Graph Algorithms in a Lazy Functional Programming Language

Yugo Kashiwagi[*]
Semiconductor Design and Development Center, Hitachi, Ltd.
5-20-1 Josuihoncho Kodaira, Tokyo 187, Japan
kasiwagi%crl.hitachi.junet@uunet.uu.net

David S. Wise[†]
Computer Science Department, Indiana University
101 Lindley Hall, Bloomington, IN 47405-4101
dswise@iuvax.cs.indiana.edu

CR categories and Subject Descriptors:
D.1.1 [**Applicative (Functional) Programming Techniques**]; G.2.2 [**Graph Theory**]: Graph Algorithms; E.1 [**Data Structures**]: Lists; C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Parallel processors.
General Term: Algorithms.
Additional Key Words and Phrases: Haskell, lazy evaluation, fixed point.

Abstract

Solutions to graph problems can be formulated as the fixed point of a set of recursive equations. Traditional algorithms solve these problems by using pointers to build a graph and by iterating side effects to arrive at the fixed point, but this strategy causes serious problems of synchronization under a parallel implementation. In denying side effects, functional programming avoids them, but it also precludes known algorithms that are uniprocessor-optimal.

Functional programming provides another, translation scheme that computes the fixed point without relying on the operational concept of a "store". In this approach, laziness plays an essential role to build a cyclic data structure, a graph, and to implement iteration as streams. The resulting algorithm is not optimal on uniprocessors but, avoiding side effects, the algorithm suggests a promising, more general approach to multiprocessor solutions.

0. Introduction

This paper considers directed graphs, in which edges are ordered pairs: the first, source of the arrow, and the second, the sink. Here we call the source, "parent," and the sink, "child," as if the graphs were trees or dags (directed acyclic graphs). Of course the graphs are not, in general, thus restricted, but the terminology is apt because the algorithms locally treat nodes as if they were in a tree—up to the penultimate step where the fixed point is discovered.

Graph algorithms [1] are often defined by a set of recursive equations. Dataflow equations in conventional compiler construction [2] are typical examples.

Let $n_1, \ldots, n_m$ be nodes of graph,

$$\begin{aligned}(n_i) &= \{n_j \mid n_j \text{ is a parent node of } n_i\},\\ (n_i) &= \{n_j \mid n_j \text{ is a child node of } n_i\},\end{aligned}$$

and $\mathcal{F}$ be a function of three arguments: a value and two sets of values. Then, the general form for a defining equation of a function f on graph nodes can be presented as follows:

$$f(n_i) = \mathcal{F}(f(n_i), \{f(n_j) \mid n_j \in \quad (n_i)\}, \{f(n_j) \mid n_j \in \quad (n_i)\}). \qquad (1 \le i \le m)$$

Then a fixed point, f, of $\lambda f.\lambda n.\mathcal{F}(fn, f \quad n, f \quad n)$ will characterize the solution.

---

* Research performed as a Graduate Student at Indiana University on a Hitachi Fellowship.

† Research reported herein was sponsored, in part, by the National Science Foundation under Grant Number DCR 90-027092

Given appropriate initial values, $f_0(n_i)$, repetitive approximation according to the following equations computes the solution, if it exists.

$$f_1(n_i) = \mathcal{F}(f_0(n_i), \emptyset, \emptyset),$$
$$f_{k+1}(n_i) = \mathcal{F}(f_k(n_i)\{f_k(n_j) \mid n_j \in \quad (n_i)\}, \{f_k(n_j) \mid n_j \in \quad (n_i)\}).$$

The solution builds a graph structure from input data, assigns a chain of approximations for each node, and finds an isolated fixed point from each chain.

A procedural language would implement the graph structure with pointers, and chains of approximations with side effects, which are incompatible with lazy evaluation [3], [4]. However, parallel implementation [5] forces an absence of side effects in order to avoid conflicts in data access.

It is possible, of course, to simulate side effects within lazy functional languages. A graph, viewed as the state of "store", can be passed as an extra argument to and returned from each updating function. This strategy generates a stream of graphs $G_1, G_2, \ldots$ that can be represented "in place" for efficient storage management. Current implementations, however, would require copying and reconstruction of the graph structure for each update, and often yield an expensive implementation and obscure code.

The problem in the above approach is trying to simulate the "store", the notion inherited from procedural languages. This paper seeks a more abstract approach to graph problems. Instead of simulating pointers and side effects, the desired solution should use only mechanisms natural to lazy functional languages. For instance, lazy binding can represent the cyclic data structure of a graph. Chains of approximation can be conveniently represented as streams [6], the direct analog of Lucid's histories [9].

The resulting program turned out to be a direct translation of the defining equations, not easily available from non-lazy languages. Although the resulting algorithm is uniprocessor sub-optimal, it successfully eliminates expensive global update of the graph, giving a possibility of further optimization, especially on parallel processors. This would confirm the choice of lazy functional expression of parallel algorithms.

The remainder of this paper is organized as follows. Section 1 presents a generic graph package for a modest selection of three typical graph algorithms and its applications. The programs are presented in the lazy functional language Haskell [7]. Section 2 discusses storage management problems, where the granularity of data structures plays an essential role. Section 3 presents a method to share graphs among different algorithms. Section 4 discusses efficiency and opportunities for further optimization.

1. Implementation of a graph package and its applications

It is not the purpose of this paper to introduce Haskell since a decent introduction is available elsewhere [8]. However, an insightful reader might barge ahead without it because much of the surface syntax is readable. Haskell is a strongly typed, modular, lazy, functional programming language. All code is declarative.

The module **Graph** is the implementation of a graph package as a Haskell module. The package can be applied to algorithms whose fixed point can be determined from local information, like the chain itself and the size of the graph. Three modules, **StrongComponent**, **FindCycle**, and **WeakComponent** are implementations of a strong component algorithm, a graph acyclicity test, and a weak component algorithm, respectively, using **Graph**. Each application uses different fixed-point conditions.

1.1 Graph package

1.1.1 Type parameters

The package uses three data types.

The *Identifier* type (**id**) identifies nodes of the graph. This type requires an equality test to distinguish each node. It is often a subrange of integers elsewhere, but it need not be here.

The *Chain* type (**c**) is the type of elements of the chain attached to each node. Its counterpart in a procedural implementation is the history of updated data attached to each node [9].

The *Result* type (**r**) is the type of the final result of the algorithm for each node.

1.1.2 Data structures

The input is a list of edges. An edge is a pair of node identifiers, each of whose first element indicates the parent node, and whose second, the child node.

```
-------------------------------------------------------------------------------
-- Graph Algorithm Generator in Haskell
-------------------------------------------------------------------------------
module Graph(GraphData, GraphSkel, GraphInitFun, GraphStepFun,
             GraphResultFun, GraphResult,
             graphAlgorithm) where
  -- GraphData: Input data for graph algorithm
  type (Eq id) => GraphData id = (id,       id     )
                              -- (parentId, childId)
  -- GraphSkel: Skeleton node of graph
  type (Eq id) => GraphSkel id = (id, [id],      [id])
                              -- (id, parentIds, childIds)
  -- GraphNode: Node of the graph (not exported)
  type (Eq id) => GraphNode id c = (id, [c]  )
                                -- (id, chain)
  -- GraphInitFun: Init function of the iteration
  type (Eq id) => GraphInitFun id c = Integer -> [id] -> id -> c
                                   -- size    -> ids  -> id -> init
  -- GraphStepFun: Step function of the iteration
  type GraphStepFun c =
     c        -> [c]     -> [c]      -> c
  -- previous -> parents -> children -> next
  -- GraphResultFun: Function to compute the result
  type (Eq id) => GraphResultFun id c r = Integer -> [id] -> [c]   -> r
                                       -- size    -> ids  -> chain -> result
  -- GraphResult: The result of the algorithm
  type (Eq id) => GraphResult id r = (id, r)
```

Module 1. Graph(Type declarations)

```
  ---------------------------------------------------- functions for pairs --
  -- pairToList: Conversion from a homogeneous pair to a list
  pairToList :: (a, a) -> [a]
  pairToList    (a, b) =  [a, b]
  -- get1st, get2nd: Projections from a homogeneous pair
  get1st, get2nd :: (a, a) -> a
  get1st            (a, b) =  a
  get2nd            (a, b) =  b
  -- test1st, test2nd: Comparators into a homogeneous pair
  test1st, test2nd :: (Eq a) => a -> (a, a) -> Bool
  test1st                       a    (b, c) =  a == b
  test2nd                       a    (b, c) =  a == c
  ----------------------------------------------- functions for graph node --
  -- testNode: Checks id field of the graph node
  testNode :: (Eq id) => id -> GraphNode id c -> Bool
  testNode               key   (id, _)        =  key == id
  -- getNode: Searches a graph with the key (The order of arguments are
  --          permuted for currying)
  getNode :: (Eq id) =>
              [GraphNode id c] -> id  -> GraphNode id c
  getNode      nodes              key =  head (filter (testNode key) nodes)
  -- getId: Id field accessor
  getId :: (Eq id) => GraphNode id c -> id
  getId               (id, _)        =  id
  -- getChain: Chain field accessor
  getChain :: (Eq id) => GraphNode id c -> [c]
  getChain               (_, chain)     =  chain
```

Module 1. Graph(Help fnctions)

The algorithm builds three lists: *skeleton* list, *node* list, and *result* list. Elements of these lists correspond to graph nodes, and have a component of identifier type standing for a node. (The choice of the underlying *list* structure is made for convenience. Alternatively, complete binary trees or arrays give more operational efficiency than lists, but simplicity dictates their use here.) The program defines these types as **GraphSkel**, **GraphNode**, **GraphResult**, respectively.

```
-- graphAlgorithm: Applies the graph algorithm
graphAlgorithm :: (Eq id) =>
   [GraphData id] -> GraphInitFun id c -> GraphStepFun c
-- edges          -> initFunction      -> stepFunction
                               -> GraphResultFun id c r -> [GraphResult id r]
                            -- -> resultFunction         -> result
graphAlgorithm edges init step getResult = result where
  -- ids: Set of id's in the graph (symbol table)
  ids = nub (concat (map pairToList edges)) -- nub drops duplicates
  -- Number of nodes in the graph
  size = length ids
  -- Skeleton nodes of the graph
  skeletonNodes :: (Eq id) => [GraphSkel id]
  skeletonNodes = map makeSkeletonNode ids
  -- Skeleton builder
  makeSkeletonNode :: (Eq id) => id -> GraphSkel id
  makeSkeletonNode               id =  (id, parentIds, childIds) where
    parentIds = map get1st (filter (test2nd id) edges)
    childIds  = map get2nd (filter (test1st id) edges)
  -- The graph
  graph = map makeGraphNode skeletonNodes
  -- Graph builder
  makeGraphNode :: (Eq id) =>
                     GraphSkel id            -> GraphNode id c
  makeGraphNode      (id, parentIds, childIds) = (id, chain) where
    -- Parent nodes and child nodes of this node
    parentChains = map (getChain . (getNode graph)) parentIds
    childChains  = map (getChain . (getNode graph)) childIds
    -- The iteration
    chain = hd : tl where
      hd = init size ids id
      tl = next hd parentChains childChains
    -- Inductive Step
    next :: c ->     [[c]] -> [[c]]    -> [c]
    next    previous parents  children =  current : rest where
      current = step previous (map head parents) (map head children)
      rest    = next current  (map tail parents) (map tail children)
  -- Get result
  result = map makeResult graph
  makeResult :: (Eq id) =>
                  GraphNode id c -> GraphResult id r
  makeResult      (id, chain)    =  (id, getResult size ids chain)
```

Module 1. Graph(Main program)

Besides the identifier field, these list-element types have the following sub-fields.

Skeleton has two sub-fields, which are lists of the identifiers of parent nodes and of child nodes. Taken together, these data describe the structure of the graph.

Node has one sub-field, a chain of data, that approximates the fixed point step-by-step.

Result has one sub-field, which is the result for the node. In general, it will be defined from, or itself be, the fixed point of the chain.

1.1.3 Arguments of the main function

The main function of the program, **graphAlgorithm**, takes four arguments. The first parameter, **edges**, is input data, a list of edges. Three others are functions. First two build a chain, and the last retrieves the result from it.

The *Initialization* function, **init**, computes the first element of the chain from the identifier of the node and from global information, the list of all the identifiers and the size of the graph.

The *Step* function, **step**, computes next element of the chain from the prefix of the chain.

The *Result* function, **getResult**, finds the fixed point and extracts the result from the chain of the node and global information.

1.1.4 Skeleton of the graph

The first stage of the algorithm extracts the skeleton of the graph. It also builds the set of node identifiers, and counts the size of the graph. Some standard Haskell functions require explanation here. **Concat** concatenates a list of lists into a single list. **Nub** drops all the duplicate elements from a list. These functions are used to obtain the set of node identifiers from the raw data of edges.

Filter has two arguments, say *predicate* and *list.* It selects elements of the list on which predicate is **true**. This function is used to obtain the set of parent identifiers and child identifiers for each node.

This part of the program requires nothing beyond the equality test on identifiers. If the identifiers were implemented as a totally ordered set, then a sorted array would be a more efficient implementation than our list.

1.1.5 Building the graph

The second stage establishes links among nodes using the function **makeGraphNode**. The function **makeGraphNode**, mapped on the skeleton of the graph, searches the chains of parents and children from the graph itself, and binds them to local variables **parentChains** and **childChains**. Giving direct access to the chains of parents and children, these bindings serve as links of the graph. Laziness is essential to build the graph, a cyclic data structure, from the flat data structure of the skeleton.

The search is performed only once for each parent and each child of the node, independently of the iteration in the chain. Once the chains of parents and children are directly accessible, it is easy to build the chain from **init** and **step**.

1.1.6 Retrieval of the result

To retrieve the result from chains of each node, the function **getResult** is mapped over the graph. The programmer must be certain that the function **getResult** converges, on only a bounded traversal of the chain. Laziness is once again essential because the chain is likely to be defined well beyond the desired fixed point.

Unlike **init** and **step**, the function **getResult** is not trivial. The condition of convergence is not immediate from the defining equations. That is, a separate proof is required to guarantee the isolated fixed point of the algorithm, just as Floyd-Hoare proof of strong correctness requires a separate proof of termination.

1.2 Examples

To illustrate the package, this section presents three examples of its use, each showing subtly different behavior at fixed points.

1.2.1 Strong component algorithm

A strong component algorithm is given by the following equations:

$$ancestors(n_i) = \{n_i\} \cup (\bigcup_{n_j \in \ \ (n_i)} ancestors(n_j)),$$

$$descendants(n_i) = \{n_i\} \cup (\bigcup_{n_j \in \ \ (n_i)} descendants(n_j)),$$

$$strongcomponent(n_i) = ancestors(n_i) \cap descendants(n_i).$$

The strong component of a node is both an ancestor and a descendant of the node. The node is both an ancestor and descendant of itself. Parents' ancestors are recursively ancestors. Children's descendants are recursively descendants.

The initialization function seeds the chain with the singleton set of the node, itself. The step function extends the union transitively to ancestors of parents and descendants of children.

The fixed point is identified by one set occurring twice, consecutively in the chain. Once the fixed point is found, it is a routine to extract the resulting set. The fixed-point condition is justified by Lemma 1.

Lemma 1

The fixed point for each node in the strong component algorithm is indicated by one set occurring twice (consecutively) in the chain.

```
---------------------------------------------------------------------------
-- Strong Component Algorithm using general graph builder
---------------------------------------------------------------------------
module StrongComponent where
  import Graph(..)
  -- general purpose function (should be in prelude)
  -- select: Filter a list with a boolean list
  select :: [Bool] ->    [a]     -> [a]
  select    []           []      =  []
  select    (True:rest)  (hd:tl) =  hd : select rest tl
  select    (False:rest) (_:tl)  =  select rest tl
  -- get1st, get2nd: Projectons from a homogeneous pair
  get1st, get2nd :: (a, a) -> a
  get1st            (a, b) =  a
  get2nd            (a, b) =  b
  -- BitMap: Bitmap of the graph nodes
  type BitMap = [Bool]
  -- Bitmap pair (for ancestors and descendants)
  type BMPair = (BitMap, BitMap)
  -- bmAnd, bmOr: Bitmap functions
  bmAnd, bmOr :: BitMap -> BitMap -> BitMap
  bmAnd          a         b      =  zipWith (&&) a b
  bmOr           a         b      =  zipWith (||) a b
  -- makeUnit: Creates unit vector for given id
  makeUnit :: (Eq id) => id -> [id] -> BitMap
  makeUnit               id    ls    =  map (== id) ls
  -- strongComponent
  strongComponent :: (Eq id) => [GraphData id] -> [GraphResult id [id]]
  strongComponent               edges         =
    graphAlgorithm edges sCInit sCStep sCResult
  -- Initialization
  sCInit :: (Eq id) =>
              Integer -> [id] -> id -> BMPair
  sCInit      _          ids     id =  (initialValue, initialValue) where
    initialValue = makeUnit id ids
  -- Step
  sCStep :: BMPair -> [BMPair] -> [BMPair] -> BMPair
  sCStep    (a, d)    parents     children =  (ancestors, descendants) where
    ancestors   = foldl1 bmOr (a : (map get1st parents))
    descendants = foldl1 bmOr (d : (map get2nd children))
  -- Result
  sCResult :: (Eq id) =>
                Integer -> [id] -> [BMPair] -> [id]
  sCResult      _          ids     chain    =
      select (bmAnd ancestors descendants) ids where
    -- final result
    (ancestors, descendants) = findFixedpoint chain
    -- fixedpoint finder
    findFixedpoint :: [BMPair]                  -> BMPair
    findFixedpoint    (hd1:hd2:_) | hd1 == hd2 =  hd1
    findFixedpoint    (_:tl)                   =  findFixedpoint tl
```

Module 2. StrongComponent

Proof

The m^{th} ancestor is propagated and first added to the ancestor set as the m^{th} element of the chain.

We show by contradiction that $A_m = A_{m+1}$ identifies a fixed point. Suppose that the n^{th} element of the ancestor set A_m and the $(m+1)^{\text{st}}$ element of the ancestor set A_{m+1} are the same, and for $(m+2)^{\text{nd}}$ element of the ancestor set, $A_{m+2} \neq A_{m+1}$ holds. This implies the node has a $(m+2)^{\text{nd}}$ ancestor without having $(m+1)^{\text{st}}$ ancestor, a contradiction. By symmetry, the same argument holds for descendants. ∎

```
-------------------------------------------------------------------------------
-- Graph Acyclicity Algorithm using general graph builder
-------------------------------------------------------------------------------
module FindCycle where
  import Graph(..)
  -- maximum with default value 0
  max0 :: [Integer] -> Integer
  max0   []         =  0
  max0   ls         =  maximum ls
  -- findCycle
  findCycle :: (Eq id) => [GraphData id] -> [GraphResult id Bool]
  findCycle               edges          =
    graphAlgorithm edges fCInit fCStep fCResult
  -- Initialization
  fCInit :: (Eq id) => Integer -> [id] -> id -> Integer
  fCInit               _           _      _  =  1
  -- Step
  fCStep :: Integer -> [Integer] -> [Integer] -> Integer
  fCStep    _          parents      _         =  max0 parents + 1
  -- Result (True if no cycle, False if cycle is found)
  fCResult :: (Eq id) =>
                Integer -> [id] -> [Integer] -> Bool
  fCResult      size       _       chain     =  testConvergence chain where
    testConvergence :: [Integer]                -> Bool
    testConvergence    (hd1:hd2:_) | hd1 == hd2 =  True
    testConvergence    (hd:_)      | hd > size  =  False
    testConvergence    (_:tl)                   =  testConvergence tl
```

Module 3. FindCycle

1.2.2 Acyclicity test of a graph

The following equation defines depth of a node from a root, when the depth is bounded.

$$depth(n_i) = \max_{n_j \in \ (n_i)} depth(n_j) + 1.$$

Here, $\max_{x \in \emptyset} x$ is defined to be 0. That is, if a node does not have any parents, its depth is 1. The acyclicity algorithm builds chains of depths. If a chain converges, the associated node has no cycle above it. If it diverges, the node has a cycle above it.

The initialization function returns 1. The step function computes the maximum of the depths of parents plus one, according to the definition of *depth* above.

The result for each node is a boolean value, showing whether there is a cycle above the node. The value depends on whether the chain converges or not. The termination of this algorithm is justified by the following lemma:

Lemma 2

The fixed point for each node in the acyclicity test algorithm is indicated by the same value occurring twice in the chain. If a value in the chain exceeds the size of the graph, the chain diverges.

Proof

By an argument similar to the proof of Lemma 1, two equal consecutive elements in the chain gives the fixed point. All that remains is to bound the search of the chain. If the depth exceeds the size of the graph, there exists $(n+1)^{st}$ parent of the node, where n is the size of the graph. So there is at least an node whose ancestor is itself in the graph. And the chain infinite. ▮

This algorithm shows that an appropriate criterion for the divergence test can handle these run-away cases. It is worth noting that child links are never used in this example. And, since the evaluation scheme is lazy, these links are not even computed.

```
-------------------------------------------------------------------------------
-- Weak Component Algorithm using general graph builder
-------------------------------------------------------------------------------
module WeakComponent where
  import Graph(..)
  -- get position
  position :: (Eq id) => id -> [id] -> Integer
  position               id    ids  =  posSub id ids 0 where
    posSub :: id -> [id] -> Integer            -> Integer
    posSub    id    (hd:_)  n       | id == hd =  n
    posSub    id    (_:tl)  n                  =  posSub id tl (n+1)
  -- WeakComponent
  weakComponent :: (Eq id) => [GraphData id] -> [GraphResult id integer]
  weakComponent               edges          =
      graphAlgorithm edges wCInit wCStep wCResult
  -- Initialization
  wCInit :: (Eq id) => Integer -> [id] -> id -> Integer
  wCInit               _          ids     id =  position id ids
  -- Step
  wCStep :: Integer -> [Integer] -> [Integer] -> Integer
  wCStep    previous   parents      children  =
    maximum (previous : (parents ++ children))
  -- Result
  wCResult :: (Eq id) => Integer -> [id] -> [Integer] -> Integer
  wCResult               size       _       chain     =  chain !! (size-1)
```

Module 4. WeakComponent

1.2.3 Weak component algorithm

The fixed point of the following equation assigns a unique number for each set of nodes in the same weak component.

$$val(n_i) = \max(\max_{n_j \in \ (n_i)} val(n_j), \max_{n_j \in \ (n_i)} val(n_i), i).$$

This equation is based on the following idea:

a) Assign a unique numeric label for each node.
b) Propagate it to parents and children, and take the local maximum as the signature value for each node.

If two nodes are weakly connected, the value converges to the maximum value assigned to the nodes in the component. The initialization function gives a unique label for each node. The step function takes the maximum of the labels of parents, children, and the node itself.

The fixed-point condition is as follows:

Lemma 3

The fixed point is reached at the m^{th} element in the chain, where m is the span of the component.

Proof

Let two nodes, n_i and n_j, be connected and the length of the shortest (undirected) path from n_i and n_j be k. Then at the k^{th} element of the chain, the value is at least the maximum of original numeric labels of two nodes.

The span is defined as the maximum of such shortest paths. Hence, after m (span of the component) steps, the value reaches the maximum numeric label in the component. ▌

In the algorithm, the size of the entire graph is used as a bound on the fixed point. This bounded search is not an efficient algorithm unless there is an "efficient" estimate on the span of the graph. Tarjan [10] gives an almost linear uniprocessor algorithm for weak component using side effects. However good it is on a uniprocessor, Tarjan's algorithm is ill-suited to asynchronous multiprocessing just as ours is suboptimal on a uniprocessor. To achieve such an optimal algorithm within the framework of parallelism or lazy functional programming is still an open problem.

2. Data structure vs binding

The code presented in this paper uses data recursion [11]. Data recursion can be implemented by bindings, as in the program presented here, or by data structures. A *data-structure solution* would build the links of the graph explicitly as links *within* the data structure. A *binding solution* builds the links of the graph implicitly as local bindings.

The data-structure solution was explored before developing the code presented above, but abandoned for reasons now discussed. In the data structure solution, the nodes are declared by the following code.

```
data GraphNode id c = Node id c [GraphNode id c] [GraphNode id c]
```

In this code, **Node** is a constructor, like *cons*, and is the signature of the type **GraphNode**, whose third field is the list of parent nodes, and whose fourth field is the list of child nodes. With this data declaration, the graph is constructed as an explicit data structure by the following code.

```
-- The graph
graph = map makeGraphNode skeleton
-- Graph builder
makeGraphNode :: (Eq id) =>
                   (id, [id],      [id]     )
                     -> GraphNode id c
makeGraphNode      (id, parentIds, childIds) =
                        Node id chain parents children
  -- Parent nodes and child nodes of this node
  parents  = map (getNode graph) parentIds
  children = map (getNode graph) childIds
```

Although this solution seems to be more intuitive and easily understood, there is currently a problem with the requirement for intermediate storage in its implementation.

In the data-structure solution, all the information of parent nodes and child nodes are locally bound in the scope of **makeGraphNode**. Consequently, all the ancestors and descendants are accessible from the local scope of **makeGraphNode**. This builds a large, mutually-recursive data structure that persists as long as there remains access to any of its nodes.

On the other hand, the binding solution binds only the chains of parents and children in the scope of **makeGraphNode**. This avoids redundant binding and keeps data structure small. So the garbage collector can reclaim nodes as soon as the last access to a node is released.

It is true that a large data structure is still bound in an outer environment to be passed to local functions. However, such a binding can be released at earlier stage by a program transformation. This technique applied to the environment structure has been called *lambda lifting* [12]; we generalize it to apply to recursively specified data types, like streams.

A transformation rule from the "data structure" to "binding" style can be stated as follows:

a) Find a binding of a structured data.
b) Analyze the scope of the binding to find out which of the components of the structure is actually used.
c) Replace the binding to the whole structure by the bindings to the components, themselves.

3. Sharing graphs

In traditional graph algorithms, the data structure of a graph is shared by several algorithms. This section shows how to implement this sharing in functional languages. The technique can be considered as a kind of "distributive law."

The function **graphAlgorithm**, curried [13] to its first argument, can be applied to several graph algorithms. In Haskell code, the expression `(graphAlgorithm edges)` serves this purpose. This value can be shared by several graph algorithms, each of which supplies its own initialization function, step function and result function. This enables sharing the skeleton of the graph, avoiding its recomputation. Without this sharing of the skeleton, the algorithm would rebuild it each time an algorithm were applied.

In traditional algorithms, sharing comes for free by binding the graph— cycles and all—as global data. This idea should be rejected in lazy functional languages because of the memory inefficiency discussed in the previous section. To attain the space performance competing with traditional algorithms, however, it is

```
module Product(product, productSharingChain) where
import Graph(GraphStepFun, GraphInitFun, GraphResultFun)
proj1 :: (a,b) ->a
proj1 (a,b) = a
proj2 :: (a,b) -> b
proj2 (a,b) = b
product :: (Eq id) => (GraphInitFun id (a,b) -> GraphStepFun id (a,b) ->
                        GraphResultFun id (a,b) (c,d) -> GraphResult id (c,d))
                       -> GraphInitFun id a ->     GraphInitFun id b
                       -> GraphStepFun id a ->     GraphStepFun id b
                       -> GraphResultFun id a c -> GraphResultFun id b d
                       -> GraphResult id (c,d)
product generator init1 init2 step1 step2 result1 result2 =
  generator combinedInit combinedStep combinedResult where
    combinedInit :: (Eq id) => GraphInitFun id (a,b)
    combinedInit size ids id = (init1 size ids id, init2 size ids id)
    combinedStep :: (Eq id) => GraphStepFun id (a,b)
    combinedStep (previous1, previous2) parents children =
      (step1 previous1 (map proj1 parents) (map proj1 children),
       step2 previous2 (map proj2 parents) (map proj2 children))
    combinedResult :: (Eq id) => GraphResultFun id (a,b) (c,d)
    combinedResult size ids chain=
      (result1 size ids (map proj1 chain),
       result2 size ids (map proj2 chain))
productSharingChain :: (Eq id) => (GraphInitFun id a -> GraphStepFun id a ->
                                    GraphResultFun id a (b,c) ->
                                    GraphResult id (b,c))
                                   -> GraphResultFun id a b
                                   -> GraphResultFun id a c
                                   -> GraphResult id (b,c)
productSharingChain generator result1 result2 =
    generator combinedResult where
  combineResult :: (Eq id) => GraphResultFun id a (b,c)
  combinedResult size ids chain=
    (result1 size ids chain, result2 size ids chain)
```

Module 5. Product

still necessary to share the links of the graph as well as skeleton of the graph among algorithms applied to the same graph.

This leads us to a notion of "Cartesian" product of algorithms. We can apply a set-theoretic "product of functions" [14] to algorithms, viewed as functions from inputs to outputs. It is illustrated in the module **Product**.

For example, the product of the acyclicity algorithm and the strong component algorithm is defined as follows.

```
productAlgorithm edges =
  product (graphAlgorithm edges) fCInit   sCInit
                                 fCStep   sCStep
                                 fCResult sCResult
```

This program executes two algorithms simultaneously sharing common data structures.

The function **product** combines two chains into a chain of pairs sharing the links of the graph. Two algorithms may share a chain, being different only in the way they retrieve their result. Such a combination is implemented by another product function, **productSharingChain**.

The module **Product** implements the product of two algorithms. The product of three algorithms could be defined similarly, but not as a nested product. However, Haskell's strict typing prevents us from defining product in full generality, nesting products on different numbers of factors. It is common for algorithms sharing a graph to have different chain types and result types, and strong typing prohibits the heterogeneous lists that result from their product.

Streams nicely capture the idea of sequential execution in lazy functional languages, just as histories capture it in Lucid. However, in strongly typed languages, each value in a list is required to have the same type, which is often not the case in common programming practice. This problem may be handled by a syntactic macro that enables a programmer to define generic procedure for arbitrary tuples.

4. Performance

Although some algorithms presented here are uniprocessor sub-optimal, they successfully remove the expensive simulation of side effects, and made it possible to share a graph between algorithms. Thus, they might perform comparatively well under massive parallelism. We have considered this carefully, but have no multiprocessor implementation for a test of that thesis.

It is easy to give rough uniprocessor bounds of the performance of the algorithm. Establishing the links of the graph requires $O(ne)$ steps where n is the number of nodes and e is the number of edges. Operations to compute chain elements are executed at most $O(nk)$ times where k is the maximum length of the chain before fixed point is found. In **StrongComponent**, each chain element requires n steps to create. These considerations show that the upper bound of the performance of **StrongComponent** is cubic and that of **FindCycle** and **WeakComponent** is quadratic.

However, the actual preformance of these algorithms is not easily analyzed from just the program, itself. As the language is lazy, the program builds only those data structures actually accessed. For example, in the program **FindCycle**, child links are never established. Hence, the actual performance depends on "how the algorithms are used" rather than "how they are implemented." In this sense, lazy algorithms are passive entities like data structures, rather than active like programs. Precise analysis requires either probabilistic assumptions on input data or a specific access pattern from outside the algorithm.

Moreover, one must not confuse the above analysis of these algorithms' (suboptimal) performance on uniprocessors with their performance on multiprocessors, the intended targets. Unfortunately, the analysis of parallel algorithms is still in its infancy, so we have neither the notation nor the facility to extract measures from this code, alone. Their performance depends also on the number of processors in the run-time environment, the architecture of their interconnection, and the scheduling algorithm used to partition a problem. We *can* assert, because the algorithms are presented functionally, that there is plently of latitude for the scheduler to divide its problem, to fit whatever resource or architecture constraints might be in force on a particular site.

Further improvements are necessary to make such an algorithm compete with conventional algorithms, Since these algorithms can be considered as lazy data structures, a good strategy is to reduce the number of probes from outside. In retrieving results, not all of the nodes' results may be required in the final result of the algorithm. Clever transformation of the algorithm may further improve performance; for example, once the identifiers in the strong components of a node are known (in the strong component algorithm), there is no need to explore other nodes in the same component.

5. Conclusion

This paper discusses an implementation of graph algorithms in the lazy functional language, Haskell. The program makes full use of mechanisms of lazy evaluation, eliminating expensive simulation of pointers and side effects of conventional languages. Their performance is not optimal, compared with some known, best algorithms for uniprocessors. However, their absence of side effects and local binding indicates a promising approach to programming with parallel processors. Some possibility of further improvement has been discussed, but further investigation is necessary.

Some interesting points are raised from our experience in developing these algorithms.

(1) Derivation of a graph algorithm from equations

The algorithm provides a direct translation from defining equation to the program. This makes the proof of the program mathematical and nearly transparent. Such a translation is not so obvious in conventional languages, including non-lazy functional languages. The difficulty for graph equations is the lack of a base case for inductive reasoning or a recursive proof, which is not essential for lazy structures (e.g. streams).

(2) Memory efficiency of lazy programs

As discussed in the paper, large recursive data structures prevent efficient memory management, and should be transformed (lambda-lifted) into bindings. It is desirable that this translation be handled by a compiler. Although the optimization seems to be intractable in general, simple cases might be handled by analyzing simple data dependency among fields of structures. On the other hand, data structures make the program more readable.

(3) Heterogeneous data structures

Strongly typed streams successfully capture the notion of sequential algorithms as long as the type of all partial results is homogeneous. They fail where partial results are heterogeneous, as in the abstract, second-order product of the graph algorithms. To generalize their product, a type-safe method is needed that extends to to tuples of arbitrary arity.

The graph problems considered in this paper arose when the first author was writing a verifier for the static syntax of Haskell in Haskell. Since the static analysis and the type checking of this language is mutually recursive among all the declarations, and since Haskell is a language highly recursive in structure and in style, it was anticipated that it would be an excellent tool for building its own type-checker. However, effective implementation of graph algorithms, originally a peripheral issue, here became a problem of focus. The general strategy for Haskell implementation of graph algorithms, (e.g. cyclicity check) was extended to others eventually to be required in an optimizing compiler (e.g. dataflow analysis).

After some consideration of traditional algorithms, we reformulated a general approach to graph problems as a few mutually recursive equations, whose solution is a fixed point. Thus, Haskell uses mutually recursive, lazy chains to isolate that fixed point—something that it does well. It remains to show that its multiprocessor implementation also performs well.

Acknowledgement

The programs presented here were syntactically tested with the Prototype Glasgow Haskell Compiler Version 0.402. We thank Cordelia V. Hall for helping us with initial debugging.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman [1983]. *Data Structures and Algorithms,* Addison-Wesley, Reading, Massachusetts.
2. A. V. Aho, R. Sethi and J. D. Ullman [1986], *Compilers Principles, Techniques, and Tools,* Addison-Wesley, Reading, Massachusetts.
3. D. P. Friedman and D. S. Wise [1976]. *Cons should not evaluate its arguments,* in S. Michaelson and R. Milner (Eds), *Automata, Languages and Programming,* Edinburgh University Press, Edinburgh, pp. 257–284.
4. P. Henderson and J. Morris, Jr. [1976]. *A lazy evaluator, Proc. 3rd ACM Symp. on Principles of programming Languages,* pp. 95-103.
5. D. B. Skillcorn [1990]. *Architecture-Independent Parallel Computation,* Computer 23, 12, IEEE.
6. P. J. Landin [1965]. *A correspondence between ALGOL60 and Church's lambda notation, Comm. ACM. 8,2. Aug. 89-101.*
7. P. Hudak and P. Wadler (Eds.) [1990]. *Report on the Programming Language Haskell,* Version 1.0.
8. P. Hudak Para-Functional Programming [1991]. in B. Szymanski (Eds.) *Parallel Functional Programming Languages and Environments*, Addison Wesley.
9. W. W. Wadge, E. A. Ashcroft [1985]. *Lucid, the Dataflow Programming Language,* Academic Press.
10. R. E. Tarjan [1983]. *Data Structures and Network Algorithms,* SIAM, Philadelphia.
11. S. D. Johnson. [1984]. *Synthesis of Digital Designs from Recursion Equations,* The MIT Press.
12. T. Johnsson [1985]. *Lambda lifting: transforming programs to recursive equations,* in Jouannaud (Eds), *Conference on Functional Programming Languages and Computer Architecture*, Nancy, LNCS 201. Springer Verlag.
13. J. K. Stoy [1979]. *Denotational Semantics. The Scott-Stracey Approach to Programming Language Theory,* The MIT Press.
14. P. R. Halmos [1960]. *Naive Set Theory,* Van Nostrand, Princeton.