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 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*$_2$ ')'
- Tuples: (*Expr*$_1$, ..., *Expr*$_n$)
- 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.

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

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 ; ... f (A)   →   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) = 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)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 => 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*$_1$) ... (*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*$_n$ $\Rightarrow$ *E*)

Performing this step n times yields that

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

is equivalent to

def *f* = (*args*$_1$ $\Rightarrow$ (*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?

```
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	$x, y, z \in \mathcal{N}$			
Terms	D, E, F	$::=$	x	names
		$\mid$	$\lambda x.E$	abstractions
		$\mid$	$D\ E$	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:

$$\lambda f.\lambda x.f\ E\ x \quad \equiv \quad (\lambda f.(\lambda x.((f\ (E))\ (x)))).$$

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

Evaluation of Lambda Terms

Evaluation of lambda terms is by the β-reduction rule.

$$\beta: \qquad (\lambda x.D)E \;\rightarrow\; [E/x]\,D$$

$[E/x]$ is substitution, which will be explained in detail later.

Example:

$$\begin{array}{lcl}
(\lambda x.x)(\lambda y.y) & \rightarrow & \lambda y.y \\
 & & \\
(\lambda f.\lambda x.f\ (f\ x))(\lambda y.y)z & \rightarrow & (\lambda x.(\lambda y.y)(\lambda y.y)x)z \\
 & \rightarrow & (\lambda y.y)((\lambda y.y)z) \\
 & \rightarrow & (\lambda y.y)z \\
 & \rightarrow & z
\end{array}$$

Term Equivalence

Question: Are these terms equivalent?

$$\lambda x.x \quad \text{and} \quad \lambda y.y$$

What about

$$\lambda x.y \quad \text{and} \quad \lambda x.z$$

?

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

Free And Bound Names

Definition The free names $\mathrm{fn}(E)$ of a term E are those names which occur in E at a position where they are not in the scope of a definition in the same term.

Formally, $\mathrm{fn}(E)$ is defined as follows.

$$\begin{array}{lcl}
\mathrm{fn}(x) & = & \{x\} \\
\mathrm{fn}(\lambda x.E) & = & \mathrm{fn}(E)\backslash\{x\} \\
\mathrm{fn}(F\,E) & = & \mathrm{fn}(F) \cup \mathrm{fn}(E).
\end{array}$$

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

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

Renaming

- ☐ The spelling of bound names is not significant.
- ☐ We regard terms D and E which are convertible by renaming of bound names as equivalent, and write $D \equiv E$
- ☐ This is expressed formally by the following α-renaming rule:

$$\alpha: \qquad \lambda x.E \;\equiv\; \lambda y.[y/x]E \qquad (y \notin \mathrm{fn}(E))$$

Formally, $\equiv$ is the smallest *congruence* which contains the equality of rule α.

Substitutions

- ☐ We now have the means to define substitution formally:

$$\begin{array}{lcll}
[D/x]\ x & = & D & \\
[D/x]\ y & = & y & (x \neq y) \\
[D/x]\ \lambda x.E & = & \lambda x.E & \\
[D/x]\ \lambda y.E & = & \lambda y.[D/x]E & (x \neq y, y \notin \mathrm{fn}(D)) \\
[D/x]\ (F\ E) & = & ([D/x]F)\ ([D/x]E) &
\end{array}$$

- ☐ 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,

  $$[y/x]\lambda y.x \;\not\equiv\; \lambda y.y \qquad \text{!!!}$$

- ☐ We have to α-rename $\lambda y.x$ first before applying the substitution:

  $$\begin{array}{lcll}
  [y/x]\lambda y.x & \equiv & [y/x]\lambda z.x & \text{by } \alpha \\
   & \equiv & \lambda z.y &
  \end{array}$$

- ☐ 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 $\twoheadrightarrow$ for reduction in an arbitrary number of steps. Formally:

$$E \twoheadrightarrow E' \quad \text{iff} \quad \exists n \geq 0. E \equiv E_0 \rightarrow \ldots \rightarrow E_n \equiv E'$$

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

Exercise: Define:

$$\begin{array}{lcl}
S & \overset{\text{def}}{\equiv} & \lambda f.\lambda g.\lambda x.fx(gx) \\
K & \overset{\text{def}}{\equiv} & \lambda x.\lambda y.x
\end{array}$$

Can SKK be reduced to a normal form?

Combinators

- ☐ Lambda calculus gives one the possibility to define new functions using λ 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 λ-definable function can be expressed as some combination of the *combinators* S and K.

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 (S, K, 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 λ-calculus is *confluent*: If $E \twoheadrightarrow E_1$ and $E \twoheadrightarrow E_2$, then there exists a term E_3 such that $E_1 \twoheadrightarrow E_3$ and $E_2 \twoheadrightarrow E_3$.
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

$$\Omega \stackrel{\text{def}}{\equiv} (\lambda x.(xx))(\lambda x.(xx))$$

Then

$$\begin{array}{lll} \Omega & \rightarrow & (\lambda x.(xx))(\lambda x.(xx)) \\ & \rightarrow & (\lambda x.(xx))(\lambda x.(xx)) \\ & \rightarrow & \ldots \end{array}$$

- ☐ 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 $I \stackrel{\text{def}}{\equiv} \lambda x.x$ and consider:

$$\begin{array}{ll} & (\lambda x.I)\ \Omega \\ \rightarrow & I \end{array}$$

in a single step. But one could also reduce:

$$\begin{array}{ll} & (\lambda x.I)\ \Omega \\ \rightarrow & (\lambda x.I)\ \Omega \\ \rightarrow & \ldots \end{array}$$

by always doing the $\Omega \rightarrow \Omega$ 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 $\lambda x.E$ even if E was still reducible.
These terms are called *weak head normal forms* or *values*.
They are characterized by the following grammar.

$$\text{Values} \quad V \quad ::= \quad x \mid \lambda x.E$$

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* C is a term where exactly one subterm is replaced by a "hole", written $[\,]$.
$C[E]$ denotes the term which results if the hole of context C is filled with term E.

- ☐ Examples of contexts:

$$[\,] \qquad \lambda x.\lambda y.[\,] \qquad \lambda x.f\ [\,]$$

- ☐ Previously, we have admitted reduction anywhere in a term without explicitly saying so.
- ☐ Let's formalize this:

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

Definition: A term E *reduces* to a term E', written $E \rightarrow E'$ if there exists a context C and terms D, D' such that

$$\begin{array}{lll} E & \equiv & C[D] \\ E' & \equiv & C[D'] \\ D & \rightarrow_\beta & D' \end{array}$$

- So much for general reduction.
- Now, to define an evaluation strategy, we *restrict* the possible set of contexts in the definition of $\rightarrow$.
- The restriction can be expressed by giving a *grammar* which describes permissible contexts.
- Such contexts are called *reduction contexts* and we let the letter R range over them

Call-By-Name

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

$$R \ ::= \ [\,] \mid R\,E$$

Definition: A term E *reduces* to a term E' using the call-by-name strategy, written $E \rightarrow_{\mathsf{cbn}} E'$ if there exists a reduction context R and terms D, D' such that

$$\begin{array}{lcl} E & \equiv & R[D] \\ E' & \equiv & R[D'] \\ D & \rightarrow_\beta & D' \end{array}$$

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 $\rightarrow_{\mathsf{cbn}}$ 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 $K\ I\ \Omega$ with the call-by-name strategy, where

$$\begin{array}{lcl} K & \stackrel{\text{def}}{\equiv} & \lambda x.\lambda y.x \\ I & \stackrel{\text{def}}{\equiv} & \lambda x.x \\ \Omega & \stackrel{\text{def}}{\equiv} & (\lambda x.(xx))(\lambda x.(xx)) \end{array}$$

Theorem: (Standardization) Call-by-name reduction is weakly complete: Whenever $E \twoheadrightarrow V$ then $E \twoheadrightarrow_{\mathsf{cbn}} V'$.
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 R are necessary?

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

$$\begin{array}{ll} & (\lambda f.f(fy))((\lambda x.x)(\lambda x.x)) \\ \rightarrow & (\lambda x.x)(\lambda x.x)((\lambda x.x)(\lambda x.x)y) \\ \rightarrow & (\lambda x.x)((\lambda x.x)(\lambda x.x)y) \\ \rightarrow & (\lambda x.x)((\lambda x.x)y) \\ \rightarrow & (\lambda x.x)y \\ \rightarrow & y \end{array}$$

- Note that the argument $(\lambda x.x)(\lambda x.x)$ is evaluated twice.

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

$$\begin{array}{ll} & (\lambda f.f(fy))((\lambda x.x)(\lambda x.x)) \\ \rightarrow & (\lambda f.f(fy))(\lambda x.x) \\ \rightarrow & (\lambda x.x)((\lambda x.x)y) \\ \rightarrow & (\lambda x.x)y \\ \rightarrow & y \end{array}$$

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:

$$\begin{array}{lll} \text{Terms} & D, E, F & ::= \; V \mid D\ E \\ \text{Values} & V, W & ::= \; x \mid \lambda x.E \end{array}$$

- As reduction rule, we have:

$$\beta_{\mathbf{V}} : \qquad (\lambda x.D)V \;\rightarrow\; [V/x]\, D$$

- As reduction contexts, we have:

$$R_V \;::=\; [\,] \mid R_V\ E \mid V\ R_V$$

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

Theorem: (Plotkin) $\rightarrow_V$ reduction is confluent.

Theorem: (Plotkin) $\rightarrow_{\mathsf{cbv}}$ is weakly complete with respect to $\rightarrow_V$.