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.

In: 2nd ACM SIGPLAN Conference on Functional Programming, Amsterdam, 1997
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.

Fegas 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 other 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 empty`
& `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
```

<table>
<thead>
<tr>
<th>generality</th>
<th>[4]</th>
<th>[13]</th>
<th>[7]</th>
<th>[14]</th>
<th>[10]</th>
<th>This paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>efficiency</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>pureness</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>clearness</td>
<td>no</td>
<td>(yes)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>reasoning</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>(yes)</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Figure 1: Treatment of graphs in functional languages.
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.

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

In the subsequent examples we will use the following nodes:

```ml
```

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:

```ml
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.

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

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

![Figure 2: A DAG and its graph expression.](image)

### 3.2 Semantics of Graph Constructors

A graph \(G = (V,E,\nu)\) of type \(\alpha\) consists of a set of nodes \(V\), a multiset (or, bag) of edges \(E \in M(V \times V)\), and a total mapping \(\nu : V \rightarrow \alpha\) 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 \(\rho \vdash e \Rightarrow v\) says that expression \(e\) evaluates in the environment \(\rho\) 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 \(\sum_i x_i\) 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):

```ml
fun addnode \((v,1)\) \(g = ([],v,1,[])\ & g\)
fun addedge \((v,w)\) \((p,u,1,s) \& g\) =
  if u\#w then \((p,u,1,s) \& g\) else
  if u\#v then \((v,p,u,1,s) \& g\)
  else \((p,u,1,s) \& \text{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.

\(\Box\)

### 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 gmapp for mapping a function to all node labels of a graph:

```ml
fun gmapp f Empty = Empty
    | gmapp f \((p,v,1,s) \& g\) =
      \((p,v,f 1,s) \& (gmapp f g)\)
```

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, \(g = (p,v,1,s) \& g'\): 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 [25] 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
canonically represented we rather need to select a specific representation (among several equivalent) via a pattern, namely one with a certain node inserted last. With the above graph semantics we can easily define such a pattern as a language primitive. We write an environment mapping variables $x_1 \to v_1, \ldots, x_n \to v_n$, and an assertion $\rho \vdash v \Rightarrow \rho'$ says that pattern $p$ matched against value $v$ in the environment $\rho$ results in the binding(s) (that is, variable environment) $\rho'$ [19]. Let $G = (V, \rightarrow, (v, s_1), \ldots, (v, s_n), (p_1, v), \ldots, (p_m, v), \nu \cup \{v, l\})$. Then:

$$\rho \vdash (p, v, l, s) \iff \rho \vdash v \Rightarrow \{s \mapsto s_n, p \mapsto p_m, l \mapsto l, g \mapsto (V \setminus \{v\}, E, v, v)\}$$

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 $p, A, 1, s, k \Rightarrow g$ to the expression $(\{A, A, 1, 1\}) \& \text{Empty}$ binds node $l$ in list $s$ even though it was placed in the predecessor list. One could think of binding $l$ in both lists, $s$ and $p$, but this would contradict the intuition that the rule $(p, A, 1, s, k \Rightarrow (p, A, 1, s, k \Rightarrow g)$ 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 $l$ were bound in $p$ and $s$, the result expression would add two self loops to $l$.

As another example, consider matching $(p, A, 1, s, k \Rightarrow g$ to the expression $\exp\text{dag}$. We get $l = \alpha \ast +, p = [0], s = [A, A, 1], k = 1(\longrightarrow^+)$ 2. Thus we can define \text{Suc} and \text{Del} by:

\begin{align*}
\text{Suc} v (p, v, l, s, k \Rightarrow g) &= s \\
\text{Del} v (p, v, l, s, k \Rightarrow g) &= g
\end{align*}

Since the described pattern matching process not only computes bindings, but also performs an implicit reorganization of the matched value, we call $\& k$ 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 $\& k$ as a language extension, instead we can always replace a function definition

$$\text{fun } f \ldots v \ldots ((p, v, l, s) \& k) = e$$

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

$$\text{fun } f \ldots v \ldots g' = \quad \text{let } \text{val } ((p, v, l, s) \& k) = \text{context } (v, g') \quad \text{in } e \text{ end}$$

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 $\&$ 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 $f : 'a \times 'b \to 'a$, unordered fold is defined:

$$\text{fun } \text{ufold } f \text{ u Empty } = u \\
| \text{ufold } f \text{ u } (c, g) = f (c, \text{ufold } f \text{ u } g)$$

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

$$\text{val } \text{greve } = \quad \text{ufold } (\text{fn } ((p, v, l, s), g) \Rightarrow (s, v, l, p) \& g) \text{ Empty }$$

$$\text{fun } \text{map } f = \quad \text{ufold } (\text{fn } ((p, v, l, s), g) \Rightarrow (p, v, l, s) \& g) \text{ Empty }$$
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 import-
tance to many graph algorithms (already indicated by their
name: depth-first, breadth-first, best-first, etc.).

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 (1) 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, prede-
cessors:

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 l1. Since, in
general, the length of l is varying, the results in l1 have to be
accumulated in some way. This is achieved by a parameter
function b which is (list-) folded along l1, 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 compu-
tation 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 forget-
ting 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 meaning-
ful 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 gfoldI f d b u v
  (c as (_, v, lab, _)) & g =
  let val (r, gl) = gfold f d b u (f c) g
  in (d (lab, r), gl) end

and gfold _ _ _ _ u [] g = (u, g)

| gfold f d b u (v::l) g =
  let val (x, gl) = gfoldI f d b u v g
  val (y, g2) = gfold f d b u l g
  in (b (x, y), g2) end

handle Match => gfold 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 gfoldI cap-
tures the following case: when a node passed to gfoldI has
already been consumed by a recursive call to gfoldI at the
time it is to be processed, it causes a Match exception (in
gfoldI). In that case gfoldI simply takes the next node
in the list. (Those who do not like programming with ex-
ceptions might note that their use is not essential here. Al-
ternatively, we execute the second RHS of gfoldI only if
gmember v g is true, otherwise we call gfoldI f d b u l
1.) Now gfold performs gfoldI and drops the graph result:

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

In essence, gfoldI 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 1 g = gfoldI dfs Branch Cons [] 1 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

fun gfoldI f d b u v Empty = u

from the definition of gfoldI to obtain a more general typ-
ing. As gfoldI is actually of type

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

adjoining 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
gfoldI which processes just one thread.

5.3 Multiple Access Graph Fold

An obvious generalization of gfoldI is to allow for multiple
accesses to nodes which can be accomplished by re-inserting
the currently matched node v with only incoming edges (ex-
cept 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

1 Actually, these nodes are forgotten and not remembered.
loops (edges from a node to itself) require careful treatment within the fold operator: when a node v is processed, that is, a function f 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) = mfoldn d b u (z,v) g
    val (y,g2) = mfoldn d b u (z,l) g
    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,_) =
  | 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 mfold 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 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 pathfrom s g =
  backtrack conspaths append [nil] [s] g

fun allpaths g = fold append
  (map (fn v=>pathfrom 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, Lauchbury [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]:

67
fun postorder (Branch (v,f)) = postorder f @ [v]
  and postorder [] = []
| postorder (t::f) = postorder t @ postorder f

fun topsort g = 
    rev (postorder (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 postorder.

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 (postorder y @ [x])
  = [x] @ (rev (postorder y))
  = x::(rev o postorder y)
  = e (x,N y)

N (b (x,y))
  = rev (postorder (x::y))
  = rev (postorder x @ postorder y)
  = (rev o postorder) y @ (rev o postorder) x
  = (rev o postorder) 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 Shapir’s strongly-connected
components algorithm as given in [7, 14]:

fun acc g = dfs (rev (postorder ((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 (postorder ... g)). We can save the inter-
mediate graph resulting from the edge reversal by fusing the
definition of dfs with that of grev. To do this we use a *du-
ality 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 acc gives (using the opti-
mized version of topsort):

fun acc 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 per-
istent graphs, that is, graphs that are non-destructively
updated through applications of the & constructor and by
decomposition. To our knowledge, data structures for per-
istent 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 repre-
sentation 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 ad-
jacency lists. By using functional arrays we ensure that any
update to the graph does not invalidate older graph ver-
sions. 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,1,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[v] := 1
(iii) P[v] := p and \( \forall w \in s : P[w] := v : P[w] \).

Thus, adding a node context takes O(c) steps plus the time
to locate all the lists S[u] and P[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,1,s) \& 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
deletions of (p,v,1,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[v] := \emptyset
(ii) \( \forall u \in p : S[u] := \text{drop } v : S[u] \)
(iii) \( \forall w \in s : P[w] := \text{drop } v : P[w] \).

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, \& is \( O(c^2) \)
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 adja-
cy list can be noticed at the earliest when that list is
requested by (another) context match. So in the implement-
atation of \& 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[v] and
P[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 $p$, $q$, and $s$ now takes $O(n + c)$ steps ($O(c)$ in the single-threaded case), and this is also the complexity of $\delta$. 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: a function like $\text{gfold}$ 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 $\text{dfs}$, 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, $\text{dfs}$, and $\text{DAG}$ 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 $\text{ML}$ 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 defined in the paper, that is, they are defined through $\text{ufold}$, $\text{gfold}$, and $\text{mfold}$ which are based on $\&$ and $\&$ (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, $\text{gfold}$: 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 $\text{gfold}$. 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, $\text{mfold}$ can be improved by locally storing traversed edges in a hash table.

The case for $\text{ufold}$ 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 $\delta$ cannot be used in pattern matching. So to implement $\text{ufold}$ we rather need a function like $\text{matchany}$ that matches an arbitrary node context. Now a simple implementation for $\text{matchany}$ will search for any (for example, the first) node that has a valid stamp. Used repeatedly, this leads to a running time of $\text{ufold}$ 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 $\text{ufold}$. The implementation also uses a local imperative array $M$ similar to the predefined $\text{gfold}$.

We ran the implementations of $\text{grev}$ and $\text{dfs}$ 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.

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
<th>ratios</th>
</tr>
</thead>
<tbody>
<tr>
<td>functional</td>
<td>0.68</td>
<td>13.04</td>
<td>50.49</td>
<td>68.388</td>
</tr>
<tr>
<td>predefined ufold</td>
<td>0.16</td>
<td>0.98</td>
<td>3.25</td>
<td>10.25</td>
</tr>
<tr>
<td>imperative</td>
<td>0.01</td>
<td>0.10</td>
<td>0.13</td>
<td></td>
</tr>
<tr>
<td>functionalized</td>
<td>0.30</td>
<td>2.04</td>
<td>4.72</td>
<td>20.36</td>
</tr>
</tbody>
</table>

Figure 4: Running times of $\text{grev}$.

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

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
<th>ratios</th>
</tr>
</thead>
<tbody>
<tr>
<td>functional</td>
<td>0.08</td>
<td>0.72</td>
<td>1.58</td>
<td>8.12</td>
</tr>
<tr>
<td>predefined gfold</td>
<td>0.02</td>
<td>0.17</td>
<td>0.26</td>
<td>2.3</td>
</tr>
<tr>
<td>imperative</td>
<td>0.01</td>
<td>0.06</td>
<td>0.13</td>
<td></td>
</tr>
<tr>
<td>functionalized</td>
<td>0.21</td>
<td>0.76</td>
<td>2.28</td>
<td>13.21</td>
</tr>
</tbody>
</table>

Figure 5: Running times of $\text{dfs}$.

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

The function $\text{evaldag}$ 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 $\text{mfold}$ has to make intensive use of graph constructors.

---

*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.*
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


Appendix

Proof of Theorem 2. We perform an induction on 1 and g.

1 = [] 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.

g = Empty, 1 = v::1' gfoldn causes a call gfoldl h d b u v Empty raising a Match exception that is handled to return gfoldn h d b u 1' Empty. Since no element of 1' can be matched in the empty graph, we know by induction on 1' and by the previous case that gfold h d b u 1 Empty = u. For the same reason, gfold h e f u' 1 Empty = u', and for this case the theorem follows from the third premise.

g = c & g', 1 = v::1' Here gfoldn first causes a call gfoldl 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 1' g, and since in that case the corresponding expression gfoldl h e f u' v g also raises Match and yields gfoldn h e f u' 1' g, we can assume the theorem by induction.

Otherwise, the gfoldl 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 gfoldl h d b u s g' (where s = f e 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

N r = r' (1)

After the gfoldl expression has been evaluated, a pair (y,g2) is computed by gfoldn h d b u 1' g1. Similarly, a pair (y',g2) is given by the expression gfoldn h e f u' 1' g1. Note that the graphs g1 and g2 are actually identical in the two folds since the graph decomposition is only affected by h, 1/1', and g/g' which are identical in the corresponding fold expressions. Now we can apply the induction hypothesis again and get

N y = y' (2)

Finally, the result of gfoldl h d b u 1 g is b (x,y) = b (d (lab,r),y). Likewise, the result of gfoldl h e f u' 1 g is f (e (lab,r'),y'). Now we can conclude:

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

Proof of Theorem 3. The proof is by induction on 1 and g. For (1 = []) and (g = Empty, 1 = x::1') the result of gfoldl is always u, and thus the theorem is true for these cases.

Thus consider the case (g ≠ Empty, 1 = v::1'). gfoldlbed computes a pair (x,g1) through a call to gfoldl bved. There g can be written (due to active pattern matching) as3 (p,v,lab,s) & g' which means that the pair (r,g2) in gfoldl bved is given by the expression gfoldn bved d b u p g'. Since gved g = (s,v,lab,p) & (grev g') the corresponding expression caused by gfoldl bved d b u 1 g can be written as gfoldl bved 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 gfoldl bved is given by the expression gfoldn bved d b u p g' (grev g'). Now the theorem follows by applying the induction hypothesis to gfoldn bved d b u p g' and gfoldn bved d b u p (grev g') and to the remaining calls gfoldl bved d b u 1' (grev gl) in gfoldn, respectively, gfoldl bved d b u 1' gl in gfoldn bved.

If, on the other hand, v is contained in s, the gfoldn expression caused by gfoldl bved d b u 1 g can be written as gfoldl bved 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 gfoldl bved is given by the expression gfoldn bved 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 called gfoldl 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 bved d b u p' (grev g') yields the same result as gfoldl bved d b u p (grev g'), so we can again apply the induction hypothesis, and the theorem is proved.

□

fun id x = x
fun pl (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.

3The 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 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

infix 5 &

fun (pred,n,l,suc) & (Empty i) =
  (pred,n,l,suc) & (Full (array (i,0),array (i,1),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 stampedSucc = map (fn x=>(x,getPosStamp (na,x))) suc
          fun update (a,i) = a
          in
          update (a,v::l) = update (a,v,fn adj=>(n,stamp0::adj),l)
          end

fun context (n,Empty _) = raise Match
  | context (n,Full (na,la,pred,suc)) =
      if getStamp (na,n)>0 then
        let val 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)) in
        Full (apply (na,n,stampFalse),la,pre,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 unfold f u (Empty _) = u
  | unfold f u (Full (na,la,pred,suc)) =
    let val V = toImpArray na
      val n = Array.length V
    fun unfold x =
      if x<n then
        let val c = (select pl (fn (v,_)=>Array.sub (V,v)>0) (sub (pred,x)),
          x,sub (la,x),
          select pi (fn (v,i)=>Array.sub (V,v)>0) (sub (suc,x)))
        val _ = Array.update (V,x,"1")
        val r = unfold (x+1)
        in f (c,r) end
      else
        u
      in unfoldi 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 gfoldl v = (Array.update (V,v,"1));
      let val l=sub (la,v)
      in
        d (1,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 (gfoldl v,gfoldn l)
        end
    and gfoldm []  = u
      | gfoldm ((v::l) = if Array.sub (V,v)<0 then gfoldm l
        else b (gfoldl 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.fromString (v+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 mfoldd [] = u
  | mfoldd (v::l) = if Array.sub (V,v)<0 then mfoldd l else b (mfold1 (NONE,v),mfoldd l)
  in
  mfoldd 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),dfs1 (Array.sub (Suc,v))))
  and dfsn [] = u
  | dfsn (v::l) = if Array.sub (V,v) then dfsn l else b (dfs1 v,dfs1 l)
  in dfsn (List.tabulate (Array.length N-i,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.