Graph Algorithms = Iteration + Data Structures?

The Structure of Graph Algorithms and a Corresponding Style of Programming[†]

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

Abstract

We encourage a specific view of graph algorithms, which can be paraphrased by "iterate over the graph elements in a specific order and perform computations in each step". Data structures will be used to define the processing order, and we will introduce recursive mapping/array definitions as a vehicle for specifying the desired computations. Being concerned with the problem of implementing graph algorithms, we outline the extension of a functional programming language by graph types and operations. In particular, we explicate an exploration operator that essentially embodies the proposed view of algorithms. Fortunately, the resulting specifications of algorithms, in addition to being compact and declarative, are expected to have an almost as efficient implementation as their imperative counterparts.

1 Introduction

Looking at graph algorithms, we observe that many of them have a very similar structure, namely iterating over nodes or edges and thereby performing computations. Since graph algorithms are ubiquitous in almost all areas of computer science it is quite natural to ask for a programming language which takes advantage of this structure by, for instance, offering particular language features.

To our knowledge, only few approaches to specialized graph programming languages exist, however: GRAMAS [12] essentially provides an ALGOL-like language, and GRAPL [10] is primarily designed for use with dynamic algorithms, that is, algorithms that change the underlying graph. A survey of early approaches can be found in [11]. Most of these languages more or less provide means to let programs look very similar to the graph algorithms they implement. Notations for expressing algorithms in a rather traditional way are also introduced in many books on graph algorithms, for example, [16, 5]. Even though SETL [13] introduces finite sets and maps as abstractions, it is intended to be a prototyping language for general purpose algorithms and large software systems and pays no special attention to graph algorithms at all. None of these languages really advances by presenting high level operations utilizing the similar structure claimed above.

From the language designer's point of view, we search for general schemes from which particular algorithms can be instantiated by "filling in the slots". One well-known example for such a scheme are *closed semirings* [9] which can be used to compute, for instance, all-pairs

[†]Appeared slightly modified in the *Proceedings of the 18th International Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 657, Springer, 1992.

shortest paths or transitive closures. Actually, this is offered by G^+ [4], a database query language restricted to the search and aggregation of paths by pattern matching. Another scheme will be identified in this paper, which can be used to express a large class of algorithms, including depth- and breadth-first search, the minimum spanning tree algorithms of Prim and Kruskal, and Dijkstra's shortest path algorithm.

Attacking graph algorithms with functional programming is somewhat extraordinary (mostly for reasons of efficiency), but recently some new, encouraging aspects have been revealed [8, 1]. There, however, motivation comes from peculiarities of functional programming, and though the applications to functional graph algorithms are certainly nice, they are rather sporadic and limited.

The purpose of this paper is, from a graph-theoretic point of view, to propose a specific view of a reasonably large class of algorithms (Section 2) and to derive a very powerful and descriptive operation from it, called *graph exploration* (Section 3). Then in Section 4 we describe the layout of a functional programming language which embodies exploration and several other features making it well-suited to succinctly expressing graph algorithms. Our contribution to functional programming is to provide a more conceptual view of arrays by mappings and to introduce a new paradigm for array definitions, namely *iteration approximation* of *recursive array definitions*. Summarizing, we shall show that functional programming is definitely appropriate for efficient implementations of graph algorithms. A short discussion follows in Section 5.

2 Main Idea

We claim that many basic graph algorithms can be characterized by the following scheme:

Visit all or some elements of the graph (that is, nodes and edges)
in a specific order and perform computations meanwhile.

There are, of course, a lot of graph algorithms that are not directly captured by that view, but we shall demonstrate that when being combined with a functional programming environment, it covers quite many interesting applications.

Next we work out the parameters of the above scheme to arrive at a concrete language construct: Given (one or more) start elements in the graph we basically need a description of (i) how to get to other elements, (ii) the order in which the elements are to be processed, and (iii) the computations to be performed.

Parameter (i) is called *expansion*. An expansion is given by an expression of the language and denotes, in general, a sequence of graph elements.

The *processing order* of graph elements is crucial to many algorithms and in fact causes some problems that do not arise when, for example, processing lists or trees: In working on a list/tree, the order of accessing the yet unvisited elements can be described by a rule involving the current element and the remaining list/subtrees (see, for example, the definitions of map and postorder in Section 4.1.3). However, being faced with the task of processing nodes in a graph we observe that nodes may have a varying number of, say, successors, and since an algorithm fitting the above scheme works on a single node at a time we have to provide a kind of buffer for nodes yielded by expansion. Now, such a buffer can be realized by a *data structure*, and if we additionally devise specific operations for inserting and retrieving/removing elements from the structure, we obtain a determinate processing order. Data structures to be used in the following are special kinds of stacks, queues, and heaps.

A *computation* yields a value which is often associated with the graph element currently being processed. These values are available in following steps. The output of an algorithm may be some or all of the computed values as well as the processing order of nodes and edges.

When iterating over a graph, we obtain (implicitly or explicitly) a sequence of traversed edges. Mostly, an additional constraint is put on these edges, namely that these edges have to form a tree. With such a constraint the corresponding iteration will be called *tree exploration* in contrast to the more general case of *graph exploration*.

3 Exploration Paradigm and Data Structures

Explorations will be defined by two functions **explore** and **exploreG** for which we specify the slots carried out above, that is, data structure, expansion, and computation. Note that **explore** denotes tree exploration whereas **exploreG** stands for graph exploration. In general, an exploration produces a set of (local) definitions which can be used in subsequent calculations.

The data structures have associated two designated operations *get* and *put* for taking a single element from and inserting multiple elements into the data structure, respectively. A *get* operation obtains a so-called *current graph element* and removes it from the data structure. The expansion expression is applied to the current graph element and yields a sequence which is inserted into the data structure by the *put* operation. Normally, these operations are executed implicitly by the exploration mechanism, but for the purpose of initialization we may use the *put* operation explicitly. There we also may apply it to a single element (instead of a sequence). We overload the *cons* operator for lists (:) to denote all *put* operations. Ambiguities can always be resolved from the context.

Let us consider a small example. In breadth-first search, the successors of a node are to be visited after its siblings. This behaviour can be realized by using a queue to which the successors of a visited node are appended and from which the front node is taken to be visited next.

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

The details of the syntax will be explained later. Here we assume that **explore** gets each node at most once. We call this *single-get behaviour*, which can be imagined as follows: The *get* operation returns an element only if it was not returned by a previous call of *get*; the *get*-call is repeated until a "new" element is found or the data structure is empty. Thus in the above example, each node[1] is returned at most once from the front of the queue though it may be appended many times. v:Queue means that the argument v is initially appended to the queue used during the exploration. The expansion is described by the function suc which yields for the current graph element the sequence of its successors. Note that in this example computations are not needed; nevertheless, the definition of bfs is not for nothing since exploration gives always some implicit results. For example, we can find a shortest path (measured in number of edges) from a node v to a node w by

```
⟨v..w⟩.tree.bfs v
```

The function tree denotes the implicitly computed tree of traversed edges, and the angle brackets select a specific path from a tree (see Section 4.3.1).

[1]The type can be inferred from the fact that suc yields a sequence of nodes. We do not go into further details of type inference here.

Table 1: Behavioural aspects of data structures and operators

get/put				
single-put	multiple-put			
		multiple-get		
single-get	single-get	via each edge ≤ 1	not-edge-limited	*DS-behaviour*
bfs, Prim, Kruskal	dfs, Dijkstra	subgraph, sp-reconstruction	Moore	*Algorithm*
Tree/Forest		Graph	Multi-Graph	*Search Structure*
explore		**exploreG**	**exploreM**	*Operator*

Like computations, expansion expressions are sometimes not needed either. If the order of accessing graph elements is known prior to the exploration, a corresponding sequence can be passed directly to the exploration. Consider, for instance, Kruskal's algorithm for computing minimum spanning trees:

kruskal = **explore** (sort cost E):Queue

The expression in parentheses computes a sequence of edges which is ordered according to **cost**, a function assumed to be defined on edges. This sequence is appended to a queue. The actual exploration simply takes the edges from the queue and performs no append at all. Apparently, in such cases the data structure could as well have been omitted.

For the two examples from above single-get behaviour and tree exploration was appropriate. Yet, nodes and edges may be even restricted to be put at most once into the data structure (which is a stronger restriction and, of course, implies single-get behaviour). There are, however, also cases, for instance, Dijkstra's algorithm, where single-get behaviour is required and where at the same time nodes or edges must be allowed to be put multiple times into the data structure. Both single-get derivatives are covered by the **explore** operator. Obviously, this is not sufficient for all imaginable kinds of explorations: For example, when exploration has to compute a subgraph, a graph structure must be built instead of a tree, and accordingly, nodes may be got more than once from the data structure (in the sense that a node may be reached via each incident edge at most once). This behaviour is supplied by the **exploreG** operator. There are even more general cases conceivable where a node, for example, may be reached via each edge multiple times. For that we can devise an **exploreM** operator, but then other behavioural aspects of data structures/explorations have to be taken into account, too, which we will not do in this paper. A summary of the preceding discussion is sketched in Table 1.

Finally, note that we deal with both directed and undirected graphs, and we shall assume in all examples that the underlying graph is alternatively directed or undirected, whichever is appropriate in the context.

4 GEL — A Functional Language with Graph Explorations

Functional languages are widely accepted, since they allow for a declarative style of programming. Hudak gives an overview of the main concepts [7], and Bird and Wadler provide a thorough introduction to functional programming in the language Miranda[2] [2].

In this section we show how to integrate the idea of exploration into a functional programming language. Our description falls into three divisions: Section 4.1 outlines the basic concepts of the **G**raph **E**xploration **L**anguage (GEL). This includes list comprehensions and mapping definitions. The exploration operators are explicated in Section 4.2. Along with that exposition, several examples for specifications of graph algorithms will be presented. In Section 4.3, we describe how to extract and how to combine information computed by the explorations.

4.1 Basic Language Elements

Our language is designed in the style of Miranda [17, 2]: GEL is a strongly-typed, higher-order functional programming language with lazy semantics. Due to limitation of space we just describe those features needed subsequently: We explain how to define and how to use functions, and we describe the concept of list comprehensions. Actually, the exploration operator will be defined in a fashion similar to list comprehensions. In addition, some graph relevant types and primitive functions are outlined, and the concept of mappings, that is, extensionally defined functions, is introduced.

4.1.1 Expressions and Functions

Expressions are used to denote values. An expression is either a function or a function application. In this paper functions are defined exclusively by simple pattern matching, that is, the different cases of the definition are listed one below another, as in[3]

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

Local definitions may be introduced by means of a **where** expression; an example will be given below. In general, function application is denoted by juxtaposition; it is left associative and has highest precedence (after composition). An exception to that rule are certain binary operators, such as + or *, which are written infix. Note that functions may be applied partially. Since functions are first-class citizens nested function applications have to be grouped appropriately by parentheses to indicate the proper arguments of functions. Annoying parentheses explosion, however, can be circumvented by the (associative) composition operator which is defined as follows:

```
f.g x = f (g x)
```

(Note that we define composition to have higher precedence than function application.) Thus a cascade of function calls can be rewritten in a convenient way, for example,

```
reverse (postorder (tree (dfs N))) = reverse.postorder.tree.dfs N
```

[2] Miranda is a trademark of Research Software Ltd.

[3] Examples for patterns involving type constructors are presented below.

4.1.2 Graph Types and Operations

In addition to atomic types like numbers or characters we need special types for graph objects, that is, nodes, edges, paths, trees, and graphs. If not specified otherwise, all expressions working on graphs are performed on a "current" graph G with node set N and edge set E. An edge is represented by a pair of nodes, and a path is equivalent to a sequence of edges. For convenience we shall assume that a forest is represented by a tree consisting of a dummy root and the trees of the forest as subtrees and can thus be treated like a tree. Note that functions defined on graphs may as well be applied to trees, and functions defined on trees are inherited by paths.

The functions suc, pred, out, and in yield for a node the successor nodes, predecessor nodes, outgoing edges, and incoming edges, respectively. Applied to an edge e = (a, b) they return the suc/out values of b, respectively the pred/in values of a. Note that in undirected graphs suc coincides with pred and in coincides with out.

reverse is a general purpose operation which reverses all edges in a graph (and by inheritance in a tree and a path, too). Applied to a path, reverse also reverses the order of edges in the corresponding sequence representation. Applied to a sequence, reverse reverses the order of the elements.

We assume that graphs together with the functions reflecting their structure (such as suc) are predefined, that is, we do not consider so-called *dynamic algorithms* where graphs are changed by operations during their run for this contradicts the functional programming style and would destroy referential transparency. Nevertheless, as we will see, it is possible to compute derived graphs which can be used in further calculations.

4.1.3 Constructed Types and Pattern Matching

When we introduced function definitions we already made use of a very simple form of pattern matching. For the definition of functions on constructed types, such as lists or trees, it is very convenient to use pattern matching for exploiting the knowledge about the structure of the arguments. For lists, we have the patterns [ ] (empty list) and a:b (list consisting of head a and tail, that is, rest sequence, b). Now the function map, which applies a function to each element of a sequence and returns the sequence of results, can succinctly be defined as follows:

```
map f [ ] = [ ]
map f (a:b) = f a : map f b
```

For trees, we have the patterns • (empty tree) and r◁st (tree with root r and a sequence of subtrees st). A traversal of a tree, returning a list of its nodes in postorder, can be defined by

```
postorder • = [ ]
postorder (r◁st) = foldr ++ [r] (map postorder st)
```

Note that ++ denotes list concatenation and that [r] is the sequence consisting of the single element r. The expression foldr op unit s reduces the sequence s in a right associative way by repeated application of the binary function op whith unit as a start element, for example, foldr op unit [a, b, c] = a op (b op (c op unit)). So in the above definition the postorder sequences of nodes obtained from the subtrees are all concatenated, and the root of the tree is appended as the last element.

4.1.4 Sequences and List Comprehensions

A simple way to define a sequence of values is to specify a subrange or to use an identifier to which a sequence is bound (for example, N denotes the sequence of nodes of the current graph). Furthermore, we make use of *list comprehensions* [17] to define sequences. Consider the set of odd squares defined by $\{n^2 \mid n \in \{1, \ldots, 100\} \wedge n \text{ is odd}\}$ using common notation of set theory. This can be expressed by a list comprehension as:

```
[n*n | n ← [1..100]; odd n]
```

The first expression describes the results of the sequence, and the phrases following the bar are called *qualifiers*. In the example, the first qualifier is a *generator* producing values which are subject to restriction by the second qualifier, which is a *filter*.

4.1.5 Mappings

Mappings are functions that are extensionally defined by a sequence of value-pairs.[4] To keep referential transparency mappings have to be defined in one place and do not change over time, that is, like functions they always denote the same value. Several means to define mappings in a functional setting are discussed by Steele and Hillis [15], Wadler [18], and Hudak [6]. A new way is, of course, disclosed by exploration. This will be demonstrated soon. Basically, in our framework a mapping can be defined by several equations which may include syntactically sugared comprehensions, as in

```
mp f 0 = 1
   f n = n*n | n ← [1..100]
```

In case of multiple definitions for one and the same argument the last one is taken as the definition. We define mapping definitions to be strict, that is, the defining expressions are evaluated instantly so that the mapping's bindings are immediately available afterwards.

Note that mappings must be defined on atomic types. If generators are given by a name, say, A[5] or B, and the body which defines mapping values is a constant or the identity function, we allow an abbreviated form of definition:

```
mp f a = 1 | a ← A
   f b = b | b ← B
```

can be written as

```
mp f A = 1
   f B = B
```

An essential part of functional programming is the possibility to define functions by recursive definitions. The semantics is defined either by reduction rules or by the least fixed point which can be approximated by iteratively applying the recursive definition to an already obtained approximation. Now we propose recursive definitions also for mappings where the semantics of such a recursive definition is defined to be an operator μ which maps sequences of named values (that is, environments) to mappings. For an example consider the following definition:

[4] Mappings essentially embody much the same concept as arrays — but on a more conceptual level. For example, programming gets easier with mappings for they can be applied directly like functions.

[5] We use capital letters to indicate sequence-valued variables.

```
[i ← getvalues]
with h i  = h i + 1
     h ⊥ = 0
```

The first line is a generator (as used in usual list comprehension) defining a sequence of values each of which is named i (**getvalues** is assumed to be defined elsewhere). We call a list comprehension consisting only of qualifiers an *environment comprehension*. The second line tells to increase h by one at each occurrence of i, and the third line defines h to yield 0 when being undefined (this is nothing but an initialization expression). The meaning of the specification is the mapping h obtained by iteratively applying the definition to the environment consisting of each of the named values augmented by the current approximation of h. So h counts the occurrences of values in a sequence, and the definition is just another solution to the well-known histogram problem [15, 18, 7, 1].

As a more sophisticated example we show how to express closed semiring applications with recursive mapping definitions. According to Mehlhorn [9], a closed semiring is a set S with $\{0,1\} \in S$ and $\oplus, \otimes : S \times S \to S$ where $(S, \oplus, 0)$ is a commutative monoid, $(S, \otimes, 1)$ is a monoid, $\otimes$ distributes over $\oplus$, 0 is a null-element w.r.t. $\otimes$, infinite sums exist, commutativity holds for infinite sums, and $\otimes$ distributes over infinite sums. Moreover, * denotes the closure operation which is defined by $\forall s \in S : s^\star = \sum_{i \geq 1} s^i$. Given a graph $G = (N, E)$ and a labelling of edges $c : E \to S$, Kleene's algorithm for solving general path problems can be realized by applying appropriate mapping definitions to a sequence of triples built using an environment comprehension:

```
[k ← [0..n]; i ← N; j ← N] with
     a k i j = if k=0 then (if (i, j) ∈ E then c (i, j) else 0)
          else if (i=j and j=k) then a k i j ⊕ 1
          else a (k-1) i j ⊕ (a (k-1) i k ⊗ (a (k-1) k k)* ⊗ a (k-1) k j)
mp d i j = a 0 i j | i ← N; j ← N
```

For the above definition to work we must assume n = |N| and that integers can be treated like nodes. Then it is clear that the all pairs shortest path problem can be solved by choosing $0 = \infty$, $1 = 0$, $\oplus$ = min, and $\otimes = +$ leaving the costs in mapping d. Transitive closure is treated accordingly. Of course, this solution is not very space efficient (needing $O(n^3)$ space), but we just wanted to give another example for recursive mapping definitions. Anyhow, in the spirit of this paper it would be more appropriate to create an own operator for semiring applications.

4.2 Specifying Graph Algorithms by Explorations

We have already seen that the exploration mechanism relies on appropriate data structures with *get* and *put* operations. Here we consider the data structures *stack*, *queue*, and *heap* with the respective (*get*, *put*) operations (*top*, *push*), (*front*, *append*), and (*min*, *insert*). Note that *top*, *front*, and *min* perform an implicit *pop*, *dequeue*, and *deletemin*, respectively.

The **explore** operation is defined in a style very similar to list comprehensions, that is, the data structure and expansion parameters of **explore** can be viewed as generators which yield values that can successively be bound to variables (and may be subject to further restriction by appropriate filters). In the bfs example of Section 3 bindings were not required, but if we, for example, want to consider only edges with a certain property, we eventually have to refer to the current and expanded nodes.

```
bfs' v = explore a ← v:Queue; b ← suc; cost a b < 100
```

If, as in the above example, the exploration operator is used within a function definition, the argument to the function being defined always constitutes an implicit binding. The bindings of a and b can operationally be regarded as resulting from a nested for-loop where a is taken from the queue and b is successively bound to the values of suc a. Considering the innermost loop, in each step a separate binding triple (v, a, b) is built. The filter given by the last expression refines the exploration in that only qualifying triples are considered. This means, that only qualifying nodes and edges are expanded, used in computations, and are put into the data structure. Likewise, only qualifying edges are traversed. In addition, a terminating condition can be attached to an exploration (by using the key word **until**) with the effect that exploration stops when the condition becomes true.

Next, look at the definition of depth-first search.

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

Comparing this definition with that of bfs, it becomes evident how exploration focuses on the very essentials of graph algorithms. This again facilitates the comparison of different algorithms. We feel that this characteristic together with the succinctness and clarity is a strong argument for the whole exploration approach. Furthermore, this feature may also be of educational use.

In Section 3 we already encountered Kruskal's algorithm. Instead of sorting edges explicitly we can delegate the work to a heap by initially inserting all edges into it:

```
kruskal = explore E:Heap(cost)
```

Note that the heap data structure needs a parameter function (defined on the items to be inserted into the heap) according to which the heap order is arranged. Prim's algorithm for computing minimum spanning trees extends a tree by selecting always the smallest edge incident to that tree. We begin by initializing a heap with an edge incident to the start node that has minimum cost among all outgoing edges.

```
prim v = explore e:Heap(cost); out
           where e = find min cost (out v)
```

The expression find agg f denotes a function which applies an aggregate function (agg) to a sequence and returns the first element of the sequence which is mapped by f to the aggregated value. In the above **where**-expression, an outgoing edge of v with minimum cost is computed.

4.2.1 Computations

Named, computed values for nodes and edges can as well be regarded as extensionally defined functions, that is, mappings. Adopting that view, each update performed in an iteration step approximates the final mapping a little bit, and the whole process of step-by-step computation is the operational description of another paradigm for mapping definitions. Hence we can view **explore** as a mapping operator applicable to mapping specifications. Here the data structure and the expansion expression serve the purpose of extracting a sequence of values out of the graph.

Consider, for example, the task of computing the level of each node, that is, the distance (measured in number of edges) from the start node. This is accomplished by a breadth-first search:

```
bfs" v =  explore  a ← v:Queue; b ← suc
            with level v = 0
                 level b = level a + 1
```

The key word **with** indicates that in addition to the information computed implicitly by **explore** (see Section 4.3) the specified mappings are also calculated. Since we know that within an exploration only mappings can be defined we omit the key word **mp**.

As an example for employing a heap data structure referring to a mapping which is approximated during the exploration itself consider Dijkstra's algorithm for computing a shortest path tree.

```
dijkstra v =  explore  a ← v:Heap(dist); b ← suc
                with dist v  = 0
                     dist b  = min (dist b) (dist a + cost a b)
                     dist ⊥ = ∞
```

Here, the parameter function of the heap is defined within the same exploration. This has two implications: First, changes in the mapping trigger reorganization activities. For example, decreasing dist for node b by δ has the effect of performing *decrease*(δ, b, Heap(dist)). In particular, this means that on a conceptual level we can ignore duplicates in the heap since multiple copies of the same item in one heap always have the same key. Second, note carefully that **explore** (viewed as a mapping definition operator) is *not* strict in its sequence (that is, environment) argument since the sequence may refer to mappings being calculated.

Looking at the definition for dist more closely, we observe that, except for the argument v, dist is initially totally undefined. So, in general, we would fail to evaluate dist b, and the whole iteration would not work. Of course, we can circumvent this problem by providing an initialization expression given by a default value for a mapping which is to be taken whenever that mapping is evaluated to $\perp$ (undefined).

4.2.2 Non-Tree Explorations

Until now we have only used tree explorations together with single-get behaviour. As an example for a graph exploration and multiple-get behaviour consider the computation of a subgraph spanned by a set of nodes M:

```
subgraph M = exploreG M:Stack; b ← suc; b ∈ M
```

The actual computation of the subgraph happens rather implicitly by traversing only edges the endnodes of which pass the membership test.

A similar example is the reconstruction of shortest paths between two nodes. Assume that we have the mapping dist available giving for each pair of nodes the length of a shortest path connecting the nodes. Since in general there may be more than one shortest path the result can be viewed as a subgraph consisting of all edges belonging to some shortest path.

```
paths v w = exploreG a ← v:Queue; b ← suc; dist a w = cost a b + dist b w
```

Note that in both examples the data structures only serve as buffers and that the processing orders are not important. This means that we could have interchanged Stack and Queue.

4.3 Using Explorations

We assume that in addition to mappings, a graph exploration computes the following information (called *aspects*): Node and edge sequence (in the order visited) and the search tree/forest (**explore**) or graph (**exploreG**). Aspects can be accessed by applying the respective functions nodes, edges, tree, forest, or graph to expressions containing the exploration defining them. Mappings are selected in the same way.

To make this clear, we note that explorations are in fact definitions. Thus applying an expression to an exploration is nothing but syntactic sugar. That is, instead of writing

```
let (nodes, edges, tree, f, g, ...) =
    explore ...with f = ... g = ...
in expr
```

we allow the more compact notation

```
expr (explore ...with f = ... g = ...)
```

Examples are given in the sequel.

4.3.1 Selections

To obtain a shortest path by means of the dijkstra exploration defined above we have to select a path from the search tree that has been built during the exploration. Therefore we define a very general *selection operation* ⟨a..b⟩ which, applied to a tree t, returns the (unique) path in t from node a to node b (only, of course, if it exists). By inheritance, selection also applies to paths. Moreover, it is convenient to allow selections even on lists; note that in this case a and b must be integers denoting proper positions in the list (positions are numbered starting with 1). We allow two special forms of selections: (i) ⟨a..⟩ asks for the subtree rooted at a or the subpath (subsequence) starting with (at position) a and (ii) ⟨a⟩ extracts the element a or, if a is an integer and ⟨a⟩ is applied to a sequence, the element at position a. Finally, the function last denotes the position of the last element of a sequence or a path, so the tail of a sequence can be denoted by ⟨2..last⟩ as well as by ⟨2..⟩. For sequences, a similar notation was chosen by Tarjan [16]. We illustrate selection by several examples. First, a shortest path can now be found by *path selection*:

```
⟨v..w⟩.tree.dijkstra v
```

The length of the shortest path is given by the dist value of w. In order to apply dist to w *element selection* is needed:

```
dist.⟨w⟩.tree.dijkstra v
```

Note that w could as well have been selected from the sequence of traversed nodes, that is, in the above expression, tree could have been interchanged with nodes.

An example for *sequence selection* is the search for the k next nodes to a given node v which have a certain property, say prop:

```
⟨1..k⟩ [a ← nodes.dijkstra v; prop a]
```

Here we made use of the fact that nodes yields the sequence of nodes in the order visited by Dijkstra's algorithm, that is, in monotonically increasing distance from the start node. This

example also shows that exploration is well-integrated into the functional language, which sometimes helps solving graph problems that are not directly expressible as explorations.

Subtree selection is used when asking, for example, for all nodes for which the shortest path from v leads via x:

```
⟨x..⟩.tree.dijkstra v
```

For computing the network center, we need the notion of a node's eccentricity, which is defined as the distance to the node that is farthest away [3]. Here again we use element selection:

```
eccentricity v = dist.⟨last⟩.nodes.dijkstra v
center = find min eccentricity N
```

The definition of center now finds for the sequence of nodes in the current graph a node with minimum eccentricity.

4.3.2 Repeated Application of Explorations

In the examples considered so far we have only used single elements as initial values for explorations. Next we demonstrate that collections of initial values are truly useful. Consider, for example, the definition of dfs. Applied to one node it may happen that only a subset of all nodes in the graph is reached (if, for instance, the graph G is not strongly connected). To obtain a complete spanning forest it is extremely helpful that we can apply dfs to N, the sequence of all nodes:

```
dfs N
```

Then, by definition of **explore**, N is pushed onto the stack used by dfs and we first get a tree rooted at the first node of N. When no more nodes are reachable, those nodes of N that are contained in the first tree are ignored and taken from the stack, and exploration continues with the next node not visited yet. This is repeated until all nodes are contained in a spanning tree. For this reason we allow arguments of function definitions to be as well single elements as sequences. Note that in this example the search tree being built is actually a forest.

For another application, consider the task of finding the k nodes having a distance as large as possible to a set of nodes P. (This is of use for a bank robber who seeks for a hideout shunning police stations.) We can achieve this by the following function definition:

```
farthest k P = ⟨last-k+1..last⟩.nodes.dijkstra P
```

Indeed, this works because the elements of P are all inserted into the heap, and because dist is initially set to 0 for all elements of P[6] and to the minimum distance to any element of P afterwards. Hence, the later nodes are expanded, the larger are their associated dist-values.

To consider a less criminal application, let H be a sequence of nodes at which hospitals are located. Then we can partition the set of nodes according to their nearest hospital. Recall that a forest is represented by a tree with a dummy root, so the desired partition is given by the subtrees of

```
forest.dijkstra H
```

[6] Here the abbreviations for mapping definitions with sequences introduced in Section 4.1.5 are really helpful.

4.3.3 Changing the Underlying Graph

Sometimes graph operations need to be performed on another than the default graph. This is done by simply providing a derived graph as an additional first argument which is introduced by the key word **ingraph**.

Consider, for instance, Sharir's algorithm for computing strong components [14]. We first have to compute a reversed postorder sequence of nodes visited by depth-first search. This sequence, denoted by M, is used as an initialization expression for a second run of dfs which is not performed on the default graph G but on the reverse graph, that is, the graph derived from G by reversing all edges.

```
forest.dfs ingraph (reverse G) M
    where M = ⟨2..⟩.reverse.postorder.forest.dfs N
```

Note that the selection $\langle 2..\rangle$ is needed to drop the dummy root from M. Finally, we obtain a forest of trees each of which represents a strong component.

Our last example is the once-around-the-minimum-spanning-tree approximation for the travelling salesman problem as described in [9]: Essentially, we have to perform a depth-first search on a minimum spanning tree. This is realized by a function for traversing a tree by revisiting a root r whenever a traversal of a subtree of r is completed.

```
walkaround (r◁[ ]) = [r]
walkaround (r◁(a:b)) = [r] ++ walkaround a ++ walkaround (r◁b)

tsp = walkaround.tree.kruskal
```

5 Discussion

We have shown how to express and how to apply graph algorithms in a functional language using a special operator tailored to graph iteration. The graph exploration scheme not only facilitates the succinct formulation of many well-known graph algorithms — it also reveals the main similarities and differences between them.

The concept of mappings in combination with **explore** as a mapping operator has lead to an applicative description of graph algorithms. This is an improvement especially for algorithms computing mappings, which are traditionally formulated in an imperative style.

The language itself is still under development. The major design issue is to effect a compromise between the size of the language (that is, the number of different concepts) and its expressiveness (and efficiency). We use topological sorting as an example to indicate how the introduction of new language features allows to express certain algorithms more appropriately (and efficiently). Topological sorting is defined by the following exploration (the topologically sorted list of nodes is obtained by the expression nodes.topsort):

```
zeroin = [n | n ← N; pred n = [ ]]

topsort = explore  a ← zeroin:Queue; b ← suc; indeg a = 0}
          with indeg b  = indeg b – 1
               indeg ⊥ = count.pred
```

Now, consider the graph describing the complete order $<$ on $\{1,\ldots,n\}$. If suc yields the successors in decreasing order, during a run of topsort up to $O(n^2)$ nodes will be removed

from the queue without being used in the computation. Although the program runs in time $O(n+m)$, it would be nicer (that is, more efficient) to insert only those nodes into the queue for which indeg is 0. In other words, what we need is a means to fix a condition to a specific part of exploration: If we were able, for instance, to add a modifier {PUT} to a condition meaning that only those elements are put into the data structure for which the condition is true, we could reformulate topological sorting by:

```
topsort = explore  a ← zeroin:Queue; b ← suc; {PUT indeg b = 0}
          with indeg b  = indeg b – 1
               indeg ⊥ = count.pred
```

The impact of the condition modifier {PUT} is to fix the condition on the specified part/action of the exploration cycle, here, the *put* operation. Modifiers such as {GET} or {MAP} are also conceivable. In combination with an **exploreM** operator for expressing Moore's shortest path algorithm, a {PUT} modifier is also useful to insert nodes into a queue only if not already present.

Another language feature under consideration is the use of default names for generators, for example, curr for the current graph element generated by data structures and next for the expanded graph elements. On the one hand, this may lead to an even more compact notation for explorations, on the other hand this facilitates "inheritance" of (optional) slots[7], that is, we can attach additional slots or mappings to applications of an exploration (for which those slots are not defined), for example:

```
shortest_path v w = ⟨v..w⟩.tree.dijkstra v until curr = w
bfs" v =  bfs v
          with level v    = 0
               level next = level curr + 1
```

As far as implementation is concerned, we note that mappings defined in explorations can be realized by arrays in the following way: First we can determine the types of the domain (for example, N) and range, and thus we can allocate memory accordingly. Then from the iterative style of explorations we observe that updates (induced by the mapping specification) are strongly serialized and thus can be performed in place. Assuming decent implementations of the data structures we conclude that programs in GEL can be almost as efficient as in imperative languages (disregarding union/find operations which are, in general, necessary to enforce tree constraints).

Moreover, there is potential for optimizations of applied explorations. For example, selecting a path from a tree computed by an exploration can be used to infer an additional, inherited **until** condition, for example,

```
⟨v..w⟩.tree.dijkstra v
```

can be transformed to

```
⟨v..w⟩.tree.dijkstra v until curr = w.
```

Of course, this would be of great use only for a version of GEL with eager evaluation.

[7] Slots are those parts of explorations which are introduced by a keyword.

Acknowledgements

Thanks to Ralf Hartmut Güting for encouraging me to work on this subject. Also, many thanks to him, to Bernd Meyer, Gerd Westerman, and Joe Lavinus for their comments on early language proposals.

References

[1] Barth, P.S., Nikhil, R.S., Arvind: M-Structures: Extending a Parallel, Non-strict, Functional Language with State, *Functional Programming and Computer Architecture*, LNCS 523, Springer-Verlag 1991, 538–566.

[2] Bird, R., Wadler, P.: *Introduction to Functional Programming*, Prentice Hall, 1988.

[3] Buckley, F., Harary, F.: *Distance in Graphs*, Addison-Wesley, 1989.

[4] Cruz, I.F., Mendelzon, A.O., Wood, P.T.: G^+: Recursive Queries Without Recursion, *Proc. 2nd Int. Conf. on Expert Database Systems*, 1988, 645–666.

[5] Ebert, J.: *Effiziente Graphenalgorithmen*, Akademische Verlagsgesellschaft, 1981.

[6] Hudak, P.: Arrays, Non-Determinism, Side-Effects, and Parallelism: A Functional Perspective, *Proc. Workshop on Graph Reduction*, LNCS 279, Springer-Verlag 1986, 312–327.

[7] Hudak, P.: Conceptions, Evolution, and Application of Functional Programming Languages, *ACM Computing Surveys, Vol. 21*, No. 3, 1989, 359–411.

[8] Kashiwagi, Y., Wise, D.S.: Graph Algorithms in a Lazy Functional Programming Language, *Proc. 4th Int. Symp. on Lucid and Intensional Programming*, 1991, 35–46.

[9] Mehlhorn, K.: *Data Structures and Algorithms 2*: *Graph Algorithms and NP-Completeness*, Springer-Verlag 1984.

[10] Nagl, M.: GRAPL — A Programming Language for Handling Dynamic Problems on Graphs, *Proc. 5th Int. Workshop on Graph Theoretic Concepts in Computer Science*, Hanser Verlag 1979, 25–45.

[11] Pape, U.: Graphen-Sprachen — Eine Übersicht, *Proc. 1st Int. Workshop on Graph Theoretic Concepts in Computer Science*, Hanser Verlag 1975, 11–27.

[12] Pape, U.: GRAMAS — A Graph Manipulation System, *Proc. 5th Int. Workshop on Graph Theoretic Concepts in Computer Science*, Hanser Verlag 1979, 47–63.

[13] Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., Schonberg, E.: *Programming with Sets — An Introduction to SETL*, Springer-Verlag, 1986.

[14] Sharir, M.: A Strong-Connectivity Algorithm and its Application in Data Flow Analysis, *Computers and Mathematics with Applications 7*, No. 1, 1981, 67–72.

[15] Steele, G.L. Jr. and Hillis, D.W.: Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing, *ACM Conf. on Lisp and Functional Programming*, 1986, 279–297.

[16] Tarjan, R.E.: *Data Structures and Network Algorithms*, SIAM, 1983.

[17] Turner, D. A.: Miranda: A Non-strict Functional Language with Polymorphic Types, *Functional Programming and Computer Architecture*, LNCS 201, Springer-Verlag 1985, 1–16.

[18] Wadler, P.: A New Array Operation, *Proc. Workshop on Graph Reduction*, LNCS 279, Springer-Verlag 1986, 328–335.