CSci 450: Organization of Programming Languages
Using Stepwise Refinement 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 function for Square Root.

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 Top-Down Stepwise Refinement

A useful and intuitive design process for a small program is to begin with a high-level solution and gradually fill in the details. We call this process top-down stepwise refinement.

Let's consider an example.

Developing a square root package

Consider the problem of computing the nonnegative square root of a nonnegative number x. Mathematically, we want to find the number y such that

$y \geq 0$ and $y^2 = x$.

A common algorithm in mathematics for computing the above y is to use Newton's method of successive approximations, which has the following steps for square root:

1. Guess at the value of y.

2. If the current approximation (guess) is sufficiently close (i.e. good enough), return it and stop; otherwise, continue.
3. Compute an improved guess by averaging the value of y and x/y, then go back to step 2.

To encode this algorithm in Lua, we work top down to decompose the problem into smaller parts until each part can be solved easily. We begin this *top-down stepwise refinement* by defining a function

```
sqrt_iter(guess,x)
```

to compute the square root of `x` iteratively, where `guess` is the current approximation.

We can encode step 2 of the above algorithm in Lua as follows:

```
local function sqrt_iter(guess,x)
   if good_enough(guess,x) then
      return guess
   else
      return sqrt_iter(improve(guess,x),x)
   end
end
```

We have two cases:

- When the current approximation `guess` is sufficiently close to `x`, we return `guess`.

  We abstract this decision into a separate function:

  ```
  good_enough(guess,x)
  ```

- When the approximation is not yet close enough, we reduce the problem to another application of `sqrt_iter` itself to an improved approximation.

  We abstract the improvement process into a separate function:

  ```
  improve(guess,x)
  ```

  To ensure termination of `sqrt_iter`, the argument `improve(guess,x)` on the recursive call must get closer to a value that satisfies its base case.

The function `improve` takes the current `guess` and `x` and carries out step 3 of the algorithm, thus averaging `guess` and `x/guess`, as follows:

```
local function improve(guess,x)
   return average(guess,x/guess)
end
```

Here we abstract `average` into a separate function as follows:

```
local function average(x,y)
   return (x + y) / 2
end
```

The new guess is closer to the square root than the previous guess. Thus the algorithm will terminate assuming a good choice for function `good_enough`, which guards the base case of the `sqrt_iter` recursion.

How should we define `good_enough`? Given that we are working with the limited precision of computer floating point arithmetic, it is not easy to choose an appropriate test for all situations. Here we simplify this and use a tolerance of 0.001.

We thus postulate the following definition for `good_enough`:

```
 local function good_enough(guess,x)
     return math.abs(square(guess)- x) < 0.001
  end
```

In the above, `math.abs` is the built-in absolute value function defined in the Lua `math` library. We define `square` as the following simple function (but could replace it by just `guess * guess`).

```
local function square(x)
   return x * x
end
```

What is a good initial guess? It is sufficient to just use 1. So we can define an overall square root function `sqrt` as follows:

```
local function sqrt(x)
   if type(x) == "number" and x >= 0 then
      return sqrt_iter(1,x)
   else
      error("Square root cannot be computed for "
            .. tostring(x), 2)
   end
end
```

Packaging the functions

Because the only public function needed is `sqrt`, we can make all the other functions local to `sqrt`. That is, we put their definitions inside the body of `sqrt`. We must define each of the functions before it is called by another function.

```
local function sqrt(x)

   local function square(x)
      return x * x
```

```
        end

        local function good_enough(guess,x)
           return math.abs(square(guess)- x) < 0.001
        end

        local function average(x,y)
           return (x + y) / 2
        end

        local function improve(guess,x)
           return average(guess,x/guess)
        end

        local function sqrt_iter(guess,x)
           if good_enough(guess,x) then
              return guess
           else
              return sqrt_iter(improve(guess,x),x)
           end
        end

        if type(x) == "number" and x >= 0 then
           return sqrt_iter(1,x)
        else
           error("Square root cannot be computed for "
                 .. tostring(x), 2)
        end
     end
```

An alternative would be to put these functions in a Lua module and just export function `sqrt`.

Top-down stepwise refinement

The program design strategy known as *top-down stepwise refinement* is a relatively intuitive design process that has long been applied in the design of structured programs in imperative procedural languages. It is also useful in the functional programming setting, as is shown by the square root example above.

In Lua, we can apply top-down stepwise refinement as follows.

1. Start with a high-level solution to the problem consisting of one or more functions. For each function, identify its signature and functional requirements (i.e., its inputs, outputs, and termination condition).

Some parts of each function are abstracted as “pseudocode” expressions or as-yet-undefined function calls.

2. Choose one of the incomplete parts. Consider its signature and functional requirements. Refine the incomplete part by breaking it into subparts or, if simple, defining it directly in terms of Lua expressions (including calls to library functions).

   When refining an incomplete part, consider the various options according to the relevant design criteria (e.g., time, space, generality, understandability, elegance, etc.)

   The refinement of the function may require a refinement of the data being passed. If so, back up in the refinement process and readdress previous design decisions as needed.

   If it not possible to design an appropriate refinement, back up in the refinement process and readdress previous design decisions.

3. Continue step 2 until all parts are fully defined in terms of Lua code and data and the resulting set of functions meets all required criteria.

For as long as possible, we should stay with terminology and notation that is close to the problem being solved. We can do this by choosing appropriate function names and signatures and data types.

For stepwise refinement to work well, we must be willing to back up to earlier design decisions when appropriate. We should keep good documentation of the intermediate design steps.

The stepwise refinement method can work well for small programs, but it may not scale well to large, long-lived, general purpose programs. In particular, stepwise refinement may lead to a module structure in which modules are tightly coupled and not robust with respect to changes in requirements. A combination of techniques may be needed to develop larger software systems.

Files

- Square root code adapted from SICP