ [\,]\quad g & = & [\,] \\ \mathit{bfs}\ (v{:}\mathit{vs})\ (c\ \overset{v}{\&}\ g) & = & v{:}\mathit{bfs}\ (\mathit{vs}{+\!\!+}\mathit{suc}\ c)\ g \\ \mathit{bfs}\ (v{:}\mathit{vs})\ g & = & \mathit{bfs}\ \mathit{vs}\ g \end{array}$$

The algorithm works very much like depth-first search, except for the treatment of newly found successors, and this reflects exactly the way in which the nodes to be visited are implicitly ordered: for *dfs* these are kept in a stack, that is, newly discovered nodes are put in front of previously discovered (but not yet visited) ones, for *bfs* the nodes are kept in a queue, that is, new nodes are appended to old ones.

Using lists to realize a queue is not very efficient due to the append operation taking linear time in the size of its first argument. However, there are several queue implementations available that guarantee constant time operations either amortized over all operations (Gries, 1981; Burton, 1982) or even for a single operation (Hood & Melville, 1981; Chuang & Goldberg, 1993; Okasaki, 1995). For our purpose, an amortized constant-time queue implementation is sufficient; this keeps *bfs* a linear algorithm. Moreover, we can enhance the above definition in the same way as *dfs* or *df* by aborting the search on encountering the empty graph.

To build a breadth-first spanning tree we again have to keep more information than just the order of nodes. Before we present the algorithm we make two observations. First, it is quite difficult to efficiently build a breadth-first spanning tree represented, for example, as a *Tree Node* value as was done by *df* (see also (Okasaki, 2000)). The problem is that the expressions denoting such trees have to be built bottom-up whereas the recursion in *bfs* delivers nodes in a way that is per se suited for top-down construction. Second, such a representation is not so important any-how because one of the most important uses of a breadth-first spanning tree is to find shortest paths[3] (from the root to any other node), and this is supported by inward directed trees, that is, trees whose edges point from the successors toward predecessors: finding a shortest path from node s to node t can be achieved by (i) computing the breadth-first spanning tree rooted at s, (ii) locating node t in it, and (iii) following the edges from t to the root. Then the reverse list of traversed nodes/edges gives the shortest path.

Now an inward directed tree can be represented simply as a mapping with domain and range of type *Node* mapping nodes to their predecessors. Since such a mapping is built incrementally either during breadth-first search or after it using a list of traversed edges, we cannot use a monolithic array for implementing it. In fact, the array construct proposed in Johnsson (1998) could be used to build up such a tree, but this requires the reformulation of the whole algorithm so that it follows the array construction, and this destroys the simplicity and elegance of the functional *bfs* algorithm. Instead we can use a binary search tree, but this adds a logarithmic factor on each operation for (i) building up the spanning tree and (ii) for reconstructing the shortest path after that.

The latter problem can be addressed by not just mapping nodes to their predecessor, but to the whole path to the root, which we call the *root path* or *r-path* for short. This does not really make the implementation more complex: to insert u as a predecessor of v instead of just inserting u with key v into the tree, we first locate the root path already stored at u, say p, and then insert $u{:}p$ with key v into the tree. In this way we only need to locate t in the inward directed tree, and we can just reverse the list of stored nodes to obtain the shortest path from s to t. Note that this representation causes only minimal space overhead: since common prefixes of paths (that is, common suffixes of r-paths) are shared, this representation is linear in the number of stored nodes. However, the complexity of computing the breadth-first spanning tree and thus also for computing shortest paths is still $O(n \log n + m)$.

Now a further improvement is to represent a breadth-first spanning tree by a list (instead of a tree) of r-paths from each node to the root. We call this kind of tree a *root-path tree*. Again, to have a linear space requirement the r-paths should share common suffixes. This can be achieved quite easily by keeping the r-paths in the queue of *bfs*. Below we define the function *bft* that takes a single node v and computes the breadth-first spanning tree rooted at v as a list of r-paths by calling the function *bf* which works much like *bfs* but uses a queue of r-paths as its first argument. The expression *map* $({:}p)$ $(\mathit{suc}\ c)$ extends the root path p by v's

[3] Here the distance is measured in number of edges. A shortest-path algorithm that uses edge weights is presented in Section 4.3.

successors and thus yields a new root path for each of v's successors. These new root paths are appended to the rest of the queue ps.

$$
\begin{aligned}
&\textbf{type } \mathit{Path} \;\; = [\mathit{Node}] \\
&\textbf{type } \mathit{RTree} = [\mathit{Path}]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{bft} :: \mathit{Node} \to \mathit{Graph}\ a\ b \to \mathit{RTree} \\
&\mathit{bft}\ v \;=\; \mathit{bf}\ [[v]]
\end{aligned}
$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathit{bf} :: [\mathit{Path}] \to \mathit{Graph}\ a\ b \to \mathit{RTree}} \\
\mathit{bf}\ [\,] & g & = & [\,] \\
\mathit{bf}\ (p@(v{:}\_){:}ps) & (c \overset{v}{\&} g) & = & p{:}\mathit{bf}\ (ps{+\!\!+}\mathit{map}\ ({:}p)\ (\mathit{suc}\ c))\ g \\
\mathit{bf}\ (p{:}ps) & g & = & \mathit{bf}\ ps\ g
\end{array}
$$

Again the optimizations for termination on empty graphs and for using a more efficient queue implementation apply.

Now the algorithm for finding the shortest path between two nodes s and t first computes the breadth-first spanning tree rooted at s. This spanning tree is represented as a list of root paths, and from these the first one that has t as a first element is extracted. This root path has then only to be reversed to obtain the shortest path.

$$
\begin{aligned}
&\mathit{first} :: (a \to \mathit{Bool}) \to [a] \to a \\
&\mathit{first}\ p \;=\; \mathit{head}\ .\ \mathit{filter}\ p
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{esp} :: \mathit{Node} \to \mathit{Node} \to \mathit{Graph}\ a\ b \to \mathit{Path} \\
&\mathit{esp}\ s\ t \;=\; \mathit{reverse}\ .\ \mathit{first}\ (\backslash(v{:}\_) \to v == t)\ .\ \mathit{bft}\ s
\end{aligned}
$$

This breadth-first search algorithm is much simpler than the one given by King (King, 1996), which uses two rather subtle functional programming tricks. Moreover, the computation of shortest paths is efficient, and *esp* has linear running time (assuming the usual optimization and $O(1)$ graph operations). This is not the case for King's algorithm that has to perform an uninformed search for the target node t. The reason is that King has chosen the same kind of "top-down" tree as for depth-first search.

4.3 Shortest Paths

Very closely related to the shortest-path algorithm of the preceding section is Dijkstra's algorithm for computing shortest paths in graphs with positive edge labels. The main difference is that the length of a path is now defined to be the sum of its edge labels and that a shortest path between two nodes is accordingly one that has a minimum path length. Dijkstra's algorithm can be described similar to *bfs*. The only difference is that root paths are not kept in a queue but in a heap that is ordered with regard to the lengths of the paths.

We first define a type for labeled nodes and labeled paths. A labeled root path is a list of labeled nodes representing a path from a node to the root, and each node v is paired with the length of the path to the root. Thus, the label of the first node of the r-path gives the length of the complete path (and the length of an r-path's last node, which is always the root, is always 0). As we have represented breadth-first spanning trees with lists of plain r-paths, we now represent shortest-path trees with lists of labeled r-paths. We define labeled paths as instances of the *Ord* and *Eq* classes so that we can store them in heaps. The function *getPath* extracts a path to a specific node from a labeled root tree and drops all labels.

$$
\begin{aligned}
&\textbf{type } \mathit{LNode}\ a\ \ = (\mathit{Node}, a)\\
&\textbf{type } \mathit{LPath}\ a\ \ = [\mathit{LNode}\ a]\\
&\textbf{type } \mathit{LRTree}\ a = [\mathit{LPath}\ a]\\[1ex]
&\textbf{instance } \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{LPath}\ a)\ \textbf{where}\\
&\quad ((\_,x){:}\_) == ((\_,y){:}\_) \ = \ x == y\\[1ex]
&\textbf{instance } \mathit{Ord}\ a \Rightarrow \mathit{Ord}\ (\mathit{LPath}\ a)\ \textbf{where}\\
&\quad ((\_,x){:}\_) \ < \ ((\_,y){:}\_) \ = \ x < y\\[1ex]
&\mathit{getPath} :: \mathit{Node} \ \rightarrow \ \mathit{LRTree}\ a \ \rightarrow \ \mathit{Path}\\
&\mathit{getPath}\ v \ = \ \mathit{reverse}\ .\ \mathit{map}\ \mathit{fst}\ .\ \mathit{first}\ (\backslash((w,\_){:}\_) \rightarrow w == v)
\end{aligned}
$$

The labeled root paths stored in a heap represent the currently expanded shortest-path tree. In particular, all their first nodes represent the fringe of the search, and the labels give the tentative costs for these nodes. All other nodes of the r-paths are nodes already known to belong to the shortest path tree. Now Dijkstra's algorithm works by repeatedly selecting the r-path p with the least labeled first node, say v. (This corresponds to making v permanent.) Then new r-paths are created by putting v's successors in front of p using as labels the label of v plus the cost associated with the edge leading to the successor. This is done by the function *expand*. Storing different root paths for one node in the heap presents no problem (with regard to correctness) since only the one with the least cost is selected. Should a node be selected a second time, it will simply be ignored because matching in the graph will fail. However, since the heap now contains up to $O(m)$ entries, deleting a minimum from the heap will be performed up to $O(m)$ times causing a running time of $O(m \log m)$ ($= O(m \log n)$). Altogether, this version of Dijkstra's algorithm runs in a worst case time of $O(m + m \log n)$ which asymptotically worse, at least for non-sparse graphs, than the imperative algorithm that runs in $O(m + n \log n)$.

We use a pairing heap implementation as described in Okasaki (1998). The operation *unitHeap* wraps a single value into a heap, and the operation *mergeAll* combines a list of heaps into a single heap. The operation *splitMin* applied to a non-empty heap returns a pair containing its minimum and the remaining heap without the minimum. (Even though, strictly speaking, we do not need a mergeable heap, we have chosen pairing heaps because they are reported to be very efficient in practice and the *mergeAll* operation is very convenient in describing the node expansion.)

Again to have a concise notation, we use an active pattern $x{\prec}h$ that is based on *splitMin* and matches any non-empty heap h'; it binds the minimum of h' to x and the heap without x to h.

$$
\begin{array}{l}
\mathit{expand} :: \mathit{Real}\ b \Rightarrow b\ \rightarrow\ \mathit{LPath}\ b\ \rightarrow\ \mathit{Context}\ a\ b\ \rightarrow\ [\mathit{Heap}\ (\mathit{LPath}\ b)]\\
\mathit{expand}\ d\ p\ (\_,\_,\_,s)\ =\ \mathit{map}\ (\backslash(l,v) \rightarrow \mathit{unitHeap}\ ((v,l+d){:}p))\ s
\end{array}
$$

$$
\begin{array}{ll}
\mathit{dijkstra} :: \mathit{Real}\ b \Rightarrow \mathit{Heap}\ (\mathit{LPath}\ b)\ \rightarrow\ \mathit{Graph}\ a\ b\ \rightarrow\ \mathit{LRTree}\ b & \\
\mathit{dijkstra}\ h\ g \mid \mathit{isEmptyHeap}\ h\ ||\ \mathit{isEmpty}\ g\ =\ [\,] & \\
\mathit{dijkstra}\ (p@((v,d){:}\_){\prec}h)\ (c\ \overset{v}{\&}\ g) & =\ p{:}\mathit{dijkstra}\ (\mathit{mergeAll}\ (h{:}\mathit{expand}\ d\ p\ c))\ g\\
\mathit{dijkstra}\ (\_{\prec}h)\ g & =\ \mathit{dijkstra}\ h\ g
\end{array}
$$

Note that *Real* is a subclass of *Num* containing all standard numeric types that are also a subclass of *Ord*; *Real* contains all standard numeric types except *Complex*.

Next we define an additional function *spt*, which encapsulates the construction of the initial heap, and a function *sp* for computing shortest paths.

$$
\begin{array}{l}
\mathit{spt} :: \mathit{Real}\ b \Rightarrow \mathit{Node}\ \rightarrow\ \mathit{Graph}\ a\ b\ \rightarrow\ \mathit{LRTree}\ b\\
\mathit{spt}\ v\ =\ \mathit{spt}\ (\mathit{unitHeap}\ [(v,0)])
\end{array}
$$

$$
\begin{array}{l}
\mathit{sp} :: \mathit{Real}\ b \Rightarrow \mathit{Node}\ \rightarrow\ \mathit{Node}\ \rightarrow\ \mathit{Graph}\ a\ b\ \rightarrow\ \mathit{Path}\\
\mathit{sp}\ s\ t\ =\ \mathit{getPath}\ t\ .\ \mathit{spt}\ s
\end{array}
$$

4.4 Minimum Spanning Tree

A minimum spanning tree of a labeled undirected graph is a spanning tree of minimal total edge length. Hence, in our context of directed graphs the algorithms described below work, in general, only for directed graphs that properly represent undirected graphs. We can easily convert any directed graph into one representing an undirected one with the function *undir* described in Section 3.2.

The two most popular minimum spanning tree algorithms are those of Kruskal and Prim. Kruskal's algorithm works by repeatedly taking edges in order of increasing edge length as long as they do not form a cycle. The graph is used only to get the list of edges, and the crucial part of the algorithm is an efficient implementation of a union/find data structure to enable fast cycle detection.

In contrast, Prim's algorithm performs a usual walk through the graph. It is a greedy algorithm, like Dijkstra's algorithm, which means that in each step one new part of the result is computed. Prim's algorithm keeps a heap of edges that start from the currently computed part of the minimum spanning tree, and selects in each step the smallest of these edges and extends the fringe around the tree by those edges that start from the selected edge's target node.

Before we can define Prim's algorithm we have to decide about the representation of the spanning tree, and this decision depends on the context in which the spanning tree is used. One application can be found in telecommunication: some telephone companies calculate the costs of phone calls by the length of a path between two nodes in a precomputed minimum spanning tree. This is supported again by labeled root path trees.

We first need a function for creating new root paths for the successors of the node expanded last. This function is quite similar to *expand*, but it has to consider only the edge costs instead of the costs of the complete root paths.

```
addEdges :: Real b ⇒ LPath b → Context a b → [Heap (LPath b)]
addEdges p (_, _, _, s) = map (\(l, v) → unitHeap ((v, l):p)) s
```

Now we can define Prim's minimum spanning tree algorithm. We parameterize the function *mst* also by a *Node*-value to provide some flexibility for specifying the root of the spanning tree. One can easily define a function that does not need a root by using the operation *matchAny*.

```
mst :: Real b ⇒ Node → Graph a b → LRTree b
mst v g = prim (unitHeap [(v, 0)]) g

prim :: Real b ⇒ Heap (LPath b) → Graph a b → LRTree b
prim h g | isEmptyHeap h || isEmpty g = [ ]
prim (p@((v, _):_)≺h) (c &^v g) = p:prim (mergeAll (h:addEdges p c)) g
prim (_≺h) g                     = prim h g
```

The striking similarity to Dijkstra's algorithm has been known for a long time and becomes very clear in the presented programming approach. In fact, all algorithms considered so far follow the same basic traversal scheme and differ essentially in the data structure that is used to control the traversal. This fact can be exploited in teaching graph algorithms by presenting the different algorithms as instances of this scheme, which has already been emphasized in Erwig (1992) and in Erwig (2000).

Going back to Prim's algorithm, it remains to be shown how paths can be found in a minimum spanning tree represented by a root path tree. The idea is quite simple: first, select the root paths for the two nodes, and then join their non-common parts at their least common ancestor. This path reconstruction is realized by the functions *joinPaths* and *joinAt*.

```
mstp :: Real b ⇒ LRTree b → Node → Node → Path
mstp t a b = joinPaths (getPath a t) (getPath b t)

joinPaths :: Path → Path → Path
joinPaths p q = joinAt (head p) (tail p) (tail q)

joinAt :: Node → Path → Path → Path
joinAt x (v:vs) (w:ws) | v == w = joinAt v vs ws
joinAt x p      q               = reverse p++(x:q)
```

All the algorithms described so far use the graph in a single-threaded way. Since our data type of graphs is persistent, we shall consider at least one application that uses graphs persistently, that is, different versions of a graph are employed at the same time. This is described next.

4.5 Maximum Independent Node Sets

An independent node set is a subset of the nodes of a graph such that no two nodes of this set are connected by an edge. A maximum independent node set is an independent node set of maximum cardinality. The problem of finding a maximum independent node set is in a sense the dual of the maximum clique problem which asks for a maximal set of nodes such that each pair of nodes is connected by an edge. Both problems are NP-hard. Hence there is little chance that there exist efficient algorithms for solving them. Nevertheless, there are algorithms that are much better than blindly trying all possible node subsets.

The algorithm defined below works by recursively comparing two alternatives:

1. the maximum independent node set of a graph g from which the node v with maximum degree is removed, and
2. the maximum independent node set of g from which the neighbors of v have been removed, extended by the node v itself.

Then the larger of the two sets is the maximum independent node set of g.

$$
\begin{array}{lll}
\mathit{indep} :: \mathit{Graph}\ a\ b \rightarrow [\mathit{Node}] & & \\
\mathit{indep}\ \mathit{Empty} & = & [\,] \\
\mathit{indep}\ g & = & \mathbf{if}\ \mathit{length}\ i_1 > \mathit{length}\ i_2\ \mathbf{then}\ i_1\ \mathbf{else}\ i_2 \\
& & \quad \begin{array}{lll}
\mathbf{where}\ \mathit{vs} & = & \mathit{nodes}\ g \\
\phantom{\mathbf{where}}\ m & = & \mathit{maximum}\ (\mathit{map}\ (\mathit{flip}\ \mathit{deg}\ g)\ \mathit{vs}) \\
\phantom{\mathbf{where}}\ v & = & \mathit{first}\ (\backslash v \rightarrow \mathit{deg}\ v\ g == m)\ \mathit{vs} \\
\phantom{\mathbf{where}}\ c \overset{v}{\&} g' & = & g \\
\phantom{\mathbf{where}}\ i_1 & = & \mathit{indep}\ g' \\
\phantom{\mathbf{where}}\ i_2 & = & v{:}\mathit{indep}\ (\mathit{foldr}\ \mathit{del}\ g'\ (\mathit{pre}\ c {+\!\!+} \mathit{suc}\ c))
\end{array}
\end{array}
$$

Note that *pre* is defined analogously to *suc*.

5 Conclusions

We have proposed an inductive definition of graphs that encourages the definition of graph algorithm as recursive functions. We hope that this functional style of writing graph algorithms eventually finds its way into teaching graph algorithms. Active patterns make the function definitions more succinct, but all functions can be easily rewritten without using them. We have also described an efficient implementation of inductive graphs, which shows that the alternative algorithmic style gives both efficiency and clarity.

Acknowledgments

The author thanks Chris Okasaki and the anonymous reviewers for their helpful comments.

References

Aasa, A., Holström, S. and Nilsson, C. (1988) An Efficiency Comparison of Some Representations of Purely Functional Arrays. *BIT* **28**(3):490–503.

Barendsen, E. and Smetsers, S. (1996) Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science* **6**:579–612.

Bird, R. S. (1998) *Introduction to Functional Programming Using Haskell.* Prentice-Hall International.

Briggs, P. and Torczon, L. (1993) An Efficient Representation For Sparse Sets. *ACM Letters on Programming Languages* **2**(4):59–69.

Burton, F. W. (1982) An Efficient Functional Implementation of FIFO Queues. *Information Processing Letters* **14**(5):205–206.

Burton, F. W. and Cameron, R. D. (1993) Pattern Matching with Abstract Data Types. *Journal of Functional Programming* **3**(2):171–190.

Burton, F. W. and Yang, H.-K. (1990) Manipulating Multilinked Data Structures in a Pure Functional Language. *Software – Practice and Experience* **20**(11):1167–1185.

Chuang, T.-R. and Goldberg, B. (1993) Real-Time Deques, Multihead Turing Machines, and Purely Functional Programming. *Conf. on Functional Programming and Computer Architecture* pp. 289–293.

Dietz, P. F. (1989) Fully Persistent Arrays. *Workshop on Algorithms and Data Structures.* LNCS 382, pp. 67–74.

Erwig, M. (1992) Graph Algorithms = Iteration + Data Structures? The Structure of Graph Algorithms and a Corresponding Style of Programming. *18th Int. Workshop on Graph-Theoretic Concepts in Computer Science.* LNCS 657, pp. 277–292.

Erwig, M. (1996) Active Patterns. *8th Int. Workshop on Implementation of Functional Languages.* LNCS 1268, pp. 21–40.

Erwig, M. (1997a) Fully Persistent Graphs – Which One to Choose? *9th Int. Workshop on Implementation of Functional Languages.* LNCS 1467, pp. 123–140.

Erwig, M. (1997b) Functional Programming with Graphs. *2nd ACM Int. Conf. on Functional Programming* pp. 52–65.

Erwig, M. (1998a) *A Functional Homage to Graph Reduction.* Technical Report 239. FernUniversität Hagen.

Erwig, M. (1998b) Abstract Syntax and Semantics of Visual Languages. *Journal of Visual Languages and Computing* **9**(5):461–483.

Erwig, M. (2000) Random Access to Abstract Data Types. *8th Int. Conf. on Algebraic Methodology and Software Technology.* LNCS 1816, pp. 135–149.

Erwig, M. and Peyton Jones, S. L. (2000) Pattern Guards and Transformational Patterns. *Haskell Workshop.*

Fegaras, L. and Sheard, T. (1996) Revisiting Catamorphisms over Datatypes with Embedded Functions. *23rd ACM Symp. on Principles of Programming Languages* pp. 284–294.

Gibbons, J. (1995) An Initial Algebra Approach to Directed Acyclic Graphs. *Mathematics of Program Construction.* LNCS 947, pp. 282–303.

Gries, D. (1981) *The Science of Programming.* Springer-Verlag.

Harrison, R. (1989) *Abstract Data Types in Standard ML.* John Wiley & Sons.

Hood, R. and Melville, R. (1981) Real-Time Queue Operations in Pure List. *Information Processing Letters* **13**(2):50–53.

Hudak, P. and Bloss, A. (1985) The Aggregate Update Problem in Functional Programming Systems. *12th ACM Symp. on Principles of Programming Languages* pp. 300–313.

Johnsson, T. (1998) Efficient Graph Algorithms Using Lazy Monolithic Arrays. *Journal of Functional Programming* **8**(4):323–333.

Kashiwagi, Y. and Wise, D. (1991) Graph Algorithms in a Lazy Functional Programming Language. *4th Int. Symp. on Lucid and Intensional Programing* pp. 35–46.

King, D. J. (1996) *Functional Programming and Graph Algorithms.* PhD thesis, University of Glasgow.

King, D. J. and Launchbury, J. (1995) Structuring Depth-First Search Algorithms in Haskell. *22nd ACM Symp. on Principles of Programming Languages* pp. 344–354.

Launchbury, J. (1995) Graph Algorithms with a Functional Flavour. *Advanced Functional Programing*. LNCS 925, pp. 308–331.

Okasaki, C. (1995) Simple and Efficient Purely Functional Queues and Deques. *Journal of Functional Programming* **5**(4):583–592.

Okasaki, C. (1998) *Purely Functional Data Structures.* Cambridge University Press.

Okasaki, C. (2000) Breadth-First Numbering: Lessons from a Small Exercies in Algorithm Design. *5th ACM Int. Conf. on Functional Programming* pp. 131–136.

O'Neill, M. E. and Burton, F. W. (1997) A New Method for Functional Arrays. *Journal of Functional Programming* **7**(5):487–513.

Palao Gostanza, P., Peña, R. and Núñez, M. (1996) A New Look at Pattern Matching in Abstract Data Types. *1st ACM Int. Conf. on Functional Programming* pp. 110–121.

Paulson, L. C. (1996) *ML for the Working Programmer (2nd ed.).* Cambridge University Press.

Rabhi, F. and Lapalme, G. (1999) *Algorithms: A Functional Programming Approach.* Addison-Wesley.

Reade, C. (1989) *Elements of Functional Programming.* Addison-Wesley.

Sastry, A., W., C. and Z., A. (1993) Order-of-Evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates. *Conf. on Functional Programming and Computer Architecture.*

Ullman, J. D. (1998) *Elements of ML Programming (2nd ed.).* Prentice-Hall International.

Wadler, P. (1987) Views: A Way for Pattern Matching to Cohabit with Data Abstraction. *14th ACM Symp. on Principles of Programming Languages* pp. 307–313.