Concurrency: Languages, Programming and Theory

– Functional Programming and Lambda Calculus –

Session 1 – Oct 22, 2003

Martin Odersky

EPFL-LAMP

Part I: Functional Programming

- A pure functional program consists of data, functions, and an expression which describes a result.
- Missing: variables, assignment, side-effects.
- A processor of a functional program is essentially a calculator.

Example: (transcript of a session with *scalarun*, the Scala interpreter)

```
/home/odersky/tmp> scalarun
> def gcd (a: int, b: int): int = if (b == 0) a else gcd (b, a % b)
'def gcd'
> gcd (8, 10)
2
> val x = gcd (15, 70)
val x : int = 5
> val y = gcd(x, x)
val y : int = 5
```


Why Study Functional Programming?

- FP is programming in its simplest form ⇒ easier to understand thoroughly than more complex variants.
- FP has powerful composition constructs.
- In FP, only the *value* of an expression matters since side effects are impossible. (this property is called *referential transparency*).
- Referential transparency gives a rich set of laws to transform programs.
- FP has a well-established theoretical basis in Lambda Calculus and Denotational Semantics.

Square Roots by Newton's Method

Compute the square root of a given number x as a limit of the sequence y_i given by:

$$\begin{aligned} y_0 &= 1 \\ y_{i+1} &= (y_i + x/y_i)/2 \end{aligned}$$

The $i \to i+1$ step is encoded in the function *improve*:

```
> def improve (guess: double, x: double) = (guess + x / guess) / 2
def improve : (guess : double,x : double)double
> val y0 = 1.0
val y0 : double = 1.0
> val y1 = improve (y0, 2.0)
val y1 : double = 1.5
> val y2 = improve (y1, 2.0)
val y2 : double = 1.4166666666666665
> val y3 = improve (y2, 2.0)
val y3 : double = 1.4142156862745097
```


We have to stop the iteration when the result is good enough:

```
> def abs(x: double): double = if (x ≥ 0) x else −x
def abs : (x : double)double
> def goodEnough (guess: double, x: double): boolean =
|  abs((guess ∗ guess) − x) < 0.001
def goodEnough : (guess : double,x : double)boolean
> def sqrtIter(guess: double, x: double): double =
|  if (goodEnough(guess, x)) guess else sqrtIter (improve(guess, x), x)
def sqrtIter : (guess : double,x : double)double
> def sqrt(x: double): double = sqrtIter(1.0, x)
def sqrt : (x : double)double
> sqrt (2.0)
1.4142156862745097
```


Language Elements Seen So Far

- Function Definitions:

  def *Ident Parameters* [*':' ResultType*] *"=" Expression*

- Value definitions:

  val *Ident "=" Expression*

- Function application: *Ident*'(' $Expr_1$, ..., $Expr_2$ ')'
- Numbers, operators: as in Java
- If-then-else: as in Java, but as an expression.
- Types: as in Java.

Nested Functions

If functions are used only internally by some other function we can avoid "name-space pollution" by nesting. E.g:

```
def sqrt (x : double) = {
  def improve (guess: double, x: double) = (guess + x / guess) / 2;
  def goodEnough (guess: double, x: double) =
     abs ((guess * guess) − x) < 0.001;
  def sqrtIter (guess: double, x: double): double =
    if (goodEnough (guess, x)) guess
    else sqrtIter (improve (guess, x), x);
  sqrtIter (1.0, x)
}
```

The visibility of an identifier extends from its own definition to the end of the enclosing block, including any nested definitions.

Exercise:

- ☐ The *goodEnough* function tests the absolute difference between the input parameter and the square of the guess.
- ☐ This is not very accurate for square roots of very small numbers and might lead to divergence for very large numbers (why?).
- ☐ Design a different *sqrtIter* function which stops if the *change* from one iteration to the next is a small fraction of the guess. E.g.

$$abs((x_{i+1} - x_i)/x_i) < 0.001$$

Complete:

```
def sqrtIter(guess: double, x: double): double = ?
```


Semantics of Function Application

- ☐ One simple rule: A function application *f* (*A*) is evaluated by
  - replacing the application with the function's body where
  - actual parameters *A* replace formal parameters of *f*.
- ☐ This can be formalised as a *rewriting of the program itself*:

  def *f* (*x*) = *B* ; ... *f* (*A*) $\rightarrow$ **def** *f* (*x*) = *B* ; ... [*A/x*] *B*
- ☐ Here, [*A/x*] *B* stands for *B* with all occurrences of *x* replaced by *A*.
- ☐ [*A/x*] *B* is called a *substitution*.

Rewriting Example:

Consider *gcd*:

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

Then *gcd* (*14, 21*) evaluates as follows:

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

Then *factorial*(*5*) rewrites as follows:

```
           factorial (5)
→          if (5 == 0) 1 else 5 * factorial (5 − 1)
→          5 * factorial (5 − 1)
→          5 * factorial (4)
→ ... →    5 * (4 * factorial (3))
→ ... →    5 * (4 * (3 * factorial (2)))
→ ... →    5 * (4 * (3 * (2 * factorial (1))))
→ ... →    5 * (4 * (3 * (2 * (1 * factorial (0)))))
→ ... →    5 * (4 * (3 * (2 * (1 * 1))))
→ ... →    120
```


Question:

What differences are there between the two rewrite sequences?

Tail Recursion

- ☐ Implementation note: If a function calls itself as its last action, the function's stack frame can be re-used. This is called "tail recursion".
- ☐ ⇒ Tail-recursive functions are iterative processes.
- ☐ More generally, if the last action of a function is a call to another (possible the same) function, only a single stack frame is needed for both functions. Such calls are called "tail calls".

Exercise: Design a tail-recursive version of *factorial*.

First-Class Functions

- ☐ Most functional languages treat functions as "first-class values".
- ☐ That is, like any other value, a function may be passed as a parameter or returned as a result.
- ☐ This provides a flexible mechanism for program composition.
- ☐ Functions which take other functions as parameters or return them as results are called "higher-order" functions..

Example

- Sum integers between a and b:

  def *sumInts* $(a$: *int*, b: *int*$)$: *double* =
    if $(a > b)$ *0.0* ***else*** $a +$ *sumInts* $(a +$ *1*, $b)$;

- Sum cubes of all integers between a and b:

  def *cube* $(a$: *int*$)$ = $a * a * a$;
  def *sumCubes* $(a$: *int*, b: *int*$)$: *double* =
    if $(a > b)$ *0.0* ***else*** *cube* $(a) +$ *sumCubes* $(a +$ *1*, $b)$;

- Sum reciprocals between a and b

  def *sumReciprocals* $(a$: *int*, b: *int*$)$: *double* =
    if $(a > b)$ *0* ***else*** *1.0* / $a +$ *sumReciprocals* $(a +$ *1*, $b)$;

- These are all special cases of $\sum_a^b f(n)$ for different values of f.

Summation with a higher-order function

- Can we factor out the common pattern?
- Define:

  def *sum*$(f$: *int* $\Rightarrow$ *double*, a: *int*, b: *int*$)$: *double* =
    if $(a > b)$ *0.0* ***else*** $f(a) +$ *sum*$(f, a +$ *1*, $b)$;

- Then we can write:

  def *sumInts*$(a$: *int*, b: *int*$)$ = *sum*(id, a, b);
  def *sumCubes*$(a$: *int*, b: *int*$)$ = *sum*$(cube, a, b)$;
  def *sumReciprocals*$(a$: *int*, b: *int*$)$ = *sum*$(reciprocal, a, b)$;

  where

  def *id*$(x$: *int*$)$ = x;
  def *cube*$(x$: *int*$)$ = $x * x * x$;
  def *reciprocal*$(x$: *int*$)$ = *1.0* / x;

Anonymous functions

- Parameterisation by functions tends to create many small functions.
- Sometimes it is cumbersome to have to define the functions using ***def***.
- A shorter notation makes use of *anonymous functions*, defined as follows:
  $(x_1 : T_1, ..., x_n : T_n) \Rightarrow E$ defines a function which maps its parameters $x_1, ..., x_n$ to the result of the expression E (where E may refer to $x_1, ..., x_n$).
- The parameter types T_i may be omitted if they can be reconstructed "from the context".

- Anonymous functions are not essential in Scala; an anonymous function $(x_1, ..., x_n) \Rightarrow E$ can always be expressed using a ***def*** as follows:

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

  where f is fresh name which is used nowhere else in the program.

- We also say, anonymous functions are "syntactic sugar".

Summation with Anonymous Functions

Now we can write shorter:

def *sumInts*$(a$: *int*, b: *int*$)$ = *sum*$((x \Rightarrow x), a, b)$;
def *sumCubes*$(a$: *int*, b: *int*$)$ = *sum*$((x \Rightarrow x * x * x), a, b)$;
def *sumReciprocals*$(a$: *int*, b: *int*$)$ = *sum*$((x \Rightarrow$ *1.0* / $x), a, b)$;

Can we do even better?

Hint: *a, b* appears everywhere and does not seem to take part in interesting combinations. Can we get rid of it?

Currying

Let's rewrite *sum* as follows.

def *sum*$(f$: *int* $\Rightarrow$ *double*$)$ = $\{$
  def *sumFun* $(a$: *int*, b: *int*$)$: *double* =
    if $(a > b)$ *0.0*
    else $f(a) +$ *sumFun*$(a +$ *1*, $b)$;
  sumFun
$\}$

- *sum* is now a function which returns another function;
- Namely, the specialized summing function which applies the f function and sums up the results.

Then we can define:

val *sumInts* = *sum* $(x \Rightarrow x)$;
val *sumCubes* = *sum* $(x \Rightarrow x * x * x)$;
val *sumReciprocals* = *sum* $(x \Rightarrow$ *1.0* / $x)$;

Function values can be applied like other functions:

sumReciprocals (*1*, *1000*)

Curried Application

How are function-returning functions applied?
Example:

> *sum* (*cube*) (*1*, *10*)
3025

- *sum* (*cube*) applies *sum* to *cube* and returns the "cube-summing function" (Hence, *sum* (*cube*) is equivalent to *sumCubes*).
- This function is then applied to the pair (*1*, *10*).
- Hence, function application associates to the left:

  sum (*cube*) (*1*, *10*) == (*sum* (*cube*)) (*1*, *10*)
                          == ***val*** *sc* = *sum* (*cube*) ; *sc* (*1*, *10*)

Curried Definition

- The style of function-returning functions is so useful in FP, that we have special syntax for it.
- For instance, the next definition of *sum* is equivalent to the previous one, but shorter:

```
def sum (f: (int, int) ⇒ double) (a; int, b: int): double = {
  if (a > b) 0.0
  else f(a) + sum(f)(a + 1, b)
}
```


Generally, a curried function definition

def $f\,(args_1)$ ... $(args_n)$ = E

where $n > 1$ expands to

def $f\,(args_1)$ ... $(args_{n-1})$ = (**def** $g\,(args_n)$ = E ; g)

where *g* is a fresh identifier. Or, shorter:

def $f\,(args_1)$ ... $(args_{n-1})$ = ($args_n$) ⇒ E

Performing this step n times yields that

def $f\,(args_1)$ ... $(args_{n-1})$ $(args_n)$ = E

is equivalent to

def f = $(args_1)$ ⇒ ($args_2$) ⇒ ... ($args_n$) ⇒ E

- Again, parentheses around single-name formal parameters may be dropped.
- This style of function definition and application is called *currying* after its promoter, Haskell B. Curry.
- Actually, the idea goes back further to Frege and Schönfinkel, but the name “curried” caught on (maybe because “schönfinkeled” does not sound so well.)

Exercises:

1. The *sum* function uses a linear recursion. Can you write a tail-recursive one by filling in the ??’s?

```
def sum f (a: int, b: int): double = {
  def iter (a: int, result: double): double = {
    if (??) ??
    else iter (??, ??)
  }
  iter (??, ??)
}
```


2. Write a function *product* that computes the product of the values of functions at points over a given range.

3. Write *factorial* in terms of *product*.

4. Can you write an even more general function which generalizes both *sum* and *product*?