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 $\Rightarrow$ 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* $\geq$ *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* $\left[\right.$*':' ResultType*$\left.\right]$ *"=" 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:

$$
\begin{array}{l}
\textbf{\textit{def}}\ \textit{sqrt}\ (\textit{x} : \textit{double}) = \{ \\
\quad \textbf{\textit{def}}\ \textit{improve}\ (\textit{guess}: \textit{double, x}: \textit{double}) = (\textit{guess} + \textit{x / guess})\ \textit{/ 2;} \\
\quad \textbf{\textit{def}}\ \textit{goodEnough}\ (\textit{guess}: \textit{double, x}: \textit{double}) = \\
\qquad \textit{abs}\ ((\textit{guess} * \textit{guess}) - \textit{x}) < \textit{0.001;} \\
\quad \textbf{\textit{def}}\ \textit{sqrtIter}\ (\textit{guess}: \textit{double, x}: \textit{double}): \textit{double} = \\
\qquad \textbf{\textit{if}}\ (\textit{goodEnough}\ (\textit{guess, x}))\ \textit{guess} \\
\qquad \textbf{\textit{else}}\ \textit{sqrtIter}\ (\textit{improve}\ (\textit{guess, x}), \textit{x}); \\
\quad \textit{sqrtIter}\ (\textit{1.0, x}) \\
\}
\end{array}
$$

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 $f\left(A\right)$ 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*:

$$\textbf{\textit{def}}\ f\left(x\right) = B\ ;\ \ldots\ f\left(A\right) \quad\longrightarrow\quad \textbf{\textit{def}}\ f\left(x\right) = B\ ;\ \ldots\ \left[A/x\right] B$$

- Here, $\left[A/x\right] B$ stands for *B* with all occurrences of *x* replaced by *A*.
- $\left[A/x\right] B$ is called a *substitution*.

Rewriting Example:

Consider *gcd*:

$$\textbf{\textit{def}}\ \textit{gcd}\left(a : \textit{int}, b : \textit{int}\right) = \textbf{\textit{if}}\ \left(b == 0\right)\ a\ \textbf{\textit{else}}\ \textit{gcd}\ \left(b, a \% b\right)$$

Then $\textit{gcd}\ \left(14, 21\right)$ evaluates as follows:

$$\begin{array}{ll}
 & \textit{gcd}\ \left(14, 21\right) \\
\longrightarrow & \textbf{\textit{if}}\ \left(21 == 0\right)\ 14\ \textbf{\textit{else}}\ \textit{gcd}\ \left(21, 14 \% 21\right) \\
\longrightarrow & \textit{gcd}\ \left(21, 14\right) \\
\longrightarrow & \textbf{\textit{if}}\ \left(14 == 0\right)\ 21\ \textbf{\textit{else}}\ \textit{gcd}\ \left(14, 21 \% 14\right) \\
\longrightarrow\longrightarrow & \textit{gcd}\ \left(14, 7\right) \\
\longrightarrow & \textbf{\textit{if}}\ \left(7 == 0\right)\ 14\ \textbf{\textit{else}}\ \textit{gcd}\ \left(7, 14 \% 7\right) \\
\longrightarrow\longrightarrow & \textit{gcd}\ \left(7, 0\right) \\
\longrightarrow & \textbf{\textit{if}}\ \left(0 == 0\right)\ 7\ \textbf{\textit{else}}\ \textit{gcd}\ \left(0, 7 \% 0\right) \\
\longrightarrow & 7
\end{array}$$

Another rewriting example:

Consider *factorial*:

$$\textbf{\textit{def}}\ \mathit{factorial}\ (n : \mathit{int}) : \mathit{int} = \textbf{\textit{if}}\ (n == 0)\ 1\ \textbf{\textit{else}}\ n * \mathit{factorial}\ (n - 1)$$

Then *factorial*(5) rewrites as follows:

$$\begin{array}{ll}
 & \mathit{factorial}\ (5) \\
\longrightarrow & \textbf{\textit{if}}\ (5 == 0)\ 1\ \textbf{\textit{else}}\ 5 * \mathit{factorial}\ (5 - 1) \\
\longrightarrow & 5 * \mathit{factorial}\ (5 - 1) \\
\longrightarrow & 5 * \mathit{factorial}\ (4) \\
\longrightarrow \ldots \longrightarrow & 5 * (4 * \mathit{factorial}\ (3)) \\
\longrightarrow \ldots \longrightarrow & 5 * (4 * (3 * \mathit{factorial}\ (2))) \\
\longrightarrow \ldots \longrightarrow & 5 * (4 * (3 * (2 * \mathit{factorial}\ (1)))) \\
\longrightarrow \ldots \longrightarrow & 5 * (4 * (3 * (2 * (1 * \mathit{factorial}\ (0))))) \\
\longrightarrow \ldots \longrightarrow & 5 * (4 * (3 * (2 * (1 * 1)))) \\
\longrightarrow \ldots \longrightarrow & 120
\end{array}$$

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".
- $\Rightarrow$ 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:

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

- Then we can write:

$$\textbf{\textit{def}}\ \textit{sumInts}\big(a: \textit{int}, b: \textit{int}\big) = \textit{sum}\big(\textit{id, a, b}\big);$$
$$\textbf{\textit{def}}\ \textit{sumCubes}\big(a: \textit{int}, b: \textit{int}\big) = \textit{sum}\big(\textit{cube, a, b}\big);$$
$$\textbf{\textit{def}}\ \textit{sumReciprocals}\big(a: \textit{int}, b: \textit{int}\big) = \textit{sum}\big(\textit{reciprocal, a, b}\big);$$

where

$$\textbf{\textit{def}}\ \textit{id}\big(x: \textit{int}\big) = x;$$
$$\textbf{\textit{def}}\ \textit{cube}\big(x: \textit{int}\big) = x * x * x;$$
$$\textbf{\textit{def}}\ \textit{reciprocal}\big(x: \textit{int}\big) = 1.0\ /\ x;$$

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:

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

  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:

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

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.

$$
\begin{array}{l}
\textbf{\textit{def}}\ \textit{sum}\big(\textit{f}: \textit{int} \Rightarrow \textit{double}\big) = \big\{ \\
\quad \textbf{\textit{def}}\ \textit{sumFun}\ \big(\textit{a}: \textit{int},\ \textit{b}: \textit{int}\big): \textit{double} = \\
\quad\quad \textbf{\textit{if}}\ \big(a > b\big)\ \textit{0.0} \\
\quad\quad \textbf{\textit{else}}\ \textit{f}\big(a\big) + \textit{sumFun}\big(a + \textit{1},\ b\big); \\
\quad \textit{sumFun} \\
\big\}
\end{array}
$$

- *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:

$> \textit{sum}\ (\textit{cube})\ (1, 10)$
3025

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

$$\begin{aligned} \textit{sum}\ (\textit{cube})\ (1, 10) \quad &== \quad (\textit{sum}\ (\textit{cube}))\ (1, 10) \\ &== \quad \textbf{\textit{val}}\ \textit{sc} = \textit{sum}\ (\textit{cube})\ ;\ \textit{sc}\ (1, 10) \end{aligned}$$

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:

$$
\begin{array}{l}
\textbf{\textit{def}}\ \textit{sum}\ \big(\textit{f}: \big(\textit{int, int}\big) \Rightarrow \textit{double}\big)\ \big(\textit{a; int, b}: \textit{int}\big): \textit{double} = \big\{ \\
\quad \textbf{\textit{if}}\ \big(a > b\big)\ \textit{0.0} \\
\quad \textbf{\textit{else}}\ \textit{f}\big(a\big) + \textit{sum}\big(\textit{f}\big)\big(a + \textit{1, b}\big) \\
\big\}
\end{array}
$$

Generally, a curried function definition

$$\textbf{\textit{def}}\ f\ (args_1)\ ...\ (args_n) = E$$

where $n > 1$ expands to

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

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

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

Performing this step n times yields that

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

is equivalent to

$$\textbf{\textit{def}}\ f = (args_1) \Rightarrow (\ args_2) \Rightarrow ...\ (\ args_n) \Rightarrow E$$

- 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?

$$
\begin{array}{l}
\textbf{\textit{def}}\ \textit{sum f}\ \left(a : \textit{int},\ b : \textit{int}\right) : \textit{double} = \{ \\
\quad \textbf{\textit{def}}\ \textit{iter}\ \left(a : \textit{int},\ \textit{result} : \textit{double}\right) : \textit{double} = \{ \\
\quad\quad \textbf{\textit{if}}\ \left(??\right)\ ?? \\
\quad\quad \textbf{\textit{else}}\ \textit{iter}\ \left(??,\ ??\right) \\
\quad \} \\
\quad \textit{iter}\ \left(??,\ ??\right) \\
\}
\end{array}
$$

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*?