CSci 450: Org. of Programming Languages
Labeled Digraph ADT in Haskell

H. Conrad Cunningham

27 October 2017

Contents

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of October 2017 is a recent version of Firefox from Mozilla.

TODO:

- Better motivate the ADT
- Better integrate this case study into the evolving textbook

- Add student outcomes, exercises, better references, etc.

Labeled Digraph ADT in Haskell

Introduction

In this case study, we seek to develop a family of graph data structures that can support the implementation of Adventure games similar to the Wizard's Adventure game given in Chapter 5 of the book *Land of Lisp: Learn to Program in Lisp, One Game at a Time* [Barski 2011]. We develop this family as an abstract data type and implement it as modules in Haskell.

This abstract data type represents *doubly labeled directed graphs*. A directed graph (digraph) includes vertices (nodes) and edges directed from one node to another. The graph allows arbitrary data (i.e., labels) to be attached to both vertices and edges in the directed graph.

Note: In this case study we use the acronym *ADT* to refer to an *abstract data type*. We also use an *algebraic data type* to denote the graph type provided by a Haskell module and perhaps use other algebraic data types in implementing the module.

The objectives of this case study are to:

a. specify a Labeled Digraph abstract data type using a constructive model (sets and interface invariant for the, preconditions/postconditions for each operation)
b. implement the abstract data type as a module in Haskell with at least two distinct implementations, giving appropriate implementation invariants for each

One source of information on digraphs and their specification we referenced is Chapter 10 of the book *Abstract Data Types: Specifications, Implementations, and Applications* [Dale-Walker 1996].

Specification

Notation

We use the following notation and terminology to describe the abstract data type's model and its semantics. (We choose notation that can readily be used in comments in the Haskell program.)

- `(ForAll x, y :: p(x,y))` is true if and only if predicate `p(x,y)` is true for all values of `x` and `y`.

- `(Exists x, y :: p(x,y))` is true if and only if there is at least one pair of values `x` and `y` for which `p(x,y)` is true.
- `(# x, y :: p(x,y))` yields a count of pairs `(x,y)` for which `p(x,y)` is true.
- `<=>` denotes logical equivalence. `p <=> q` is true if and only if the logical (Boolean) values `p` and `q` are equal (i.e., both true or both false).
- `x IN C` is true if and only if value `x` is member of a collection `C` (such as a set, bag, or sequence). Similarly, `x NOT_IN C` denotes the negation of `x IN C`.
- A type consists of a set of values and a set of operations. We sometimes say a value is `IN` a type to mean the value is `IN` the set associated with the type.
- For sets `C` and `D`, `C UNION D` denotes set union, that is, a set that includes all the element of both `C` and `D`.
- For sets `C` and `D`, `C INTERSECT D` denotes set intersection, that is, a set that includes all elements that are both in `C` and in `D`.
- For sets `C` and `D`, `C - D` denotes set difference, that is, the set `C` with all elements of set `D` removed.
- For sets `C` and `D`, `C SUBSET_OF D` denotes that `C` is subset of `D`, that is, all the elements of `C` also occur in `D`.
- A tuple such as `(x,*)` appearing in a collection such as `{ (x,*) }` denotes element `x` grouped with all possible values of the second component. Note: We could also write `{ (x,*) }` using q quantification as:

  `{ (x,c) :: c IN some_domain }`
- A *function* is a special case of a *relation* and relation is a set of ordered pairs (tuples). We sometimes manipulate functions or relations using set notation for convenience.
- A *total function* is defined for all elements of its domain. A *partial function* is defined for a subset of the elements of its domain.

Abstract model

We parameterize the abstract model for the Labeled Digraph ADT with the following types.

- `VertexType` is the set of possible vertices (i.e., vertex identifiers).
- `VertexLabelType` is the set of possible labels on vertices. (Values of this type may have several components.)

- `EdgeLabelType` is the set of possible labels on edges. (Values of this type may have several components.)

We model the state of the instance of the Labeled Digraph ADT with an abstract value `G` such that `G = (V,E,VL,EL)` with `G`'s components satisfying the following *Labeled Digraph Properties*.

- `V` is a finite subset of values from the set `VertexType`. `V` denotes the vertices (or nodes) of the digraph.
- `E` is a binary relation on the set `V`. A pair `(v1,v2) IN E` denotes that there is a directed edge from `v1` to `v2` in the digraph.

  Note that this model allows at most one (directed) edge from a vertex `v1` to vertex `v2`.
- `VL` is a total function from set `V` to the set `VertexLabelType`.
- `EL` is a total function from set `E` to the set `EdgeLabelType`.

Interface invariant

We define the following interface invariant for the Labeled Digraph ADT:

> *Any valid labeled digraph instance `G`, appearing in either the arguments or return value of a public ADT operation, must satisfy the Labeled Digraph Properties.*

Constructive semantics

We specify the various ADT operations below using their type signatures, preconditions, and postconditions. Along with the interface invariant, these comprise the (implementation-independent) specification of the ADT.

In these assertions, for a digraph `g` that satisfies the invariants, `G(g)` denotes its abstract model`(V,E,VL,EL)` as described above. The value `Result` denotes the return value of function.

- Constructor `new_graph` creates and returns a new instance of the graph ADT.
  - Precondition:

    `True`
  - Postcondition:

    `G(Result) == ({},{},{},{})`
- Accessor `is_empty g` returns true if and only if graph `g` is empty.
  - Precondition:

```
G(g) = (V,E,VL,EL)
```

  - Postcondition:

```
Result == (V == {} && E == {})
```

- Mutator `add_vertex g nv nl` inserts vertex `nv` with label `nl` into graph `g` and returns the resulting graph.
  - Precondition:

```
G(g) = (V,E,VL,EL) && nv NOT_IN V
```

  - Postcondition:

```
G(Result) == (V UNION {nv}, E, VL UNION {(nv,nl)}, EL)
```

- Mutator `remove_vertex g ov` deletes vertex `ov` from graph `g` and returns the resulting graph.
  - Precondition:

```
G(g) =  (V,E,VL,EL) && ov IN V
```

  - Postcondition:

```
G(Result) == (V', E', VL', EL')
    where V'  = V  - {ov}
          E'  = E  - {(ov,*),(*,ov)}
          VL' = VL - {(ov,*)}
          EL' = EL - {((ov,*),*),((*,ov),*)}
```

- Mutator `update_vertex g ov nl` changes the label on vertex `ov` in graph `g` to be `nl` and returns the resulting graph.
  - Precondition:

```
G(g) =  (V,E,VL,EL) && ov IN V
```

  - Postcondition:

```
G(Result) == (V - {ov}, E, VL', EL)
    where VL' = (VL - {(ov,VL(ov))}) UNION {(ov,nl)}
```

- Accessor `get_vertex g ov` returns the label from vertex `ov` in graph `g`
  - Precondition:

```
G(g) = (V,E,VL,EL) && ov IN V
```

  - Postcondition:

```
Result == VL(ov)
```

- Accessor `has_vertex g ov` returns true if and only if `ov` is a vertex of graph `g`.
  - Precondition:

```
G(g) = (V,E,VL,EL) && ov IN VertexLabelType
```

  - Postcondition:

```
G(Result) == ov IN V
```

- Mutator `add_edge g v1 v2 nl` inserts an edge from vertex `v1` to vertex `v2` in graph `g` and returns the resulting graph.
  - Precondition:

```
G(g) = (V,E,VL,EL) && v1 IN V && v2 IN V &&
(v1,v2) NOT_IN E
```

  - Postcondition:

```
G(Result) == (V, E', VL, EL')
    where E'  = E  UNION {(v1,v2)}
          EL' = EL UNION {((v1,v2),nl)}
```

- Mutator `remove_edge g v1 v2` deletes the edge from vertex `v1` to vertex `v2` from graph `g` and returns the resulting graph.
  - Precondition:

```
G(g) =  (V,E,VL,EL) V - {ov} && (v1,v2) IN E
```

  - Postcondition:

```
G(Result) == (V, E - {(v1,v2)}, VL, EL - { ((v1,v2),*) }
```

- Mutator `update_edge g v1 v2 nl` changes the label on the edge from vertex `v1` to vertex `v2` in graph `g` to have label `nl` and returns the resulting graph.
  - Precondition:

```
G(g) = (V,E,VL,EL) && (v1,v2) IN E
```

  - Postcondition:

```
G(Result) == (V, E, VL, EL')
    where EL' == (EL - {((v1,v2),*)}) UNION {((v2,v2),nl)
```

- Accessor `get_edge g v1 v2` returns the label on the edge from vertex `v1` to vertex `v2` in graph `g`.
  - Precondition:

```
G(g) = (V,E,VL,EL) && (v1,v2) IN E
```

  - Postcondition:

```
Result == EL((v1,v2))
```

- Accessor `has_edge g v1 v2` returns true if and only if there is an edge from a vertex `v1` to a vertex `v2` in graph `g`.

  - Precondition:

    ```
    G(g) = (V,E,VL,EL)
    ```

  - Postcondition:

    ```
    Result == (v1,v2) IN E
    ```

- Accessor `all_vertices g` returns a sequence of all the vertices in graph `g`. The returned sequence is represented by a builtin Haskell list.

  - Precondition:

    ```
    G(g) = (V,E,VL,EL)
    ```

  - Postcondition:

    ```
    (ForAll ov: ov IN Result <=> ov IN V) &&
    length(Result) == size(V)
    ```

- Accessor `from_edges g v1` returns a sequence of all vertices `v2` such that there is an edge from vertex `v1` to vertex `v2` in graph `g`. The returned sequence is represented by a builtin Haskell list.

  - Precondition:

    ```
    G(g) = (V,E,VL,EL) && v1 IN V
    ```

  - Postcondition:

    ```
    (ForAll v2: v2 IN Result <=> (v1,v2) IN E) &&
    length(Result) == (# v2 :: (v1,v2) IN E)
    ```

  Function `from_edges g v1` should return `[]` when `v1` does not appear in `g`, so that it can work well with the Wizard's Adventure game.

- Accessor `all_vertices_labels g` returns a sequence of all pairs `(v,l)` such that `v` is a vertex and `l` is it's label in graph `g`. The returned sequence is represented by a builtin Haskell list.

  - Precondition:

    ```
    G(g) = (V,E,VL,EL)
    ```

  - Postcondition:

    ```
    (ForAll v, l: (v,l) IN Result <=> (v,l) IN VL) &&
    length(Result) == size(VL)
    ```

- Accessor `from_edges_labels g v1` returns a sequence of all pairs `(v2,l)` such that there is an edge `(v1,v2)` labeled with `l` in graph `g`.

  - Precondition:

    ```
    G(g) = (V,E,VL,EL) && v1 IN V
    ```

  - Postcondition:

```
    (ForAll v2, l :: (v2,l) IN Result <=> ((v1,v2),l) IN EL) &&
    length(Result) == (# v2 :: (v1,v2 ) IN E)
```

Function `from_edges_labels g v1` should return `[]` when `v1` does not appear in `g`, so that it can work well with the Wizard's Adventure game.

Haskell module abstract interface

Below we state the header for a Haskell module `Digraph_XXX` that implements the Labeled Digraph ADT. The module name suffix `XXX` denotes the particular implementation for a data representation, but the signatures and semantics of the operations are the same regardless of representation.

The module exports data type `Digraph`, but its constructors are not exported. This allows modules that import `Digraph_XXX` to use the data type without revealing how the data type is implemented. (If we had `Digraph(..)` in the export list, then the data type and all its constructors would be exported.)

```haskell
module DigraphADT_XXX
  ( Digraph              -- constrain ops (Eq a, Show a, Show b, Show c)
  , new_graph            -- Digraph a b c
  , is_empty             -- Digraph a b c -> Bool
  , add_vertex           -- Digraph a b c -> a -> b -> Digraph a b c
  , remove_vertex        -- Digraph a b c -> a -> Digraph a b c
  , update_vertex        -- Digraph a b c -> a -> b -> Digraph a b c
  , get_vertex           -- Digraph a b c -> a -> b
  , has_vertex           -- Digraph a b c -> a -> Bool
  , add_edge             -- Digraph a b c -> a -> a -> c -> Digraph a b c
  , remove_edge          -- Digraph a b c -> a -> a -> Digraph a b c
  , update_edge          -- Digraph a b c -> a -> a -> c -> Digraph a b c
  , get_edge             -- Digraph a b c -> a -> a -> c
  , has_edge             -- Digraph a b c -> a -> a -> Bool
  , all_vertices         -- Digraph a b c -> [a]
  , from_edges           -- Digraph a b c -> a -> [a]
  , all_vertices_labels  -- Digraph a b c -> [(a,b)]
  , from_edges_labels    -- Digraph a b c -> a -> [(a,c)]
  )
where  -- definitions for the types and functions
```

Note: The Glasgow Haskell Compiler (GHC) release 8.2 (July 2017) and the Cabal-Install package manager release 2.0 (August 2017) support a new mixin package system called Backpack. This extension would enable us to define an abstract module "DigraphADT" as a signature file with the above interface. Other modules can then implement this abstract interface thus giving a more explicit and flexible definition of this abstract data type.

List Implementation

Type parameters

The Haskell List representation uses the following values for the type parameters:

- `VertexType` is an instance of Haskell classes `Eq` and `Show` (i.e., can be compared for equality and converted to strings),
- `VertexLabelType` is an instance of Haskell class `Show`,
- `EdgeLabelType` is an instance of Haskell class `Show`.

That is, vertices can be compared for equality. Vertices and both the vertex and edge labels can be displayed as strings.

Note: It may be desirable to require `VertexType` to be from class `Ord` (totally ordered) and `VertexLabelType` and `EdgeLabelType` to be from class `Eq`. These were not necessary for the List implementation, but were necessary for the Map implementation.

Labeled digraph representation

The List implementation represents a labeled digraph as an instance of the Haskell algebraic data type `Digraph`, in particular data constructor `(Graph vs es)`.

In an instance `(Graph vs es)`:

- `vs` is a list of tuples `(v,vl)` where
  - `v` has `VertexType` and represents a vertex of the digraph
  - `vl` has `VertexLabelType` and is the unique label associated with vertex `v`
  - a vertex `v` occurs at most once in `vs` (i.e., vs encodes a function from vertices to vertex labels)
- `es` is a list of tuples `((v1,v2),el)` where
  - `v1` and `v2` are vertices occurring in `vs`, representing a directed edge from `v1` to `v2`
  - `el` has `EdgeLabelType` and is the unique label associated with edge `(v1,v2)`
  - an edge `(v1,v2)` occurs at most once in `vs` (i.e., es encodes a function from edges to edge labels)

In terms of the abstract model, `vs` encodes `VL` directly and, because `VL` is a total function on `V`, it encodes `V` indirectly. Similarly, es encodes `EL` directly and `E` indirectly.

Implementation invariant

Given the above description, we then define the following implementation (representation) invariant for the list-based version of the Labeled Digraph ADT:

> *Any Haskell* ***`Digraph`*** *value* ***`(Graph vs es)`*** *with abstract model* ***`G = (V,E,VL,EL)`****, appearing in either the arguments or return value of an operation, must also satisfy the following:*

```
(ForAll v, l :: (v,l) IN vs  <=> (v,l) IN VL ) &&
(ForAll v1, v2, m :: (v1,v2,m) IN es  <=> ((v1,v2),m) IN EL )
```

Haskell module

The Haskell module for the list representation of the labeled digraph graph ADT is in file `DigraphADT_List.hs`. Its test driver module is in file `DigraphADT_TestList.hs`.

Map Implementation

Type parameters

The Haskell Map representation uses the following values for the type parameters:

- `VertexType` is an instance of Haskell classes `Ord` and Show (i.e., can be compared and also converted to strings)
- `VertexLabelType` is an instance of Haskell classes `Eq` and `Show`.
- `EdgeLabelType` is an instance of Haskell classes `Eq` and `Show`.

Note: In the List version of this ADT, `VertexType` is required to be in classes `Show` and `Eq` (instead of `Ord`). The two label types did not require `Eq`. However, the use of the Map module for implementation in this version requires the new type constraints.

Labeled digraph representation

This implementation represents a labeled digraph as an instance of the Haskell algebraic data type `Digraph`, in particular data constructor `(Graph m)`, where `m` is from the `Data.Map.Strict` collection. (This collection is implemented as a balanced tree.)

An instance of `(Graph m)` corresponds to the abstract model as follows:

- The keys for `Map m` are from `VertexLabelType`.

- `Map m` is defined for all keys `v1` in vertex set `V` and undefined for all other keys.
- For some vertex `v1`, the value of `m` at key `v1` is a pair `(l,es)` where
- `l` is an element of `VertexLabelType` and is the unique label associated with `v1`, that is, `l = VL(v1)`.
    - `es` is the list of all tuples `(v2,el)` such that `(v1,v2) IN E`, `el IN EdgeLabelType`, and `el = EL((v1,v2))`. That is, `(v1,v2)` is an edge and `el` is its unique label.

Implementation invariant

Given the above description, we then define the following implementation (representation) invariant for the list-based version of the Labeled Digraph ADT:

> *Any Haskell `Digraph` value `(Graph m)` with abstract model `G = (V,E,VL,EL)`, appearing in either the arguments or return value of an operation, must also satisfy the following:*

```
(ForAll v1, l, es ::
    ( m(v1) defined && m(v1) == (l,es) ) <=>
    ( VL(v1) == l &&
        (ForAll v2, el :: (v2,el) IN es <=>
                          EL((v1,v2)) == el) ) )
```

Haskell module

The Haskell module for the Map representation of the labeled digraph graph ADT is in file `DigraphADT_Map.hs`. Its test driver module is in file `DigraphADT_TestMap.hs`.

Acknowledgements

In Spring 2017, I adapted and revised this document from comments in my Spring 2015 Haskell implementations of the Labeled Digraph abstract data type. (In addition to the list- and map-based Haskell implementations, I also developed a list-based implementation in Elixir in Spring 2015 and two Scala-based implementations in Spring 2016.)

In Summer and Fall 2017, I continue to revise this document. I plan eventually to integrate it into evolving textbook "Introduction to Functional Programming Using Haskell."

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to

HTML, PDF, and other forms as needed. The HTML version of this document requires use of a browser that supports the display of MathML.

References

[**Barski 2011**] Conrad Barski. "Building a Text Game Engine," *Land of Lisp: Learn to Program in Lisp, One Game at a Time*, pp. 69-84, No Starch Press, 2011. (The Common Lisp example in this chapter is similar to the classic Adventure game; the underlying data structure is a labeled digraph.)

[**Bird-Wadler 1998**] Richard Bird and Philip Wadler. *Introduction to Functional Programming*, Second Edition, Addison Wesley, 1998. [First Edition, 1988]

[**Cunningham 2017**] H. Conrad Cunningham. *Notes on Data Abstraction*, 1996-2017.

[**Dale-Walker 1996**] Nell Dale and Henry M. Walker. "Directed Graphs or Digraphs," Chapter 10, In *Abstract Data Types: Specifications, Implementations, and Applications*, pp. 439-469, D. C. Heath & Co, 1996.

Terms and Concepts

Use of Haskell module hiding features to implement the abstract data type's interface, applying specification concepts, using mathematical concepts to model the data abstraction (graphs, sets, sequences, bags, functions, relations), graph data structure.

The specification model for the abstract data type uses a constructive semantics – an abstract model, invariants, preconditions, and postconditions.