Exploring Languages with Interpreters and Functional Programming Chapter 7

H. Conrad Cunningham

04 April 2022

Contents

Professor of Computer and Information Science

University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

7 Data Abstraction

7.1 Chapter Introduction

Chapter 2 introduced the concepts of procedural and data abstraction. Chapter 6 focuses on procedural abstraction and modular design and programming. This chapter focuses on data abstraction. 4 The goals of this chapter are to:

- illustrate use of data abstraction
- reinforce and extend the concepts of modular design and programming using Haskell modules

The chapter uses the development of a rational arithmetic package to illustrate data abstraction.

7.2 Data Abstraction Review

As defined in Chapter 2, *data abstraction* is the separation of the logical properties of *data* from the details of how the data are represented.

In data abstraction, programmers primarily focus on the problem's data and secondarily on its actions. Programmers first identify the key data entities and develop the programs around those and the operations needed to create and update them.

Data abstraction seeks to make a program robust with respect to change in the data.

7.3 Using Data Abstraction

As in Chapter 6, let's begin the study of this design technique with an example.

7.3.1 Rational number arithmetic

For this example, let's implement a group of Haskell functions to perform rational number arithmetic, assuming that the Haskell library does not contain such a data type. We focus first on the operations we want to perform on the data.

In mathematics we usually write rational numbers in the form $\frac{x}{y}$ where x and y are integers and $y \neq 0$.

For now, let us assume we have a special type `Rat` to represent rational numbers and a constructor function

```
makeRat :: Int -> Int -> Rat
```

to create a Haskell rational number instance from a numerator `x` and a denominator `y`. That is, `makeRat x y` constructs a Haskell rational number with mathematical value $\frac{x}{y}$, where $y \neq 0$.

Let us also assume we have selector functions `numer` and `denom` with the signatures:

```
numer, denom :: Rat -> Int
```

Functions `numer` and `denom` take a valid Haskell rational number and return its numerator and denominator, respectively.

Requirement: For any `Int` values `x` and `y` where $y \neq 0$, there exists a Haskell rational number `r` such that `makeRat x y == r` and rational number values $\frac{\texttt{numer r}}{\texttt{denom r}} = \frac{\texttt{x}}{\texttt{y}}$.

Note: In this example, we use fraction notation like $\frac{\texttt{x}}{\texttt{y}}$ to denote the mathematical value of the rational number. In constrast, `r` above denotes a Haskell value representing a rational number.

We consider how to implement rational numbers in Haskell later, but for now let's look at rational arithmetic implemented using the constructor and selector functions specified above.

Given our knowledge of rational arithmetic from mathematics, we can define the operations for unary negation, addition, subtraction, multiplication, division, and equality as follows. We assume that the operands `x` and `y` are values created by the constructor `makeRat`.

```haskell
negRat :: Rat -> Rat
negRat x = makeRat (- numer x) (denom x)

addRat, subRat, mulRat, divRat :: Rat -> Rat -> Rat  -- (1)
addRat x y = makeRat (numer x * denom y + numer y * denom x)
                     (denom x * denom y)
subRat x y = makeRat (numer x * denom y - numer y * denom x)
                     (denom x * denom y)
mulRat x y = makeRat (numer x * numer y) (denom x * denom y)
divRat x y  -- (2) (3)
    | eqRat y zeroRat = error "Attempt to divide by 0"
    | otherwise       = makeRat (numer x * denom y)
                                (denom x * numer y)

eqRat :: Rat -> Rat -> Bool
eqRat x y = (numer x) * (denom y) == (numer y) * (denom x)
```

The above code:

1. combines the type signatures for all four arithmetic operations into a single declaration by listing the names separated by commas
2. introduces the parameterless function `zeroRat` to abstract the constant rational number value 0

Note: We could represent zero as `makeRat 0 1` but choose to introduce a separate abstraction.

3. calls the `error` function for an attempt to divide by zero

These arithmetic functions do not depend upon any specific representation for rational numbers. Instead, they use rational numbers as a *data abstraction* defined by the type `Rat`, constant `zeroRat`, constructor function `makeRat`, and selector functions `numer` and `denom`.

The goal of a data abstraction is to separate the logical properties of *data* from the details of how the data are represented.

7.3.2 Rational number data representation

Now, how can we represent rational numbers?

For this package, we define type synonym `Rat` to denote this type:

```
type Rat = (Int, Int)
```

For example, `(1,7)`, `(-1,-7)`, `(3,21)`, and `(168,1176)` all represent the value $\frac{1}{7}$.

As with any value that can be expressed in many different ways, it is useful to define a single *canonical* (or *normal*) form for representing values in the rational number type `Rat`.

It is convenient for us to choose a Haskell rational number representation `(x,y)` that satisfies all parts of the following **Rational Representation Property**:

- `(x,y)` $\in$ `(Int,Int)`
- `y > 0`
- if `x == 0`, then `y == 1`
- `x` and `y` are relatively prime
- rational number value is $\frac{\mathtt{x}}{\mathtt{y}}$

By *relatively prime*, we mean that the two integers have no common divisors except 1.

This representation keeps the magnitudes of the numerator `x` and denominator `y` small, thus reducing problems with overflow arising during arithmetic operations.

This representation also gives a unique representation for zero. For convenience, we define the name `zeroRat` to represent this constant:

```
zeroRat :: (Int,Int)
zeroRat = (0,1)
```

We can now define constructor function `makeRat x y` that takes two `Int` values (for the numerator and the denominator) and returns the corresponding Haskell rational number in this canonical form.

```
makeRat :: Int -> Int -> Rat
makeRat x 0 = error ( "Cannot construct a rational number "
                      ++ show x ++ "/0" )          -- (1)
makeRat 0 _ = zeroRat
makeRat x y = (x' `div` d, y' `div` d)             -- (2)
    where x' = (signum' y) * x                     -- (3,4)
          y' = abs' y
          d  = gcd' x' y'
```

In the definition of `makeRat`, we use features of Haskell we have not used in the previous examples. the above code:

1. uses the infix `++` (read "append") operator to concatenate two strings

   We discuss `++` in the chapter on infix operations.

2. puts backticks (`) around an alphanumeric function name to use that function as an infix operator

   The function `div` denotes integer division. Above the `div` operator denotes the integer division function used in an infix manner.

3. uses a `where` clause to introduce `x'`, `y'`, and `d` as local definitions within the body of `makeRat`

   These local definition can be accessed from within `makeRat` but not from outside the function. In contrast, `sqrtIter` in the Square Root example is at the same level as `sqrt'`, so it can be called by other functions (in the same Haskell module at least).

   The `where` feature allows us to introduce new definitions in a top-down manner—first using a symbol and then defining it.

4. uses *type inference* for local variables `x'`, `y'`, and `d` instead of giving explicit type definitions

   These parameterless functions could be declared

   ```
   x', y', d :: Int
   ```

   but it was not necessary because Haskell can infer the types from the types involved in their defining expressions.

   Type inference can be used more broadly in Haskell, but explicit type declarations should be used for any function called from outside.

We require that `makeRat x y` satisfy the *precondition*:

```
y /= 0
```

The function generates an explicit error exception if it does not.

As a *postcondition*, we require `makeRat x y` to return a result `(x',y')` such that:

- `(x',y')` satisfies the Rational Representation Property
- rational number value is $\frac{\texttt{x}}{\texttt{y}}$

Note: Together the two postcondition requirements imply that $\frac{\texttt{x'}}{\texttt{y'}} = \frac{\texttt{x}}{\texttt{y}}$.

The function `signum'` (similar to the more general function `signum` in the Prelude) takes an integer and returns the integer `-1`, `0`, or `1` when the number is negative, zero, or positive, respectively.

```
signum' :: Int -> Int
signum' n | n == 0     =  0
          | n > 0      =  1
          | otherwise  = -1
```

The function `abs'` (similar to the more general function `abs` in the Prelude) takes an integer and returns its absolute value.

```
abs' :: Int -> Int
abs' n | n >= 0    =  n
       | otherwise = -n
```

The function `gcd'` (similar to the more general function `gcd` in the Prelude) takes two integers and returns their greatest common divisor.

```
gcd' :: Int -> Int -> Int
gcd' x y = gcd'' (abs' x) (abs' y)
    where gcd'' x 0 = x
          gcd'' x y = gcd'' y (x `rem` y)
```

Prelude operation `rem` returns the remainder from dividing its first operand by its second.

Given a tuple `(x,y)` constructed by `makeRat` as defined above, we can define `numer (x,y)` and `denom (x,y)` as follows:

```
numer, denom :: Rat -> Int
numer (x,_) = x
denom (_,y) = y
```

The preconditions of both `numer (x,y)` and `denom (x,y)` are that their arguments `(x,y)` satisfy the Rational Representation Property.

The postcondition of `numer (x,y) = x` is that the rational number values $\frac{\texttt{x}}{\texttt{numer (x,y)}} = \frac{\texttt{x}}{\texttt{y}}$.

Similarly, the postcondition of `denom (x,y) = y` is that the rational number values $\frac{\texttt{denom (x,y)}}{y} = \frac{\texttt{x}}{\texttt{y}}$.

Finally, to allow rational numbers to be displayed in the normal fractional representation, we include function `showRat` in the package. We use function `show`, found in the Prelude, here to convert an integer to the usual string format and use the list operator ++ to concatenate the two strings into one.

```
showRat :: Rat -> String
showRat x = show (numer x) ++ "/" ++ show (denom x)
```

Unlike `Rat`, `zeroRat`, `makeRat`, `numer`, and `denom`, function `showRat` (as implemented) does not use knowledge of the data representation. We could optimize it slightly by allowing it to access the structure of the tuple directly.

7.3.3 Rational number modularization

There are three groups of functions in this package:

1. the six public rational arithmetic functions `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat`
2. the public type `Rat`, constant `zeroRat`, public constructor function `makeRat`, public selector functions `numer` and `denom`, and string conversion function `showRat`
3. the private utility functions called only by the second group, but just reimplementations of Prelude functions anyway

7.3.3.1 Module `RationalCore` As we have seen, data type `Rat`; constant `zeroRat`; functions `makeRat`, `numer`, `denom`, and `showRat`; and the functions' preconditions and postconditions form the *interface* to the *data abstraction*.

The data abstraction hides the information about the representation of the data. We can *encapsulate* this group of functions in a Haskell module as follows. This source code must also be in a file named `RationalCore.hs`.

```
module RationalCore
    (Rat, makeRat, zeroRat, numer, denom, showRat)
where
    -- Rat,makeRat,zeroRat,numer,denom,showRat definitions
```

In terms of the information-hiding approach, the secret of the `RationalCore` module is the rational number data representation used.

We can encapsulate the utility functions in a separate module, which would enable them to be used by several other modules.

However, given that the only use of the utility functions is within the data representation module, we choose not to separate them at this time. We leave them as local functions in the data abstraction module. Of course, we could also eliminate them and use the corresponding Prelude functions directly.

7.3.3.2 Module `Rational`

Similarly, functions `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat` use the core data abstraction and, in turn, extend the interface to include rational number arithmetic operations.

We can encapsulate these in another Haskell module that imports the module giving the data representation. This module must be in a file named `Rational1.hs`.

```
module Rational1
  ( Rat, zeroRat, makeRat, numer, denom, showRat,
    negRat, addRat, subRat, mulRat, divRat, eqRat )
where
    import RationalCore
    -- negRat,addRat,subRat,mulRat,divRat,eqRat definitions
```

Other modules that use the rational number package can import module `Rational1`.

7.3.3.3 Modularization critique

The modularization described above:

- enables a module to be reused in several different programs
- offers robustness with respect to change

  The data representation and arithmetic algorithms can change independently.

- allows multiple implementations of each module as long as the public (abstract) interface is kept stable
- enables understanding of one module without understanding the internal details of modules it uses
- costs some in terms of extra code and execution efficiency

  But that probably does not matter given the benefits above and the code optimizations carried out by the compiler.

However, the modularization does not hide the representation fully because it uses a concrete data structure—a pair of integers—to represent a rational number. In chapter 21, we see how to use a user-defined data type to hide the representation fully.

7.3.4 Alternative data representation

In the rational number data representation above, constructor `makeRat` creates pairs in which the two integers are relatively prime and the sign is on the numerator. Selector functions `numer` and `denom` just return these stored values.

An alternative representation is to reverse this approach, as shown in the following module (in file `RationalDeferGCD.hs`.)

```haskell
module RationalDeferGCD
    (Rat, zeroRat, makeRat, numer, denom, showRat)
where

type Rat = (Int,Int)

zeroRat :: (Int,Int)
zeroRat = (0,1)

makeRat :: Int -> Int -> Rat
makeRat x 0 = error ( "Cannot construct a rational number "
                      ++ show x ++ "/0" )
makeRat 0 y = zeroRat
makeRat x y = (x,y)

numer :: Rat -> Int
numer (x,y) = x' `div` d
    where x' = (signum' y) * x
          y' = abs' y
          d  = gcd' x' y'

denom :: Rat -> Int
denom (x,y) = y' `div` d
    where x' = (signum' y) * x
          y' = abs' y
          d  = gcd' x' y'

showRat :: Rat -> String
showRat x = show (numer x) ++ "/" ++ show (denom x)
```

This approach defers the calculation of the greatest common divisor until a selector is called.

In this alternative representation, a rational number `(x,y)` must satisfy all parts of the following **Deferred Representation Property**:

- `(x,y)` ∈ `(Int,Int)`
- `y /= 0`
- if `x == 0` , then `y == 1`
- rational number value is $\frac{\mathtt{x}}{\mathtt{y}}$

Furthermore, we require that `makeRat x y` satisfies the *precondition*:

```haskell
y /= 0
```

The function generates an explicit error condition if it does not.

As a *postcondition*, we require `makeRat x y` to return a result `(x',y')` such that:

- `(x',y')` satisfies the Deferred Representation Property
- rational number value is $\frac{\texttt{x}}{\texttt{y}}$

The preconditions of both `numer (x,y)` and `denom (x,y)` are that `(x,y)` satisfies the Deferred Representation Property.

The postcondition of `numer (x,y) = x'` is that the rational number values $\frac{\texttt{x'}}{\texttt{numer (x,y)}} = \frac{\texttt{x}}{\texttt{y}}$.

Similarly, the postcondition of `denom (x,y) = y` is that the rational number values $\frac{\texttt{denom (x,y)}}{y'} = \frac{\texttt{x}}{\texttt{y}}$.

Question:

> What are the advantages and disadvantages of the two data representations?

Like module `RationalCore`, the design secret for this module, `RationalDeferGCD`, is the rational number data representation.

Regardless of which approach is used, the definitions of the arithmetic and comparison functions do not change. Thus the `Rational` module can import data representation module `RationalCore` or `RationalDeferGCD`.

Figure 7.1 shows the dependencies among the modules we have examined in the rational arithmetic example.

We can consider the `RationalCore` and `RationalDeferGCD` modules as two concrete instances (Haskell `module`s) of a more abstract module we call `RationalRep` in the diagram.

The module `Rational` relies on the abstract module `RationalRep` for an implementation of rational numbers. In the Haskell code above, there are really two versions of the Haskell module `Rational` that differ only in whether they import `RationalCore` or `RationalDeferGCD`.

Chapter 21 introduces user-defined (algebraic) data types. Instead of concrete data types (e.g., the `Int` pairs used by the type alias `Rat`), we can totally hide the details of the data representation using modules.

7.3.5 Haskell information-hiding modules

In the Rational Arithmetic example, we defined two information-hiding modules:

1. "RationalRep", whose secret is how to represent the rational number data and whose interface consists of the data type `Rat`, constant `zeroRat`, operations (functions) `makeRat`, `numer`, `denom`, and `showRat`, and the constraints on these types and functions

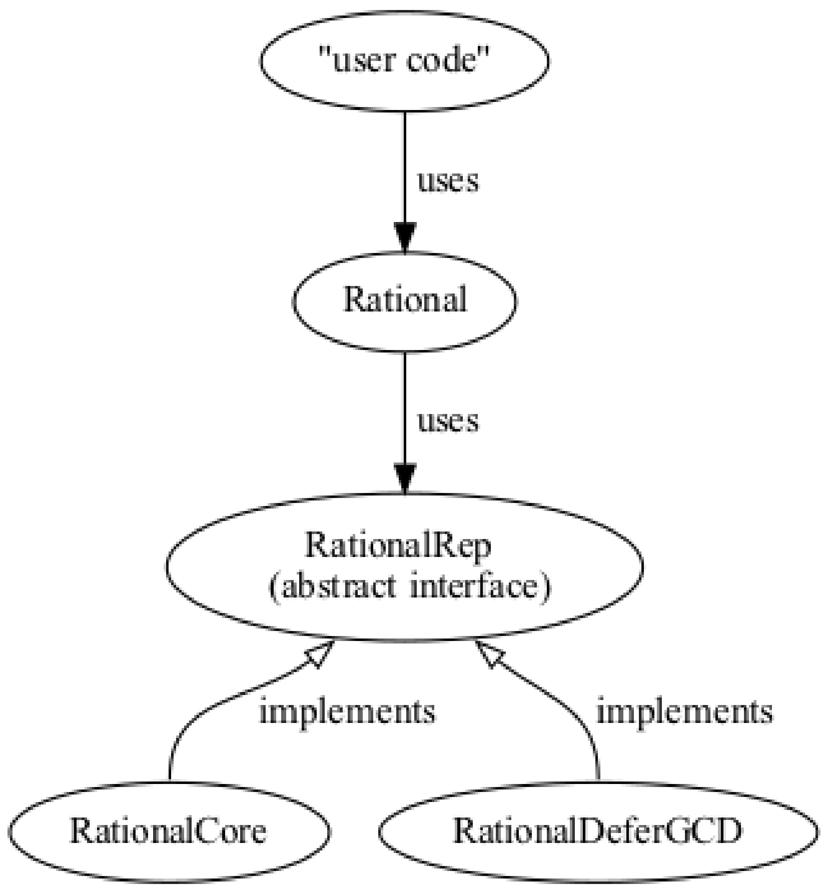

Figure 7.1: Rational package module dependencies.

2. “Rational”, whose secret is how to implement the rational number arithmetic and whose interface consists of operations (functions) `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat`, the other module’s interface, and the constraints on these types and functions

We developed two distinct Haskell modules, `RationalCore` and `RationalDeferGCD`, to implement the “RationalRep” information-hiding module.

We developed one distinct Haskell module, `Rational`, to implement the “Rational” information-hiding module. This module can be paired (i.e., by changing the `import` statement) with either of the other two variants of “RationalRep” module. (Source file `Rational1.hs` imports module `RationalCore`; source file `Rational2.hs` imports module `RationalDeferGCD`.)

Unfortunately, Haskell 2010 has a relatively weak module system that does not support multiple implementations as well as we might like. There is no way to declare that multiple Haskell modules have the same interface other than copying the common code into each module and documenting the interface carefully. We must also have multiple versions of `Rational` that differ only in which other module is imported.

Together the Glasgow Haskell Compiler (GHC) release 8.2 (July 2017) and the Cabal-Install package manager release 2.0 (August 2017) support a new extension, the Backpack mixin package system. This new system remedies the above shortcoming. In this new approach, we would define the abstract module “RationalRep” as a signature file and require that `RationalCore` and `RationalDeferGCD` conform to it.

Further discussion of this new module system is beyond the scope of this chapter.

7.3.6 Rational number testing

Chapter 12 discusses testing of the Rational modules designed in this chapter. The test scripts for the following modules are in the files shown:

- Module `RationalRep`
  - `TestRatRepCore.hs` for module instance `RationalCore`
  - `TestRatRepDefer.hs` for module instance `RationalDeferGCD`
- Module `Rational`
  - `TestRational1.hs` for `Rational` using `RationalCore`.
  - `TestRational2.hs` for `Rational` using `RationalDeferGCD`.

7.4 Module invariants

As we see in the rational arithmetic example, a module that provides a data abstraction must ensure that the objects it creates and manipulates maintain their integrity—always have a valid structure and state.

- The `RationalCore` rational number representation satisfies the Rational Representation Property.
- The `RationalDeferGCD` rational number representation satisfies the Deferred Representation Property.

These properties are *invariants* for those modules. An invariant for the data abstraction can help us design and implement such objects.

Invariant: A logical assertion that must always be true for every "object" created by the public constructors and manipulated only by the public operations of the data abstraction.

Often, we separate an invariant into two parts.

Interface invariant: An invariant stated in terms of the public features and abstract properties of the "object".

Implementation (representation) invariant: A detailed invariant giving the required relationships among the internal features of the implementation of an "object"

An interface invariant is a key aspect of the *abstract interface* of a module. It is useful to the users of the module, as well to the developers.

7.4.1 RationalRep modules

In the Rational Arithmetic example, the *interface invariant* for the "RationalRep" abstract module is the following.

RationalRep Interface Invariant: For any valid Haskell rational number `r`, all the following hold:

- `r` ∈ `Rat`
- `denom r > 0`
- if `numer r == 0`, then `denom r == 1`
- `numer r` and `denom r` are relatively prime
- the (mathematical) rational number value is $\frac{\texttt{numer r}}{\texttt{denom r}}$

We note that the *precondition* for `makeRat x y` is defined above without any dependence upon the concrete representation.

```
y /= 0
```

We can restate the *postcondition* for `makeRat x y = r` generically to require both of the following to hold:

- `r` satisfies the RationaRep Interface Invariant
- rational number `r` 's value is $\frac{\texttt{x}}{\texttt{y}}$

The preconditions of both `numer r` and `denom r` are that their argument `r` satisfies the RationalRep Interface Invariant.

The postcondition of `numer r = x'` is that the rational number value $\frac{\texttt{x'}}{\texttt{denom r}}$ is equal to the rational number value of `r`.

Similarly, the postcondition of `denom r = y'` is that the rational number value $\frac{\texttt{numer r}}{y'}$ is equal to the rational number value of `r`.{.haskell}

An implementation invariant guides the developers in the design and implementation of the internal details of a module. It relates the internal details to the interface invariant.

7.4.1.1 `RationalCore`

We can state an implementation invariant for the `RationalCore` module.

RationalCore Implementation Invariant: For any valid Haskell rational number `r`, all the following hold:

- `r == (x,y)` for some `(x,y)` ∈ `Rat`
- `y > 0`
- if `x == 0`, then `y == 1`
- `x` and `y` are relatively prime
- rational number value is $\frac{\texttt{x}}{\texttt{y}}$

The implementation invariant implies the interface invariant given the definitions of data type `Rat` and selector functions `numer` and `denom`. Constructor function `makeRat` does the work to establish the invariant initially.

7.4.1.2 `RationalDeferGCD`

We can state an implementation invariant for the `RationalDeferGCD` module.

RationalDeferGCD Implementation Invariant: For any valid Haskell rational number `r`, all the following hold:

- `r == (x,y)` for some `(x,y)` ∈ `Rat`
- `y /= 0`
- if `x == 0`, then `y == 1`
- rational number value is $\frac{\texttt{x}}{\texttt{y}}$

The implementation invariant implies the interface invariant given the definitions of `Rat` and of the selector functions `numer` and `denom`. Constructor function `makeRat` is simple, but the selector functions `numer` and `denom` do quite a bit of work to establish the interface invariant.

7.4.2 Rational modules

The `Rational` abstract module extends the `RationalRep` abstract module with new functionality.

- It imports the public interface of the `RationalRep` abstract module and exports those features in its own public interface. Thus it must maintain the interface invariant for the `RationalRep` module it uses.
- It does not add any new data types or constructor (or destructor) functions. So it does not need any new invariant components for new data abstractions.
- It adds one unary and four binary arithmetic functions that take rational numbers and return a rational number. It does so by using the data abstraction provided by the `RationalRep` module. These must preserve the `RationalRep` interface invariant.
- It adds an equality comparison function that takes two rational numbers and returns a `Bool`.

7.5 What Next?

Chapter 6 examined procedural abstraction and stepwise refinement and used the method to develop a square root package.

This chapter (7) examined data abstraction and used the method to develop a rational number arithmetic package. The chapters explored concepts and methods for modular design and programming using Haskell, including preconditions, postconditions, and invariants.

We continue to use these concepts, techniques, and examples in the rest of the book. In particular:

- Chapter 12 examines how to test the modules developed in this chapter.
- Chapter 22 explores the data abstraction concepts and techniques in more depth. In particular, it examines a detailed case study of an abstract data type.

The next chapter, Chapter 8, examines the substitution model for evaluation of Haskell programs and explores efficiency and termination in the context of that model.

7.6 Chapter Source Code

The Haskell source code for this chapter includes the following:

- Two versions of a lower-level "RationalnRep" module that gives implementations of rational number given in the following files.
  - `RationalCore.hs`.

- `RationalDeferGCD.hs`.)

- An upper-level rational arithmetic module given in the following files.
  - `Rational1.hs`, a variant that imports the `RationalCore` module
  - `Rational2.hs`, a variant that imports the `RationalDeferGCD` module

7.7 Exercises

For each of the following exercises, develop and test a Haskell function or set of functions.

1. Develop a Haskell module (or modules) for line segments on the two-dimensional coordinate plane using the *rectangular coordinate* system.

   We can represent a line segment with two points—the starting point and the ending point. Develop the following Haskell functions:

   - constructor `newSeg` that takes two points and returns a new line segment
   - selectors `startPt` and `endPt` that each take a segment and return its starting and ending points, respectively

   We normally represent the plane with a *rectangular coordinate* system. That is, we use two axes—an `x` *axis* and a `y` *axis*—intersecting at a right angle. We call the intersection point the *origin* and label it with 0 on both axes. We normally draw the `x` axis horizontally and label it with increasing numbers to the right and decreasing numbers to the left. We also draw the `y` axis vertically with increasing numbers upward and decreasing numbers downward. Any point in the plane is uniquely identified by its `x`-coordinate and `y`-coordinate.

   Define a data representation for points in the rectangular coordinate system and develop the following Haskell functions:

   - constructor `newPtFromRect` that takes the `x` and `y` coordinates of a point and returns a new point
   - selectors `getx` and `gety` that takes a point and returns the `x` and `y` coordinates, respectively
   - display function `showPt` that takes a point and returns an appropriate `String` representation for the point

   Now, using the various constructors and selectors, also develop the Haskell functions for line segments:

   - `midPt` that takes a line segment and returns the point at the middle of the segment

- display function `showSeg` that takes a line segment and returns an appropriate `String` representation

Note that `newSeg`, `startPt`, `endPt`, `midPt`, and `showSeg` can be implemented independently from how the points are represented.

2. Develop a Haskell module (or modules) for line segments that represents points using the *polar coordinate* system instead of the rectangular coordinate system used in the previous exercise.

   A polar coordinate system represents a point in the plane by its *radial coordinate* `r` (i.e., the distance from the *pole*) and its *angular coordinate* `t` (i.e., the angle from the *polar axis* in the reference direction). We sometimes call `r` the *magnitude* and `t` the *angle*.

   By convention, we align the rectangular and polar coordinate systems by making the origin the pole, the positive portion of the `x` axis the polar axis, and let the first quadrant (where both `x` and `y` are positive) be the smallest positive angles in the reference direction. That is, with a traditional drawing of the coordinate systems, we measure and the radial coordinate `r` as the distance from the origin measure the angular coordinate `t` counterclockwise from the positive `x` axis.

   Using knowledge of trigonometry, we can convert among rectangular coordinates `(x,y)` and polar coordinates `(r,t)` using the equations:

   ```
   x = r * cos(t)
   y = r * sin(t)
   r = sqrt(x^2 + y^2)
   t = arctan2(y,x)
   ```

   Define a data representation for points in the polar coordinate system and develop the following Haskell functions:

   - constructor `newPtFromPolar` that takes the magnitude `r` and angle `t` as the polar coordinates of a point and returns a new point
   - selectors `getMag` and `getAng` that each take a point and return the magnitude `r` and angle `t` coordinates, respectively
   - selectors `getx` and `gety` that return the `x` and `y` components of the points (represented here in polar coordinates)
   - display functions `showPtAsRect` and `showPtAsPolar` to convert the points to strings using rectangular and polar coordinates, respectively,

   Functions `newSeg`, `startPt`, `endPt`, `midPt`, and `showSeg` should work as in the previous exercise.

3. Modify the solutions to the previous two line-segment module exercises to enable the line segment functions to be in one module that works properly

if composed with either of the two data representation modules. (The solutions may have already done this.)

4. Modify the solution to the previous line-segment exercise to use the Backpack module system.
5. Modify the modules in the previous exercise to enable the line segment module to work with both data representations in the same program.
6. Modify the solution to the Rational Arithmetic example to use the Backpack module system.
7. State preconditions and postconditions for the functions in abstract module `Rational`.

7.8 Acknowledgements

In Summer and Fall 2016, I adapted and revised much of this work from my previous materials:

- Discussion of the Rational Arithmetic modules mostly from chapter 5 of my *Notes on Functional Programming with Haskell* [4], from my Lua-based implementations, and from section 2.1 of Abelson and Sussman's *Structure and Interpretation of Computer Programs* [1].
- Discussion of modular design and programming issues from my Data Abstraction [5] and Modular Design [6] notes, which drew ideas over the past 25 years from a variety of sources [2,3,7–19].

In 2017, I continued to develop this work as Sections 2.6-2.7 in Chapter 2, Basic Haskell Functional Programming, of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I divided the previous Basic Haskell Functional Programming chapter into four chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. Previous sections 2.1-2.3 became the basis for new Chapter 4, First Haskell Programs; previous Section 2.4 became Section 5.3 in the new Chapter 5, Types; and previous sections 2.5-2.7 were reorganized into new Chapter 6, Procedural Abstraction, and Chapter 7, Data Abstraction (this chapter).

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

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.

7.9 Terms and Concepts

TODO: Update

Haskell `module`, module exports and imports, module dependencies, rational number arithmetic, data abstraction, properties of data, data representation, precondition, postcondition, invariant, interface invariant, implementation or representation invariant, canonical or normal forms, relatively prime, information hiding, module secret, encapsulation, interface, abstract interface, type inference.

7.10 References

[1] Harold Abelson and Gerald Jockay Sussman. 1996. *Structure and interpretation of computer programs (SICP)* (Second ed.). MIT Press, Cambridge, Massachusetts, USA. Retrieved from https://mitpress.mit.edu/sicp/

[2] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. 1981. A procedure for designing abstract interfaces for device interface modules. In *Proceedings of the 5th international conference on software engineering*, IEEE, San Diego, California, USA, 195–204.

[3] Timothy Budd. 2000. *Understanding object-oriented programming with Java* (Updated ed.). Addison-Wesley, Boston, Massachusetts, USA.

[4] H. Conrad Cunningham. 2014. *Notes on functional programming with Haskell*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/Notes_FP_Haskell/Notes_on_Functional_Programming_with_Haskell.pdf

[5] H. Conrad Cunningham. 2019. *Notes on data abstraction*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/DataAbstraction/DataAbstraction.html

[6] H. Conrad Cunningham. 2019. *Notes on modular design*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ModularDesign/ModularDesign.html

[7] H. Conrad Cunningham, Yi Liu, and Jingyi Wang. 2010. Designing a flexible framework for a table abstraction. In *Data engineering: Mining, information, and intelligence*, Yupo Chan, John Talburt and Terry M. Talley (eds.). Springer, New York, New York, USA, 279–314.

[8] H. Conrad Cunningham and Jingyi Wang. 2001. Building a layered framework for the table abstraction. In *Proceedings of the ACM symposium on applied computing*, Las Vegas, Nevada, USA.

[9] H. Conrad Cunningham, Cuihua Zhang, and Yi Liu. 2004. Keeping secrets within a family: Rediscovering Parnas. In *Proceedings of the international conference on software engineering research and practice (SERP)*, CSREA Press, Las Vegas, Nevada, USA, 712–718.

[10] Nell Dale and Henry M. Walker. 1996. *Abstract data types: Specifications, implementations, and applications.* D. C. Heath, Lexington, Massachusetts, USA.

[11] Cay S. Horstmann. 1995. *Mastering object-oriented design in C++.* Wiley, Indianapolis, Indiana, USA.

[12] Cay S. Horstmann and Gary Cornell. 1999. *Core Java 1.2: Volume I—Fundamentals.* Prentice Hall, Englewood Cliffs, New Jersey, USA.

[13] Bertrand Meyer. 1997. *Object-oriented program construction* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[14] Hanspeter Mossenbock. 1995. *Object-oriented programming in Oberon-2.* Springer, Berlin, Germany.

[15] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (December 1972), 1053–1058.

[16] David L. Parnas. 1979. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering* SE-5, 1 (1979), 128–138.

[17] David L. Parnas. 1979. The modular structure of complex systems. *IEEE Transactions on Software Engineering* SE-11, 13 (1979), 128–138.

[18] Pete Thomas and Ray Weedom. 1995. *Object-oriented programming in Eiffel.* Addison-Wesley, Boston, Massachusetts, USA.

[19] David M. Weiss. 2001. Introduction: On the criteria to be used in decomposing systems into modules. In *Software fundamentals: Collected papers by David L. Parnas*, Daniel M. Hoffman and David M. Weiss (eds.). Addison-Wesley, Boston, Massachusetts, USA.