CSE 130: Programming Languages Oct. 9, 2000

Lecture Notes on functional programming

Instructor: Daniele Micciancio

This notes supplement chapter 14 in the text-book. See the text-book for a general discussion about functional programming, and an introduction to the SCHEME functional programming language.

1 Functions and the λ-calculus

In mathematics, functions are usually defined using the following notation

$$f(x) : x + 1$$

or equivalently

$$f : x \mapsto x + 1$$

meaning that function f maps x to $x + 1$. This notation serves two purposes:

- Define a function, in this case the increment function
- Give a name to this function, in this case f.

In our study of programming languages, and functional programming in particular, is useful to separate these two issues. The λ-calculus gives a notation that can be used to define functions without explicitly naming them. For example, one can write

$$(\lambda x.x + 1)$$

to denote the increment function that maps x to $x + 1$. Functions can be applied to arguments:

$$((\lambda x.x + 1)3) = 4$$

$$((\lambda x.x^2)7) = 49$$

If we want to give a name to a function we have to do it explicitly

$$\text{inc} \equiv (\lambda x.x + 1)$$

$$(\text{inc}5) = 6$$

The λ notation can be used to describe functions that take functions are arguments and give functions are result. For example

$$\text{funsq} \equiv (\lambda f.(\lambda x.(f(fx))))$$

represent a function funsq that on input any function f returns another function

$$(\text{funsq}f) : x \mapsto f(f(x)).$$

The ability to pass and return functions as values increases substantially the expressive power of a language. So far, the syntax of our functional language can be described by the grammar (λ-calculus):

$$\begin{array}{rcll} S & ::= & x & (x \in V \text{ a variable name}) \\ & | & (\lambda x.S) & \text{Function abstraction} \\ & | & (S\ S) & \text{Function application} \end{array}$$

(We also used numbers and arithmetic operations, but let's forget about them for a while).

Is this enough to write programs? It doesn't seem so. in a programming language we typically have many other things: constants (numerals, boolean), arithmetic operations, conditional expressions (if-then-else), loops. Even to describe functions, this language seems quite limited: functions only take one argument!

Surprisingly, we don't need all of this, and the ability to pass and return functions as values is enough to get all other typical programming constructs. In this lecture we will study how some of the above features can be emulated using higher order functions. Of course, for efficiency and practicality reasons, real programming languages will include many of the above features, but showing that they are not strictly necessary is an instructive excercise to understand the power of programming with higher order functions.

2 Functions of several variables

Let's start with something simple. Say we want to define a function of two arguments:

$$\text{sum}(x, y) = x + y$$

(Assume for the moment that we have numbers and arithmetics.)

The addition function can be described using a technique called Currying (from Haskell Curry), in which arguments are passed to the function one at a time. Define

$$\text{sum} \equiv (\lambda x.(\lambda y.x + y))$$

Notice that *sum* is a function of one variable that on input 3 returns a function $(\lambda y.3 + y)$ as the result. We can now apply $(\lambda y.x + y)$ to 4 to get $3 + 4 = 7$, the sum of a and b! Combining the two steps we have

$$((\text{sum } 3)\ 4) = 7$$

In general, functions of n variables

$$f : x_1, \ldots, x_n \mapsto f(x_1, \ldots, x_n)$$

can be represented by the λ-term

$$S \equiv (\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.f(x_1, \ldots, x_n)))))$$

If we want to apply the function represented by S to arguments $a_1, \ldots, a_n$, we can write:

$$((((Sa_1)\ a_2)\ \ldots)\ a_n) \equiv f(a_1, \ldots, a_n)$$

As an example that does not use arithmetics, we can define the composition fuction $f, g \mapsto f \circ g$ as

$$\text{comp} \equiv (\lambda f.(\lambda g.(\lambda x.(f\ (g\ x)))))$$

Even if functions of several arguments can be represented as functions of only one argument using the Currying technique, it is convenient to extend our notation to allow for multivariate functions as follows:

$$S ::= (\lambda x_1 \ldots x_n.S) | (S\ S_1\ \ldots\ S_n)$$

3 Boolean constants and conditional expressions

The boolean values can be represented as λ-terms as follows:

$$\textbf{true} = (\lambda xy.x)$$

$$\textbf{false} = (\lambda xy.y)$$

Using this representation, the conditional expression *if* S *then* S_1 *else* S_2 can be simply expressed as

$$(S\ S_1\ S_2)$$

because

$$(\textbf{true}\ S_1\ S_2) = S_1$$

and

$$(\textbf{false}\ S_1\ S_2) = S_2$$

Using conditionals we can also easily define the usual boolean operations

$$\textbf{not} = (\lambda z.(z\ \textbf{false}\ \textbf{true}))$$

$$\textbf{and} = (\lambda z_1 z_2.(z_1\ z_2\ \textbf{false}))$$

$$\textbf{or} = (\lambda z_1 z_2.(z_1\ \textbf{true}\ z_2)).$$

In the homeworks, you will be asked to define some other simple boolean function.

4 Pairs and tuples

In class we showed also how to represent pairs $[x, y] \equiv (\lambda z.(z\ x\ y))$ as functions that return the first element of the pair on input **true**, and the second element on input **false**. We also defined constructor and selectors:

$$\textbf{pair} = (\lambda xy.(\lambda z.(z\ x\ y)))$$

$$\textbf{first} = (\lambda x.(x\ \textbf{true}))$$

$$\textbf{second} = (\lambda x.(x\ \textbf{false})).$$

with the property that

$$(\textbf{first}\ (\textbf{pair}\ x\ y)) = x$$

and

$$(\textbf{second}\ (\textbf{pair}\ x\ y)) = y$$

Pairs can be used to build more complicate data structures, e.g. tuples can be represented applying the pair constructor repeatedly

$$[x_1, x_2, \ldots, x_n] \equiv (\textbf{pair}\ x_1\ (\textbf{pair}\ x_2\ (\textbf{pair}\ x_3\ \ldots (\textbf{pair}\ x_{n-1}\ x_n)))).$$

Using these techniques it is also possible to represent numerals, and arithmetic operations on numerals.

5 Recursion

Next week we will study how to realize recursion using λ-terms. The idea is the following. Consider a recursive definition:

$$f(x) = if\ x = 0\ then\ 1\ else\ x * f(x-1)$$

defining the factorial function $f(n) = n!$. You can think of the right hand side of this definition as a higher order function T that on input a function f, returns another function

$$T(f) \equiv \lambda x.(if\ x = 0\ then\ 1\ else\ x * f(x-1)).$$

Notice that the factorial function $f(n) = n!$ satisfies the equation

$$f = T(f)$$

i.e., if we replace the factorial function $\lambda x.x!$ instead of f in $T(f)$, what we obtain is again the factorial function

$$(\lambda x.(if\ x = 0\ then\ 1\ else\ x * (x-1)!) = (\lambda x.x!).$$

We say that $(\lambda x.x!)$ is a *fix-point* of T.

We have seen that given a higher order function $T(f)$ representing a recursive definition for f, the solution to the equation $f = T(f)$ gives the desired recursive function.

Let $A = (\lambda xy.y\ (x\ x\ y))$ and define the "fix-point" operator

$$\Theta = AA = (\lambda xy.y\ (x\ x\ y))(\lambda xy.y\ (x\ x\ y)).$$

Notice that

$$\begin{aligned}(\Theta\ T) &= (A\ A\ T)\\ &= ((\lambda xy.(y\ (x\ x\ y)))\ A\ T)\\ &= (T\ (A\ A\ T))\\ &= (T\ (\Theta\ T))\end{aligned}$$

i.e., $(\Theta\ T)$ is a fix-point for T. If T is the transformation

$$T = \lambda f.\lambda x.(if\ x = 0\ then\ 1\ else\ x * f(x-1))$$

then $(\Theta\ T)$ is the factorial function.