Tail Recursion

Review: Evaluating a Function Application

One simple rule : One evaluates a function application $\texttt{f}(e_1, ..., e_n)$

- by evaluating the expressions $e_1, \ldots, e_n$ resulting in the values $v_1, ..., v_n$, then
- by replacing the application with the body of the function `f`, in which
- the actual parameters $v_1, ..., v_n$ replace the formal parameters of `f`.

Application Rewriting Rule

This can be formalized as a *rewriting of the program itself*:

$$\begin{array}{ll} & \texttt{def f}(x_1, ..., x_n) = B;\ \ ...\ \ \texttt{f}(v_1, ..., v_n) \\ \rightarrow & \\ & \texttt{def f}(x_1, ..., x_n) = B;\ \ ...\ \ [v_1/x_1, ..., v_n/x_n]\, B \end{array}$$

Here, $[v_1/x_1, ..., v_n/x_n]\, B$ means:

The expression B in which all occurrences of x_i have been replaced by v_i.

$[v_1/x_1, ..., v_n/x_n]$ is called a *substitution*.

Rewriting example:

Consider `gcd`, the function that computes the greatest common divisor of two numbers.

Here's an implementation of `gcd` using Euclid's algorithm.

```
def gcd(a: Int, b: Int): Int =
  if (b == 0) a else gcd(b, a % b)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
```

$\longrightarrow$ `if (21 == 0) 14 else gcd(21, 14 % 21)`

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
```

$\rightarrow$ `if (21 == 0) 14 else gcd(21, 14 % 21)`

$\rightarrow$ `if (false) 14 else gcd(21, 14 % 21)`

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
```

$\longrightarrow$ `if (21 == 0) 14 else gcd(21, 14 % 21)`

$\longrightarrow$ `if (false) 14 else gcd(21, 14 % 21)`

$\longrightarrow$ `gcd(21, 14 % 21)`

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
```

$\rightarrow$ `if (21 == 0) 14 else gcd(21, 14 % 21)`

$\rightarrow$ `if (false) 14 else gcd(21, 14 % 21)`

$\rightarrow$ `gcd(21, 14 % 21)`

$\rightarrow$ `gcd(21, 14)`

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 21 else gcd(14, 21 % 14)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
↠ gcd(7, 0)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
↠ gcd(7, 0)
→ if (0 == 0) 7 else gcd(0, 7 % 0)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
↠ gcd(7, 0)
→ if (0 == 0) 7 else gcd(0, 7 % 0)
→ 7
```

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)

factorial(4)
```

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

```
factorial(4)
```

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)`

$\longrightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\longrightarrow\!\!\!\!\!\rightarrow$ `4 * factorial(3)`

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)`

$\longrightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\longrightarrow\!\!\!\!\!\rightarrow$ `4 * factorial(3)`

$\longrightarrow\!\!\!\!\!\rightarrow$ `4 * (3 * factorial(2))`

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0)))`

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * 1)))`

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * 1)))`

$\twoheadrightarrow$ `120`

What are the differences between the two sequences?

Tail Recursion

Implementation Consideration: If a function calls itself as its last action, the function's stack frame can be reused. This is called *tail recursion*.

$\Rightarrow$ Tail recursive functions are iterative processes.

In general, if the last action of a function consists of calling a function (which may be the same), one stack frame would be sufficient for both functions. Such calls are called *tail-calls*.

Tail Recursion in Scala

In Scala, only directly recursive calls to the current function are optimized.

One can require that a function is tail-recursive using a `@tailrec` annotation:

```
@tailrec
def gcd(a: Int, b: Int): Int = ...
```

If the annotation is given, and the implementation of `gcd` were not tail recursive, an error would be issued.

Exercise: Tail recursion

Design a tail recursive version of `factorial`.

Higher-Order Functions

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) 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) 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) 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) 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) 1 else 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:

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

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

1. Write a `product` function that calculates the product of the values of a function for the points on a given interval.
2. Write `factorial` in terms of `product`.
3. Can you write a more general function, which generalizes both `sum` and `product`?

Currying

Motivation

Look again at the summation functions:

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

Question

Note that `a` and `b` get passed unchanged from `sumInts` and `sumCubes` into `sum`.

Can we be even shorter by getting rid of these parameters?

Functions Returning Functions

Let's rewrite `sum` as follows.

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

`sum` is now a function that returns another function.

The returned function `sumF` applies the given function parameter `f` and sums the results.

Stepwise Applications

We can then define:

```
def sumInts       = sum(x => x)
def sumCubes      = sum(x => x * x * x)
def sumFactorials = sum(fact)
```

These functions can in turn be applied like any other function:

```
sumCubes(1, 10) + sumFactorials(10, 20)
```

Consecutive Stepwise Applications

In the previous example, can we avoid the `sumInts`, `sumCubes`, … middlemen?

Of course:

```
sum (cube) (1, 10)
```

Consecutive Stepwise Applications

In the previous example, can we avoid the `sumInts`, `sumCubes`, … middlemen?

Of course:

```
sum (cube) (1, 10)
```

- `sum(cube)` applies `sum` to `cube` and returns the *sum of cubes* function.
- `sum(cube)` is therefore equivalent to `sumCubes`.
- This function is next applied to the arguments `(1, 10)`.

Consecutive Stepwise Applications

In the previous example, can we avoid the `sumInts`, `sumCubes`, … middlemen?

Of course:

```
sum (cube) (1, 10)
```

- `sum(cube)` applies `sum` to `cube` and returns the *sum of cubes* function.
- `sum(cube)` is therefore equivalent to `sumCubes`.
- This function is next applied to the arguments `(1, 10)`.

Generally, function application associates to the left:

```
sum(cube)(1, 10)   ==   (sum (cube)) (1, 10)
```

Multiple Parameter Lists

The definition of functions that return functions is so useful in functional programming that there is a special syntax for it in Scala.

For example, the following definition of `sum` is equivalent to the one with the nested `sumF` function, but shorter:

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

Expansion of Multiple Parameter Lists

In general, a definition of a function with multiple parameter lists

$$\texttt{def f}(\texttt{args}_1)...(\texttt{args}_n) = \texttt{E}$$

where $n > 1$, is equivalent to

$$\texttt{def f}(\texttt{args}_1)...(\texttt{args}_{n-1}) = \{\texttt{def g}(\texttt{args}_n) = \texttt{E}; \texttt{g}\}$$

where g is a fresh identifier. Or for short:

$$\texttt{def f}(\texttt{args}_1)...(\texttt{args}_{n-1}) = (\texttt{args}_n \Rightarrow \texttt{E})$$

Expansion of Multiple Parameter Lists (2)

By repeating the process *n* times

$$\texttt{def f}(\texttt{args}_1)...(\texttt{args}_{n-1})(\texttt{args}_n) = \texttt{E}$$

is shown to be equivalent to

$$\texttt{def f} = (\texttt{args}_1 \Rightarrow (\texttt{args}_2 \Rightarrow ...(\texttt{args}_n \Rightarrow \texttt{E})...))$$

This style of definition and function application is called *currying*, named for its instigator, Haskell Brooks Curry (1900-1982), a twentieth century logician.

In fact, the idea goes back even further to Schönfinkel and Frege, but the term "currying" has stuck.

More Function Types

Question: Given,

```
def sum(f: Int => Int)(a: Int, b: Int): Int = ...
```

What is the type of `sum` ?

More Function Types

Question: Given,

```
def sum(f: Int => Int)(a: Int, b: Int): Int = ...
```

What is the type of `sum` ?

Answer:

```
(Int => Int) => (Int, Int) => Int
```

Note that functional types associate to the right. That is to say that

```
Int => Int => Int
```

is equivalent to

```
Int => (Int => Int)
```

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 (???) ???
    else loop(???, ???)
  }
  loop(???, ???)
}
```

Example: Finding Fixed Points

Finding a fixed point of a function

A number `x` is called a *fixed point* of a function `f` if

```
f(x) = x
```

For some functions `f` we can locate the fixed points by starting with an initial estimate and then by applying `f` in a repetitive way.

```
x, f(x), f(f(x)), f(f(f(x))), ...
```

until the value does not vary anymore (or the change is sufficiently small).

Programmatic Solution

This leads to the following function for finding a fixed point:

```
val tolerance = 0.0001
def isCloseEnough(x: Double, y: Double) =
  abs((x - y) / x) / x < tolerance
def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
    val next = f(guess)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

Return to Square Roots

Here is a *specification* of the `sqrt` function:

> `sqrt(x)` = the number `y` such that `y * y = x`.

Or, by dividing both sides of the equation with `y`:

> `sqrt(x)` = the number `y` such that `y = x / y`.

Consequently, `sqrt(x)` is a fixed point of the function `(y => x / y)`.

First Attempt

This suggests to calculate `sqrt(x)` by iteration towards a fixed point:

```
def sqrt(x: Double) =
  fixedPoint(y => x / y)(1.0)
```

Unfortunately, this does not converge.

Let's add a `println` instruction to the function `fixedPoint` so we can follow the current value of `guess`:

First Attempt (2)

```
def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
    val next = f(guess)
    println(next)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

`sqrt(2)` then produces:

```
2.0
1.0
2.0
1.0
```

Average Damping

One way to control such oscillations is to prevent the estimation from varying too much. This is done by *averaging* successive values of the original sequence:

```
def sqrt(x: Double) = fixedPoint(y => (y + x / y) / 2)(1.0)
```

This produces

```
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
1.4142135623746899
```

In fact, if we expand the fixed point function `fixedPoint` we find a similar square root function to what we developed last week.

Functions as Return Values

The previous examples have shown that the expressive power of a language is greatly increased if we can pass function arguments.

The following example shows that functions that return functions can also be very useful.

Consider again iteration towards a fixed point.

We begin by observing that $\sqrt{x}$ is a fixed point of the function `y => x / y`.

Then, the iteration converges by averaging successive values.

This technique of *stabilizing by averaging* is general enough to merit being abstracted into its own function.

```
def averageDamp(f: Double => Double)(x: Double) = (x + f(x)) / 2
```

Exercise:

Write a square root function using `fixedPoint` and `averageDamp`.

Final Formulation of Square Root

```
def sqrt(x: Double) = fixedPoint(averageDamp(y => x/y))(1.0)
```

This expresses the elements of the algorithm as clearly as possible.

Summary

We saw last week that the functions are essential abstractions because they allow us to introduce general methods to perform computations as explicit and named elements in our programming language.

This week, we've seen that these abstractions can be combined with higher-order functions to create new abstractions.

As a programmer, one must look for opportunities to abstract and reuse.

The highest level of abstraction is not always the best, but it is important to know the techniques of abstraction, so as to use them when appropriate.

Scala Syntax Summary

Language Elements Seen So Far:

We have seen language elements to express types, expressions and definitions.

Below, we give their context-free syntax in Extended Backus-Naur form (EBNF), where

- `|` denotes an alternative,
- `[...]` an option (0 or 1),
- `{...}` a repetition (0 or more).

Types

```
Type         = SimpleType | FunctionType
FunctionType = SimpleType '=>' Type
             | '(' [Types] ')' '=>' Type
SimpleType   = Ident
Types        = Type {',' Type}
```

A *type* can be:

- A *numeric type*: `Int`, `Double` (and `Byte`, `Short`, `Char`, `Long`, `Float`),
- The `Boolean` type with the values `true` and `false`,
- The `String` type,
- A *function type*, like `Int => Int`, `(Int, Int) => Int`.

Later we will see more forms of types.

Expressions

```
Expr         = InfixExpr | FunctionExpr
             | if '(' Expr ')' Expr else Expr
InfixExpr    = PrefixExpr | InfixExpr Operator InfixExpr
Operator     = ident
PrefixExpr   = ['+' | '-' | '!' | '~' ] SimpleExpr
SimpleExpr   = ident | literal | SimpleExpr '.' ident
             | Block
FunctionExpr = Bindings '=>' Expr
Bindings     = ident [':' SimpleType]
             | '(' [Binding {',' Binding}] ')'
Binding      = ident [':' Type]
Block        = '{' {Def ';'} Expr '}'
```

Expressions (2)

An *expression* can be:

- An *identifier* such as `x`, `isGoodEnough`,
- A *literal*, like `0`, `1.0`, `"abc"`,
- A *function application*, like `sqrt(x)`,
- An *operator application*, like `-x`, `y + x`,
- A *selection*, like `math.abs`,
- A *conditional expression*, like `if (x < 0) -x else x`,
- A *block*, like `{ val x = math.abs(y) ; x * 2 }`
- An *anonymous function*, like `x => x + 1`.

Definitions

```
Def          = FunDef  |  ValDef
FunDef       = def ident {'(' [Parameters] ')'}
               [':' Type] '=' Expr
ValDef       = val ident [':' Type] '=' Expr
Parameter    = ident ':' [ '=>' ] Type
Parameters   = Parameter {',' Parameter}
```

A *definition* can be:

- A *function definition*, like `def square(x: Int) = x * x`
- A *value definition*, like `val y = square(2)`

A *parameter* can be:

- A *call-by-value parameter*, like `(x: Int)`,
- A *call-by-name parameter*, like `(y: => Double)`.