# Concurrency: Theory, Languages and Programming

# – Encoding FP in Lambda Calculus –

# Session 3 – Nov 05, 2002

Martin Odersky

EPFL-LAMP

# Program Equivalence

**Question:** When are two lambda terms $M$ and equivalent, in the following sense:

Exchanging $M$ by in a program does not change the behavior of the program

This notion is called *operational equivalence*, written $M$ .

It is formalized as follows.

$$M \qquad \text{iff} \qquad M$$

Here, $M$ means that evaluation of $M$ terminates.

Formally, $M$ iff $M$ .

# perational Equivalence and $\beta$-Conversio

-Reduction also gives rise to another program equivalence, called *convertibility*.

Define: $M$ iff $M$ $M$ $M$ $M$

Then is the smallest congruence that includes reduction .

Also, we have that: $M$ $M$

**Question:** : Name two terms $M$ such that $M$ but not $M$ ?

# 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.

$$\begin{aligned} \textit{not}\ x &= \textbf{\textit{if}}\ (x)\ \textit{false}\ \textbf{\textit{else}}\ \textit{true} \\ x \quad\ y &= \textbf{\textit{if}}\ (x)\ \textit{true}\ \textbf{\textit{else}}\ y \\ x\ \&\&\ y &= \textbf{\textit{if}}\ (x)\ y\ \textbf{\textit{else}}\ \textit{false} \end{aligned}$$

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

$$x.\quad y.\ \textbf{\textit{if}}\ (B)\ x\ \textbf{\textit{else}}\ y$$

## That is:

$$\mathit{true} \qquad x.\ \ y.\ x$$

$$\mathit{false} \qquad x.\ \ y.\ y$$

$$\textbf{\textit{if}}\ c\ x\ y \qquad c\ x\ y$$

**Example:**

$$\textbf{\textit{if}}\ (\mathit{true})\ D\ \textbf{\textit{else}}\ E \qquad \mathit{true}\ D\ E$$

$$(\ \ x\ .\ \ y.\ x)\ D\ E$$

$$(\ \ y\ .\ D)\ E$$

$$D$$

**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):

$$\textit{data List a} = \textit{Nil} \quad \textit{Cons a} \; (\textit{List a})$$

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

$$\textbf{case}\ \textit{xs of Nil} \qquad\qquad E \qquad \textit{Cons x xs} \qquad E$$

Here, the expression of the second branch, $E$ , 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:

```
car xs =
  case xs of
    Nil            Nil
    Cons y ys      x
```

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

*xs*        *a.   b.***case** *xs of Nil*     *a*     *Cons x xs*     *b x xs*

That is:

| | |
|---|---|
| *Nil* | *a.   b. a* |
| *Cons x xs* | *a.   b. b x xs* |

or, equivalently:

| | |
|---|---|
| *Cons* | *x.   xs.   a.   b. b x 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:

| | |
|---|---|
| *car* | *xs. xs Nil* (  *y.   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:

*data Nat* = *Zero*    *Succ Nat*

Hence:

| | |
|---|---|
| *Zero* | *a*.  *b*.*a* |
| *Succ x* | *a*.  *b*.*b x* |

**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:

$$\textbf{val}\ x = D\ ;\ E \qquad\qquad (\quad x.E\,)\ D$$

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

$$\textbf{def}\ f\ x = D\ ;\ E \qquad\qquad \textbf{val}\ f = \quad x.D\ ;\ E$$

$$(\quad f.E\,)\ (\quad x.D\,)$$

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

# Fixed Points to the Rescue

If we have a recursive definition of

***val*** *f* = *E*

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

Another way to characterize solutions to this equation is to say that these solutions are fixed points of the function .

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

**Proposition:** The solutions of are exactly the fixed points of

**Proof:** is a solution of the equation

iff

iff

iff is a fixed point of .

# Fixed Point Operators

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

Then we can encode potentially recursive definitions as follows:

$$\textbf{def}\ f\ x = D\ ;\ E \qquad \textbf{val}\ f = Y\ (\ f.\ \ x.D\ )\ ;\ E$$

$$(\ \ f.E\ )\ (Y\ (\ \ f.\ \ x.D\ )\ )$$

Remains the question whether  exists.

**Proposition:** Let

Then is a fixed point operator:

**Proof:** By repeated -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
has every -term as a solution.
Can we characterize the fixed point computed by ?

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

**Exercise:** Find the least fixed point of (which is also the least solution of the equation ).

# Connection to Domain Theory

The definition of least fixed points is made precise in the field of *domain theory*.

Domain theory gives -terms meaning by mapping them to mathematical functions.

Divergent terms are modeled by a value , which stands for “undefined”.

Domain theory introduces a partial ordering on values which makes smaller than any defined value.

The fixed points computed by are the smallest with respect to this ordering.

# Summary

We have seen the basic theory of -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 (or ).

-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

-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.