Concurrency: Theory, Languages and Programming

– Functions and Data –

Session 3 – Nov 05, 2003

Martin Odersky

EPFL-LAMP

Part I: Data in Scala

- So far, we have been only dealing with predefined (numeric) datatypes and with functions.
- We now explain how to define new kinds of data.
- Example: Scala does not have a type of rational numbers, but it is easy to define one, using a class.


```
class Rational(n: int, d: int) {
  private def gcd(x: int, y: int): int = {
    if (x == 0) y
    else if (x < 0) gcd(-x, y)
    else if (y < 0) -gcd(x, -y)
    else gcd(y % x, x);
  }
  private val g = gcd(n, d);
  val numer: int = n/g;
  val denom: int = d/g;
  def +(that: Rational) = new Rational(
    numer * that.denom + that.numer * denom,
    denom * that.denom);
  ... }
```


Creating and Accessing Objects

- Here's a program that prints the sum of all numbers $1/i$ where i ranges from 1 to 10.

```
var i = 1;
var x = new Rational(0, 1);
while (i ≤ 10) {
  x = x + new Rational(1,i);
  i = i + 1;
}
System.out.println(x.numer + "/" + x.denom);
```

- The + operation converts both its operands to strings and returns the concatenation of the result strings. It thus corresponds to + in Java.

Case Classes

- A *case class* is defined like a normal class, except that the definition is prefixed with the modifier ***case***.
- Example:

  abstract class *Expr;*
  case class *Number(n: int)* ***extends*** *Expr;*
  case class *Sum(e1: Expr, e2: Expr)* ***extends*** *Expr;*

- This introduces *Number* and *Sum* as case classes which extend class *Expr*.

The ***case*** modifier in front of a class definition has the following effects.

- Case classes implicitly come with a constructor function, with the same name as the class. In our example, the following functions are implicitly added:

  def *Number(n: int) =* ***new*** *Number(n);*
  def *Sum(e1: Expr, e2: Expr) =* ***new*** *Sum(e1, e2);*

  Hence, one can now construct expression trees more concisely:

  Sum(Sum(Number(1), Number(2)), Number(3))

- Case classes allow the constructions of *patterns* which refer to the case class constructor.

Pattern Matching

- Pattern matching is a generalization of C or Java's *switch* statement to class hierarchies.
- For instance, here is an implementation of *eval* using pattern matching.

  def *eval(e: Expr): int = e match* {
    case *Number(x)* $\Rightarrow$ *x*
    case *Sum(l, r)* $\Rightarrow$ *eval(l)* $+$ *eval(r)*
  }

- In this example, there are two cases.
- Each case associates a pattern with an expression.
- Patterns are *matched* against the selector values *e*.

- For instance, *Number(n)* matches all values of the form *Number(V)*, for arbitrary values *V*.
- The *pattern variable n* is bound to the value *V*.
- In general, patterns are built from
  - Case class constructors, e.g. *Number*, *Sum*, whose arguments are again patterns,
  - pattern variables, e.g. *n*, *e1*, *e2*,
  - the "wildcard" pattern _,
  - constants, e.g. *1*, *true*, "abc", *MAXINT*.

  Pattern variables always start with a lower-case letter, so that they can be distinguished from constant identifiers, which start with an upper case letter.

Part II: Program Equivalence

- **Question:** When are two lambda terms M and N equivalent, in the following sense:
- Exchanging M by N in a program does not change the behavior of the program
- This notion is called *operational equivalence*, written $M \cong N$.
- It is formalized as follows.

$$M \cong N \quad \text{iff} \quad \forall C. C[M] \Downarrow \quad \Leftrightarrow \quad C[N] \Downarrow .$$

- Here, $M \Downarrow$ means that evaluation of M terminates.
- Formally, $M \Downarrow$ iff $\exists V. M \twoheadrightarrow V$.

Operational Equivalence and β-Conversion

- β-Reduction also gives rise to another program equivalence, called *convertibility*.
- Define: $M =_\beta N$ iff $\exists M'. M \twoheadrightarrow M' \wedge N \twoheadrightarrow M'$.
- Then $=_\beta$ is the smallest congruence that includes reduction $\rightarrow$.
- Also, we have that: $M =_\beta N \quad \Rightarrow \quad M \cong N$
- **Question:** : Name two terms M, N such that $M \cong N$ but not $M =_\beta N$?

Part III: Church Encodings

- The treatment so far covered *pure* lambda calculus which consists of just functions and their applications.
- Actual programming languages add to this primitive data types and their operations, named value and function definitions, and much more.
- We can model these constructs by extending the basic calculus.
- But it is also possible to *encode* these constructs in the basic calculus itself.
- These encodings will be presented in the following.
- We will assume in general call-by-name evaluation, but will also work out modifications needed for call-by-value.

Encoding of Booleans

- An abstract type of booleans is given by the two constants *true* and *false* as well as the conditional ***if***.
- Other constructs can be written in terms of these primitives. E.g.

  not x = ***if*** (*x*) *false* ***else*** *true*
  x || *y* = ***if*** (*x*) *true* ***else*** *y*
  x && *y* = ***if*** (*x*) *y* ***else*** *false*

- **Idea:** : The encoding of a boolean value $B \in \{$*true,false*$\}$ is the binary function

  $\lambda x.\lambda y.$ ***if*** (B) x ***else*** y

That is:

$$\mathit{true} \stackrel{\text{def}}{\equiv} \lambda x.\ \lambda y.\ x$$

$$\mathit{false} \stackrel{\text{def}}{\equiv} \lambda x.\ \lambda y.\ y$$

$$\textbf{\textit{if}}\ c\ x\ y \stackrel{\text{def}}{\equiv} c\ x\ y$$

Example:

$$\begin{aligned}
\textbf{\textit{if}}\ (\mathit{true})\ D\ \textbf{\textit{else}}\ E &\stackrel{\text{def}}{\equiv} \mathit{true}\ D\ E \\
&\stackrel{\text{def}}{\equiv} (\lambda x\,.\lambda y.\ x)\ D\ E \\
&\rightarrow (\lambda y\,.\ D)\ E \\
&\rightarrow D
\end{aligned}$$

Question: What changes to this encoding are necessary if the evaluation strategy is call-by-value?

Encoding of Lists

The encoding of Booleans can be generalized to arbitrary algebraic data types.
Example: Consider the type of lists (as defined in Haskell):

$$\mathit{data\ List\ a} = \mathit{Nil} \mid \mathit{Cons\ a}\ (\mathit{List\ a})$$

This defines a type of lists with (nullary) constructor *Nil* and (curried binary) constructor *Cons*.
A list *xs* can be accessed using a case-expression

$$\textbf{\textit{case}}\ \mathit{xs\ of\ Nil}\ \mathit{Arrow}\ E_1 \mid \mathit{Cons\ x\ xs} \Rightarrow E_2$$

Here, the expression of the second branch, E_2, can refer to the variables *x* and *xs* defined in the *Cons* pattern.

All other functions over lists can be written in terms of the case-expression.
For instance, function *car* which equals *head* except that it avoids errors, can be written as:

$$\begin{aligned}
&\mathit{car\ xs} = \\
&\quad \textbf{\textit{case}}\ \mathit{xs\ of} \\
&\qquad \mathit{Nil} \quad \Rightarrow \mathit{Nil} \\
&\quad \mid\ \mathit{Cons\ y\ ys} \Rightarrow x
\end{aligned}$$

Question: How can lists be encoded?
Same principle as before: Equate a list with the case-expression that accesses it.

$$\mathit{xs} \stackrel{\text{def}}{\equiv} \lambda a.\lambda b.\textbf{\textit{case}}\ \mathit{xs\ of\ Nil} \Rightarrow a \mid \mathit{Cons\ x\ xs} \Rightarrow b\ x\ \mathit{xs}$$

That is:

$$\mathit{Nil} \stackrel{\text{def}}{\equiv} \lambda a.\lambda b.\ a$$

$$\mathit{Cons\ x\ xs} \stackrel{\text{def}}{\equiv} \lambda a.\lambda b.\ b\ x\ \mathit{xs}$$

or, equivalently:

$$\mathit{Cons} \stackrel{\text{def}}{\equiv} \lambda x.\lambda \mathit{xs}.\lambda a.\lambda b.\ b\ x\ \mathit{xs}$$

The pattern-bound names *x* and *xs* are now passed as parameters to the case branch that accesses them.
Example: : *car* is coded as follows:

$$\mathit{car} \stackrel{\text{def}}{\equiv} \lambda \mathit{xs}.\ \mathit{xs\ Nil}\ (\lambda y.\lambda \mathit{ys}.y)$$

Exercise: Church-encode function *isEmpty* which returns true iff the given list is empty.

Encoding of Numbers

The encoding for lists generalizes to arbitrary data types which are defined in terms of a finite number of constructors.
For instance, whole numbers don't present any new difficulties.
To see this, note that natural numbers can be coded as algebraic data types as follows:

$$\textit{data Nat} = \textit{Zero} \mid \textit{Succ Nat}$$

Hence:

$$\begin{aligned} \textit{Zero} &\stackrel{\text{def}}{\equiv} \lambda a.\lambda b.a \\ \textit{Succ}\ x &\stackrel{\text{def}}{\equiv} \lambda a.\lambda b.b\ x \end{aligned}$$

Note: Church encodings do not reflect types. In fact *Zero*, *Nil*, and *true* are all mapped to the same term!

Encoding of Definitions

A non-recursive value definition ***val*** *x = D ; E* can be encoded as:

$$\textbf{val}\ x = D\ ;\ E \stackrel{\text{def}}{\equiv} (\lambda x.E)\ D$$

Caveat: With a call-by-name strategy, *D* might be evaluated more than once.
Let's try an analogous principle for function definitions:

$$\begin{aligned} \textbf{def}\ f\ x = D\ ;\ E &\stackrel{\text{def}}{\equiv} \textbf{val}\ f = \lambda x.D\ ;\ E \\ &\stackrel{\text{def}}{\equiv} (\lambda f.E)\ (\lambda x.D) \end{aligned}$$

But this fails if *f* is used recursively in *D*! (Why?)

Fixed Points to the Rescue

If we have a recursive definition of

$$\textbf{val}\ f = E$$

where *E* refers to *f*, we can interpret this as a solution to the equation

$$f = E$$

Another way to characterize solutions to this equation is to say that these solutions are fixed points of the function $\lambda f.E$.

Definition: A *fixed point* of a function f is a value x such that

$$f\ x = x$$

Proposition: The solutions of $f = E$ are exactly the fixed points of $\lambda f.E$
Proof: F is a solution of the equation

$$f = E$$

iff

$$F = [F/f]E$$

iff

$$F = (\lambda f.E)\ F$$

iff F is a fixed point of $\lambda f.E$.

Fixed Point Operators

Let's assume the existence of a *fixed point operator* Y. For every function f, $Y\,f$ evaluates to a fixed point of f. That is,

$$Y\,f \quad = \quad f\,(Y\,f)$$

Then we can encode potentially recursive definitions as follows:

$$\begin{aligned} \textbf{\textit{def}}\ f\ x = D\ ;\ E &\stackrel{\text{def}}{\equiv} \textbf{\textit{val}}\ f = Y\ (\lambda f.\lambda x.D)\ ;\ E \\ &\stackrel{\text{def}}{\equiv} (\lambda f.E)\ (Y\ (\lambda f.\lambda x.D)) \end{aligned}$$

Remains the question whether Y exists.

Proposition: Let

$$Y \stackrel{\text{def}}{\equiv} \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

Then Y is a fixed point operator:

$$Y\,f \quad = \quad f\,(Y\,f)$$

Proof: By repeated β-reduction.

Least Fixed Points

In fact, an equation will in general have several solutions, and a function will in general have several fixed points.

Example: The equation $f \;=\; f$
has every λ-term as a solution.
Can we characterize the fixed point computed by Y?

Proposition: Among all the fixed points of a function f, $Y\,f$ will return the one which diverges most often. This is also called the *least fixed point* of the function f.

Exercise: Find the least fixed point of $\lambda f.f$ (which is also the least solution of the equation $f \;=\; f$).

Connection to Domain Theory

- The definition of least fixed points is made precise in the field of *domain theory*.
- Domain theory gives λ-terms meaning by mapping them to mathematical functions.
- Divergent terms are modeled by a value $\perp$, which stands for "undefined".
- Domain theory introduces a partial ordering on values which makes $\perp$ smaller than any defined value.
- The fixed points computed by Y are the smallest with respect to this ordering.

Summary

- We have seen the basic theory of λ-calculus, and how it can express functional programming.
- Two main variants: Call-by-value and call-by-name.
- In each case, evaluation is described by reduction of function applications, using rule β (or β_V).
- λ-calculus has two important properties, which make it well suited as a basis of deterministic programming languages:
  - **Confluence:** Every term can be reduced to at most one value.
  - **Standardization:** There exists a deterministic reduction strategy which always reduces a term to a value, provided it can be done at all.

Outlook

- λ-calculus is ideally suited as a basis for functional programming.
- But it is less well suited as basis for imperative programming with side effects (essentially, need to introduce and carry along a data structure describing global state).
- It is not suitable at all as a basis for reactive systems with concurrent evaluation.
- Two new issues:
  - **Non-determinism:** If programs can have several behaviors, confluence no longer holds.
  - **Non-termination:** Operational equivalence needs to be adapted for programs that do not terminate.