Language Engineering - Problem Sheet 2

Dom Hutchinson

October 28, 2018

Question 1.1 - Introduction
Consider a domain-specific language for directed graphs. A *vertex* is any natural number. An *edge* consists of a *source* and a *target* vertex. A *graph* is a set of vertices V and a bag of edges E, where all the vertices in the edges E are contained in V . A bag (sometimes called a multiset) is a structure like set except that duplicate elements are allowed.
Graphs can be composed out of four operations, empty, vertex , connect, and overlay, described as follows.

- $empty :: Graph$ where empty is a graph that contains no vertices and no edges.
- $vertex :: IntGraph$, where vertex v is a graph containing the vertex v, and no edges.
- $overlay :: GraphGraphGraph$, where overlay x y is a graph whose vertices are the union of the vertices in x and the vertices in y, and whose edges are the edges in x followed by the edges in y.
- $connect :: GraphGraphGraph$, where the vertices in connect x y are all those in overlay x y, and the edges are the edges in overlay x y followed by an edge from each vertex in x to each vertex in y.

Together, these combinators make it possible to express all kinds of graph structures.

Question 1.2 - Deep Embedding
A deep embedding uses a datatype to represent a domain-specific language, where each constructor of the datatype corresponds to an operation. These constructors are called core constructors of the language.

Question 1.2.1
Define a datatype *Graph* that encodes a deep embedding of the graph language. Introduce a core constructor for each of the four operations.

My Solution 2.1

```
data Graph = Empty
           | Vertex Int
           | Overlay Graph Graph
           | Connect Graph Graph
```

Question 1.2.2
Define a function $vertices :: Graph \to [Int]$, where vertices g is a list of the vertices in the graph g.

My Solution 2.2

```
vertices :: Graph -> [Int]
vertices (Empty)          = []
vertices (Vertex n)       = [n]
vertices (Overlay g1 g2) = nub ((vertices g1) ++ (vertices g2))
vertices (Connect g1 g2) = nub ((vertices g1) ++ (vertices g2))
```

Question 1.2.3
Define a function *edges* :: *Graph* → *Int*, where edges g is the number of edges in the graph g. Hint: You may need to use your vertices function.

My Solution 2.3

```
edges :: Graph -> Int
edges (Empty)          = 0
edges (Vertex n)       = 0
edges (Overlay g1 g2) = (edges g1) + (edges g2)
edges (Connect g1 g2) = (edges g1) + (edges g2)
                        + (length(vertices g1) * length(vertices g2))
```

Question 1.2.4
Define a function *roots* :: *Graph* → [*Int*], where roots g is a list of the roots of the graph g. A root of a graph g is a vertex of g which does not appear as the target of any edge in g.

My Solution 2.4

```
roots :: Graph -> [Int]
roots (Empty)          = []
roots (Vertex n)       = [n]
roots (Overlay g1 g2) = nub (removeAll (intersect (vertices g1) (vertices g2))
                         (nub ((roots g1) ++ (roots g2))))
                         ++(intersect (roots g1) (roots g2))
roots (Connect g1 g2) = removeAll (intersect (vertices g1) (vertices g2))
                         (roots g1)

-- removes given element from list
remove :: Int -> [Int] -> [Int]
remove i [] = []
remove i (x:xs)
  | i == x      = remove i xs
  | otherwise = x : (remove i xs)

-- remove list of elements from list
removeAll :: [Int] -> [Int] -> [Int]
removeAll [] ys = ys
removeAll (x:xs) ys = removeAll xs (remove x ys)
```

Question 1.3 - Shallow Embedding

Question 1.3.1
Provide a shallow embedding that produces the same semantics as *vertices* by redefining an appropriate type for *Graph*, and defining the behaviour of *empty*, *vertex* , *overlay*, and *connect*.

My Solution 3.1

```
type Vertices = [Int]
type Graph' = Vertices

empty :: Graph
empty = []

vertex :: Int -> Graph
vertex n = [n]

overlay :: Graph -> Graph -> Graph
overlay g1 g2 = nub (g1++g2)

connect :: Graph -> Graph -> Graph
connect g1 g2 = nub (g1++g2)
```

Question 1.3.2
Redefine *Graph* again, as well as all the operations, to produce the same semantics as *edges*. Hint: Your semantic domain should be a pair of values, one containing the information for *vertices*, and the other for *edges*.

My Solution 3.2

```
type Edges = Int
type Graph = (Edges, Vertices)

empty :: Graph
empty = (0,[])

vertex :: Int -> Graph
vertex n = (0, [n])

overlay :: Graph -> Graph -> Graph
overlay (e1, v1) (e2,v2) = (e1+e2, nub (v1++v2))

connect :: Graph -> Graph -> Graph
connect (e1, v1) (e2, v2) = (e1+e2+(length v1 * length v2), nub(v1++v2))
```

Question 1.3.3
Discuss why the shallow definition of *edges* more efficient than the deep definition of *edges*.

My Solution 3.3
Using the shallow definition we don't need to recalculate the vertices of a graph each time we want to referrence it.

Question 1.4 - Classy Embedding
Instead of redefining *Graph* for each semantics, it is better to provide a type class that captures all the shallow operations, and use instances to provide each of the different semantics.

Question 1.4.1
Define an appropriate type class called *Graphy*, and show how the semantics of *vertices* and *edges* can be given using new types called *Vertices* and *Edges* respectively.

My Solution 4.1

```
class Graphy a where
  empty   :: a
  vertex  :: Int -> a
  overlay  :: a -> a -> a
  connect  :: a -> a -> a
```

Question 1.4.2
Provide an instance of your type class that allows a semantics of the shallow embedding to be the deep embedding. Hint: this is the instance *Graphy Graph*.

My Solution 4.2

```
instance Graphy Vertices where
  empty = Vertices []
  vertex x = Vertices [x]
  overlay (Vertices g1) (Vertices g2) = Vertices (nub (g1++g2))
  connect (Vertices g1) (Vertices g2) = Vertices (nub (g1++g2))

instance Graphy Edges where
  empty = Edges (0,[])
  vertex x = Edges (0,[x])
  overlay (Edges (e1,v1)) (Edges (e2,v2)) = Edges (e1+e2, nub(v1++v2))
  connect (Edges (e1,v1)) (Edges (e2,v2)) = Edges (e1+e2+(length v1 * length v2)

instance Graphy Graph where
  empty = Empty
  vertex x = Vertex x
  overlay g1 g2 = Overlay g1 g2
  connect g1 g2 = Connect g1 g2
```

Question 1.5 - Smart Constructors
A *smart constructor* is a function that provides new operations which are defined in terms of existing ones.

Question 1.5.1
Extend the language with a new smart constructor called $ring :: Graphy\ graph \to [Int] \to graph$ which produces a graph where all the elements in the list are converted into vertices and connected into a complete cycle connecting each element to the next, and the last to the first.

My Solution 5.1

TODO

Question 1.5.2
Instead of defining this function in terms of existing operations, consider the implications of adding a new core constructor *Ring* to the *Graph* data type.
a) Outline the code that needs to be changed when adding a core constructor rather than a smart constructor.
b) Explain the benefits of using smart constructors over core constructors when new semantic functions are added to a given language.
c) Explain why using core constructors can potentially lead to more efficient implementations.

My Solution 5.2
TODO

Question 1.6 - Comparing Approaches

Question 1.6.1
Discuss the changes required in order to add a new semantics of a deep embedding.

My Solution 6.1
To add a semantics to a deep embedding only requires the definition of a new function.

Question 1.6.2
Discuss the changes required in order to add a new operation to a deep embedding.

My Solution 6.2
To add a new operation requires the data types to be updated and all functions which use these datatypes to be updated as well.

Question 1.6.3
Discuss the changes required in order to add a new semantics of a shallow embedding.

My Solution 6.3
To add a new semantic to shallow embedding requires all functions to be redefined, or updated to now calculate a tuple output which includes the new semantics.

Question 1.6.4
Discuss the changes required in order to add a new operation to a shallow embedding.

My Solution 6.4
Add a new operation requires only the definition of a new function.

Question 1.6.5
Discuss the advantages gained by using type classes for a shallow embedding.

My Solution 6.5
Type classes allows you to define a different instance of each semantics, rather than only have one semantics or using tuples.

Question 1.6.6
Discuss the advantages gained by using smart constructors for a deep embedding.

My Solution 6.6
A smart constructor allows you to add new operations without having to redefined existing ones, as it is defined in terms of existing operations.

Question 1.7 - Reinterpretting Graphs
The graph language can be used to construct a graph that is interpreted traditionally either as an adjacency list or as an adjacency matrix.

Question 1.7.1
Write a function $lists :: Graph \to Map\ Int\ [Int]$ that realises the deep *Graph* DSL as a traditional adjacency list implementation.

My Solution 7.1

```
lists :: Graph -> Map Int [Int]
lists Empty = Map.empty
lists (Vertex n) = Map.singleton n []
lists (Overlay g1 g2) = Map.union (lists g1) (lists g2)
lists (Connect g1 g2) = Map.map (nub) (Map.union (Map.map (++ (Map.keys (lists g
```

Question 1.7.2
Write a new typeclass instance for *Graphy* that realises the shallow *Graph* DSL as a traditional adjacency list implementation.

My Solution 7.2

TODO

Question 1.7.3
Discuss why it is harder to write the interpretation using the shallow embedding rather than the deep embedding.

My Solution 7.3
TODO

Question 1.7.4
Write a function $mat :: Graph \to MatGraph$ that realises the *Graph* DSL as a traditional adjacency matrix implementation (you can define the datatype *MatGraph* however you want).

My Solution 7.4

```
type MatGraph = ([Int], [[Int]])

-- Treates dupes as seperate vertices
mat :: Graph -> MatGraph
mat Empty = ([], [])
mat (Vertex n) = ([n], [[0]])
mat (Overlay g1 g2) = (v1++v2, (overlay' (length e1) e1 (length e2) e2))
  where
    v1 = fst (mat g1)
    v2 = fst (mat g2)
    e1 = snd (mat g1)
    e2 = snd (mat g2)
mat (Connect g1 g2) = (v1++v2, connect' (length e1) e1 (length e2) e2)
  where
    v1 = fst (mat g1)
    v2 = fst (mat g2)
    e1 = snd (mat g1)
    e2 = snd (mat g2)

-- Pads edges with zeroes
--          len(e1)   e1     len(e2)    e2          out
overlay' :: Int -> [[Int]] -> Int -> [[Int]] -> [[Int]]
overlay' _ [] _ [] = []
overlay' n [] m (y:ys) = ((zeroes n)++y) : (overlay' n [] m ys)
overlay' n (x:xs) m ys = (x++(zeroes m)) : (overlay' n xs m ys)
```

```
-- Creates list of n zeroes
zeroes :: Int -> [Int]
zeroes 0 = []
zeroes n = 0:(zeroes (n-1))

-- pads edges with 1s
connect' :: Int -> [[Int]] -> Int -> [[Int]] -> [[Int]]
connect' _ [] _ [] = []
connect' n [] m (y:ys) = ((zeroes n)++y) : (overlay' n [] m ys)
connect' n (x:xs) m ys = (x++(ones m)) : (overlay' n xs m ys)

-- creates list of n ones
ones :: Int -> [Int]
ones 0 = []
ones n = 1:(ones (n-1))
```