FGL - A Functional Graph Library

User Manual

Martin Erwig
FernUniversität Hagen

October 31, 1997

1 Overview

The functional graph library (FGL) provides a set of ML signatures, structures, and functors that realize several interfaces and implementations of functional graphs for the use within ML programs. This document describes a preliminary version of the library; many features that are missing will be added in future.

The motivation for designing and implementing FGL was the unfortunate situation that although graphs are being used very frequently in (functional) programs, graph libraries for functional languages are not available. Even for imperative languages there exist only few libraries supporting graphs [4, 6, 5], and the design process cannot be considered as completed yet [7].

Some of the problems that make a design and a straightforward implementation difficult are:

Heterogeneity There are quite different types/views of graphs. On the one hand, offering an interface for each view might lead to a large number of interfaces that is difficult to comprehend: a user/programmer cannot easily select the most appropriate graph type for his or her current application. On the other hand, giving only very few interfaces might lead to oversimplification of certain graph types and there might be eventually not an appropriate interface reflecting a specific application.

Adaptability Related to the previous issue is the problem that a graph operation might require quite intensive changes in its type, parameters and, of course, implementation, when adapted in a different context. An important question is: how can changes in an operation be easily or even automatically propagated to derived operations.

Extensibility Extend a graph type by specific requirements, for example, nodes are embedded in the Euclidean plane, ideally by only specifying the changes and leaving the rest of the implementation untouched.

Diversity There exists a vast amount of even elementary graph operations, and the number multiplies when taking into account specialized algorithms for that are designed to work only for special cases.

These are general problems which pose non-trivial challenges to software engineering and design. In addition, there are specific problems with the integration of graphs into functional programming languages. These are discussed in some detail in [2] where as a solution an inductive graph view was proposed. With FGL we provide a functional graph library based on that approach. What we describe here is just a first step: the base library and even more the set of applications built on that misses many parts. However, the overall structure and the design goals are fixed, so that we can release a preliminary version of the library hoping to get feedback of users that helps to improve and further extend the library.

To obtain a powerful and flexible, yet usable and manageable, graph library we mainly follow some rules for coping with the large name space that seems to be unavoidable in a language like ML lacking general mechanisms for overloading:[1]

Use Naming Scheme Names for signatures, structures, and functors are selected in a well defined and transparent way. General signatures get single names, specialized or derived entries use a sequence of prepended qualifiers to the main name. Names of structures match those of signatures, and different implementations of the same interface use additional appended qualifiers. To keep often used names short default qualifiers are omitted. A more detailed description of the naming scheme is given in the Appendix.

Resist Overspecialization Define only very general interfaces. Sometimes, specialized implementations support only a subset of operations. Instead of introducing a new interface describing properly the implementation it is often better to use the general interface and leave some operations undefined (that is, they simply raise an exception `NotImplemented`).

In Section 2 we describe the main types used in the graph interfaces, and in Section 3 we explain the operations on graphs. Section 4 shows all the different implementations that are currently available. In Section 5 we describe some (standard) graph algorithms and applications that are implemented on top of the library. This should also provide some help in how to use the library. Finally, Section 6 lists known bugs and items that should be included in the next versions of FGL.

2 Graph Types

Graphs consist of nodes with edges connecting them. Some thoughts about node types are given in [2]. As yet, nodes are simply characterized by the following interface:

```
signature GRAPH_NODE =
sig
  eqtype node = int
end
```

To control the abnormal dynamic behavior of graph operations some exceptions are introduced:

[1]This is a rather pragmatic approach; alternatively we can follow a different line of developing concepts and tools for managing related interfaces. However, this leads into a research direction of its own and distracts the focus from graph libraries.

```
signature GRAPH_EXCEPTIONS =
sig
  exception Node
  exception Edge
  exception NotImplemented
end
```

Exception `Node` is raised on trying to insert an already existing node into a graph, and exception `Edge` is raised when an edge cannot be inserted because one of its incident nodes is not in the graph. Finally, exception `NotImplemented` is typically used in functions that are not supported by a particular implementation.

Now the actual graph types come (currently) in two versions: (i) for (edge-) labeled graphs and (ii) for unlabeled graphs. First of all, `('a,'b) graph` denotes the type of graphs with node labels of type `'a` and edge labels of type `'b`. In addition, there are types describing outgoing edges (`out`) and the so-called *adjacency* of a node (`adj`). The *context* of a node is a four-tuple consisting of the incoming edges, the node itself, its label, and its outgoing edges. A *decomposition* (of a graph g) is a context c together with a (remaining) graph g' so that g can be obtained from g' by inserting the context c, that is, the node of c together with the incoming and outgoing edges. A *forward decomposition* is much like a decomposition except that only the outgoing edges are contained in it (the node itself is also omitted, only the node label is retained).

```
signature GRAPH_TYPES =
sig
  include GRAPH_NODE
  type ('a,'b) graph
  type 'b out              = ('b * node) list
  type ('a,'b) adj         = 'a * 'b out
  type ('a,'b) context     = 'b out * node * 'a * 'b out
  type ('a,'b) decomp      = ('a,'b) context * ('a,'b) graph
  type ('a,'b) fwd_decomp  = ('a,'b) adj * ('a,'b) graph
end
```

The types for unlabeled graphs are very similar except that the type parameter `'b` for edge labels is missing. (There is also no type `out`, since this simply reduces to `node list`.)

```
signature UNLAB_GRAPH_TYPES =
sig
  include GRAPH_NODE
  type 'a graph
  type 'a adj        = 'a * node list
  type 'a context    = node list * node * 'a * node list
  type 'a decomp     = 'a context * 'a graph
  type 'a fwd_decomp = 'a adj * 'a graph
end
```

An own type for unlabeled graphs is strictly not necessary, since unlabeled graphs can be considered as a special case of labeled graphs where the type of edge labels is, for example, `unit`. However, in addition to convenience, unlabeled graphs have more efficient implementations,

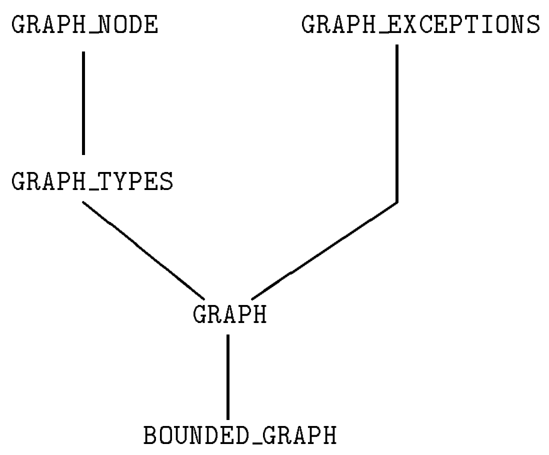

Figure 1: Graph Interfaces.

and it is therefore helpful to have unlabeled graph directly available.

The same is certainly true for undirected graphs, and interfaces and implementations for undirected graphs are among the most urgent extensions to this library.

The relationships between the different signatures are briefly summarized in Figure 1 (`GRAPH` and `BOUNDED_GRAPH` are explained in the next section.)

3 Operations on Graphs

An overview of all graph operations is given in Figure 2. Next we describe all functions in the following subsections.

3.1 Graph Composition and Decomposition

The operations `empty`, `embed`, and `match` offer an inductive view of graphs: `empty` denotes the empty graph, and `embed` inserts a node into a graph and adds edges to predecessors and successors in the graph as specified by the context argument. This means the result of the expression `embed ((p,v,l,s),g)` is a new graph which differs from `g` in two ways: (i) it contains node `v` with label `l`, and (ii) for each pair `(x,w)` in the predecessor list `p` (respectively, in the successor list `s`) it contains an edge `(w,v)` (respectively, an edge `(v,w)`) with edge label `x`. If the node is already contained in the graph, a `Node` exception is raised, and if any of the nodes specified as predecessors or successors is not in the graph, an `Edge` exception is raised.

The operation `match` retrieves a specific node with its label and its incoming and outgoing edges from a graph, it also returns the remaining graph, that is, the graph without that node and its incident edges. Hence, `match` is the (deterministic) inverse operation of `embed`. If the node is not contained in the graph, a `Match` exception is raised.

```
val empty : ('a,'b) graph
val embed : ('a,'b) context * ('a,'b) graph -> ('a,'b) graph
val match : node * ('a,'b) graph -> ('a,'b) decomp
```

There are some specialized versions of `match`: `matchFwd` returns only the outgoing edges

```
signature GRAPH =
sig
  include GRAPH_EXCEPTIONS
  include GRAPH_TYPES
  val empty       : ('a,'b) graph
  val embed       : ('a,'b) context * ('a,'b) graph -> ('a,'b) graph
  val match       : node * ('a,'b) graph -> ('a,'b) decomp
  val matchFwd    : node * ('a,'b) graph -> ('a,'b) fwd_decomp
  val matchAny    : ('a,'b) graph -> ('a,'b) decomp
  val matchAnyFwd : ('a,'b) graph -> ('a,'b) fwd_decomp
  val matchOrd    : node * ''b list * ''b list * ('a,''b) graph -> ('a,''b) decomp
  val matchOrdFwd : node * ''b list * ('a,''b) graph -> ('a,''b) fwd_decomp
  val context     : node * ('a,'b) graph -> ('a,'b) context
  val fwd         : node * ('a,'b) graph -> ('a,'b) adj
  val bwd         : node * ('a,'b) graph -> ('a,'b) adj
  val suc         : node * ('a,'b) graph -> node list
  val pred        : node * ('a,'b) graph -> node list
  val ufold       : (('a,'b) context * 'c -> 'c) -> 'c -> ('a,'b) graph -> 'c
  val gfold       : (('a,'b) context -> node list) -> ('a * 'c -> 'd) ->
                    ('d * 'c -> 'c) -> 'c -> node list -> ('a,'b) graph -> 'c
  val nodes       : ('a,'b) graph -> node list
  val labNodes    : ('a,'b) graph -> (node * 'a) list
  val noNodes     : ('a,'b) graph -> int
  val isEmpty     : ('a,'b) graph -> bool
  val newNodes    : int -> ('a,'b) graph -> node list
  val mkgr        : 'a list * (node * node * 'b) list -> ('a,'b) graph
  val adj         : ('a,'b) graph -> (node * ('a,'b) adj) list
end
```

Figure 2: Operations on labeled graphs.

(and the node label without the node itself). This is important for efficiency reasons because if in a program access to predecessors is never needed, specialized and more efficient graph implementations supporting only successor information can be used. The operation `matchAny` is a non-deterministic variant of `match`: an arbitrary node is selected and decomposed from the graph; `matchAnyFwd` does the same as `matchAny`, but again just returns outgoing edges. The function `matchOrd` computes the same result as `match`, but returns the incoming and outgoing edges of the found context sorted by edge labels according to the (partial) order specified by the additional list arguments: assume that `matchOrd (v,op,os,g)` returns the context `((p,v,l,s),g')` where `(x,u)` and `(y,w)` are contained in `p` (respectively, `s`). Now if `x` appears before `y` in `op` (`os`), then `(x,u)` appears before `(y,w)` in `p` (`s`). Finally, `matchOrdFwd` works like `matchFwd` with additional sorting of the outgoing edges.

```
val matchFwd    : node * ('a,'b) graph -> ('a,'b) fwd_decomp
val matchAny    : ('a,'b) graph -> ('a,'b) decomp
val matchAnyFwd : ('a,'b) graph -> ('a,'b) fwd_decomp
```

```
val matchOrd    : node * ''b list * ''b list * ('a,''b) graph -> ('a,''b) decomp
val matchOrdFwd : node * ''b list * ('a,''b) graph -> ('a,''b) fwd_decomp
```

3.2 Graph Inspection

There are a couple of operations that just retrieve information from a graph without decomposing the graph itself: First, `context` finds the context of a specific node much like `match` does, but without returning the reduced graph. This is, in general, much more efficient than just computing `match` and than simply not using the graph result. The functions `fwd` and `bwd` return the label of a node together with its outgoing, respectively, incoming, edges, and `suc` and `pred` give just the list of successor, respectively, predecessor nodes.

```
val context : node * ('a,'b) graph -> ('a,'b) context
val fwd     : node * ('a,'b) graph -> ('a,'b) adj
val bwd     : node * ('a,'b) graph -> ('a,'b) adj
val suc     : node * ('a,'b) graph -> node list
val pred    : node * ('a,'b) graph -> node list
```

3.3 Graph Fold

With `embed` and `match` rather arbitrary recursions over graphs can be expressed. Instead the operators `ufold` and `gfold` encapsulate specific recursion schemes.

```
val ufold : (('a,'b) context * 'c -> 'c) -> 'c -> ('a,'b) graph -> 'c
val gfold : (('a,'b) context -> node list) -> ('a * 'c -> 'd) ->
             ('d * 'c -> 'c) -> 'c -> node list -> ('a,'b) graph -> 'c
```

The behavior of `ufold` is most easily specified as follows:

```
ufold f u empty         = u
ufold f u (embed (c,g)) = f (c,ufold f u g)
```

With `ufold` we can reverse, for example, the direction of all edges in a graph, map a function to all node labels, or test for node membership:

```
val rev    = ufold (fn ((p,v,l,s),g)=>embed ((s,v,l,p),g) empty
fun gmap f = ufold (fn ((p,v,l,s),g)=>embed ((p,v,f l,s),g) empty
fun memb v = ufold (fn ((_,w,_,_),b)=>v=w orelse b) false
```

Whereas `ufold` moves in an arbitrary order through the graph (the `u` stands for *u*nordered), `gfold` decomposes graphs in a specific order. When folding a data type value one always moves "forward" from the current constructor (node) to the contained values (that is, successors). In contrast, graphs also provide access to nodes' predecessors. So we are able to specify 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. Now fold operates on a node `v` in two steps: first, fold is recursively applied to `l` 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`. Since `gfold` consumes nodes in a single-threaded way (that is, once a node is visited, it is decomposed from the graph and cannot be visited again), we have to account for cases when a node is reached via two different predecessors. This is dealt with as follows: if a node is tried to be matched that has already been decomposed from the graph, a `Match` exception is raised, and this exception is handled in `gfold` by simply trying the next node. The implementation of `gfold` uses two auxiliary functions, `gfold1` and `gfoldn`, to fold from one particular node, respectively, from a list of nodes:

```
fun gfold1 f d b u v g =
    let val (c as (_,_,lab,_),g1) = match (v,g)
        val (r,g2) = gfoldn f d b u (f c) g1
     in
        (d (lab,r),g2)
    end
and gfoldn f d b 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. 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)
```

It is interesting to note that `gfold` essentially describes depth-first search on graphs. With

```
fwd (_,_,_,s) = s
```

`gfold fwd` performs a depth-first graph traversal where the remaining parameters determine what is to be computed during the search. For instance, with `CONS = op ::` and

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

a depth-first spanning forest is computed by

```
fun dfs l g = gfold fwd BRANCH CONS [] l g
```

3.4 Miscellaneous Graph Functions

The function `nodes` gives a list of all nodes contained in a graph, and `labNodes` gives the same list together with node labels. The operation `noNodes` yields the number of nodes in the graph, and `isEmpty` simply checks for the empty graph (actually: `isEmpty g` = `noNodes` g=0). An expression newNodes (n,g) generates a list of n new nodes, that is, a list of nodes that are *not* contained in g. The function mkgr takes a list of n node labels and a list of edges with labels and creates a graph where the node labels are assigned sequentially to the nodes $0..n-1$. Finally, adj computes for a graph a list of so-called *adjacencies*, that is, a list of nodes with their labels and all outgoing edges.

```
val nodes    : ('a,'b) graph -> node list
val labNodes : ('a,'b) graph -> (node * 'a) list
val noNodes  : ('a,'b) graph -> int
val isEmpty  : ('a,'b) graph -> bool
val newNodes : int -> ('a,'b) graph -> node list
val mkgr     : 'a list * (node * node * 'b) list -> ('a,'b) graph
val adj      : ('a,'b) graph -> (node * ('a,'b) adj) list
```

3.5 Specialized Graphs

From the general interface more specialized graph types and corresponding interfaces can be derived. One example is the type of unlabeled graphs which has the operations shown in Figure 3.

The meanings of the operations are the same as for labeled graphs. Of course, there are some functions which inherently depend on edge labels and which are thus missing in UNLAB_GRAPH, for example, matchOrd and matchOrdFwd.

In Section 4 we will describe, among others, array implementations of graphs. These require the size of a graph (in the sense of an upper bound to the number of nodes) being specified already during the creation of the graph. This means that the empty operation should get an additional int-parameter. However, in order to keep a high degree of "interface sharing" we introduce a new function emptyBound instead of changing the type of empty. Thus the interface is simply specified by:

```
signature BOUNDED_GRAPH =
sig
  include GRAPH
  val emptyBound : int -> ('a,'b) graph
end
```

Of course, the same can be done for unlabeled graphs:

```
signature UNLAB_BOUNDED_GRAPH =
sig
  include UNLAB_GRAPH
  val emptyBound : int -> 'a graph
end
```

There is also an array implementation that does not allow the creation of new nodes at all (there graphs have to be build using mkgr). For these the interface is actually the same as for usual graphs, however, some functions are simply not implemented. To document that the implemented type is different, though, we define a synonym for the general signatures:

```
signature STATIC_GRAPH = GRAPH
signature UNLAB_STATIC_GRAPH = UNLAB_GRAPH
```

```
signature UNLAB_GRAPH =
sig
  include UNLAB_GRAPH_TYPES
  include GRAPH_EXCEPTIONS
  val empty       : 'a graph
  val embed       : 'a context * 'a graph -> 'a graph
  val match       : node * 'a graph -> 'a decomp
  val matchFwd    : node * 'a graph -> 'a fwd_decomp
  val matchAny    : 'a graph -> 'a decomp
  val matchAnyFwd : 'a graph -> 'a fwd_decomp
  val context     : node * 'a graph -> 'a context
  val fwd         : node * 'a graph -> 'a adj
  val bwd         : node * 'a graph -> 'a adj
  val suc         : node * 'a graph -> node list
  val pred        : node * 'a graph -> node list
  val ufold       : ('a context * 'b -> 'b) -> 'b -> 'a graph -> 'b
  val gfold       : ('a context -> node list) -> ('a * 'b -> 'c) ->
                    ('c * 'b -> 'b) -> 'b -> node list -> 'a graph -> 'b
  val nodes       : 'a graph -> node list
  val labNodes    : 'a graph -> (node * 'a) list
  val noNodes     : 'a graph -> int
  val isEmpty     : 'a graph -> bool
  val newNodes    : int -> 'a graph -> node list
  val mkgr        : 'a list * (node * node) list -> 'a graph
  val adj         : 'a graph -> (node * 'a adj) list
end
```

Figure 3: Operations on unlabeled graphs.

4 Graph Implementations

The interfaces described in the previous section are realized by several different implementations. Common to all implementations is that a graph is always represented by an adjacency list structure. This is advantageous for many algorithms and, in particular, for sparse graphs. A matrix representation is not difficult to realize (and it will be certainly included in one of the next library versions), since we have already a functional array implementation that is used by one of the graph implementations.

Currently, there are three implementations for directed functional graphs available:

- Balanced Binary Search Trees
- Bounded Functional Arrays
- Static Functional Arrays

Details of the implementations and also some benchmark results concerning their efficiency in different applications are given in [1]. Here we simply list all the implementations that can be used, and we especially document the unimplemented parts of their interfaces.[2]

The following guidelines might be helpful to find the most appropriate implementation.

1. Choose only among implementations that provide all operations you need. (This is rather obvious.)

2. Among those that qualify choose the most specialized one, that is, prefer unlabeled over labeled graphs and forward over full context implementation.

3. The tree implementations are generally more versatile and also more reliable on large graphs whereas the array implementations are faster in some specific cases.

4. For more detailed informations on efficiency, look into [1].

In the following the "`Fwd`" appendix to a structure/functor name[3] is used for graph representations storing only successor information. These representations are generally more efficient, but on the other hand, they provide only a subset of operations.

4.1 Balanced Binary Search Tree Implementation

The tree representation is the most general and also the most stable and versatile implementation of functional graph. It is considered the "standard representation". The following structures are offered by the tree representation:

```
structure Graph: GRAPH
structure GraphFwd : GRAPH
structure UnlabGraph : UNLAB_GRAPH
structure UnlabGraphFwd : UNLAB_GRAPH
```

The following functions of interfaces `GRAPH`, respectively, `UNLAB_GRAPH`, are *not* implemented by `GraphFwd`, respectively, `UnlabGraphFwd`:

```
match          context    ufold
matchAny       bwd        gfold
matchOrd       pred
matchFwd
matchAnyFwd
matchOrdFwd
```

This is because the predecessor information is not available. Note that in order to decompose a node from a graph it is necessary to remove that node from the successor lists of all predecessors, and these can only be found in a forward representation by scanning all nodes, resulting in a poor performance. Therefore, neither of the functions `matchFwd`, `matchAnyFwd`, and `matchOrdFwd` is supplied.

[2] This is an important issue because we have decided to keep only a small number of interfaces which means that restrictions of specialized implementations cannot always be seen at the interface level, but have to be documented explicitly.

[3] For the rules of assigning (and understanding!) names of signatures, structures and functors, see the description of the naming scheme in the Appendix.

4.2 Bounded Functional Array Implementation

Currently, only one functor (the most general one) is offered by the bounded array implementation:

```
functor BoundedGraphArray (FunArray:FUN_ARRAY) : BOUNDED_GRAPH
```

Note that `empty` is not undefined, but denotes an empty graph that can be extended to up to 50 nodes, that is, `empty` = `emptyBounded 50`.

It is important to note that the representation is based on an appropriate realization of functional arrays. The library provides an implementation which has been specifically designed for representing graph adjacency lists:

```
structure CArray:FUN_ARRAY
```

This implementation uses array "caches" to speed up lookup operations (this is described in some detail in [2] and [1]). In addition, some specific array functions are needed by the graph functor. The functional arrays must provide the following 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 apply        : 'a array * int * ('a -> 'a) -> 'a array
  val firstIndex   : 'a array * ('a -> bool) -> int
  val toList       : ('a -> bool) * (int * 'a -> 'b) -> 'a array -> 'b list
  val toImpArray   : 'a array -> 'a Array.array
  val fromList     : 'a list -> 'a array
  val fromImpArray : 'a Array.array -> 'a array
end
```

4.3 Static Functional Array Implementation

Static graphs are supported by four implementations based on functional arrays:

```
functor StaticGraphArray (FunArray:FUN_ARRAY) : STATIC_GRAPH
functor StaticGraphArrayFwd (FunArray:FUN_ARRAY) : STATIC_GRAPH
functor UnlabStaticGraphArray (FunArray:FUN_ARRAY) : UNLAB_STATIC_GRAPH
functor UnlabStaticGraphArrayFwd (FunArray:FUN_ARRAY) : UNLAB_STATIC_GRAPH
```

The following functions of interfaces `STATIC_GRAPH`, respectively, `UNLAB_STATIC_GRAPH`, are *not* implemented by `StaticGraphArrayFwd`, respectively, `UnlabStaticGraphArrayFwd`:

```
match         context      ufold
matchAny      bwd          gfold
matchOrd      pred
```

As for the tree implementation the reason is that the predecessor information is not available. Note, however, that the functions `matchFwd`, `matchAnyFwd`, and `matchOrdFwd` *are* implemented. This is possible here, since node decomposition happens by an internal stamping technique that does not require to access predecessors.

Finally, not implemented by either functor is, of course, the function

```
newNodes
```

5 Applications

The fact that we have different basic graph implementations has a certain impact on the kind of interfaces for application operations. Instead of simply having one single signature with a type for each application function we need, in general, a functor for each application reflecting the dependency of the application operation on the underlying graph implementation. In particular, it is not sufficient to give just the "best" implementation (that is, based on the most specific applicable graph representation) because we might want to use an operation on a graph together with another operation that is not defined on that specialized graph representation.

5.1 Build

We have two functions to construct lists of edges that can be used to build graphs. The definitions are contained in the file

```
applic/build.sml
```

which defines a structure `Build` with the following two functions:

```
val edgeList      : int * int -> (int * int * unit) list
val unlabEdgeList : int * int -> (int * int) list
```

The expression `edgeList (n,d)` simply builds a list of $n \times d$ edges with edge label `():unit`. The node identifiers are in the range $0..n-1$ and each node has d outgoing edges. `unlabEdgeList (n,d)` has the same effect, but produces unlabeled edges. The edge lists can be used by the function `mkgr` to construct regular graphs of a certain density.

These functions have been used to generate example graphs for benchmarking the different graph implementations.

5.2 Clique

Functions to construct complete (labeled) graphs are contained in the file

```
applic/clique.sml
```

which defines the structure `Clique` with the functions:

```
val clique  : int -> (unit,unit) graph
val cliqueA : int -> (unit,unit) graph
```

The function `clique` is based on the tree implementation, and the function `cliqueA` is based on the bounded array implementation. The latter is possible, since the argument of clique can be well used as the upper bound on the graph size.

These functions have been used to test the efficiency of the `embed` graph constructor.

5.3 DFS

Currently, there are direct implementations of depth-first search for all kinds of graph types available. The definitions are contained in the file

```
applic/dfs.sml
```

We have used depth-first search especially to construct depth-first spanning trees (that is, forests). The type for these are included in a structure:

```
structure DFSTree =
struct
  datatype tree = BRANCH of int * tree list
end
```

The implementation of the function `dfs` is given by two functors:

```
functor DFS (G:GRAPH)
functor DFSu (G:UNLAB_GRAPH)
```

Each functor defines the function `dfs` with a slightly different type:

```
val dfs : ('a,'b) graph -> tree list   (* in DFS  *)
val dfs : 'a graph -> tree list        (* in DFSu *)
```

Future extensions should include operations that are derived from `dfs`, see for example, [3].

5.4 Independent Node Sets

The file

```
applic/indep.sml
```

defines the structure `Indep` that contains a function to compute a maximal independent node set.

```
val indep : ('a,'b) graph -> node list
```

The algorithm used to implement `indep` uses graphs in a non-single-threaded way. This functions has been used to benchmark the persistent behavior of different graph implementations.

5.5 Simple Paths

An example of how to program graph backtracking is provided in the file

```
applic/paths.sml
```

where the functor `Paths` defines a function to compute all simple paths in a graph.

```
val paths : ('a,'b) graph -> node list list
```

5.6 Reversing Edges

The file

```
applic/rev.sml
```

defines three functors `REV`, `REVu`, and `REVd` that define different implementations of the reverse function

```
val rev : ('a,'b) graph -> ('a,'b) graph   , respectively
val rev : 'a graph -> 'a graph
```

5.7 Combinator Graph Reduction

Graph reduction shows a slightly larger application of the graph library. The graph reducer is defined in a functor `GrRed` in file

```
applic/red.sml
```

In addition, there are some utilities to translate lambda terms into combinator expressions and graphs representing them. These are contained in the file

```
applic/lamtr.sml
```

In order to work (actually, play) with the graph reducer one can write lambda expressions as terms of the data type `lambda` (defined in structure `LambdaCalc` in file `lamtr.sml`) and translate these into graphs with the function `toGraph` (defined in functor `LambdaTrans` in file `lamtr.sml`). Evaluation of these graphs can then be done with the function

```
val eval : node -> (node_label,edge_label) graph ->
           value * (node_label,edge_label) graph
```

This means that the evaluation of a graph `g` starting at root `r` is obtained by the expression `graph r g`. The result is a value and the reduced graph. The types `value`, `node_label`, and `edge_label` are defined in the structure `Value` in file `red.sml`.

5.8 No Shortest Paths Yet ...

It is not difficult to give implementations of other standard graph algorithms, like shortest paths or minimum spanning trees. We have deliberately omitted such functions from the library, since a direct translation of well-known algorithms, such as Dijkstra's or Prim's, would re-introduce an imperative style into FGL, which we are trying to avoid.

Functional descriptions of those algorithms are currently under development and will certainly be included in future versions of FGL.

6 To Do

There remain lots of things to be done and improved in FGL, and the following list is not meant to be complete.

1. *Define more graph types.* The most important example is probably a type for undirected graphs.
2. *Include more graph representations.* In particular, a matrix representation, which is suited well for dense graphs, is missing at the moment.
3. *Make the implementation correct or adjust the semantics.* This relates to the fact that the semantics given in [2] require that self-loops are reported only as successors. However, in the current implementation loops are actually reported twice because scanning each returned predecessor list to eliminate loops would require too much work compared to the rare case of self-loops. Although there are reasons to have the semantics as they are defined, adapting the implementations to give this behavior makes the functional graph structures too inefficient. It is not clear right now how to proceed.
4. *Define much more algorithms and applications.*
5. *Translate FGL to Haskell/HUGS.*
6. *Implement a graphical user interface.* It is very convenient to construct and to view graphs in a graphical way.

Suggestions to the above list and, of course, also in general are welcome. It would be great to include contributions from other people.

References

[1] M. Erwig. Fully Persistent Graphs – Which One to Choose? In *9th Int. Workshop on Implementation of Functional Languages*, pages 101–118, 1997.

[2] M. Erwig. Functional Programming with Graphs. In *2nd ACM SIGPLAN Int. Conf. on Functional Programming*, pages 52–65, 1997.

[3] 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.

[4] K. Mehlhorn and S. Näher. LEDA: A Platform for Combinatorial and Geometric Computing. *Communications of the ACM*, 38:69–102, 1995.

[5] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, Reading, MA, 1996.

[6] S. Näher and C. Uhrig. The LEDA User Manual, Version R 3.4. Technical report, Max Planck Institute, Saarbrücken, Germany, 1996.

[7] K. Weihe. Reuse of Algorithms: Still a Challange to Object-Oriented Programming. In *12th ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 34–48, 1997.

Appendix: Naming Scheme

We briefly describe here a general naming scheme for ML signatures, structures, and functors. The goals for inventing and adopting a naming scheme are:

- A disciplined and structured way of using names aids in keeping control over a large name set.
- Having defaults allows users to start using general modules (with simple/plain names) and go on with specialized modules later.

General Structure of Signature Names

First we adopt the conventions from the SML'97 Standard Library: signatures are written in capital letters separating different words by underscores. In addition, we suggest a "semantic" structuring of signatures. The general form of a signature name is:

(*modifier_*)**sig*

where a *modifier* describes a specialized signature. Certain modifiers are complementary to each other, and a set of such modifiers can be grouped into a class. We can then identify one default modifier for each such class which is to be assumed if no modifier of that class is contained in the name.

In the case of graphs we have, for example, the following modifier classes (default modifiers listed first):

Edge-Type	`DIR` (directed), `UNDIR` (undirected)
Edge-Labels	`LAB` (labeled), `UNLAB` (unlabeled)
Dynamics	`DYN` (dynamic), `BOUNDED` (bounded size), `STATIC` (fixed size)

Examples signature names are:

`GRAPH`	(directed, labeled, dynamic graphs)
`UNLAB_GRAPH`	(directed, unlabeled, dynamic graphs)
`STATIC_GRAPH`	(directed, labeled, fixed-sized graphs)

(From the default conventions it follows that we should never have signature names like `LAB_GRAPH` or `DIR_DYN_GRAPH`.)

Splitting and Sharing of Signature Names

In order to allow for sharing of specifications, parts of a signature might be given in separate places. We suggest to split, if possible, by the kind of names specified (types, exceptions, values). For example,

`GRAPH_TYPES`	(denotes type definitions)
`GRAPH_EXCEPTION`	(denotes exception definitions)

If part of such a section is to be shared and thus needs a signature of its own, or if a "semantic separation" makes sense (that is, take several definitions of different name kinds), use the corresponding defined name (in capital letters) as the appended part of the signature, for example,

`GRAPH_NODE`	(contains definitions about nodes)

Names of Structures and Functors

In general, there are different structures/functors implementing one signature. We assume that there is a preferred or default implementation of each signature. The name of that structure/functor is essentially the same as the signature (up to the SML naming conventions, that is, structure names are written with a leading capital letter and without underscores). However, there might be different ways of implementing one interface, and there might also be specialized implementations that support only special and restricted usages of a type (for example, forward-graphs) or that are more efficient in specific situations. These informations should be given as appendices to the "main" structure/functor name.

Examples of implementation specific details (for graphs):

Structure	`Tree` (balanced binary search tree), `Array` (functional array)
Access	`Full` (successors and predecessors), `Fwd` (forward = only successors)

Thus some names for graph implementations are:

`Graph`	(directed, labeled, dynamic graphs; tree structure, full access)
`UnlabGraphFwd`	(directed, unlabeled, dynamic graphs; tree structure, only successors)
`BoundedGraphArray`	(directed, labeled, bounded graphs; array structure, full access)

Additional Remarks

Of course, a naming scheme is by no means unique, since specialized implementations often only support limited access, that is, they can raise a new interface. Likewise different interfaces give opportunities to specialized implementation. Thus it is not always obvious where and when to introduce new names or whether to better omit them.

We found the following general strategy to be helpful.

1. Try to keep the number of interfaces (and names) small.

2. Rather raise `NotImplemented` in functions of specialized implementations than to introduce a whole new interface.

3. Better add a new function with a different name when the type of a function changes than introduce a whole new interface with just that function's type changed.