Exploring Languages with Interpreters and Functional Programming Chapter 23

H. Conrad Cunningham

21 October 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires use of a browser that supports the display of MathML. A good choice as of October 2018 is a recent version of Firefox from Mozilla.

23 Data Abstraction Revisited

23.1 Chapter Introduction

This chapter revisits the specification, design, and implementation of data abstraction modules in Haskell. It follows the general approach introduced in Chapter 7 but uses algebraic data types (Chapter 21) to represent the data. An algebraic data enables the Haskell module implementing the abstraction to encapsulate the details of the data structure.

The goals of this chapter are to:

- reinforce the methods for specification and design of data abstractions
- illustrate how to use Haskell modules and algebraic data types to enforce the encapsulation of a module's implementation secrets
- introduce additional concepts and terminology for data abstractions

The concepts and terminology in this chapter are mostly general. They are applicable to most any language. Here we look specifically at Haskell. (I have implemented basically the same data abstraction module in Scala and Elixir.)

23.2 Terminology

Chapter 7 used the term *data abstraction*.

This chapter uses the related term *abstract data type* to refer to a data abstraction. The data abstraction module defines and exports a user-defined type (i.e. an algebraic data type) and a set of operations (i.e. functions) on that type. The type is abstract in the sense that its concrete representation is hidden; only the module's operations may manipulate the representation directly.

For convenience, this chapter sometimes uses acronym *ADT* to refer to an abstract data type.

In Chapters 6 and 7, we explored the concepts of contracts, which include preconditions and postconditions for the functions in the module and interface and implementation) invariants for the data created and manipulated by the module. For convenience, this chapter refers to these as the *abstract model* for the ADT.

23.3 Example: Doubly Labelled Digraph

In this chapter, we develop a family of *doubly labelled digraph* data structures.

As a *graph*, the data structure consists of a finite set of *vertices (nodes)* and a set of *edges*. Each edge connects two vertices. (Some writers require that the set of vertices be nonempty, but here we prefer to allow an empty graph to have no vertices. But the question remains whether such a graph with no vertices is pointless concept.)

As a *directed graph* (or *digraph*), each pair of vertices has at most one edge connecting them; the edge has a direction from one of the edges to the other.

As a *doubly labelled* graph, each vertex and each edge has some user-defined data (i.e. labels) attached.

This chapter draws on the discussion of digraphs and their specification in Chapters 1 and 10 of the Dale and Walker book *Abstract Data Types* [Dale 1996].

23.4 Use Case

For what purpose can we use a doubly labelled digraph data structure?

One concrete use case is to represent the game world in an implementation of an adventure game.

For example, in the Wizard's Adventure Game from Chapter 5 of *Land of Lisp: Learn to Program in Lisp, One Game at a Time* [Barski 2011], the game's rooms become vertices, passages between rooms become edges, and descriptions associated with rooms or passages become labels on the associated vertex or edge (as shown in Figure 23-1).

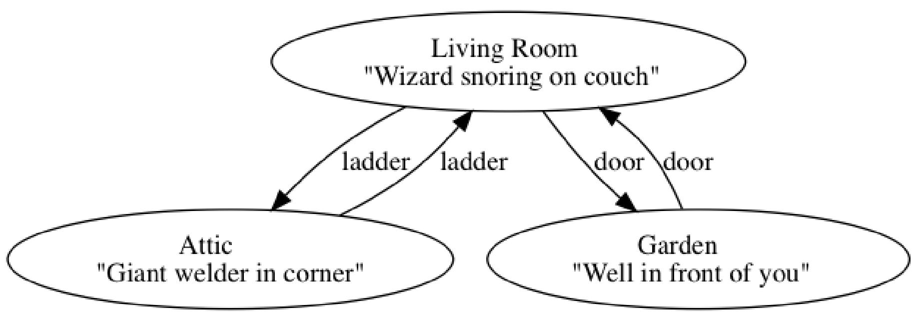

Figure 23-1: Labelled Digraph for Wizard's Adventure Game

Aside: By using a digraph to model the game world, we disallow multiple passages directly from one room to another. By changing the graph to a *multigraph*, we can allow multiple directed edges from one vertex to another.

The Adventure game must create and populate the game world initially, but it does not typically modify the game world during play. It maintains the game state (e.g. player location) separately from the game world. A player moves from room to room during play; the labelled digraph gives the static structure and descriptions of the game world.

23.5 Defining ADTs

How can we define an abstract data type?

The behavior of an ADT is defined by a set of operations that can be applied to an *instance* of the ADT.

Each operation of an ADT can have inputs (i.e. parameters) and outputs (i.e. results). The collection of information about the names of the operations and their inputs and outputs is the *interface* of the ADT.

23.5.1 Specification

To specify an ADT, we need to give:

1. the *name* of the ADT
2. the *sets* (or domains) upon which the ADT is built. These include the type being defined and the auxiliary types (e.g. primitive data types and other ADTs) used as parameters or return values of the operations.
3. the *signatures* (syntax or structure) of the operations
    - name
    - input sets (i.e. the types, number, and order of the parameters)
    - output set (i.e. the type of the return value)
4. the *semantics* (or meaning) of the operations

Note: In this chapter, we more state the specification of the data abstraction more systematically than in Chapter 7. But we are doing essentially the same things we did in Chapter 7.

23.5.2 Operations

We categorize an ADT's operations into four groups depending upon their functionality:

- A *constructor* (sometimes called a creator, factory, or producer function) constructs and initializes an instance of the ADT.
- A *mutator* (sometimes called a modifier, command, or setter function) returns the instance with its state changed.
- An *accessor* (sometimes called an observer, query, or getter function) returns information from the state of an instance without changing the state.
- A *destructor* destroys an instance of the ADT.

We normally list the operations in that order.

For a language with immutable data structures like Haskell, a mutator returns a distinct new instance of the ADT with a state that is a modified version of the original instance's state. That is, we are taking an applicative (or functional or referentially transparent) approach to ADT specifications.

Note: Of course, in an imperative language, a mutator can change the state of an instance in place. That may be more efficient, but it tends to be less safe. It also tends to make concurrent use of an abstract data type more problematic.

Technically speaking, a destructor is not an operation of the ADT. We can represent the other types of operations as functions on the sets in the specification. However, we cannot define a destructor in that way. But destructors are of pragmatic importance in the implementation of ADTs, particularly in languages that do not have automatic storage reclamation (i.e. garbage collection).

23.5.3 Approaches to semantics

There are two primary approaches for specifying the semantics of the operations:

- The *axiomatic* (or *algebraic*) approach gives a set of logical rules (properties or axioms) that relate the operations to one another. The meanings of the operations are defined implicitly in terms of each other.
- The *constructive* (or *abstract model*) approach describes the meaning of the operations explicitly in terms of operations on other abstract data types. The underlying *model* may be any well-defined mathematical model or a previously defined ADT.

In some ways, the axiomatic approach is the more elegant of the two approaches. It is based in the well-established mathematical fields of abstract algebra and category theory. Furthermore, it defines the new ADT independently of other ADTs. To understand the definition of the new ADT it is only necessary to understand its axioms, not the semantics of a model.

However, in practice, the axiomatic approach to specification becomes very difficult to apply in complex situations. The constructive approach, which builds a new ADT from existing ADTs, is the more useful methodology for most practical software development situations.

In this chapter, we use the constructive approach.

23.6 Specification of Labelled Digraph ADT

Now let's look at a constructive specification of the doubly labelled digraph.

First, we specify the ADT as an implementation-independent abstraction. The *secret* of the ADT module is the data structure used internally to implement the doubly labelled digraph.

Then, we examine two implementations of the abstraction:

- using Haskell lists to represent the vertex and edge sets
- using a Haskell `Map` to map a vertex to the set of outgoing edges from that vertex

Before we specify the ADT, let's define the mathematical notation we use. We choose notation that can readily be used in comments in program.

23.6.1 Notation

We use the following notation and terminology to describe the abstract data type's model and its semantics.

- `(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 *Cartesian product* of two sets `C` and `D` is the set of all *ordered pairs* `(x,y)` where `x IN C` and `y 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 a quantification as:

  ```
  { (x,c) :: c IN some_domain }
  ```

- A *relation* on sets `C` and `D` is a subset of the Cartesian product of `C` and `D`. That is, a set of tuples.
- A *function* on sets `C` and `D` is a special case of a relation on `C` and `D` where each value from `C` occurs in at most one tuple in the relation.
- 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.

23.6.2 Sets

The abstract data type being defined is *named* `Digraph`.

We specify that this abstract data type be represented by a Haskell algebraic data type `Digraph a b c`, which has three *type* parameters (i.e. sets):

1. `VertexType`, the set of possible vertices (i.e. vertex identifiers) in the `Digraph`
2. `VertexLabelType`, the set of possible labels on vertices in the `Digraph`
3. `EdgeLabelType`, the set of possible labels on edges in the `Digraph`

Given this ADT defines a digraph, edges can be identified by ordered pairs (tuples) of vertices.

Values the above types, in particular the labels, may have several components.

23.6.3 Signatures

We define the following operations on the Labelled Digraph ADT (shown below as Haskell function signatures).

Given the primary use case described above, we specify a constructor that to create an empty graph (`new_graph`), a mutator to add a new vertex (`add_vertex`), and mutator to add a new edge between existing vertices (`add_edge`).

We also specify mutators to remove vertices (`remove_vertex`) and edges (`remove_edge`) and to update the labels on vertices (`update_vertex`) and edges (`update_edge`). (Note: In the identified use case, these are likely used less often than the mutators that add new vertices and edges.)

Constructors:

```
new_graph :: Digraph a b c
```

Mutators:

```
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
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
```

We specify query functions to check whether the labelled digraph is empty (`is_empty`), has a given vertex (`has_vertex`), and has an edge between two vertices (`has_edge`).

We specfiy accessors to retrieve the label associated with a given vertex (`get_vertex`) and edge (`get_edge`).

Given the identified use case, we also specify accessors to return lists of all vertices in the graph (`all_vertices`) and of just their labels (`all_vertices_labels`) and to return lists of all outgoing edges from a vertex (`from_edges`) and of just their labels (`from_edges_labels`).

Accessors:

```
is_empty      :: Digraph a b c -> Bool
get_vertex    :: Digraph a b c -> a -> b
has_vertex    :: Digraph a b c -> a -> Bool
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)]
```

Given the identified use case and that Haskell uses garbage collection, no destructor seems to be needed in most cases.

Destructors: None

23.6.4 Semantics

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

- `V` is a finite subset of values from the set `VertexType`. `V` denotes the vertices (or nodes) of the digraph.

- Any two elements of `V` can be compared for *equality*.
- `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`. It allows a directed edge from a vertex to itself.

  Also, because vertices can be compared for equality, any two edges can also be compared for equality.
- `VL` is a total function from set `V` to the set `VertexLabelType`.
- `EL` is a total function from set `E` to the set `EdgeLabelType`.

23.6.4.1 Interface invariant

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

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

23.6.4.2 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 (i.e. its abstract interface).

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)
    ```

  Note: 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. We should redefine the precondition and postcondition to specify this behavior.

- 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)` labelled 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)
    ```

Note: 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. We should redefine the precondition and postcondition to specify this behavior.

23.6.5 Haskell module abstract interface

Below we state the header for a Haskell module Digraph_XXX that implements the Labelled 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 knowing 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.

The intention of this interface is to constrain the type parameters of Digraph a b c so that:

- Type a (i.e. type VertexType) must be in Haskell class Eq. This is essentially required by the interface invariant (i.e. the Labelled Digraph Properties).
- Types a, b, and c (i.e. types VertexType, VertexLabelType, and EdgeLabelType) must be in Haskell class Show. This contraint enables the vertices and labels to be displayed as text.

It may be desirable (or necessary) for an implementation to further constrain the type parameters. For example, some implementations may need to constrain VertexType to be from class Ord (i.e. totally ordered).

```haskell
module DigraphADT_XXX
  ( Digraph       --constraints (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.

23.7 List Implementation

This section gives an implementation of the ADT that uses Haskell lists to represent the vertex and edge sets.

23.7.1 Labelled digraph representation

We represent the List implementation of the Labelled Digraph ADT as an instance of the Haskell algebraic data type `Digraph` as shown below. (Remember that type variable `a` is `VertexType`, `b` is `VertexLabelType`, and `c` is `EdgeLabelType`.)

```
data Digraph a b c = Graph [(a,b)] [(a,a,c)]
```

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 `es` (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.

Of course, there are many other ways to represent the graph as lists. This representation is biased for a context where, once built, the labelled digraph is relatively static and the most frequent operations are the retrieval of labels attached to vertices or edges. That is, it is biased toward the Adventure game use case.

Given that all the type parameters must be of class `Show`, we also define `Digraph` to also be of class `Show` as defined below.

```
instance (Show a, Show b, Show c) =>
                    Show (Digraph a b c) where
    show (Graph vs es) =
        "(Digraph " ++ show vs ++ ", " ++ show es ++ ")"
```

23.7.2 Implementation invariant

Given the above description, we then define the following implementation (representation) invariant for the list-based version of the Labelled 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 )
```

23.7.3 Haskell implementation

The code in this section shows a list-based implementation for several of the operations related to vertices.

The Haskell module for the list representation of the Labelled Digraph ADT is in source file `DigraphADT_List.hs`. A simple smoke test driver module is in source file `DigraphADT_TestList.hs`.

The implementations of constructor `new_graph` and accessor `is_empty` are straightforward.

```
new_graph :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c
new_graph = Graph [] []

is_empty :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c -> Bool
```

```
is_empty (Graph [] _ ) = True
is_empty _             = False
```

Function `has_vertex` just needs to search through the list of vertices to determine whether or not the vertex occurs. It relies upon `VertexType` being in class `Eq`.

```
has_vertex :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c -> a -> Bool
has_vertex (Graph vs _) ov =
    not (null [ n | (n,_) <- vs, n == ov])
```

Because of lazy evaluation, the list comprehension only needs to evaluate far enough to find the occurrence of the vertex in the list.

To add a new vertex and its label to the graph, `add_vertex` must return a new graph with the new vertex-label pair added to the head of the vertex list. To meet the specification, it must not allow a vertex to be added if the vertex already occurs in the list.

```
add_vertex :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c -> a -> b -> Digraph a b c
add_vertex g@(Graph vs es) nv nl
    | not (has_vertex g nv) = Graph ((nv,nl):vs) es
    | otherwise             = error has_nv
    where has_nv =
        "Vertex " ++ show nv ++ " already in digraph"
```

Function `remove_vertex` is a bit trickier with this representation. To remove an existing vertex and its label from the graph, `remove_vertex` must return a new graph with that vertex's tuple removed from the list of vertices and with any outgoing edges also removed from the list of edges.

```
remove_vertex :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c -> a -> Digraph a b c
remove_vertex g@(Graph vs es) ov
    | has_vertex g ov = Graph ws fs
    | otherwise       = error no_ov
    where ws    = [ (w,m)      | (w,m) <- vs, w /= ov ]
          fs    = [ (v1,v2,m) |
                    (v1,v2,m) <- es, v1 /= ov, v2 /= ov ]
          no_ov = "Vertex " ++ show ov ++ " not in digraph"
```

The implementation of `remove_vertex` filters all occurrences of the vertex from the list of vertices. Given the implementation invariant, this is not necessary. However, this potentially adds some safety to the implementation at the possible expense of execution time.

For an existing vertex in the list of vertices, function `update_vertex` replaces the old label with the new label. Like `remove_vertex`, it potentially processes the entire list of vertices and makes the change to all occurrences, when the implementation invariant would allow it to stop on the first (and only) occurrence.

```
update_vertex :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c -> a -> b -> Digraph a b c
update_vertex g@(Graph vs es) ov nl
    | has_vertex g ov = Graph (map chg vs) es
    | otherwise       = error no_ov
    where chg (w,m) = (if w == ov then (ov,nl) else (w,m))
          no_ov     = "Vertex " ++ show ov ++ " not in digraph"
```

For an existing vertex, function `get_vertex` retrieves the label. Because of lazy evaluation, the search of the list of vertices stops with the first occurrence.

```
get_vertex :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c -> a -> b
get_vertex (Graph vs _)  ov
    | not (null ls) = head ls
    | otherwise     = error no_ov
    where ls    = [ l | (w,l) <- vs, w == ov]
          no_ov = "Vertex " ++ show ov ++ " not in digraph"
```

The remainder of the functions are defined in file `DigraphADT_List.hs`.

We can create an empty labelled digraph `g0` having `Int` identifiers for vertices, `Int` labels for vertices, and `Int` labels for edges as follows:

```
g0  = (new_graph :: Digraph Int Int Int)
```

Then we can add a new vertex with identifier `1` and vertex label `101` as follows:

```
g1  = add_vertex g0 1 101
```

23.7.4 Improvements to the list implementation

Based on the list-based design and implementation above, what improvements should we consider? Here are some possibilities.

1. As described above, the current list implementations of functions such as `remove_vertex` and `update_vertex` do some unnecessary work with respect to the implementation invariant. This could be eliminated.

2. The data representation (i.e. implementation invariant) could be changed to allow, for example, multiple occurrences of vertices in the vertex list. This would avoid the checks of `has_vertex` in `add_vertex` and `update_vertex`. Then, as it does above, `remove_vertex` needs to remove all occurrences of the vertex.

   Other functions would need to be modified accordingly so that they only access the first occurrence of a vertex (especially the `all_vertices` and `all_vertices_labels` functions).

A similar change could be made to the list of edges.

Note: The Labelled Diagraph ADT specification does not specify what the behavior should be when the referenced vertex or edge is not defined. The change suggested in this item gives non-error behavior to those situations. Perhaps a better alternative would be to change the general ADT specification to require specific behaviors in those cases.

3. Most of the functions throw an `error` exception when the vertex they reference does not exist. A better Haskell design would redefine these functions to return a `Maybe` or `Either` value. This would eliminate most of the `has_vertex` checks and make the functions defined on all possible inputs.

   This would require changes to the overall Labelled Digraph ADT specification and its abstract interface.

4. New functions could be added to the Labelled Digraph ADT—such as an equality check on graphs, a constructor that creates a copy of an existing graph, or functions to apply various graph algorithms.

5. Existing functions could be eliminated. For example, if the graph is only constructed and used for retrieval, then the remove and update functions could be eliminated.

23.8 Map Implementation

This section gives an implementation of the ADT that uses a Haskell `Map` to map a vertex to the set of outgoing edges from that vertex

23.8.1 Labelled digraph representation

We represent the Map implementation of the Labelled Digraph ADT as an instance of the Haskell algebraic data type `Digraph` as shown below. (Remember that type variable `a` is `VertexType`, `b` is `VertexLabelType`, and `c` is `EdgeLabelType`.)

```
import qualified Data.Map.Strict as M

data Digraph a b c = Graph (M.Map a (b,[(a,c)]))
```

In the data constructor `(Graph m)`, `m` is an instance of `Data.Map.Strict`. This collection is set of key-value pairs implemented as a balanced tree, giving logarithmic access time.

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

- The keys for the `Map m` collection are of `VertexLabelType`.

  The interface invariant requires that `VertexType` be in class `Eq`. The implementation based on `Data.Map.Strict` further constrains vertices to be in subclass `Ord` because the vertices are the keys of the `Map`.

- `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.

Given that all the type parameters must be of class `Show`, we also define `Digraph` to also be of class `Show` as defined below.

```
instance (Show a, Show b, Show c) => Show (Digraph a b c) where
    show (Graph m) = "(Digraph " ++ show (M.toAscList m) ++ ")"
```

23.8.2 Implementation invariant

Given the above description, we then define the following implementation (representation) invariant for the list-based version of the Labelled 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) ) )
```

23.8.3 Haskell module

The code in this section shows a map-based implementation for the same operations we examined for the list-based implementation.

The Haskell module for the map representation of the Labelled Digraph ADT is in source file `DigraphADT_Map.hs`. A simple smoke test driver module is in source file `DigraphADT_TestMap.hs`.

Constructor `new_graph` and accessors `is_empty` and `has_vertex` are just wrappers for functions from `Data.Map.Strict`.

```haskell
new_graph :: (Ord a, Show a, Show b, Show c) =>
               Digraph a b c
new_graph = Graph M.empty

is_empty :: (Ord a, Show a, Show b, Show c) =>
                Digraph a b c -> Bool
is_empty (Graph m) = M.null m

has_vertex :: (Ord a, Show a, Show b, Show c) =>
                Digraph a b c -> a -> Bool
has_vertex (Graph m) ov = M.member ov m
```

To add a new vertex and label to the graph, `add_vertex` must return a graph with the new key-value pair inserted into the existing graph's `Map`. The value consists of the label paired with a nil list of adjacent edges. To meet the specification, it must not allow a vertex to be added if the vertex already occurs in the list.

```haskell
add_vertex :: (Ord a, Show a, Show b, Show c) =>
                Digraph a b c -> a -> b -> Digraph a b c
add_vertex g@(Graph m) nv nl
    | not (has_vertex g nv) = Graph (M.insert nv (nl,[]) m)
    | otherwise             = error has_nv
    where has_nv =
        "Vertex " ++ show nv ++ " already in digraph"
```

Except for making sure the vertex to be deleted is the graph, function `remove_vertex` is just a wrapper for the `Data.Map.Strict.delete` function.

```haskell
remove_vertex :: (Ord a, Show a, Show b, Show c) =>
                    Digraph a b c -> a -> Digraph a b c
remove_vertex g@(Graph m) ov
    | has_vertex g ov = Graph (M.delete ov m)
    | otherwise       = error no_ov
    where no_ov = "Vertex " ++ show ov ++ " not in digraph"
```

If the argument vertex is in the graph, then function `update_vertex` retrieves its old label and edge list and then reinserts the new label paired with the same edge list.

```haskell
update_vertex :: (Ord a, Show a, Show b, Show c) =>
                    Digraph a b c -> a -> b -> Digraph a b c
update_vertex g@(Graph m) ov nl
    | has_vertex g ov =
        Graph (M.insert ov (upd (M.lookup ov m)) m)
    | otherwise       = error no_ov
    where upd (Just (ol,edges)) = (nl,edges)
          upd _                 = error no_entry
          no_ov    = "Vertex " ++ show ov ++ " not in digraph"
```

```
            no_entry =
                "Missing/malformed value for vertex " ++ show ov
```

For an existing vertex, function `get_vertex` retrieves the associated value and extracts the label.

```
    get_vertex :: (Ord a, Show a, Show b, Show c) =>
                      Digraph a b c -> a -> b
    get_vertex g@(Graph m)  ov
        | has_vertex g ov = getlabel (M.lookup ov m)
        | otherwise       = error no_ov
        where getlabel (Just (ol,_)) = ol
              no_ov = "Vertex " ++ show ov ++ " not in digraph"
```

The remainder of the functions are defined in file `DigraphADT_Map.hs`.

The Map-based functions can be called in the same manner as the List-based function, except that the vertices must be in class `Ord`{.haskell_.

23.8.4 Improvements to the map implementation

All the improvements suggested for the list-based implementation apply to the map-based implementation except for the first.

For large graphs, the map-based implementation should perform better than the list-based implementation.

For large graphs with many outgoing edges on each vertex, it might be useful to implement the edge-list itself with a `Map`.

23.9 What Next?

This chapter revisited the issues of specification, design, and implementation of data abstractions as modules in Haskell. It used a labelled digraph data structure as the example.

Although we may not specify all subsequent Haskell modules as systematically as we did in this chapter, we do use the modular style of programming in the various interpreters developed in Chapter 41 and following.

In the future, we plan to implement a Adventure game on top of the ADT implemented in this chapter.

23.10 Exercise Set A

1. Restate the preconditions and postconditions for functions `from_edges` and `from_edges` so that they must return empty lists when the argument

vertex `v1` is not in the vertex set. (See the notes on these operations in the semantic specification above.)

2. Develop a comprehensive test script for the Labelled Digraph ADT implementations using blackbox, module-level, functional testing as described in Chapters 11 and 12.

3. Adapt the Haskell Labelled Digraph ADT interface and it two implementations to use GHC's Backpack module system.

4. Specify a similar Labelled Digraph ADT as a Java interface.

5. Give two different implementations of the Labelled Digraph ADT in Java using the specification from the previous exercise.

6. Specify a similar Labelled Digraph ADT as a Python 3 module.

7. Give two different implementations of the Labelled Digraph ADT in Python 3 using the specification from the previous exercise.

8. Choose one of the improvements described in the "Improvements in the list implementation" subsection and change the specification and list implementation as needed for the improvement.

9. Choose one of the improvements and change the specification and map implementation as needed for the improvement.

10. Specify a doubly labelled directed multigraph data structure to replace the doubled labelled digraph. (That is, allow multiple directed edges from one vertex to another.)

11. Give an implementation of the doubly labelled directed multigraph specified in the previous exercise.

23.11 Mealy Machine Simulator Project

In this project, you are asked to design and implement Haskell modules to represent Mealy Machines and to simulate their execution.

This kind of machine is a useful abstraction for simple controllers that listen for input events and respond by generating output events. For example in an automobile application, the input might be an event such as "fuel level low" and the output might be command to "display low-fuel warning message".

In the theory of computation, a *Mealy Machine* is a *finite-state automaton* whose output values are determined both by its current state and the current input. It is a *deterministic finite state transducer* such that, for each state and input, at most one transition is possible.

Appendix A of the Linz textbook [Linz 2017] defines a Mealy Machine mathematically by a tuple

$M = (Q, \Sigma, \Gamma, \delta, \theta, q_0)$

where

Q is a finite set of internal states
Σ is the input alphabet (a finite set of values)
Γ is the output alphabet (a finite set of values)
$\delta : Q \times \Sigma \longrightarrow Q$ is the transition function
$\theta : Q \times \Sigma \longrightarrow \Gamma$ is the output function
q_0 is the initial state of M (an element of Q)

In an alternative formulation, the transition and output functions can be combined into a single function:

$\delta : Q \times \Sigma \longrightarrow Q \times \Gamma$

We often find it useful to picture a finite state machine as a *transition graph* where the states are mapped to vertices and the transition function represented by directed edges between vertices labelled with the input and output symbols.

23.12 Exercise Set B

1. Specify, design, and implement a general representation for a Mealy Machine as a Haskell module implementing an abstract data type. It should hide the representation of the machine and should have, at least, the following public operations.

   - `newMachine s` creates a new machine with initial (and current) state `s` and no transitions.

     Note: This assumes that the state, input, and output sets are exactly those added with the mutator operations below. An alternative would be to change this function to take the allowed state, input, and output sets.

   - `addState m s` adds a new state `s` to machine `m` and returns an `Either` wrapping the modified machine or an error message.

   - `addTransition m s1 in out s2` adds a new transition to machine `m` and returns an `Either` wrapping the modified machine or an error message. From state `s1` with input `in` the modified machine outputs `out` and transitions to state `s2`.

   - `addResets m` adds all reset transitions to machine `m` and returns the modified machine. From state `s1` on input `in` the modified machine outputs `out` and transitions to state `s2`. This operation makes the transition function a total function by adding any missing transitions from a state back to the initial state.

- `setCurrent m s` sets the current state of machine `m` to `s` and returns an `Either` wrapping the modified machine or an error message.
- `getCurrent m` returns the current state of machine `m`.
- `getStates m` returns a list of the elements of the state set of machine `m`.
- `getInputs m` returns a list of the input set of machine `m`.
- `getOutputs m` returns a list of the output set of machine `m`.
- `getTransitions m` returns a list of the transition set of machine `m`. Tuple `(s1,in,out,s2)` occurs in the returned list if and only if, from state `s1` with input `in`, the machine outputs `out` and moves to state `s2`.
- `getTransitionsFrom m s` returns an `Either` wrapping a list of the set of transitions enabled from state `s` of machine `m` or an error message.

2. Given the above implementation for a Mealy Machine, design and implement a separate Haskell module that simulates the execution of a Mealy Machine. It should have, at least, the following new public operations.

    - `move m in` moves machine `m` from the current state given input `in` and returns an `Either` wrapping a tuple `(m',out)` or an error message. The tuple gives the modified machine `m'` and the output `out`.
    - `simulate m ins` simulates execution of machine `m` from its current state through a sequence of moves for the inputs in list `ins` and returns an `Either` wrapping a tuple `(m',outs)` or an error message. The tuple gives the modified machine `m'` after the sequence of moves and the output list `outs`.

    Note: It is possible to use a Labelled Digraph ADT module in the implementation of the Mealy Machine.

3. Implement a Haskell module that uses a different representation for the Mealy Machine. Make sure the simulator module still works correctly.

23.13 Acknowledgements

In Spring 2017, I created a Labelled Digraph ADT document by adapting and revising comments from the Haskell implementations of the Labelled Digraph abstract data type. I had specified the ADT and developed the implementations as my solution for Assignment #1 in CSci 556 (Multiparadigm Programming) in Spring 2015. I also included some content from my notes on Data Abstraction [Cunningham 2017].

(In addition to the list- and map-based Haskell implementations of the Labelled Digraph ADT, I developed a list-based implementation in Elixir in Spring 2015 and two Scala-based implementations in Spring 2016.)

In Spring 2017, I also created a Mealy Machine Simulator Exercise document by adapting and revising a project I had assigned in the Scala-based offering of CSci 555 (Functional Programming) in Spring 2016.

In 2018, I merged and revised these documents to become new Chapter 23, Data Abstraction Revisited, in the textbook *Exploring Languages with Interpreters and Functional Programming*.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

23.14 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 labelled digraph.)

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

[Dale 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.

[Linz 2017]: Peter Linz. *Formal Languages and Automata*, 6th Edition, Jones & Bartlett, 2017.

23.15 Terms and Concepts

Data abstraction; abstract data type (ADT), instance; specification of ADTs using name, sets, signatures, and semantics; constructor, accessor, mutator, and destructor operations; axiomatic and constructive semantics; abstract model (contract, precondition, postcondition, interface and implementation invariant, abstract interface); use of Haskell module hiding features to implement the abstract data type's interface; using mathematical concepts to model the data abstraction (graph, digraph, labelled graph, multigraph, set, sequence, bag, total and partial functions, relation); graph data structure; adventure game.

Mealy Machine, simulator, finite-state automaton (machine), deterministic finite state transducer, state, transition, transition graph.