Higher-Order Functions

Principles of Functional Programming

Higher-Order Functions

Functional languages treat functions as *first-class values*.

This means that, like any other value, a function can be passed as a parameter and returned as a result.

This provides a flexible way to compose programs.

Functions that take other functions as parameters or that return functions as results are called *higher order functions*.

Example:

Take the sum of the integers between `a` and `b`:

```
def sumInts(a: Int, b: Int): Int =
  if a > b then 0 else a + sumInts(a + 1, b)
```

Take the sum of the cubes of all the integers between `a` and `b` :

```
def cube(x: Int): Int = x * x * x

def sumCubes(a: Int, b: Int): Int =
  if a > b then 0 else cube(a) + sumCubes(a + 1, b)
```

Example (ctd)

Take the sum of the factorials of all the integers between `a` and `b` :

```
def sumFactorials(a: Int, b: Int): Int =
  if a > b then 0 else fact(a) + sumFactorials(a + 1, b)
```

These are special cases of

$$\sum_{n=a}^{b} f(n)$$

for different values of f.

Can we factor out the common pattern?

Summing with Higher-Order Functions

Let's define:

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if a > b then 0
  else f(a) + sum(f, a + 1, b)
```

We can then write:

```
def sumInts(a: Int, b: Int)       = sum(id, a, b)
def sumCubes(a: Int, b: Int)      = sum(cube, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

where

```
def id(x: Int): Int   = x
def cube(x: Int): Int = x * x * x
def fact(x: Int): Int = if x == 0 then 1 else x * fact(x - 1)
```

Function Types

The type `A => B` is the type of a *function* that takes an argument of type `A` and returns a result of type `B`.

So, `Int => Int` is the type of functions that map integers to integers.

Anonymous Functions

Passing functions as parameters leads to the creation of many small functions.

- Sometimes it is tedious to have to define (and name) these functions using `def`.

Compare to strings: We do not need to define a string using `def`. Instead of

```
def str = "abc"; println(str)
```

We can directly write

```
println("abc")
```

because strings exist as *literals*. Analogously we would like function literals, which let us write a function without giving it a name.

These are called *anonymous functions*.

Anonymous Function Syntax

Example: A function that raises its argument to a cube:

```
(x: Int) => x * x * x
```

Here, `(x: Int)` is the *parameter* of the function, and `x * x * x` is it's *body*.

- The type of the parameter can be omitted if it can be inferred by the compiler from the context.

If there are several parameters, they are separated by commas:

```
(x: Int, y: Int) => x + y
```

Anonymous Functions are Syntactic Sugar

An anonymous function $(x_1 : T_1, ..., x_n : T_n) \Rightarrow E$ can always be expressed using `def` as follows:

$$\mathtt{def}\ f(x_1 : T_1, ..., x_n : T_n) = E; f$$

where f is an arbitrary, fresh name (that's not yet used in the program).

- One can therefore say that anonymous functions are *syntactic sugar*.

Summation with Anonymous Functions

Using anonymous functions, we can write sums in a shorter way:

```
def sumInts(a: Int, b: Int)  = sum(x => x, a, b)
def sumCubes(a: Int, b: Int) = sum(x => x * x * x, a, b)
```

Exercise

The `sum` function uses linear recursion. Write a tail-recursive version by replacing the `???`s.

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  def loop(a: Int, acc: Int): Int =
    if ??? then ???
    else loop(???, ???)
  loop(???, ???)
```