CSci 450: Organization of Programming Languages
Using Data Abstraction in Lua

H. Conrad Cunningham

13 September 2016

Acknowledgements: Adapted from my notes *Introduction to Functional Programming Using Haskell* (under development, 2016) and example Lua modules for Rational Arithmetic.

Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of September 2016 is a recent version of Firefox from Mozilla.

Using Data Abstraction

How can we make a program robust with respect to change in the form of its data? A good technique is data abstraction. Let's begin with an example.

Rational number arithmetic

For this example, let's implement a group of Lua functions to perform rational number (fraction) arithmetic.

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

For now, let's assume we have a Lua constructor function

```
makeRat(x,y)
```

to create a rational number instance from its numerator `x` and denominator `y`. That is, `makeRat(x, y)` constructs rational number $\frac{\mathtt{x}}{\mathtt{y}}$.

Further, let us assume we have selector functions `numer(r)` and `denom(r)` that each take a rational number argument and return its numerator and denominator, respectively. That is, they satisfy the equalities:

```
numer(makeRat(x,y)) == x
denom(makeRat(x,y)) == y
```

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

Given the knowledge of rational arithmetic from mathematics, we can define the operations for unary negation, addition, subtraction, multiplication, and division.

```
local function negRat(r)
   return makeRat(- numer(r), denom(r))
end

local function addRat(r,s)
   return makeRat(numer(r) * denom(s) + numer(s) * denom(r),
                  denom(r) * denom(s)                       )
end

local function subRat(r,s)
   return makeRat(numer(r) * denom(s) - numer(s) * denom(r),
                  denom(r) * denom(s)                       )
end

local function mulRat(r,s)
   return makeRat(numer(r) * numer(s), denom(r) * denom(s))
end

local function divRat(r,s)
   return makeRat(numer(r) * denom(s), denom(r) * numer(s))
end
```

We can also define a function `showRat` to convert a rational number to a convenient string representation.

```
local function showRat(r)
   return tostring(numer(r)) .. "/" .. tostring(denom(r))
end
```

Because the comparison operations are similar to each other, they are good candidates for us to use a higher-order function. We can abstract out the common pattern of comparisons into a function that takes the corresponding integer comparison as an argument.

To compare two rational numbers, we can express their values in terms of a common denominator (e.g., `denom(x) * denom(y)`) and then compare the numerators using the integer comparisons. We can thus abstract the comparison into a higher-order function `compare` that takes an appropriate integer relational operator and returns a function that compares the two numerators accordingly.

```
local function compare(comp)
   return
      function(r,s)
         local x, y = numer(r) * denom(s), denom(r) * numer(s)
         return comp(x,y)
      end
end
```

Then we can define functions for the six relational operators as follows:

```
local eqRat  = compare(function(x,y) return x == y end)
local neqRat = compare(function(x,y) return x ~= y end)
local ltRat  = compare(function(x,y) return x < y  end)
local leqRat = compare(function(x,y) return x <= y end)
local gtRat  = compare(function(x,y) return x > y  end)
local geqRat = compare(function(x,y) return x >= y end)
```

All these rational arithmetic functions use the constructor function `makeRat` and the selector functions `numer` and `denom` assumed above. They do not depend upon any specific representation for rational numbers.

The above functions work on rational numbers as a *data abstraction* defined by the constructor function `makeRat` and selector functions `numer` and `denom`.

Rational number data representation

Now, how can we represent rational numbers?

We can represent a rational number as a two-element array (table) with numerator at index 1 and denominator at index 2. For example, `{1,7}`, `{-1,-7}`, `{3,21}`, and `{168,1176}` all represent $\frac{1}{7}$.

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 rational number values.

It is convenient for us to choose a rational number representation `{x,y}` that satisfies the following *implementation (representation) invariant*:

> `y > 0`, `x` and `y` are relatively prime, and zero is denoted by `{0,1}`.

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

By *invariant*, we mean a property that always holds except temporarily while being operated upon. The constructor must make it true and it remains true before and after all mutator and accessor operations. It is true before any explicit destructor operation.

An *implementation invariant* is an invariant that is stated in terms of the concrete data structures used.

This representation has the advantage that the magnitudes of the numerator `x` and denominator `y` are kept small, thus reducing problems with overflow arising during arithmetic operations.

We thus provide a function for constructing rational numbers in this canonical form. We define constructor `makeRat` as follows. This function checks whether arguments are integers for Lua before 5.3.

```
local function makeRat(x,y)
   if type(x) == "number" and type(y) == "number" and
         x == math.floor(x) and y == math.floor(y) and y ~= 0 then
      return newRat(x,y)
   else
      error("Cannot construct rational number " ..
            tostring(x) .. "/" .. tostring(y)     )
   end
end
```

The `makeRat` constructor delegates the actual construction of the rational number to function `newRat`. Function `newRat` assumes that it is called with two appropriate integers.

```
local function newRat(x,y)
   if x == 0 then
      return {0,1}
   else
      local xx = signum(y) * x
      local yy = math.abs(y)
      local d  = gcd(xx,yy)
      return {xx/d, yy/d}
   end
end
```

The function `signum` takes a number and returns the integer `-1`, `0`, or `1` when the number is negative, zero, or positive, respectively.

```
local function signum(n)
   if n == 0 then
      return 0
   elseif n > 0 then
      return 1
   else
      return -1
   end
```

```
end
```

The function gcd takes two integers and returns their greatest common divisor.

```
local function gcd(x,y)
   local function gcdaux(x,y)
      if y == 0 then
         return x
      else
         return gcdaux(y, x % y) -- tail recursive
      end
   end

   return gcdaux(math.abs(x), math.abs(y))
end
```

Function gcd uses a tail recursive internal function gcdaux to compute the gcd on the absolute values of its two integer arguments. It uses the Euclidean Algorithm. (Operation % returns the remainder from dividing its first operand by its second.)

Given makeRat defined as above, we can define numer and denom as follows:

```
local function numer(r)
   return r[1]
end

local function denom(r)
   return r[2]
end
```

These functions access the actual data structure used to represent the rational numbers.

Data abstraction and modules

There are three groups of functions defined in this package:

1. the twelve public rational arithmetic functions negRat, addRat, subRat, mulRat, divRat, showRat, eqRat, neqRat, ltRat, leqRat, gtRat, and geqRat.
2. the public constructor function makeRat and public selector functions numer and denom.
3. the private utility functions called only by the second group signum, gcd, and newRat.

As we have seen, `makeRat`, `numer`, and `denom` in group 2 form the *interface* to the *data abstraction* that hides the information about the representation of the data. We can *encapsulate* the group 2 and 3 functions in a Lua *module*.

We put this source code in a script file named (say) `rationalCore.lua`. We define the functions so that a function is defined before it is used, e.g., in the order `signum`, `gcd`, `newRat`, `makeRat`, `numer`, and `denom`.

As the last executable statement in the module's script, we return the module table with the public functions defined with appropriate field names.

```
return { makeRat = makeRat, numer = numer, denom = denom }
```

Questions:

- Should we also include the function `newRat` in the public interface? What are the advantages and disadvantages of doing so?
- Should we also include the functions `signum` and `gcd` in the public interface? Or should we, instead, put them in a separate module?

The rational arithmetic functions in group 1 use the core data abstraction and, in turn, extend the interface to include rational number arithmetic operations.

We can encapsulate these in another module `rational`. This Lua module loads the data representation module an defines local variables for the module's operations.

```
local ratco = require("rationalCore")
local makeRat, numer, denom =
   ratco.makeRat, ratco.numer, ratco.denom
```

The module returns both the group 1 and group 2 functions.

```
return { makeRat = makeRat, numer   = numer,  denom   = denom,
         negRat  = negRat,  addRat  = addRat, subRat  = subRat,
         mulRat  = mulRat,  divRat  = divRat, showRat = showRat,
         eqRat   = eqRat,   neqRat  = neqRat, ltRat   = ltRat,
         leqRat  = leqRat,  gtRat   = gtRat,  geqRat  = geqRat
       }
```

Other modules that use the rational number package can load Lua module `rational`. This module can be reused wherever needed.

This modular approach to program design and implementation offers the potential of scalability and robustness with respect to change.

One key to this *information hiding* approach to design is to identify the aspects of a software system that are most likely to change from one version to another and make each a design *secret* of some module.

The secret of the `rationalCore` module is the rational number data representation used. The secret of the `rational` module itself is the methods used for rational number arithmetic.

Another key is the use of an *abstract interface*. The interface to each module focuses on providing operations that are general, not specific to a particular implementation.

Let's now consider changes to the data representation.

Alternative rational number data representation

In the rational number data representation above (i.e., module `rationalCore`), 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 in functions `newRat`, `numer`, and `denom`, as shown in the module `rationalDeferGCD`.

```
local function newRat(x,y)
   if x == 0 then
      return {0,1}
   else
      return {x,y}
   end
end

local function numer(r)
   local x,y = r[1], r[2]
   local xx = signum(y) * x
   local yy = math.abs(y)
   local d  = gcd(xx,yy)
   return xx / d
end

local function denom(r)
   local x,y = r[1], r[2]
   local xx = signum(y) * x
   local yy = math.abs(y)
   local d  = gcd(xx,yy)
   return yy / d
end
```

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

The implementation invariant for this rational number representation requires that, for `{x,y}`, `y` $\neq$ `0` and zero is represented by `{0,1}`.

Furthermore, function `numer` and `denom` satisfy the equalities

```
numer (makeRat(x,y)) == x'
denom (makeRat(x,y)) == y'
```

where `y' > 0`, `x'` and `y'` are relatively prime, and $\frac{x}{y} = \frac{x'}{y'}$.

Question:

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

Like module `rationalCore`, the design secret for module `rationalDeferGCD` is the rational number data representation.

Another rational number data representation

Another alternative to `rationalCore` and `rationalDeferGCD` is a module that uses a *closure* instead of an array to represent a rational number. (We use a parameterless closure, which is also called a *thunk*.)

We call build module `rationalClo` around the following altered definitions.

```
local function newRat(x,y)
   if x == 0 then
      return -- return thunk (closure)
         function() return 0, 1 end -- two returns
   else
      return -- return thunk (closure)
         function()
            local xx = signum(y) * x
            local yy = math.abs(y)
            local d  = gcd(xx,yy)
            return xx/d, yy/d -- two returns
         end
   end
end

local function numer(r)
   local x, _ = r() -- force thunk (closure)
   return x
end

local function denom(r)
   local _, y = r() -- force thunk (closure)
   return y
end
```

The implementation invariant for this rational number representation requires that, for some `r`,

> For `x, y = r()`: `y` $\neq$ `0`, `x` and `y` are relatively prime, and if `x == 0` then `y == 1`.

Like modules `rationalCore` and `rationalDeferGCD`, the design secret for module `rationalDeferGCD` is the rational number data representation.

Regardless of which approach we use, the definitions of the arithmetic and comparison functions do not change. Thus the `rational` module can load data representation module `rationalCore`, `rationalDeferGCD`, or `rationalClo` and get the same result.

Files

1. Rational arithmetic module – outer layer implementation of Rational Arithmetic abstraction

2. Rational number data representation using two-element arrays – module `rationalCore` implementing primitive layer

   partial test script

3. Rational number data representation using array but deferring GCD – module `rationalDeferGCD` implementing primitive layer

   partial test script

4. Rational number data representation using closures – module `rationalClo` implementing primitive layer

   partial test script