Concurrency: Theory, Languages and Programming

– Encoding FP in Lambda Calculus –

Session 3 – Nov 05, 2002

Martin Odersky

EPFL-LAMP

- **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-Conversio

- 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: $M =_\beta N \Rightarrow M \cong N$
- **Question:** : Name two terms M, N such that $M \cong N$ but not $M =_\beta N$?

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.

- 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:

$$\begin{aligned} true &\stackrel{\text{def}}{\equiv} \lambda x.\, \lambda y.\, x \\ false &\stackrel{\text{def}}{\equiv} \lambda x.\, \lambda y.\, y \\ \textbf{if}\ c\ x\ y &\stackrel{\text{def}}{\equiv} c\ x\ y \end{aligned}$$

Example:

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

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):

data List a = Nil | *Cons a* (*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

case *xs of Nil* $Arrow$ E_1 | *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:

car xs =
  case *xs of*
    Nil $\Rightarrow$ *Nil*
  | *Cons y ys* $\Rightarrow$ *x*

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

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

That is:

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

or, equivalently:

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

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:

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

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

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:

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

Hence:

$$\mathit{Zero} \stackrel{\text{def}}{\equiv} \lambda a.\lambda b.a$$
$$\mathit{Succ}\ x \stackrel{\text{def}}{\equiv} \lambda a.\lambda b.b\ x$$

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:

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

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

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

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

Fixed Points to the Rescue

If we have a recursive definition of

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

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 $\lambda f.E$.

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

$$Yf \;=\; f\,(Yf)$$

Then we can encode potentially recursive definitions as follows:

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

$\stackrel{\text{def}}{\equiv}$ $(\lambda f.E)\;(Y\;(\lambda f.\lambda x.D))$

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:

$$Yf \;=\; f\,(Yf)$$

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, Yf 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 $f \;=\; f$).

- 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 $\bot$, which stands for "undefined".
- Domain theory introduces a partial ordering on values which makes $\bot$ smaller than any defined value.
- The fixed points computed by Y are the smallest with respect to this ordering.

- 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.