Concurrency: Theory, Languages and Programming

– Functional Programming and Lambda Calculus –

Session 2 – Oct 30rd, 2002

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 *siris*, the Scala interpreter)

```
/home/odersky/tmp    siris
    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, one 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 as a limit of the sequence given by:

The 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* , ..., *Expr* ')'

Tuples: (*Expr* , ..., *Expr*)

Numbers, operators: as in Java

If-then-else: as in Java, but as an expression.

Types: as in Java, but written upper case.

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) =
  def improve (guess, x) = (guess    x / guess) / 2
  def goodEnough (guess, x) = abs ((guess   guess)    x)    0.001f
  def sqrtIter (guess, x) =
    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.

Complete:

def *sqrtIter* (*guess* : *Double*, *x* : *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 $f\ (x) = B\ ;\ \ldots\ f\ (A)$ **def** $f\ (x) = B\ ;\ \ldots\ [A/x]\ B$

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*:

$$\textbf{def}\ gcd\,(a : Int,\ b : Int) = \textbf{if}\ (b \quad\quad 0)\ a\ \textbf{else}\ gcd\ (b,\ a \quad b)$$

Then $gcd\ (14,\ 21)$ evaluates as follows:

$$
\begin{array}{l}
gcd\ (14,\ 21) \\
\textbf{if}\ (21 \quad\quad 0)\ 14\ \textbf{else}\ gcd\ (21,\ 14 \quad 21) \\
gcd\ (21,\ 14) \\
\textbf{if}\ (14 \quad\quad 0)\ 21\ \textbf{else}\ gcd\ (14,\ 21 \quad 14) \\
gcd\ (14,\ 7) \\
\textbf{if}\ (7 \quad\quad 0)\ 14\ \textbf{else}\ gcd\ (7,\ 14 \quad 7) \\
gcd\ (7,\ 0) \\
\textbf{if}\ (0 \quad\quad 0)\ 7\ \textbf{else}\ gcd\ (0,\ 7 \quad 0) \\
7
\end{array}
$$

Another rewriting example:

Consider *factorial*:

def *factorial* (*n* : *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 $\sum$ for different values of

.

Summation with a higher-order function

Can we factor out the common pattern?

Define:

```
def sum(f: (Int)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:

$(x\ : T\ , ..., x\ : T\ \Rightarrow E)$ defines a function which maps its parameters $x\ , ..., x$ to the result of the expression E (where E may refer to $x\ , ..., x$).

The parameter types T may be omitted if they can be reconstructed “from the context”.

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

> ***def*** $f(x\ : T\ , ..., x\ : T\ ) = E\ ; f$

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      x), a, b)
def sumCubes (a : Int, b : Int) = sum ((x      x    x    x), a, b)
def sumReciprocals (a : Int, b : Int) = sum ((x      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 )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      x)
val sumCubes = sum (x      x    x    x)
val sumReciprocals = sum (x      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*) ... (*args*) = *E*

where expands to

def *f* (*args*) ... (*args*) = (***def*** *g* (*args*) = *E ; g*)

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

def *f* (*args*) ... (*args*) = (*args* *E*)

Performing this step times yields that

def *f* (*args*) ... (*args*) (*args*) = *E*

is equivalent to

def *f* = (*args* (*args* ... (*args* *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*?

Part II: Lambda Calculus

Lambda Calculus is a foundation for functional programs.

It's an operational semantics, based on term rewriting.

Lambda Calculus was developed by Alonzo Church in the 1930's and 40's as a theory of computable functions.

Lambda calculus is as powerful as Turing machines. That is, every Turing machine can be expressed as a function in the calculus and vice versa

Church Hypothesis: Every computable algorithm can be expressed by a function in Lambda calculus.

Pure Lambda Calculus

Pure Lambda calculus expresses only functions and function applications.

Three term forms:

Names	$\mathcal{N}$	
Terms		names
		abstractions
		applications

We generally omit parentheses around single-name function arguments.

Function-application is left-associative.

The scope of a name extends as far to the right as possible.

Example:

Often, one uses the term *variable* instead of *name*.

Evaluation of Lambda Terms

Evaluation of lambda terms is by the -reduction rule.

is substitution, which will be explained in detail later.

Example:

Term Equivalence

Question: Are these terms equivalent?

and

What about

and

?

Need to distinguish between *bound* and *free* names.

Free And Bound Names

Definition The free names of a term are those names which occur in at a position where they are not in the scope of a definition in the same term.

Formally, is defined as follows.

All names which occur in a term and which are not free in are called *bound*.

A term without any free variables is called *closed*.

Renaming

The spelling of bound names is not significant.

We regard terms and which are convertible by renaming of bound names as equivalent, and write

This is expressed formally by the following -renaming rule:

Formally, is the smallest *congruence* which contains the equality of rule .

Substitutions

We now have the means to define substitution formally:

Substitution affects only the free names of a term, not the bound ones.

Avoiding Name Capture

We have to be careful that we do not bind free names of a substituted expression (this is called *name capture*).

For instance,

We have to -rename first before applying the substitution:

by

In the following, we will always assume that terms are renamed automatically so as to make all substitutions well-defined.

Normal Forms

Definition: We write for reduction in an arbitrary number of steps. Formally:

iff

Definition: A *normal form* is a term which cannot be reduced further.

Exercise: Define:

Can be reduced to a normal form?

Combinators

Lambda calculus gives one the possibility to define new functions using abstractions.

Question: Is that really necessary for expressiveness, or could one also do with a fixed set of functions?

Answer: (by Haskell Curry) Every closed -definable function can be expressed as some combination of the *combinators* and .

Combinator Implementation Technique

This insight has influenced the implementation of one functional language (Miranda).

The Miranda compiler translates a source program to a combination of a handful of combinators (, , and a few others for “optimizations”).

A Miranda runtime system then only has to implement the handful of combinators.

Very elegant, but “slow as continental drift”.

Confluence

If a term had more than one normal form, we'd have to worry about an implementation finding "the right one".
The following important theorem shows that this case cannot arise.

Theorem: (Church-Rosser) Reduction in -calculus is *confluent*: If and , then there exists a term such that and .
Proof: Not easy.

Corollary: Every term can be reduced to at most one normal form.
Proof: Your turn.

Terms Without Normal Forms

There are terms which do not have a normal form.

Example: Let

Then

Terms which cannot be reduced to a normal form are called *divergent*.

Evaluation Strategies

The existence of terms without normal forms raises the question of *evaluation strategies*.

For instance, let and consider:

in a single step. But one could also reduce:

by always doing the reduction.

Complete Evaluation Strategies

An evaluation strategy is a decision procedure which tells us which rewrite step to choose, given a term where several reductions are possible.

Question 1: Is there a *complete* evaluation strategy, in the following sense:

> Whenever a term has a normal form, the reduction using the strategy will end in that normal form.

?

Weak Head Normal Forms

In practice, we are not so much interested in normal forms; only in terms which are not further reducible "at the top level".

That is, reduction would stop at a term of the form even if was still reducible.

These terms are called *weak head normal forms* or *values*.

They are characterized by the following grammar.

Values

We now reformulate our question as follows:

Question 2: Is there a (weakly) complete evaluation strategy, in the following sense:

> Whenever a term can be reduced to a value, the reduction using the strategy will end in that value.

Precise Definition of Evaluation Strategy

How can we define evaluation strategies formally?

Idea: Use *reduction contexts*.

Definition: A *context* is a term where exactly one subterm is replaced by a “hole”, written .

denotes the term which results if the hole of context is filled with term .

Examples of contexts:

Previously, we have admitted reduction anywhere in a term without explicitly saying so.

Let’s formalize this:

Definition: A term *reduces at top-level* to a term , if and are the left- and right-hand sides of an instance of rule . We write in this case: .

Definition: A term *reduces* to a term ’, written if there exists a context and terms , such that

So much for general reduction.

Now, to define an evaluation strategy, we *restrict* the possible set of contexts in the definition of .

The restriction can be expressed by giving a *grammar* which describes permissible contexts.

Such contexts are called *reduction contexts* and we let the letter range over them

Call-By-Name

Definition: The *call-by-name* strategy is given by the following grammar for reduction-contexts:

Definition: A term *reduces* to a term ’ using the call-by-name strategy, written cbn if there exists a reduction context and terms , such that

Deterministic Reduction Strategies

Definition: A reduction strategy is *deterministic* if for any term at most one reduction step is possible.

Proposition: The call-by-name strategy $_{\mathrm{cbn}}$ is deterministic.

Proof: There is only one way a term can be split into a reduction context *R* and a subterm which is reducible at top-level.

Exercise: Reduce the term with the call-by-name strategy, where

Theorem: (Standardization) Call-by-name reduction is weakly complete: Whenever then cbn .

Proof: hard.

Normal Order Reduction

Question:

Modify call-by-name reduction to *normal-order reduction*, which always reduces a term to a normal form, if it has one.

Which changes to the definition of reduction contexts are necessary?

In practice, call-by-name is rarely used since it leads to duplicate evaluations of arguments. Example:

Note that the argument is evaluated twice.

A shorter reduction can often be achieved by evaluating function arguments before they are passed. In our example:

Call-By-Value

The *call-by-value* strategy evaluates function arguments before applying the function.

It is often more efficient than the call-by-name strategy. However:

Proposition: The call-by-value strategy is not (weakly) complete.

Question: Name a term which can be reduced to a value following the call-by-name strategy, but not following the call-by-value strategy.

Hence we have a dilemma: One strategy is in practice too inefficient, the other is incomplete.

First Solution: Call-By-Need Evaluation

Idea: Rather than re-evaluating arguments repeatedly, save the result of the first evaluation and use that for subsequent evaluations.

This technique is called *memoization*.

It is used in implementations of *lazy* functional languages such as Miranda or Haskell.

A formalization of call-by-need is possible, but beyond the scope of this course. See

> A Call-by-Need Lambda Calculus, Zena Ariola, Matthias Felleisen, John Maraist, Martin Odersky and Philip Wadler. *Proc. ACM Symposium on Principles of Programming Languages*, 1995.

Second Solution: Call-By-Value Calculus

Rather than tweaking the evaluation strategy to be complete with respect to a given calculus, we can also change the calculus so that a given evaluation strategy becomes complete with respect to it.

This has been done by Gordon Plotkin, in the *call-by-value* lambda calculus.

The *terms* and *values* of this calculus are defined as before. A more concise re-formulation is:

Terms	
Values	W

As reduction rule, we have:

As reduction contexts, we have:

Let be general reduction of terms with the rule, and let $_{\text{cbv}}$ be reduction only at the holes of call-by-value reduction contexts . Then we have:

Theorem: (Plotkin) reduction is confluent.

Theorem: (Plotkin) $_{\text{cbv}}$ is weakly complete with respect to .