Compositional Graphs*

Sergio Antoy[1] Michael Hanus[2]

[1] Computer Science Department, Portland State University,
P.O. Box 751, Portland, OR 97207, U.S.A.
antoy@cs.pdx.edu

[2] Institut für Informatik, Christian-Albrechts-Universität Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. In many applications, graphs are a natural representation for the structure of a problem domain. Since functional languages supports only (tree-structured) algebraic datatypes, there are various ways to represent graphs in declarative languages. In this paper we describe a representation of graphs in a functional logic language. Our representation is compositional as lists or tree structures are. This property makes our representation useful for many application domains, like graphical user interfaces or web programming.

1 Introduction

Lists and trees are ubiquitous datatypes in functional and logic programming because of their simplicity. These datatypes are compositional, i.e., one can easily define operations to concatenate lists or join trees into a new tree without changing the structure of the arguments (which might not be the case for imperative languages where such data structures are implemented with pointers). This property often simplifies the reasoning on programs.

In many application areas, the use of these simple datatypes leads to unnatural models of a problem resulting in error-prone programs that are difficult to maintain. For instance, graphical user interfaces can be considered tree-like structures since widgets can be grouped in containers that are used as widgets themselves. However, these structures may include dependencies among each other, e.g., a button widget may manipulate another widget in a different hierarchy. Thus, a graph structure is a more appropriate model in this situation.

In order to make our ideas more concrete, consider a standard representation of simple graphs as an algebraic type. For instance, in Haskell [17] a graph can be defined as a pair consisting of nodes and edges as follows:

```
data Graph = Graph [Node] [Edge]
```

* This research has been partially supported by the DAAD/NSF grant INT-9981317, the German Research Council (DFG) grant Ha 2457/1-2 and the NSF grant CCR-0110496.

Edges consist (at a minimum) of a source and a target node, i.e., we need a unique identification of nodes in order to specify the edges between them. If we identify nodes by unique integers, we obtain:

```
data Node = Node Int

data Edge = Edge Int Int
```

Depending on the application, additional information items are included into both nodes and edges, e.g., lengths of edges, names of nodes, etc., that we omit for the sake of clarity. Now we can define a simple graph instance as follows:

```
g1 = Graph [Node 1, Node 2, Node 3]
           [Edge 1 2, Edge 3 2, Edge 1 3, Edge 3 3]
```

A problem of this representation is that graph instances of this kind cannot be composed and lack desirable properties like functional abstraction. For instance, if `joinGraphs` is a function that composes two graphs by joining their nodes and edges, respectively, the expression `(joinGraphs g1 g1)` produces a non-intended graph containing supposedly different nodes with identical numbers. Renaming the node identifiers in one argument graph is not possible since these nodes might be referred from other parts of the program.

This problem could be avoided by passing a counter through all the nodes when all the graphs are defined. However, this solution leads to code that is non-reusable and difficult to both understand and maintain because two separate tasks are interleaved.

In this paper we present a solution to this problem in the functional logic language Curry. A key feature of this solution is the use of logical variables (i.e., unknown values) when defining graphs. This enables the composition of graphs similar to other data structures and, thus, supports useful abstractions, e.g., functions that computes graph structures from their input arguments. Our representation becomes handy in situations where the construction of graphs is clearly separated from the processing of the constructed graph structures. For instance, graphical user interfaces requires the definition of the structure of the interface (structure of widgets, dependencies caused by event handlers) followed by the activation of the interface. A similar situation arises in dynamic web pages where the definition of the page structure is usually separated from the processing of the web page.

This paper is structured as follows. Section 2 briefly recalls some principles of functional logic programming and the programming language Curry which we use to present concrete examples. Section 3 discusses our approach to represent graphs and its implementation in Curry. Section 4 concludes the paper with a discussion of some related work.

2 Functional Logic Programming and Curry

This section introduces both the basic ideas of functional logic programming and the elements of the programming language Curry that are necessary to understand the subsequent examples.

Functional logic programming integrates in a single programming model the most important features of functional and logic programming (see [7] for a detailed survey). Thus, functional logic languages declare algebraic datatypes, define functions by pattern matching and evaluate expressions containing logical variables. Supporting the latter requires some built-in search principle to guess the appropriate instantiations of logical variables. There exist many languages that are functional logic in this broad sense, e.g., Curry [12], Escher [13], Le Fun [2], Life [1], Mercury [20], NUE-Prolog [16], Oz [19], Toy [14], among others. However, note that only logical variables but no search is necessary for our representation of graphs.

Curry has a Haskell-like syntax [17], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$"). In addition to Haskell, Curry supports logic programming by means of free (logical) variables in both conditions and right-hand sides of defining rules. Thus, a Curry *program* consists of the definition of functions and the declaration of data types on which the functions operate. Functions are evaluated lazily and can be called with partially instantiated arguments. In general, functions are defined by conditional equations, or *rules*, of the form:

$f\ t_1 \ldots t_n$ `|` c `=` e `where` vs `free`

where $t_1, \ldots, t_n$ are *data terms* (i.e., terms without defined function symbols), the *condition* c is either a Boolean function or constraint, e is an expression and the `where` clause introduces a set of free variables. The condition c and the `where` clause are optional. Curry predefines *equational constraints* of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 can be evaluated to unifiable data terms. `success` denotes the predefined constraint that is always satisfied.

The `where` clause introduces the free variables vs occurring in c and/or e but not in the left-hand side. Similarly to Haskell, the `where` clause can also contain other local function or pattern definitions. For instance, consider the following function `last` that computes the last element of a list:

```
last l | xs ++ [e] =:= l
       = e
       where xs,e free
```

The predefined function `++` denotes the concatenation of two lists. Thus, if the condition "`xs ++ [e] =:= l`" is satisfied, `e` is the last element of the list `l`.

The operational semantics of Curry, precisely described in [8,12], is a conservative extension of both lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. Since computations are based on an optimal evaluation strategy [3,4], Curry can be considered a generalization of concurrent constraint programming [18] with a lazy (optimal) evaluation strategy. Furthermore, Curry also offers features for application programming like modules, monadic I/O, ports for distributed programming, and specialized libraries (e.g., [9,10]). We do not discuss these aspects since they are unnecessary to understand our ideas.

There exist several implementations of Curry. The examples presented in this paper were all compiled and executed by PAKCS [11], a compiler/interpreter for a large subset of Curry.

3 Functional Logic Representation of Graphs

We have seen in Section 1 that the definition of graphs with fixed node identifiers (e.g., numbers) causes problems when combining two graphs with accidentally identical node identifiers. Thus, this representation can also cause problems when defining functions to compute new graphs from a given input. For instance, consider Thompson's construction of non-deterministic finite automata (NFA) from regular expressions. If we represent an NFA as a graph, the translation can be defined as a function that takes a regular expression and yields a graph with a start and a final node. In the case of alternative or concatenated regular expressions, we have to combine the graphs corresponding to the argument expressions in a specific way. In this application it is essential that each translation of an expression yields a graph with "new" node identifiers to avoid the construction of non-intended graphs during the combination.

Such problems can be avoided by leaving unspecified the concrete node identifiers when defining a graph. In a functional logic language, this is easily obtained using unbound local variables as node identifiers when defining or creating graphs. Following this idea, we define the graph `g1` of Section 1 as follows:

```
g1 = Graph [Node n1, Node n2, Node n3]
           [Edge n1 n2, Edge n3 n2, Edge n1 n3, Edge n3 n3]
  where n1,n2,n3 free
```

Since `n1`, `n2` and `n3` are local variables, `g1` becomes compositional as a list or a tree would be. For example, `(joinGraphs g1 g1)` is a graph with six different nodes.

To connect two graphs of this kind with an additional edge, one "exposes" the nodes intended for the connection:

```
g2 = (Graph [Node n1, Node n2, Node n3]
            [Edge n1 n2, Edge n3 n2, Edge n1 n3, Edge n3 n3],
      n1)
  where n1,n2,n3 free
```

The following function connects graph/node pairs with an edge provided that `addEdge` is a function that adds a new edge between two nodes of a graph:

```
connectGraphs (g,m) (h,n) = addEdge m n (joinGraphs g h)
```

Now, `(connectGraphs g2 g2)` defines a graph consisting of six nodes and nine edges. The locally defined node identifiers `n1`, `n2` and `n3` act as global identifiers in the composition.

Since unbound variables are not expressive, one may wish to instantiate them in some applications, e.g., visualization. To visualize graphs, one instantiates the node identifiers to pairwise distinct numbers or strings as usually required by

visualization tools. This can be obtained by applying the following constraint on a graph:

```
isFinalizedGraph (Graph nodes _) = numberNodes 1 nodes
  where
     numberNodes _ [] = success
     numberNodes n (Node ni : ns)
          | ni =:= n  -- assign unique identifier
          = numberNodes (n+1) ns
```

The above representation of graphs allows us a further improvement which is useful in some situations. The use of logical variables instead of concrete numbers as node identifiers in the definition of graphs is only a guideline for the programmer, i.e., it is possible to use numbers with the disadvantages mentioned at the beginning. The use of logical variables can be enforced by replacing node numbers with another type which has only "hidden" values so that the programmer cannot write down any concrete value.

The values of a datatype can be hidden by "wrapping" them with a private constructor of the datatype. This means that literal values are replaced by values of an abstract datatype that has no public constructors. The values of this datatype can only be denoted by unbound variables. For instance, consider the graph representation presented above. To hide the use of integers to identify nodes, we define in the graph library a datatype for *node identifiers*:

```
data NodeId = NodeId Int
```

where the constructor `NodeId` is not exported—a standard feature of module systems. Furthermore, we change the definition of nodes and edges so that we use the type `NodeId` wherever nodes are required:

```
data Node = Node NodeId

data Edge = Edge NodeId NodeId
```

These definitions are in the same module that declares `NodeId`. Consequently, `NodeId` can be accessed even though it is private. Finally, we adapt all the functions in the graph library where node identifiers are involved. These functions may access `NodeId` as well. In our example we slightly change the definition of `isFinalizedGraph` by replacing "`ni =:= n`" with "`ni =:= NodeId n`." This change is completely invisible to the user of the library. The coding of graphs remain identical, but this change ensures that the arguments of `Node` are exclusively unbound variables.

A further advantage of this modification is the fact that the implementor of a graph library is free to change the datatype of node identifiers, e.g., from `Int` to `String` whenever it is convenient, without affecting a client of the library.

The complete implementation of this graph representation together with a few examples is available *on-line*[3].

In order to discuss a concrete example where our technique is applied, consider Thompson's construction of NFAs from regular expressions mentioned

[3] `http://www.cs.pdx.edu/~antoy/flp/patterns/unique-names-dir/`

above. We assume that regular expressions are specified by the following datatype where the type parameter `a` specifies the set of base symbols:

```
data RegExp a = Empty                       -- empty word
              | Symbol a                    -- a single symbol
              | Alt  (RegExp a) (RegExp a)  -- alternative
              | Conc (RegExp a) (RegExp a)  -- concatenation
              | Star (RegExp a)             -- repetition
```

The transition relation of the associated NFA will be represented as a graph. The edges are marked by `Nothing` (i.e., epsilon transition) or `(Just c)` for some character `c` (for the sake of simplicity, we assume in the following that the base symbols are characters). Thus, we change the definition of edges shown above into

```
data Edge = Edge NodeId (Maybe Char) NodeId
```

The translation of a regular expression into an NFA is specified by a function `regExp2graph` of the following type:

```
regExp2graph :: RegExp Char -> (NodeId,Graph,NodeId)
```

Thus, we associate to each regular expression a graph with a start and final node for this graph. For instance, the empty word is translated into a graph with a single epsilon edge:

```
regExp2graph Empty =
            (s, Graph [Node s, Node e] [Edge s Nothing e], e)
  where s,e free
```

Similar, a regular expression representing an atomic symbol is translated into a graph with a single edge marked with this symbol:

```
regExp2graph (Symbol c) =
            (s, Graph [Node s, Node e] [Edge s (Just c) e], e)
  where s,e free
```

The most interesting cases are the translation of the remaining types of regular expressions since in these cases our graph representation becomes handy. For instance, the concatenation is translated by combining the graphs of the components sequentially, i.e., we put an epsilon transition between the component graphs (the function `addEdges` adds a list of edges to a graph):

```
regExp2graph (Conc re1 re2) =
   (s1, addEdges [Edge e1 Nothing s2] (joinGraphs g1 g2), e2)
  where (s1,g1,e1) = regExp2graph re1
        (s2,g2,e2) = regExp2graph re2
```

The remaining types of regular expressions are translated in a similar way. After translating a regular expression into an NFA represented as a graph, one can use this graph to solve the word problem by computing epsilon closures of states etc. The complete implementation of this example is available *on-line*[4].

[4] `http://www.cs.pdx.edu/~antoy/flp/patterns/unique-names-dir/`

4 Conclusion and Related Work

We have presented a representation of graphs in a functional logic languages. The main property of this representation is the compositionality of graphs similar to other standard data structures like lists or trees. This is achieved by defining node identifiers as locally defined free variables. Our representation is mainly intended for applications where the construction and the processing of the constructed graph structures are clearly separated. For instance, the translation of regular expressions into NFAs is an example of this kind.

Our representation of graphs is not only useful in graph-based applications, but also in applications where hierarchical (tree-like) data structures are appropriate but additional references inside such structures are needed. As mentioned earlier, graphical user interfaces are one class of such applications. [9] contains an application of our technique in this area. Another application area is dynamic web page generation with form-based input. HTML documents are structured as trees, but the input forms and their "submit" buttons contain dependencies between subtrees that can be appropriately described with our technique [10].

This representation of graphs is not directly available in purely functional languages since they lack free variables. As a consequence, functional approaches to GUI or HTML programming use a more imperative style and/or lack compositionality [5,15]. Erwig [6] proposes an inductive definition of graphs that supports coding graph algorithms in a functional style. His approach is specific to graphs and does not lead to appropriate descriptions of the GUI and HTML applications that we mentioned.

A remaining problem not addressed by this representation is ensuring that the variables used in different nodes are distinct. This situation occurs in other environments as well, e.g., Tcl/Tk or Perl/CGI programs. It can be improved by the use of particular program analysis techniques (e.g., sharing analysis) which is an interesting topic for future work.

References

1. H. Aït-Kaci. An overview of LIFE. In J. Schmidt and A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pages 42–58. Springer LNCS 504, 1990.
2. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 17–23, San Francisco, 1987.
3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
5. K. Claessen, T. Vullinghs, and E. Meijer. Structuring graphical paradigms in TkGofer. In *Proc. of the International Conference on Functional Programming (ICFP'97)*, pages 251–262. ACM SIGPLAN Notices Vol. 32, No. 8, 1997.

6. M. Erwig. Functional programming with graphs. In *2nd ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'97)*, pages 52–65, 1997.
7. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
8. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
9. M. Hanus. A functional logic programming approach to graphical user interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.
10. M. Hanus. High-level server side web scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.
11. M. Hanus, S. Antoy, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2002.
12. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.7). Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.
13. J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, (3):1–49, 1999.
14. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
15. E. Meijer. Server side web scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, 2000.
16. L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 15–26. Springer LNCS 528, 1991.
17. S. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org`, 1999.
18. V. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
19. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.
20. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.