Learn Functional Programming with Haskell

Oliver Krischer

October 30, 2021

Working through the classical book *Structure and Interpretation of Computer Programs* [1] using *Haskell*

Contents

List of Figures

1 Building Abstractions with Procedures

```
import Criterion.Main (defaultMain, bench, bgroup, whnf)
```

1.1 The Elements of Programming

1.1.1 Procedures as Black-Box Abstractions

In this example we are including definitions for subroutines inside the main function, in order to keep the user interface clean. We also use *lexical scoping*: all references to the input value inside the subroutines (in this case x) receive their value directly from the main argument.

```
sqrtHeron :: Double → Double
sqrtHeron x = iter 1
```

```
  where
    satisfies guess = abs (guess ↑ 2 − x) < 0.001
    improve guess = (guess + (x / guess)) / 2
    iter guess =
      if satisfies guess then guess
      else iter $ improve guess
```

1.2 Procedures and the Processes They Generate

1.2.1 Linear Recursion and Iteration

The *factorial function* is defined by the following *recurrence relation*:

$$1! = 1$$
$$n! = n \cdot (n-1)!$$

A straightforward implementation, resulting in a *recursive process*:

```
facRec :: Integer → Integer
facRec n = case n of
  1 → 1
  n → n * facRec (n − 1)
```

In contrast, here is an implementation which leads to an *iterative process*: we use the concept of *accumulation* in which we 'store' the running product in a parameter of the iteration function:

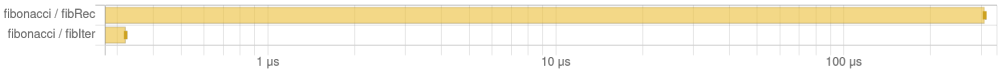
```
facIter :: Integer → Integer
facIter = iter 1
  where
    iter p n = case n of
      1 → p
      n → iter (p * n) (n − 1)
```

Observe, that we have a *recursive function definition* in **both cases**, as this is the standard way of *looping* in a pure functional programming language. But the resulting *computing process* differs in each case:

recursive: The program needs to keep track of the operations to be performed later on. Hence it has to store every frame of execution within the programs *call stack*, which will grow and shrink during execution. This will lead to a space complexity of $\mathcal{O}(n)$ for this linear process.

iterative: The program keeps track of the process with a fixed number of *values* (in our case p for the running product and n for decreasing the input value), which we repeatedly recalculate. The stack size keeps constant, resulting in a space complexity of $\mathcal{O}(1)$.

Figure 1: Benchmarking Fibonacci number generation



Another way of understanding this, is to look at the actual recursive call of the function: if there are no additional calculations to be performed (i.e. the resulting value is immediately returned), this will lead to an iterative process, provided the compiler is able to recognize and optimize this kind of *tail recursive* calls.

1.2.2 Tree Recursion

Let's have a look at another standard example for recurrence relations, the *Fibonacci numbers*:

$$Fib_0 = 0$$
$$Fib_1 = 1$$
$$Fib_n = Fib_{n-1} + Fib_{n-2}$$

This translates directly into recursive code:

```
fibRec :: Integer → Integer
fibRec n
  | n < 2 = n
  | otherwise = fibRec (n − 1) + fibRec (n − 2)
```

With that recursion expression we are calling the function twice for every $n > 1$, which leads to an exponential growth of calculation steps. The resulting process looks like a tree, in which the branches split into two at each level. In general, the number of steps required by a *tree-recursive* process will be proportional to the number of nodes in the tree, while the space required will be proportional to the maximum depth of the tree.

```
fibIter :: Integer → Integer
fibIter = iter 0 1
  where
    iter a b n = case n of
      1 → b
      n → iter b (a + b) (n − 1)
```

Here we have created an iterative implementation, *accumulating* the current sum of the last two numbers.

```
main :: IO ()
main = defaultMain
  [bgroup "fibonacci"
    [bench "fibRec" $ whnf fibRec 20
    , bench "fibIter" $ whnf fibIter 20]]
```

References

[1] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. The MIT Press, 1996. eprint: `UnofficialTexinfoFormat2016`. URL: `http://sarabander.github.io/sicp/html/index.xhtml`.