Functional Programming with Graphs

Martin Erwig
FernUniversität Hagen, Praktische Informatik IV
58084 Hagen, Germany
erwig@fernuni-hagen.de

Abstract

Graph algorithms expressed in functional languages often suffer from their inherited imperative, state-based style. In particular, this impedes formal program manipulation. We show how to model persistent graphs in functional languages by graph constructors. This provides a decompositional view of graphs which is very close to that of data types and leads to a "more functional" formulation of graph algorithms. Graph constructors enable the definition of general fold operations for graphs. We present a promotion theorem for one of these folds that allows program fusion and the elimination of intermediate results. Fusion is not restricted to the elimination of tree-like structures, and we prove another theorem that facilitates the elimination of intermediate graphs. We describe an ML-implementation of persistent graphs which efficiently supports the presented fold operators. For example, depth-first-search expressed by a fold over a functional graph has the same complexity as the corresponding imperative algorithm.

1 Introduction

Traditionally, most graph algorithms are formulated in an imperative manner, for example, in depth-first search, nodes are *marked* as being visited to prevent repetitive traversal. Most often, this imperative style is carried over when implementing graph algorithms in functional languages, for example, a set of visited nodes is threaded through successive function calls. Although this strategy can keep programs free of imperative updates, the state-based, imperative algorithm is still present, just in functional disguise. However, in order to find program transformations like in the unfold/fold approach or in the Bird/Meertens formalism, an integration in a truly functional style is needed. We follow Richard Bird who recently concluded [3]:

But if we remain within a functional formalism, then we need to reformulate standard algorithms [...]

The treatment of graphs in functional languages has now been addressed in quite different ways [4, 13, 7, 14], but there is no accepted "standard" yet. We believe that one reason for this situation is that the integration often suffers from the inherited imperative style.

This has to be seen in contrast to tree-like structures that can be directly represented by data types which present themselves rather uniformly across different functional languages: the notions of data type, constructor, pattern and pattern matching are well-known and they are present in almost all modern functional languages. Research on generalized fold operations, also called catamorphisms, [18, 22, 10] has (among other things) produced far-reaching opportunities for program transformations. In particular, the fusion of multi-pass algorithms [17, 24, 16, 12] is a profitable optimization technique.

Now it is challenging to reach a comparable status for graphs and graph algorithms. At this point one might object that graphs are application-specific structures rather than a programming language concept, and they should therefore be implemented by means of language features already present. Even if this is true, it is nonetheless important to have a uniform comprehension of graphs together with a corresponding programming style to facilitate program transformation and optimization as it is known from data types.

This paper suggests a (de)compositional view of graphs which is very close to that of data types. This gives a new flavor of defining graph algorithms and clears the way for defining general fold operations on graphs. We show how to define graph algorithms in terms of graph folds and how this facilitates program transformations and optimizations. An integration of the proposed concept into a functional language requires as its backbone an implementation of functional, or persistent, graphs. We have implemented functional graphs together with graph folds in ML, providing efficient implementations for graph operations like depth-first-search.

There are two ways to achieve such an integration: First, as a language extension. This allows the optimizations described in Section 6 to be used by a compiler, and it provides a convenient way of pattern matching (Section 4) to the user. On the other hand, it is not reasonable to expect a compiler to provide all the different graph representation that are needed to efficiently deal with graphs in specific situations. Second, by providing a graph library. With this approach it is much easier to provide (and extend) different graph implementations. We follow the latter approach since it seems to be more promising, but for the convenient presentation of examples in this paper we assume having the pattern matching capability available.

	[4]	[13]	[7]	[14]	[10]	This paper
generality	yes	yes	no	(yes)	no	yes
efficiency	no	no	yes	yes	no	yes
pureness	yes	yes	yes	no	yes	yes
clearness	no	(yes)	yes	yes	yes	yes
reasoning	no	no	no	(yes)	yes	yes

Figure 1: Treatment of graphs in functional languages.

2 Related Work

In [4] the state used by graph algorithms is simulated by functional arrays that are threaded through function calls. It is shown how to directly transfer classical algorithms into a lazy functional language, but no particular use of functional languages is made in the design of the algorithms themselves.

In contrast, in [13] algorithms are described as fixed points of recursive equations which essentially relies on lazy evaluation. Though being "more functional", the algorithms become quite complex and are rather difficult to comprehend. Both approaches do not achieve the asymptotic runtime of imperative algorithms.

In [7] we have identified some classes of graph algorithms and have introduced a few corresponding predefined operators. A graph algorithm is realized by simply providing an operator with some parameter functions and data structures. We believe that the approach reflects the structure of graph algorithms very well. However, like the previous two approaches there is not much potential for formal program manipulation. Moreover, the operator approach lacks generality.

In the proposal of [14] 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 (dfs) becomes possible. Moreover, Launchbury shows in [15] how phase fusion can be applied to eliminate intermediate results of some of these algorithms. The dfs function itself is realized nicely in a generate-and-prune manner. Monads are used to implement the state maintained during dfs (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. As yet the approach applies just to dfs.

Fegaras and Sheard investigate in [10] 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 lacks efficiency since direct access to a node requires, in general, traversal of the whole graph.

Also related is the work of Gibbons [11] 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, so that his approach is currently not usable. A summary of the preceding comparison is shown in Figure 1.

In the next section we define graph constructors and show their use in building directed graphs. In Section 4 we describe a special kind of pattern matching for graphs. Section 5 presents some fold operations. Two theorems are given in Section 6 to demonstrate the optimization of graph algorithms by simple program transformations. The implementation is described in Section 7, and some conclusions follow in Section 8.

3 A Model of Directed Graphs

We propose a (de)compositional view of graphs in the style of algebraic data types found in languages like ML or Haskell: a graph is either empty, or it is constructed by a graph G and a new node v together with edges from v to its successors in G and edges from its predecessors in G leading to v. This view is closely related to an adjacency representation of graphs. The main difference to data types is that predecessors are mentioned explicitly. We present our ideas in terms of ML, but a translation to other languages is not difficult.

3.1 Graph Constructors

There are quite different kinds of graphs, and it is almost impossible to capture all aspects in a single type. Therefore we focus in the following on directed, node-labeled multi-graphs. This, on the one hand, includes some non-trivial aspects, such as multiple edges between two nodes, and, on the other hand leaves out other details, for example, edge labels, that would only make examples longer and more difficult to read. Adaption to other graphs types is straightforward. The constructive view of graphs suggests the following two constructors:

```
Empty: 'a graph
&: 'a context * 'a graph -> 'a graph
```

The type parameter `'a` gives the type of node labels. Distinguishing between nodes and node labels is necessary whenever different nodes may have the same label, see Figure 2. (This example also shows the need for multiple edges between two nodes.)

The context of a node is the node itself together with its label and the lists of its predecessors (first component) and its successors (last component):

```
type 'a context =
     node list * node * 'a * node list
```

The requirements on the type `node` are given by the signature below. In particular, we have to create node values before we can build a graph. This is done by the **gen** function that generates any requested number of different nodes.

```
sig
  eqtype node
  val gen : int -> node list
  val new : node list -> node
end
```

In the subsequent examples we will use the following nodes:

```
val [A,B,C,D] = gen 4
```

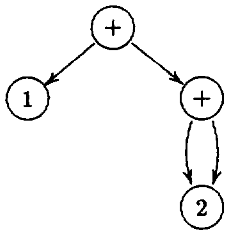
The function **new** creates a node that is not contained in a given list of nodes. This function is useful when extending a graph whose construction history is not known since we need a node value not already contained in the graph. However, to apply **new** we have to extract the nodes of a graph. This is done by the function **nodes**:

```
val nodes: 'a graph -> node list
```

As an example, we construct a cycle of three nodes and extend it by a node with an edge to some node of the cycle.

```
val cyc = ([C],A,1,[B]) & ([],B,2,[C]) &
          ([],C,3,[]) & Empty
let val N = nodes cyc
 in ([],new N,4,[hd N]) & cyc end
```

The DAG for the expression $1 + (2 + 2)$ and a graph expression for it are shown in Figure 2.



```
datatype int_expr = CON of int
                  | OP of int * int -> int

val exprDag =
    ([],D,OP (op +),[B,C]) &
    ([],C,CON 1,[]) & ([],B,OP (op +),[A,A]) &
    ([],A,CON 2,[]) & Empty
```

Figure 2: A DAG and its graph expression.

3.2 Semantics of Graph Constructors

A graph $G = (V, E, \nu)$ of type α consists of a set of nodes V, a multiset (or, bag) of edges $E \in \mathcal{M}(V \times V)$, and a total mapping $\nu : V \longrightarrow \alpha$ defining the node labels. Representing the edges of a graph as a bag of node pairs accounts for multiple edges between two nodes.

The semantics of the above graph constructors with respect to this graph model is given by inference rules as used in the definition of Standard ML [19], see Figure 3: an assertion $\rho \vdash \mathtt{e} \Rightarrow v$ says that expression e evaluates in the environment ρ to the value v. For simplicity, we assume having bags as semantic values, and we denote a bag by writing a sequence of its elements, that is, $\langle x_1, \ldots, x_n \rangle$ or $\overline{x_n}$ for short, disregarding the order of elements. The union of two bags is written like the concatenation of two lists L and L' by $L \cdot L'$.

The last two rules describe exceptional situations: trying to add a context for an already existing node results in a Node exception. Likewise, trying to add an edge between non-existing nodes raises an Edge exception. (Note that adding a tuple (v, w) to the edge bag of a graph means not only that w is a successor of v, but also that v is a predecessor of w.)

It is not difficult to define two functions **addnode** and **addedge** for adding a single node (without predecessors and successors) and a single edge (between two nodes that are known to be already contained in the graph):

```
fun addnode (v,l) g = ([],v,l,[]) & g
fun addedge (v,w) ((p,u,l,s) & g) =
    if u=v then (p,u,l,w::s) & g
           else (p,u,l,s) & (addedge (v,w) g)
```

These can be used to build any graph by first inserting all nodes and after that inserting all edges. Hence we know:

Theorem 1 (Completeness)
Any node-labeled multi-graph can be represented by a graph expression. □

4 Pattern Matching on Graphs

Regarding the free term algebra generated by **Empty** and **&**, pattern matching is the same as with other data types. For example, we can define a function **gmap** for mapping a function to all node labels of a graph:

```
fun gmap f Empty = Empty
  | gmap f ((p,v,l,s) & g) =
           (p,v,f l,s) & (gmap f g)
```

The semantics of graphs, however, suggests that some distinct graphs should be regarded as equal. In particular, many function definitions become more convenient when a kind of pattern matching could be used that abstracts away the order of node/edge insertions. Consider, for example, functions **suc** and **del** for selecting the successors of node v in a graph g, respectively, for deleting v from g. Function definitions get remarkably simple when node v is inserted last into g, that is, g = (p,v,l,s) & g': then we can simply return s, respectively, g' as result. It is an immediate corollary of Theorem 1 that we can always reorganize g to obtain the term above, since for any graph g we can always find a term for a graph g' with a specific node v (and incident edges) removed, so that g can be obtained by inserting v together with its incident edges into g'.

Miranda *laws* and the *views* mechanism proposed by Wadler [23] allow pattern matching on non-free algebraic data types by mapping data type terms to canonical representations. For our purposes instead of mapping to a canonical representation we rather need to select a specific representation (among several equivalent) via a pattern, namely one with a certain node inserted last. With the

$$\frac{}{\rho \vdash \texttt{Empty} \Rightarrow (\emptyset, \langle\rangle, \emptyset)}$$

$$\frac{\begin{array}{c}\rho \vdash \texttt{g} \Rightarrow (V, E, \nu) \qquad \rho \vdash \texttt{p} \Rightarrow \overline{p_n} \qquad \rho \vdash \texttt{v} \Rightarrow v \qquad \rho \vdash \texttt{l} \Rightarrow l \qquad \rho \vdash \texttt{s} \Rightarrow \overline{s_m} \\ v \notin V \qquad \bigcup_{i=1}^{n} \{p_i\} \subseteq (V \cup \{v\}) \qquad \bigcup_{i=1}^{m} \{s_i\} \subseteq (V \cup \{v\})\end{array}}{\rho \vdash \texttt{(p,v,l,s) \& g} \Rightarrow (V \cup \{v\}, E \cdot \langle (p_1, v), \ldots, (p_n, v), (v, s_1), \ldots, (v, s_m)\rangle, \nu \cup \{(v, l)\})}$$

$$\frac{\rho \vdash \texttt{g} \Rightarrow (V, E, \nu) \qquad \rho \vdash \texttt{v} \Rightarrow v \qquad v \in V}{\rho \vdash \texttt{(p,v,l,s) \& g} \Rightarrow [\texttt{Node}]}$$

$$\frac{\rho \vdash \texttt{g} \Rightarrow (V, E, \nu) \qquad \rho \vdash \texttt{p} \Rightarrow \overline{p_n} \qquad \rho \vdash \texttt{v} \Rightarrow v \qquad (\bigcup_{i=1}^{n} \{p_i\} \cup \bigcup_{i=1}^{m} \{s_i\}) \not\subseteq (V \cup \{v\})}{\rho \vdash \texttt{(p,v,l,s) \& g} \Rightarrow [\texttt{Edge}]}$$

Figure 3: Semantics of graph constructors.

above graph semantics we can easily define such a pattern as a language primitive. We write an environment mapping variables $\overline{x_n}$ to values $\overline{v_n}$ as $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$, and an assertion $\rho, v \vdash \texttt{p} \Rightarrow \rho'$ says that pattern p matched against value v in the environment ρ results in the binding(s) (that is, variable environment) ρ' [19]. Let $G = (V, E \cdot \langle (v, s_1), \ldots, (v, s_n), (p_1, v), \ldots, (p_m, v)\rangle, \nu \cup \{(v, l)\})$. Then:

$$\frac{\rho \vdash \texttt{v} \Rightarrow v}{\begin{array}{l}\rho, G \vdash \texttt{(p,v,l,s)} \ \underline{\&}\ \texttt{g} \Rightarrow \\ \qquad \{\texttt{s} \mapsto \overline{s_n}, \texttt{p} \mapsto \overline{p_m}, \texttt{l} \mapsto l, \texttt{g} \mapsto (V - \{v\}, E, \nu)\}\end{array}}$$

This rule says that the unbound variables p, l, and s of the pattern are bound to the corresponding values of v's context. If $v \notin V$, a special semantic object FAIL is returned [19]. FAIL is not a value, its only purpose is to direct pattern matching to the next case. If the last case returns FAIL, a Match exception is raised. The notation in the rule assumes that n and m are chosen maximally, that is, all edges incident to v are selected. Moreover, writing successors preceding predecessors in the above edge list matches self loops as successor-based, that is, matching the pattern (p,A,l,s) & g to the expression ([A],A,1,[]) & Empty binds node A in list s even though it was placed in the predecessor list. One could think of binding A in both lists, s and p, but this would contradict the intuition that the rule (p,A,l,s) $\underline{\&}$ g => (p,A,l,s) & g denotes the identity on graphs. To see this consider the application to a graph consisting of just one node with an edge to itself. If A were bound in p and s, the result expression would add two self loops to A.

As another example, consider matching ((p,B,l,s) $\underline{\&}$ g) to the expression exprDag. We get $\texttt{l} = \texttt{op +}$, $\texttt{p} = \texttt{[D]}$, $\texttt{s} = \texttt{[A,A]}$, and $\texttt{g} =$ $\begin{tikzcd} \textcircled{1} & \textcircled{+} \arrow[l] & \textcircled{2} \end{tikzcd}$. Thus we can define suc and del by:

```
fun suc v ((p,v,l,s) & g) = s
fun del v ((p,v,l,s) & g) = g
```

Since the described pattern matching process not only computes bindings, but also performs an implicit reorganization of the matched value, we call $\underline{\&}$ an *active pattern*. Note that this is not possible with laws/views (since computation is guided by external values, that is, from the outside of the pattern); in [21] a similar feature is described for $n + k$-patterns in Haskell.

The use of active patterns is actually not restricted to graphs, and it is an interesting language concept in its own right with many subtleties (the reader might have noticed the non-linear patterns), for further details see [8]. We do not require $\underline{\&}$ as a language extension, instead we can always replace a function definition

```
fun f ... v ... ((p,v,l,s) & g) = e
```

by using a predefined operation context for computing a node's context:

```
fun f ... v ... g' =
    let val ((p,v,l,s) & g) = context (v,g')
     in e end
```

The translation of active patterns in the general case when a function has more than one rule can be found in [8].

In the sequel, however, we keep using the active pattern $\underline{\&}$ for syntactic convenience.

5 Graph Folding

Whereas for data types the fold operation has a canonical form, reducing graphs can be done in quite different ways.

5.1 Unordered Fold

A first approach is to define graph folding in strong analogy to data types, that is, given a binary function f : 'a context * 'b -> 'b, unordered fold is defined:

```
fun ufold f u Empty   = u
 | ufold f u (c & g) = f (c,ufold f u g)
```

Note that we do not use the active pattern $\underline{\&}$. We can employ ufold to implement some basic functions, such as reversing edges, the function gmap from above, or testing node membership:

```
val grev =
    ufold (fn ((p,v,l,s),g)=>(s,v,l,p) & g) Empty
fun gmap f =
   ufold (fn ((p,v,l,s),g)=>(p,v,f l,s) & g) Empty
fun gmember v =
    ufold (fn ((_,w,_,_),b)=>v=w orelse b) false
```

However, the scope of ufold is somewhat limited. This is mainly because we have no control about the order of graph decomposition, but this actually seems to be of high importance to many graph algorithms (already indicated by their name: *depth-first*, *breadth-first*, *best-first*, and so on).

5.2 Linear Graph Fold

When folding a data type value one always moves "forward" from the current constructor (node) to the contained values (that is, successors). In contrast, the graph constructor & also provides access to a node's predecessors. So we have to determine the fold direction within the fold operator. We do this by a parameter function f, computing from a node's context the list of nodes (l) which are to be accessed, that is, folded, next. Two such functions which will be used in the sequel direct fold to the successors, respectively, predecessors:

```
fun fwd (p,v,l,s) = s
fun bwd (p,v,l,s) = p
```

Now fold operates on a node v in two steps: first, fold is recursively applied to the list of nodes, l, which is computed by f from v's context, yielding a list of results l'. Since, in general, the length of l is varying, the results in l' have to be accumulated in some way. This is achieved by a parameter function b which is (list-) folded along l', yielding a value r. A further parameter function d is finally applied to lab (the label of v) and r.

Another parameter is the "linearity" of nodes, that is, whether a node value can be used only once in a computation or if it might be used multiple times (when reached, for example, from different predecessors). We first consider the former option: once we have matched a node context (p,v,l,s) & g we proceed with just graph g, thus forgetting v. This is a bit dangerous since v might be tried to be matched in g later (coming from a different predecessor) thus causing a Match exception. Being aware of that fact, however, we can recover from exceptions by giving meaningful defaults. In fact, this is done in the following definition of gfold. We first define two functions for performing fold from just one node, respectively, from a list of nodes.

```
fun gfold1 f d b u v
             ((c as (_,v,lab,_)) & g) =
    let val (r,g1) = gfoldn f d b u (f c) g
     in (d (lab,r),g1) end

and gfoldn _ _ _ u []      g = (u,g)
 |  gfoldn f d b u (v::l) g =
    let val (x,g1) = gfold1 f d b u v g
        val (y,g2) = gfoldn f d b u l g1
     in (b (x,y),g2) end
    handle Match => gfoldn f d b u l g
```

In addition to the accumulated value, both functions have as a result the reduced graph. Performing successive fold calls always on these reduced graphs essentially ensures that nodes are visited only once. In a sense, the graphs passed around represent the progressive consumption of nodes from the original graph. The exception handling in gfoldn captures the following case: when a node passed to gfoldn has already been consumed by a recursive call to gfold1 at the time it is to be processed, it causes a Match exception (in gfold1). In that case gfoldn simply takes the next node in the list. (Those who do not like programming with exceptions might note that their use is not essential here. Alternatively, we execute the second RHS of gfoldn only if gmember v g is true, otherwise we call gfoldn f d b u l g.) Now gfold performs gfoldn and drops the graph result:

```
fun gfold f d b u l g = #1 (gfoldn f d b u l g)
```

In essence, gfold fwd performs depth-first search on graphs. As demonstrated in [14], many graph problems can be easily solved by first computing a depth-first spanning tree of the graph. So we show how to compute it with gfold. We will represent trees of variable degree by the following data type:

```
datatype 'a tree = Branch of 'a * 'a tree list
```

Now, dfs is simply given by (with val Cons = op ::):

```
fun dfs l g = gfold fwd Branch Cons [] l g
```

This definition for depth-first search is very different from the Haskell implementation presented in [14]. In particular, the way of maintaining the dfs-state is distinctive: instead of using state transformers, remembering already visited nodes is implicit in the graph decomposition achieved by pattern matching.[1] Note that we have deliberately omitted a case like

```
fun gfold1 f d b u v Empty = u
```

from the definition of gfold1 to obtain a more general typing. As gfold is actually of type

```
('a context -> node list) ->
('a * 'b -> 'c) -> ('c * 'b -> 'b) -> 'b ->
node list -> 'a graph -> 'b
```

adding the above case would result in a unification of 'b with 'c entailing some effort to adjust definitions like that of dfs. The similarity of gfold to dfs makes it the basis for many graph algorithms. Since we can establish general laws for gfold (see Section 6) graph algorithms become amenable to program optimization.

Linear fold is different from fold on data types: there, multiple threads to a value are possible via the use of sharing variables. In a decomposition of a value containing multiple threads, say, to a subvalue v, v is processed as many times as there are threads leading to it. This is not the case for gfold which processes just one thread.

5.3 Multiple Access Graph Fold

An obvious generalization of gfold is to allow for multiple accesses to nodes which can be accomplished by re-inserting the currently matched node v with only incoming edges (except the one via which v is reached); multiple accesses to the node are then possible through successor lists of other nodes that have not been processed yet. Node sharing and loops (edges from a node to itself) require careful treatment within the fold operator: when a node v is processed, that is,

[1] Actually, these nodes are forgotten and not remembered.

a function d is applied to v's label lab and a value r resulting from reducing the currently remaining graph, the result d (lab,r) is not just returned as a value, but is also inserted as v's label into the graph to be reduced. This ensures that the value is available at later stages of the reduction, and it furthermore avoids its recomputation. This accounts for nodes reached via more than one predecessor. When folding a node v that contains an edge to itself, v is among its own successors, and eventually fold is applied to it. Thus, v must be present in the argument graph passed to the recursive fold call, that is, v must be re-inserted into g *before* the recursive call with its original label lab and without any predecessors and successors (this guarantees termination).

Since the result type of the fold is, in general, different from the type of node labels we actually have to process a heterogeneous graph where nodes labels are either tagged SRC (not processed yet) or DEST (node carries a result value). We therefore use the following union type:

```
datatype ('a,'b) hybrid = SRC of 'a | DEST of 'b
```

Now we can define the function mfold. In mfold1 we have to remove in each step exactly one edge – the edge by which the current node v was reached. We therefore have to pass as an additional parameter the node z from which v was accessed. Since there is no such node for any of the argument nodes initially passed to mfold we use the option data type:

```
datatype 'a option = SOME of 'a | NONE
```

and apply the SOME constructor to parent nodes and pass a nullary NONE to the initial call of mfoldn. (This also hides the parameter from the interface of mfold.) For simplicity we omit the parameter f (recall the definition of gfold) and consider only a forward fold, that is, we always move to successors. Thus, when reaching a node v with a DEST-label, we can simply re-insert v with its current predecessors except z.

```
fun mfold1 d b u (z,v) ((p,v,lab,s) & g) =
    case lab of DEST w =>
                (w,(drop z p,v,DEST w,[]) & g)
              | SRC w  =>
    let val (r,g1) = mfoldn d b u (SOME v,s)
                        (([],v,SRC w,[]) & g)
        val new    = d (w,r)
     in (new,(drop z p,v,DEST new,[]) & del v g1)
    end

and mfoldn _ _ u (_,[])    g = (u,g)
 |  mfoldn d b u (z,v::l) g =
    let val (x,g1) = mfold1 d b u (z,v) g
        val (y,g2) = mfoldn d b u (z,l) g1
     in (b (x,y),g2) end
```

(drop (SOME x) p removes one occurrence of the element x from list p, and drop NONE p = p.) Now mfold first wraps up the nodes of the graph with SRC, then reduces the graph by means of mfoldn, and finally drops the graph part of the result:

```
fun mfold d b u l g =
    #1 (mfoldn d b u (NONE,l) (gmap SRC g))
```

As an example, an evaluator for expression DAGs is given by the function evalDag. (The expression filter p l selects all elements of the list l for which the predicate p yields true.)

```
fun pred v ((p,v,_,_) & _) = p
fun roots g = filter (fn v=>pred v g=[]) (nodes g)
fun evalNode (CON i,_) = i
 |  evalNode (OP f,[x,y]) = f (x,y)
fun evalDag g =
    mfold evalNode Cons [] (roots g) g
```

It seems there are only few applications of mfold: there must be a need to fold along all edges (folding along a spanning tree can be done with gfold), and the order of decomposition must be important (otherwise ufold could be used). However, some advanced examples can be found in the translation of visual programs [9].

5.4 Graph Backtracking

By passing the very same graph to all recursive fold calls of one successor list we obtain a backtracking operator:

```
fun backtrack1 d b u v ((_,v,lab,s) & g) =
    d (lab,backtrack d b u s (([],v,lab,[]) & g))

and backtrack d b u nil g = u
 |  backtrack d b u (v::l) g =
    b (backtrack1 d b u v g,backtrack  d b u l g)
```

With backtrack we can compute, for example, all simple paths in a graph (let val append = op @):

```
fun conspaths (v,l) = map (fn p=>v::p) l
fun pathsfrom s g =
    backtrack conspaths append [nil] [s] g
fun allpaths g = fold append
    (map (fn v=>pathsfrom v g) (nodes g)) []
```

(Actually, the list of paths returned by allpaths contains $|V|$-times the empty path.)

6 Program Fusion

A popular optimization technique for functional languages is to eliminate intermediate results of multi-pass algorithms. Concerning graph algorithms, Launchbury [15] gives some examples of how to fuse operations based on dfs.

The following theorem shows that program fusion also applies to algorithms specified by graph folds. (The proof is given in the Appendix.)

Theorem 2 (Promotion Theorem) *If* M *and* N *are functions such that*

```
    M (d (x,y)) = e (x,N y)
    N (b (x,y)) = f (M x,N y)
          N u = u'
```

then:

```
    N (gfold h d b u l g) = gfold h e f u' l g
```

As an application example consider the definition of topological sorting as given in [14, 15]:

```
fun postorder (Branch (v,f)) = postorderf f @ [v]
and postorderf []     = []
 | postorderf (t::f) = postorder t @ postorderf f
fun topsort g =
    rev (postorderf (dfs (nodes g) g))
```

After unfolding the definition of dfs we can apply the promotion theorem to obtain a version of topsort that does not build an intermediate tree structure. First, we match variables of the theorem: h = fwd, d = Branch, b = Cons, u = [], and N = rev o postorderf.

Next we have to invent values for the remaining variables: e = Cons, f = fn (x,y)=>y@x (append first argument to second), u' = [], and M = rev o postorder. Now we check the premises of the theorem: it is clear that N u = [] = u'. Moreover:

```
M (d (x,y))
= rev (postorder (Branch (x,y)))
= rev (postorderf y @ [x]))
= [x] @ (rev (postorderf y))
= x::(rev o postorderf) y
= e (x,N y)
```

```
N (b (x,y))
= rev (postorderf (x::y))
= rev (postorder x @ postorderf y))
= rev (postorderf y) @ rev (postorder x)
= (rev o postorderf) y @ (rev o postorder) x
= f (M x,N y)
```

Thus we obtain the following optimized version of topsort:

```
fun topsort g = gfold fwd Cons (fn (x,y)=>y@x)
                      [] (nodes g) g
```

Theorem 2 facilitates the elimination of intermediate tree structures which certainly has many applications. Yet, it is challenging to investigate unfold/fold transformations to save intermediate graph structures, too; according to Wadler [24] we could call this *degraphation*. As an example we optimize the implementation of Sharir's strongly-connected components algorithm as given in [7, 14]:

```
fun scc g = dfs (rev (postorderf
                      (dfs (nodes g) g))) (grev g)
```

The algorithm works by performing dfs on a graph with its edges reversed (**grev g**) while the argument node list of the traversal must be a reverse postorder list of the graph's nodes (**rev (postorderf ... g)**). We can save the intermediate graph resulting from the edge reversal by fusing the definition of dfs with that of grev. To do this we use a *duality theorem* that relates gfold fwd to gfold bwd. (The proof can be found in the Appendix.)

Theorem 3 (Duality Theorem)
gfold fwd d b u l (grev g) =
gfold bwd d b u l g

The application to the function scc gives (using the optimized version of topsort):

```
fun scc g = gfold bwd Branch Cons []
    (gfold fwd Cons (fn (x,y)=>y@x) []
     (nodes g) g) g
```

7 Implementation

We have implemented the proposed graph concept as an extension of ML. At the core is a data structure for persistent graphs, that is, graphs that are *non*-destructively updated through applications of the & constructor and by decomposition. To our knowledge, data structures for persistent graphs have not been investigated previously [20]. (The method of [6] cannot be used since it applies only to linked structures with nodes of constant bounded in-degree.)

Since, even for imperative graphs, no single graph representation exists that is optimal for all kinds of applications, we initially focus on a representation suited for sparse graphs and base our implementation on node-indexed arrays of adjacency lists. By using functional arrays we ensure that any update to the graph does not invalidate older graph versions. We use the version tree implementation of functional arrays [1] in which updates take constant time and index access time depends on the depth of the version tree. Extending version trees by an (imperatively updated) "cache array" that actually duplicates the array represented by the leftmost node in the version tree, index access becomes $O(1)$ for single-threaded arrays.

Let us assume for a moment that we represent a graph by three arrays L, S, and P storing node labels, successor and predecessor lists. Then adding a node context (p,v,l,s) (of size c) can be simply done by (i) setting the node label, (ii) adding successors, and (iii) adding predecessors as follows:

(i) $L[\mathtt{v}] := \mathtt{l}$
(ii) $S[\mathtt{v}] := \mathtt{s}$ and $\forall\, \mathtt{u} \in \mathtt{p} : S[\mathtt{u}] := \mathtt{v}::S[\mathtt{u}]$
(iii) $P[\mathtt{v}] := \mathtt{p}$ and $\forall\, \mathtt{w} \in \mathtt{s} : P[\mathtt{w}] := \mathtt{v}::P[\mathtt{w}]$.

Thus, adding a node context takes $O(c)$ steps plus the time to locate all the lists $S[\mathtt{u}]$ and $P[\mathtt{w}]$. In the worst case, this is $O(cu)$ where u denotes the number of updates to g. (Note that u is generally not even bounded by the number of edges.) However, in single-threaded graphs, an adjacency list is found in $O(1)$, so & is $O(c)$.

Graph decomposition as requested by a match (p,v,l,s) $\underline{\&}$ g is, in general, more complex: not only must we return v's label, its successors and predecessors, we also have to build the reduced graph resulting from the deletion of context (p,v,l,s). To do this we delete v, and we remove v from all successor (predecessor) lists of v's predecessors (successors), that is,

(i) $L[\mathtt{v}] := \perp$
(ii) $\forall\, \mathtt{u} \in \mathtt{p} : S[\mathtt{u}] := \mathtt{drop\ v}\ S[\mathtt{u}]$
(iii) $\forall\, \mathtt{w} \in \mathtt{s} : P[\mathtt{w}] := \mathtt{drop\ v}\ P[\mathtt{w}]$.

The costly operations are those in steps (ii) and (iii): we have to find $O(c)$ adjacency lists, which requires $O(cu)$ steps in general and $O(c)$ steps in the single-threaded case. The deletion of v takes $O(c)$ time for each list. Thus, $\underline{\&}$ is $O(c^2u)$ in general and $O(c^2)$ in the single-threaded case.

We can improve this implementation by exploiting the following observation: the deletion of a node in any adjacency list can be noticed at the earliest when that list is requested by (another) context match. So in the implementation of $\underline{\&}$ instead of removing v from adjacency lists, we just mark v as deleted. (This is done in an additional array V.) But now p and s cannot simply be bound to $S[\mathtt{v}]$ and $P[\mathtt{v}]$, respectively, instead only those nodes are returned that are not marked as deleted in V. This means that building the reduced graph is $O(1)$ and computing p, l, and s now takes $O(u+c)$ steps ($O(c)$ in the single-threaded case), and this is also the complexity of $\underline{\&}$. Even if this means a reduction in complexity only for non-sparse graphs, it is in any case an important improvement in practice, since in addition to smaller constants within the big-Oh expressions, we also save a lot of heap allocations.

There remains one problem with the proposed approach: assume the context of node v is deleted and, for example, S[w] contains v. Now, if later on v is re-inserted *without* w as one of its predecessors, then v still must not be considered a successor of w. But this seems to be impossible since we cannot mark v as deleted anymore. A solution is to equip nodes with a kind of "time stamps": when a node v is inserted the first time, it gets a stamp, say 1 (that is, we set V[v] := 1) and we store this stamp with each entry in an adjacency list. When v is removed, we set V[v] := $-V$[v]. When accessing nodes in an adjacency list, we return only those nodes whose stamps in the list are equal to that in V. So deleted nodes will be filtered out. Now when reinserting v we set V[v] := $-V$[v] + 1 so that "old" entries in adjacency lists still have non-matching stamps and will correctly be filtered out.

The importance of the structure lies in its behavior on single-threaded graph decompositions:[2] a function like gfold has a running time of $O(|V|+|E|)$, that is, is linear in the size of the graph. As an immediate consequence of this, algorithms, such as dfs, have the same complexity as in the imperative case.

However, the implementation of functional graphs bears a considerable overhead. To get an impression of the real behavior, we compare the functional algorithms for graph reversal, dfs, and DAG evaluation with corresponding imperative implementations. We also give the measures for a functional realization of the imperative algorithms with functional arrays.

The imperative algorithms make use of the imperative arrays of ML and represent a graph simply by two arrays for storing node labels and successors. The functional implementation of the imperative algorithms use an efficient implementation of balanced binary search trees [2] to represent functional arrays. The algorithms are slightly changed to exploit the dynamic behavior of search trees and to account for state threading. The functional algorithms are those defined in the paper, that is, they are defined through ufold, gfold, and mfold which are based on & and $\underline{\&}$ (or, context) as provided by the persistent graph implementation described above. The source code is shown in the Appendix. As expected, the functional algorithms are significantly slower than the imperative ones. This is mainly due to the intensive use of the heap caused by updates to the graph representing functional arrays.

We can improve the running time of the functional algorithms by providing predefined graph fold operations. Consider, for example, gfold: Instead of decomposing the argument graph in each step, we can use a local (imperative) array M to mark those nodes already matched during the current run of gfold: Initially, M is obtained by copying the stamp array V of the argument graph. Then each time the context of a node v is matched, M[v] is set to -1, and only those successors and predecessor of v are selected that have a stamp equal to that in M. Similarly, mfold can be improved by locally storing traversed edges in a hash table.

The case for ufold is more subtle. First, we observe that the chosen array representation of a graph forgets about its construction history. In particular, we do not know which node (context) was inserted last. This implies that with this implementation & cannot be used in pattern matching. So to implement ufold we rather need a function like matchany that matches an arbitrary node context. Now a simple implementation for matchany will search for any (for example, the first) node that has a valid stamp. Used repeatedly, this leads to a running time of ufold that is quadratic in the number of nodes. Thus, a predefined version that scans the node array in a fixed order achieves linear complexity for ufold. The implementation also uses a local imperative array M similar to the predefined gfold.

We ran the implementations of grev and dfs on a sparse graph (with a degree of 8) with 1000, 5000, and 10000 nodes. The user time spent by SML/NJ 1.09 on a SPARCstation 10 is given in Figures 4 and 5. The "functionalized"-rows show the times of a functional realization of the imperative algorithms.

	1000	5000	10000	ratios
functional	0.68s	13.04s	50.49s	68..388
predefined ufold	0.16s	0.98s	3.25s	10..25
imperative	0.01s	0.10s	0.13s	1
functionalized	0.30s	2.04s	4.72s	20..36

Figure 4: Running times of grev.

The predefined version of ufold improves the running time of grev by an order of magnitude, but it cannot compete with the imperative implementation. However, it is recognizably faster that the functionalized implementation.

	1000	5000	10000	ratios
functional	0.08s	0.72s	1.58s	8..12
predefined gfold	0.02s	0.17s	0.26s	2..3
imperative	0.01s	0.06s	0.13s	1
functionalized	0.21s	0.76s	2.28s	13..21

Figure 5: Running times of dfs.

We can see in Figure 5 that the predefined version of gfold performs quite well. It is striking that gfold seems to run much faster than ufold. This is certainly because imperative dfs has to build a dfs-tree on the heap whereas imperative graph reverse only works on its imperative arrays.

The function evaldag was applied to tree-shaped DAGs where internal nodes have two successors and predecessors. The results are shown in Figure 6.

Again, the basic functional solution is extremely slow because mfold has to make intensive use of graph constructors.

[2]Note that the complexity of the general case could be improved by employing advanced data structures for functional arrays, such as [5], although an implementation would require considerable effort.

	5050	11325	20100	ratios
functional	2.59s	7.41s	15.08s	43..49
predefined `mfold`	0.32s	0.82s	1.72s	5
imperative	0.06s	0.15s	0.32s	1
functionalized	0.46s	1.18s	2.35s	7..8

Figure 6: Running times of **evaldag**.

8 Conclusions

We have presented a new programming style for graphs that draws much of its attraction from being based on pattern matching and value decomposition, which are well-known and accepted programming concepts. The most significant difference between this and previous approaches is the departure from the imperative *view* of graph traversals, giving more opportunities for program transformation and optimization. Although more work is required on functional graphs and efficient graph operations, experiments with an initial implementation are encouraging showing that the presented approach is a reasonable and practical alternative to imperative graphs in functional languages. In particular, predefined graph operations offer much potential for further efficiency improvements.

References

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

[2] S. Adams. Efficient Sets - A Balancing Act. *Journal of Functional Programming*, 3:553–561, 1993.

[3] R. S. Bird. Functional Algorithm Design. In *Mathematics of Program Construction*, LNCS 947, pages 2–17, 1995.

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

[5] P. F. Dietz. Fully Persistent Arrays. In *Workshop on Algorithms and Data Structures*, LNCS 382, pages 67–74, 1989.

[6] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

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

[8] M. Erwig. Active Patterns. In *8th Int. Workshop on Implementation of Functional Languages*, pages 95–112, 1996. To appear in LNCS.

[9] M. Erwig and B. Meyer. Heterogeneous Visual Languages – Integrating Visual and Textual Programming. In *11th IEEE Symp. on Visual Languages*, pages 318–325, 1995.

[10] L. Fegaras and T. Sheard. Revisiting Catamorphisms over Datatypes with Embedded Functions. In *ACM Symp. on Principles of Programming Languages*, pages 284–294, 1996.

[11] J. Gibbons. An Initial Algebra Approach to Directed Acyclic Graphs. In *Mathematics of Program Construction*, LNCS 947, pages 282–303, 1995.

[12] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving Structural Hylomorphisms from Recursive Definitions. In *1st Int. Conf. on Functional Programming*, pages 73–82, 1996.

[13] Y. Kashiwagi and D. Wise. Graph Algorithms in a Lazy Functional Programming Language. In *4th Int. Symp. on Lucid and Intensional Programing*, pages 35–46, 1991.

[14] D. J. King and J. Launchbury. Structuring Depth-First Search Algorithms in Haskell. In *ACM Symp. on Principles of Programming Languages*, pages 344–354, 1995.

[15] J. Launchbury. Graph Algorithms with a Functional Flavour. In *1st Int. Spring School on Advanced Functional Programming*, LNCS 925, pages 308–331, 1995.

[16] J. Launchbury and T. Sheard. Warm Fusion: Deriving Build-Catas from Recursive Definitions. In *Conf. on Functional Programming and Computer Architecture*, pages 314–323, 1995.

[17] G. Malcom. Homomorphisms and Promotability. In *Mathematics of Program Construction*, LNCS 375, pages 335–347, 1989.

[18] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Conf. on Functional Programming and Computer Architecture*, pages 124–144, 1991.

[19] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[20] C. Okasaki. Functional Data Structures. In *Advanced Functional Programming*, LNCS 1129, pages 131–158, 1996.

[21] P. Palao Gonstanza, R. Peña, and M. Núñez. A New Look at Pattern Matching in Abstract Data Types. In *1st Int. Conf. on Functional Programming*, pages 110–121, 1996.

[22] T. Sheard and L. Fegaras. A Fold for all Seasons. In *Conf. on Functional Programming and Computer Architecture*, pages 233–242, 1993.

[23] P. Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *ACM Symp. on Principles of Programming Languages*, pages 307–313, 1987.

[24] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73:231–284, 1990.

Appendix

Proof of Theorem 2. We perform an induction on `l` and `g`.

<u>`l = []`</u> From the definition of `gfoldn` we see immediately that `gfoldn h d b u [] g = (u,g)` and thus `gfold h d b u [] g = u`. Likewise, `gfold h e f u' [] g = u'`. Now the conclusion of the theorem follows directly from the third premise.

<u>`g = Empty, l = v::l'`</u> `gfoldn` causes a call `gfold1 h d b u v Empty` raising a `Match` exception that is handled to return `gfoldn h d b u l' Empty`. Since no element of `l'` can be matched in the empty graph, we know by induction on `l'` and by the previous case that `gfold h d b u l Empty = u`. For the same reason, `gfold h e f u' l Empty = u'`, and for this case the theorem follows from the third premise.

<u>`g = c & g', l = v::l'`</u> Here `gfold` first causes a call `gfold1 h d b u v g` which either succeeds or results in a `Match` exception. In the latter case handling the exception yields `gfoldn h d b u l' g`, and since in that case the corresponding expression `gfold1 h e f u' v g` also raises `Match` and yields `gfoldn h e f u' l' g`, we can assume the theorem by induction.

Otherwise, the `gfold1` expression results in a pair `(x,g1)` with `x = d (lab,r)` where `lab` is the label of `v` and `r` is the first component of the recursive call `gfoldn h d b u s g'` (where `s = f c`). In the same way we obtain a value `r'` as the first component of the expression `gfoldn h e f u' s g'`, and we can apply the induction hypothesis to obtain

$$\texttt{N r} = \texttt{r'} \qquad (1)$$

After the `gfold1` expression has been evaluated, a pair `(y,g2)` is computed by `gfoldn h d b u l' g1`. Similarly, a pair `(y',g2)` is given by the expression `gfoldn h e f u' l' g1`. Note that the graphs `g1` and `g2` are actually identical in the two folds since the graph decomposition is only affected by `h`, `l/l'`, and `g/g'` which are identical in the corresponding fold expressions. Now we can apply the induction hypothesis again and get

$$\texttt{N y} = \texttt{y'} \qquad (2)$$

Finally, the result of `gfold h d b u l g` is `b (x,y) = b (d (lab,r),y)`. Likewise, the result of `gfold h e f u' l g` is `f (e (lab,r'),y')`. Now we can conclude:

```
  N (gfold h d b u l g)
= N (b (d (lab,r),y))
= f (M (d (lab,r)),N y)          (2nd premise)
= f (e (lab,N r)),N y)           (1st premise)
= f (e (lab,r')),y')             (ind.hyp. (1), (2))
= gfold h e f u' l g                             □
```

Proof of Theorem 3. The proof is by induction on `l` and `g`. For (`l = []`) and (`g = Empty, l = x::l'`) the result of `gfold` is always `u`, and thus the theorem is true for these cases.

Thus consider the case (`g` $\neq$ `Empty, l = v::l'`). `gfoldn bwd` computes a pair `(x,g1)` through a call to `gfold1 bwd`. There `g` can be written (due to active pattern matching) as[3] `(p,v,lab,s) & g'` which means that the pair `(r,g2)` in `gfold1 bwd` is given by the expression `gfoldn bwd d b u p g'`. Since `grev g = (s,v,lab,p) & (grev g')` the corresponding `gfold1` expression caused by `gfold fwd d b u l g` can be written as `gfold1 fwd d b u v ((s,v,lab,p) & (grev g'))` if `s` does not contain `v`. (This restriction is necessary because otherwise `v` would be "moved" through the matching from `s` into `p`.) Then the pair `(r,g2)` in the call `gfold1 fwd` is given by the expression `gfoldn fwd d b u p (grev g')`. Now the theorem follows by applying the induction hypothesis to `gfoldn bwd d b u p g'` and `gfoldn fwd d b u p (grev g')` and to the remaining calls `gfoldn fwd d b u l' (grev g1)` in `gfoldn`, respectively, `gfold bwd d b u l' g1` in `gfoldn bwd`.

If, on the other hand, `v` is contained in `s`, the `gfold1` expression caused by `gfold fwd d b u l g` can be written as `gfold1 fwd d b u v ((s',v,lab,p') & (grev g'))` where `s'` is equal to `s` with all occurrences of `v` removed and `p'` is `p` with all occurrences of `v` in `s` appended, say. This means the pair `(r,g2)` in the call `gfold1 fwd` is given by the expression `gfoldn fwd d b u p' (grev g')`. The difference to the expression above is some occurrences of `v` in `p'`. Now since `v` is not contained in `grev g'` the eventually caused `gfold1` calls with `v` will all raise a `Match` exception (which are handled by just moving in `p'` to the next node). Thus, the presence of `v` in `p'` has actually no impact on the result compared with the corresponding computation using `p`. This means that `gfoldn fwd d b u p' (grev g')` yields the same result as `gfoldn fwd d b u p (grev g')`, so we can again apply the induction hypothesis, and the theorem is proved. □

```
fun id x = x
fun p1 (x,_) = x
fun forceOpt (SOME x) = x
fun select _ _ []     = []
 | select f p (x::l) =
   if p x then f x::select f p l
          else select f p l
```

Figure 7: Some utility functions.

```
signature FUN_ARRAY =
sig
  type 'a array
  val array         : int * 'a -> 'a array
  val sub           : 'a array * int -> 'a
  val size          : 'a array -> int
  val update        : 'a array * int * 'a -> 'a array
  val toImpArray    : 'a array -> 'a Array.array
  val fromList      : 'a list -> 'a array
  val fromImpArray  : 'a Array.array -> 'a array
end
```

Figure 8: Operations on functional arrays.

[3]The case for the Match exception is identical to Theorem 2.

```
structure FunArray : FUN_ARRAY =
struct

  datatype 'a array =
      Root  of 'a Array.array
    | Node  of int * 'a * 'a array
    | Cache of int * 'a * 'a array * bool ref * 'a Array.array

  fun array (n,x) = Cache (0,x,Root (Array.array (n,x)), ref true,Array.array (n,x))

  fun search (Cache (_,_,Root a,_,_),i) = Array.sub (a,i)
   |  search (Cache (j,x,  tree,_,_),i) = if i=j then x else search (tree,i)
   |  search (Node  (_,_,Root a),i)     = Array.sub (a,i)
   |  search (Node  (j,x,  tree),i)     = if i=j then x else search (tree,i)

  and sub (tree as Cache (_,_,_,ref cache,c),i) = if cache then Array.sub (c,i) else search (tree,i)
   |  sub (tree,i) = search (tree,i)

  fun size (Root a)             = Array.length a
   |  size (Node (_,_,a))       = size a
   |  size (Cache (_,_,_,_,a)) = Array.length a

  fun update (a as Cache (_,_,_,cache,c),i,x) =
      if !cache then (cache := false; Array.update (c,i,x); Cache (i,x,a,ref true,c)) else Node (i,x,a)
   |  update (a,i,x) = Node (i,x,a)

  fun fromList l = Cache (0,hd l,Root (Array.fromList l), ref true,Array.fromList l)

  fun fromImpArray a =
      let val b = Array.array (Array.length a,Array.sub (a,0))
       in (Array.copy {src=a,si=0,len=NONE,dst=b,di=0}; Cache (0,Array.sub (a,0),Root a,ref true,b))
      end

  fun toImpArray (Cache (_,_,Root a,_,_)) =  (* is used only on unchanged arrays *)
      let val b = Array.array (Array.length a,Array.sub (a,0))
       in (Array.copy {src=a,si=0,len=NONE,dst=b,di=0}; b) end
end
```

Figure 9: Implementation of functional arrays with cache.

```
signature GRAPH =
sig
  eqtype node = int
  type 'a context = node list * node * 'a * node list
  type 'a graph
  exception Node
  val empty      : int -> 'a graph
  val &          : 'a context * 'a graph -> 'a graph
  val context    : node * 'a graph -> 'a context * 'a graph
  val matchany   : 'a graph -> 'a context * 'a graph
  val noNodes    : 'a graph -> int
  val gmap       : ('a -> 'b) -> 'a graph -> 'b graph
  val ufold      : ('a context * 'b -> 'b) -> 'b -> 'a graph -> 'b
  val gfold      : ((node * int) list * node * 'a * (node * int) list -> (node * int) list) ->
                   ('a * 'b -> 'c) -> ('c * 'b -> 'b) -> 'b -> node list -> 'a graph -> 'b
  val mfold      : ('a * 'b -> 'c) -> ('c * 'b -> 'b) -> 'b -> node list -> 'a graph -> 'b
end
```

Figure 10: Operations on functional graphs.

```
functor Graph (FunArray:FUN_ARRAY) : GRAPH =
struct

  type node = int
  type 'a context = node list * node * 'a * node list
  exception Node
  exception Edge

  (* additional array functions *)
  open FunArray
  fun apply (a,i,f)    = update (a,i,f (sub (a,i)))
  fun firstIndex (a,p) = let fun scan (i,p) = if p (sub (a,i)) then i else scan (i+1,p)
                          in scan (0,p) end
  fun arrayToList f a = let fun list (f,a,i,n) = if i<n then f (sub (a,i))::list (f,a,i+1,n) else []
                         in list (f,a,0,size a) end
  fun impArrayToList a = let fun list (a,i,n) = if i<n then Array.sub (a,i)::list (a,i+1,n) else []
                          in list (a,0,Array.length a) end

  (* stamp type and operations *)
  type stamp = int
  fun stampTrue  i = abs i+1
  fun stampFalse i = ~(abs i+1)
  fun getStamp (na,n)    = sub (na,n)
  fun getNegStamp (na,n) = let val s=sub (na,n)
                            in if s>0 then raise Node else s end
  fun getPosStamp (na,n) = let val s=sub (na,n)
                            in if s<=0 then raise Edge else s end

  datatype 'a graph = Empty of int
                    | Full  of    stamp array * 'a array
                              * (node * stamp) list array   (* pred *)
                              * (node * stamp) list array   (* suc  *)

  (* basic graph operations *)
  fun empty n = Empty n

  infixr 5 &
  fun (pred,n,l,suc)&(Empty i) =
      (pred,n,l,suc)&(Full (array (i,0),array (i,l),array (i,[]),array (i,[])))
   |  (pred,n,l,suc)&(Full (na,la,pa,sa)) =
      let val stampN      = stampTrue (getNegStamp (na,n))
          val stampedPred = map (fn x=>(x,getPosStamp (na,x))) pred
          val stampedSuc  = map (fn x=>(x,getPosStamp (na,x))) suc
          fun updAdj (a,[])   = a
            | updAdj (a,v::l) = updAdj (apply (a,v,fn adj=>(n,stampN)::adj),l)
       in
          Full (update (na,n,stampN),update (la,n,l),
                updAdj (update (pa,n,stampedPred),suc),updAdj (update (sa,n,stampedSuc),pred))
      end

  fun context (n,Empty _) = raise Match
   |  context (n,Full (na,la,pred,suc)) =
      if getStamp (na,n)>0 then
         ((select p1 (fn (v,i)=>i=getStamp (na,v)) (sub (pred,n)),n,sub (la,n),
           select p1 (fn (v,i)=>i=getStamp (na,v)) (sub (suc,n))),
          Full (apply (na,n,stampFalse),la,pred,suc))
      else
         raise Match

  fun matchany (Empty _) = raise Match
   |  matchany (g as (Full (na,la,pred,suc))) =
              (context (firstIndex (na,fn i=>i>0),g) handle Subscript => raise Match)
```

Figure 11: Functional graph implementation (Part 1).

```
 fun noNodes (Empty _)          = 0
  |  noNodes (Full (na,_,_,_)) = size na

 fun gmap f (Empty i) = Empty i
  |  gmap f (Full (na,la,pa,sa)) =
     let val x = f (sub (la,0))
         val n = ref (size la)
         val L = Array.array (!n,x)
         val _ = while (!n>0) do (n := !n-1;Array.update (L,!n,f (sub (la,!n))))
      in
         Full (na,fromImpArray L,pa,sa)
     end

(* predefined fold operations *)
fun ufold f u (Empty _) = u
 |  ufold f u (Full (na,la,pred,suc)) =
    let val V = toImpArray na
        val n = Array.length V
        fun ufoldi x =
            if x<n then
               let val c = (select p1 (fn (v,_)=>Array.sub (V,v)>0) (sub (pred,x)),
                            x,sub (la,x),
                            select p1 (fn (v,i)=>Array.sub (V,v)>0) (sub (suc,x)))
                   val _ = Array.update (V,x,~1)
                   val r = ufoldi (x+1)
                in f (c,r) end
             else
                u
     in ufoldi 0 end

fun gfold f d b u l (Empty _) = u
 |  gfold f d b u l (Full (na,la,pred,suc)) =
    let val V = toImpArray na
        fun gfold1 v = (Array.update (V,v,~1);
                        let val l=sub (la,v)
                         in
                            d (l,gfoldn (f (sub (pred,v),v,l,sub (suc,v))))
                        end)
        and gfoldn []         = u
          |  gfoldn ((v,i)::l) =
             let val j = Array.sub (V,v)
              in
                 if j<0 orelse i<>j then gfoldn l else b (gfold1 v,gfoldn l)
             end
        and gfoldm []     = u
          |  gfoldm (v::l) = if Array.sub (V,v)<0 then gfoldm l
                                                  else b (gfold1 v,gfoldm l)
     in
        gfoldm l
     end
```

Figure 12: Functional graph implementation (Part 2).

```
  fun mfold d b u l (Empty _) = u
   | mfold d b u l (Full (na,la,pred,suc)) =
     let val V = toImpArray na
         val n = Array.length V
         val U = Array.array (n,false)
         val L = Array.array (n,NONE)
         exception NotFound
         fun e2word (v,w) = Word.fromInt (v*5+w)
         val E = HashTable.mkTable (e2word,fn (v,w)=>v=w) (5*n+1,NotFound)
         fun mfold1 (pred,v) =
             (case pred of SOME z => HashTable.insert E ((z,v),true) | NONE => ();
              if Array.sub (U,v) then forceOpt (Array.sub (L,v))
              else (Array.update (U,v,true);
                    let val x=d (sub (la,v),mfoldn (v,sub (suc,v)))
                     in (Array.update (L,v,SOME x); x) end))
         and mfoldn (_,[])       = u
          |  mfoldn (z,(v,i)::l) =
             let val j = Array.sub (V,v)
                 val e = getOpt (HashTable.find E (z,v),false)
              in
                 if j<0 orelse i<>j orelse e then mfoldn (z,l)
                 else b (mfold1 (SOME z,v),mfoldn (z,l))
             end
         and mfoldm []     = u
          |  mfoldm (v::l) = if Array.sub (V,v)<0 then mfoldm l else b (mfold1 (NONE,v),mfoldm l)
       in
          mfoldm l
      end
end (* functor Graph *)
```

Figure 13: Functional graph implementation (Part 3).

```
fun dfs' d b u (N,Suc) =
    let val V = Array.array (Array.length N,false)
        fun dfs1 v = (Array.update (V,v,true); d (Array.sub (N,v),dfsn (Array.sub (Suc,v))))
        and dfsn []     = u
         |  dfsn (v::l) = if Array.sub (V,v) then dfsn l else b (dfs1 v,dfsn l)
     in dfsn (List.tabulate (Array.length N-1,id)) end
fun dfs g = dfs' Branch (op ::) [] g

fun evaldag (N,Suc) =
    let val R = Array.array (Array.length N,Array.sub (N,0))
        val _ = Array.copy {src=N,si=0,len=NONE,dst=R,di=0};
        fun eval1 v =
            case Array.sub (R,v) of
                 CON i => i
               | OP  f => let val result = f ((fn [x,y] => (x,y)) (evaln (Array.sub (Suc,v))))
                           in (Array.update (R,v,CON result);result) end
        and evaln []     = []
         |  evaln (x::l) = eval1 x::evaln l
     in evaln (roots (N,Suc)) end

fun grev (N,Suc) =
    let val R = Array.array (Array.length N,[]:int list)
        fun scan (i,[])    = ()
         |  scan (i,v::l) = (Array.update (R,v,i::Array.sub (R,v)); scan (i,l))
        val _ = Array.appi scan (Suc,0,NONE)
     in (N,R) end
```

Figure 14: Imperative graph algorithms.