Exploring Languages with Interpreters and Functional Programming
Chapter 23

H. Conrad Cunningham

4 August 2018

Contents

23 Data Abstraction Revisited 2
    23.1 Chapter Introduction .......................... 2
    23.2 Terminology .................................. 2
    23.3 Example: Doubly Labelled Digraph ............. 2
    23.4 Use Case? .................................... 3
    23.5 Defining ADTs ................................ 3
        23.5.1 Specification ............................ 3
        23.5.2 Signatures ............................... 4
        23.5.3 Approaches to semantics ................. 4
    23.6 Specification of Labelled Digraph ADT .......... 5
        23.6.1 Notation ................................ 5
        23.6.2 Abstract model .......................... 6
        23.6.3 Interface invariant ...................... 7
        23.6.4 Constructive semantics .................. 7
        23.6.5 Haskell module abstract interface ....... 11
    23.7 List Implementation ........................... 12
        23.7.1 Type parameters ........................ 12
        23.7.2 Labeled digraph representation .......... 12
        23.7.3 Implementation invariant ............... 13
        23.7.4 Haskell module ........................... 13
    23.8 Map Implementation ............................ 13
        23.8.1 Type parameters ........................ 13
        23.8.2 Labeled digraph representation .......... 14
        23.8.3 Implementation invariant ............... 14
        23.8.4 Haskell module ........................... 14
    23.9 What Next? .................................. 15
    23.10 Exercise Set A ............................... 15
    23.11 Mealy Machine Simulator Case Study ......... 15
23 Data Abstraction Revisited

23.1 Chapter Introduction

This chapter explores the design and implementation of a data abstraction module in Haskell. It follows the general approach introduced in Chapter 7 but uses an algebraic data type (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 use of methods for specification and design of data abstractions
- illustrate how to use Haskell features 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 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 the 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, nonempty set of vertices (nodes) and a set of edges. Each edge connects two vertices.
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 arbitrary data attached.


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.

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 Signatures

To specify the signatures for the operations, we use the notation for mathematical functions. By a tuple like \((X, Y)\), we mean the Cartesian product of sets \(X\) and \(Y\), that is, the set of ordered pairs where the first component is from \(X\) and the second is from \(Y\). The set to the right of the \(\rightarrow\) is the return type of the function.

We categorize the operations into one of 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. Of course, in an imperative language, a mutator can change the state of an instance.

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, then we examine two implementations of that 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 $\cup$ D denotes set union, that is, a set that includes all the element of both C and D.

• For sets C and D, C $\cap$ 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 $\subseteq$ 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.

23.6.2 Abstract model

The specification of a doubly labelled digraph involves the following sets—that is, Haskell types. (The latter three are essentially type parameters of the abstract model.)

• Digraph is the abstract data type being defined.

• 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 \( \text{VertexType} \). \( V \) denotes the vertices (or nodes) of the digraph.

• \( E \) is a binary relation on the set \( V \). A pair \((v_1,v_2) \in E\) denotes that there is a directed edge from \( v_1 \) to \( v_2 \) in the digraph.

Note that this model allows at most one (directed) edge from a vertex \( v_1 \) to vertex \( v_2 \). It allows a directed edge from a vertex to itself.

• \( VL \) is a total function from set \( V \) to the set \( \text{VertexLabelType} \).

• \( EL \) is a total function from set \( E \) to the set \( \text{EdgeLabelType} \).

23.6.3 Interface invariant

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

\[
\text{Any valid labeled digraph instance } G, \text{ appearing in either the arguments or return value of a public ADT operation, must satisfy the Labeled Digraph Properties.}
\]

23.6.4 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 \text{new_graph} creates and returns a new instance of the graph ADT.
  
  – Precondition:
    \text{True}
  
  – Postcondition:
    \( G(\text{Result}) == (\emptyset,\emptyset,\emptyset,\emptyset) \)

• Accessor \text{is_empty} \ g returns true if and only if graph \( g \) is empty.
  
  – Precondition:
    \( G(\text{g}) = (V,E,VL,EL) \)
  
  – Postcondition:
    \( \text{Result} == (V == \{} \&\& E == \{\}) \)
• Mutator add_vertex g nv nl inserts vertex nv with label nl into graph g and returns the resulting graph.
  – Precondition:
    \[ G(g) = (V,E,VL,EL) \land \text{nv NOT IN V} \]
  – Postcondition:
    \[ G(\text{Result}) = (V \cup \{\text{nv}\}, E, VL \cup \{(\text{nv},\text{nl})\}, EL) \]

• Mutator remove_vertex g ov deletes vertex ov from graph g and returns the resulting graph.
  – Precondition:
    \[ G(g) = (V,E,VL,EL) \land \text{ov IN V} \]
  – Postcondition:
    \[ G(\text{Result}) = (V \setminus \{\text{ov}\}, E \setminus \{(\text{ov},\_),(\_,\text{ov})\}, VL \setminus \{(\text{ov},\_}\}, EL \setminus \{((\text{ov},\_),\_),((\_),\text{ov})\}) \]

• Mutator update_vertex g ov nl changes the label on vertex ov in graph g to be nl and returns the resulting graph.
  – Precondition:
    \[ G(g) = (V,E,VL,EL) \land \text{ov IN V} \]
  – Postcondition:
    \[ G(\text{Result}) = (V \setminus \{\text{ov}\}, E, VL \setminus \{(\text{ov},\_}\}) \]

• Accessor get_vertex g ov returns the label from vertex ov in graph g.
  – Precondition:
    \[ G(g) = (V,E,VL,EL) \land \text{ov IN V} \]
  – Postcondition:
    Result = VL(ov)

• Accessor has_vertex g ov returns true if and only if ov is a vertex of graph g.
  – Precondition:
    \[ G(g) = (V,E,VL,EL) \land \text{ov IN VertexLabelType} \]
  – Postcondition:
    \[ G(\text{Result}) = \text{ov IN V} \]
• 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:
  \[ G(g) = (V,E,VL,EL) \land v1 \in V \land v2 \in V \land (v1,v2) \not\in E \]
  
  – Postcondition:
  \[ G(\text{Result}) = (V,E',VL,EL') \]
  where
  \[ E' = E \cup \{(v1,v2)\} \]
  \[ EL' = EL \cup \{(v1,v2),nl\} \]

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

  – Precondition:
  \[ G(g) = (V,E,VL,EL) \land \{v1,v2\} \in E \]

  – Postcondition:
  \[ G(\text{Result}) = (V,E - \{(v1,v2)\},VL,EL - \{((v1,v2),\ast)\}) \]

• 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:
  \[ G(g) = (V,E,VL,EL) \land (v1,v2) \in E \]

  – Postcondition:
  \[ G(\text{Result}) = (V,E,VL,EL') \]
  where
  \[ EL' = EL - \{((v1,v2),\ast)\} \cup \{((v1,v2),nl)\} \]

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

  – Precondition:
  \[ G(g) = (V,E,VL,EL) \land (v1,v2) \in E \]

  – Postcondition:
  \[ \text{Result} = EL((v1,v2)) \]

• 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 its 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)
Note: Function \texttt{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 \texttt{Digraph\_XXX} that implements the Labeled Digraph ADT. The module name suffix \texttt{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 \texttt{Digraph}, but its constructors are not exported. This allows modules that import \texttt{Digraph\_XXX} to use the data type without revealing how the data type is implemented.

If we had \texttt{Digraph(\ldots)} in the export list, then the data type and all its constructors would be exported.

\begin{verbatim}
module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module DigraphADT_XXX

т module Digra...
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 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

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.

23.7.2 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,v1)` where
  - `v` has `VertexType` and represents a vertex of the digraph
  - `v1` 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.

### 23.7.3 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 \((\text{Graph } vs \hspace{1em} es)\) with abstract model \(G = (V,E,VL,EL)\), appearing in either the arguments or return value of an operation, must also satisfy the following:

\[
\begin{align*}
(\forall v, l :: (v, l) \in vs \iff (v, l) \in VL) \land \\
(\forall v_1, v_2, m :: (v_1, v_2, m) \in es \iff ((v_1, v_2), m) \in EL)
\end{align*}
\]

### 23.7.4 Haskell module

The Haskell module for the list representation of the labeled digraph graph ADT is in file \(\text{DigraphADT\_List.hs}\). Its test driver module is in file \(\text{DigraphADT\_TestList.hs}\).

TODO: Update the testing code to follow the approach from Chapters 11 and 12.

### 23.8 Map Implementation

This section gives an implementation of the ADT that uses a Haskell \(\text{Map}\) to map a vertex to the set of outgoing edges from that vertex.

#### 23.8.1 Type parameters

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

- **VertexType** is an instance of Haskell classes \(\text{Ord}\) and \(\text{Show}\) (i.e. can be compared and also converted to strings)
- **VertexLabelType** is an instance of Haskell classes \(\text{Eq}\) and \(\text{Show}\).
- **EdgeLabelType** is an instance of Haskell classes \(\text{Eq}\) and \(\text{Show}\).

Note: In the List version of this ADT, **VertexType** is required to be in classes \(\text{Show}\) and \(\text{Eq}\) (instead of \(\text{Ord}\)). The two label types did not require \(\text{Eq}\). However, the use of the Map module for implementation in this version requires the new type constraints.
23.8.2 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.

23.8.3 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:

\[
\forall v1, l, es :: \begin{align*}
& (m(v1) \text{ defined } \land m(v1) = (l,es)) \iff \\
& (VL(v1) = l \land \\
& \forall v2, el :: (v2,el) \text{ IN } E \iff \\
& \quad EL((v1,v2)) = el) 
\end{align*}
\]

23.8.4 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`.

TODO: Update the testing code to follow the approach from Chapters 11 and 12.
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 $v_1$ is not in the vertex set. (See the notes on these operations in the semantic specification above.)

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

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

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

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

6. Adapt the Haskell Labelled Digraph ADT interface and its two implementations to use the Backpack module system.

23.11 Mealy Machine Simulator Case Study

In this case study, 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
- \( \Sigma \) is the input alphabet (a finite set of values)
- \( \Gamma \) is the output alphabet (a finite set of values)
- \( \delta : Q \times \Sigma \rightarrow Q \) is the transition function
- \( \theta : Q \times \Sigma \rightarrow \Gamma \) 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:

\[ \delta : Q \times \Sigma \rightarrow Q \times \Gamma \]

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** \( m \) \( s1 \) \( \text{in} \) \( s2 \) adds a new transition to machine \( m \) and returns an `Either` wrapping the modified machine or an error message. From state \( s1 \) with input \( \text{in} \) the modified machine outputs \( \text{out} \) and transitions to state \( s2 \).

   - **addResets** \( m \) adds all reset transitions to machine \( m \) and returns the modified machine. From state \( s1 \) on input \( \text{in} \) the modified machine outputs \( \text{out} \) 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 \((s_1, \text{in}, \text{out}, s_2)\) occurs in the returned list if and only if, from state \( s_1 \) with input \( \text{in} \), the machine outputs \( \text{out} \) and moves to state \( s_2 \).
- **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** \( m, \text{in} \) moves machine \( m \) from the current state given input \( \text{in} \) and returns an Either wrapping a tuple \((m', \text{out})\) or an error message. The tuple gives the modified machine \( m' \) and the output \( \text{out} \).
- **simulate** \( m, \text{ins} \) simulates execution of machine \( m \) from its current state through a sequence of moves for the inputs in list \( \text{ins} \) and returns an Either wrapping a tuple \((m', \text{outs})\) or an error message. The tuple gives the modified machine \( m' \) after the sequence of moves and the output list \( \text{outs} \).

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 Diagraph ADT document by adapting and revising comments from the Haskell implementations of the Labeled 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 labeled digraph.)


### 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 (graphs, sets, sequences, bags, total and partial functions, relations); graph data structure; adventure game.

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