Programming language design and analysis

Lambda Calculus

Marius Minea

6 October 2011

Course references:
Principles of Programming Languages, Uday Reddy, Univ. of Birmingham
Program Analysis and Understanding, Jeff Foster, Univ. of Maryland

Background. Church-Turing thesis

Lambda calculus: developed in 1930's by Alonzo Church
initially typed, then untyped fragment

Formalizing *computability*:
Lambda calculus [Church]
Turing machines [1936–37]
general recursive functions [Church, Kleene, Rosser]

Church-Turing thesis: these three computational processes are equivalent, i.e., the class of *computable* functions (by recursion or λ-calculus) are precisely the *effectively calculable* ones (by a Turing machine).

$\Rightarrow$ Lambda calculus is a *universal model of computation*.

Syntax

$$e ::= x \qquad \text{variable}$$
$$\mid \ \lambda x.e \qquad \text{function abstraction (definition)}$$
$$\mid \ e_1 \ e_2 \qquad \text{function application}$$

Basic ideas:

functions are values (no split b/w functions and args/results)
functions need not be named (λ-abstractions suffice)
functions are all one needs (can express numbers, if-then, etc.)

Syntax conventions:

the scope of the abstraction . extends as far right as possible
application is left-associative, $e_1 \ e_2 \ e_3$ means $(e_1 \ e_2) \ e_3$

Free and bound variables

The function abstraction $\lambda x.e$ *binds* the occurrence of x in e
intuitively: inside e, x is the argument; outside e it has no meaning

$FV(x) = \{x\}$
$FV(\lambda x.e) = FV(e) \setminus \{x\}$
$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$

A term is *closed* if it has no free variables.

A variable that is not free is called *bound*.

Substitutions

To correctly compute with λ expressions, we need to define substitutions.

Denote by $e_1[x \to e_2]$ the substitution of x by e_2 in e_1
(various other notations: $e_1[x := e_2], e_1[x/e_2], e_1[e_2/x]$)

Define:

$$y[x \to e] = \begin{cases} e & \text{if } y \text{ is the same as } x \\ y & \text{if } y \text{ is different from } x \end{cases}$$

$(\lambda y.e_1)[x \to e_2] =$

$$\begin{cases} \lambda y.e_1 & \text{if } y \text{ is the same as } x \\ \lambda y.(e_1[x \to e_2]) & \text{if } y \text{ is different from } x \text{ and } y \notin FV(e_2) \end{cases}$$

(otherwise occurrences of y in e_2 would be captured by $\lambda y.e_1$)

$(e_1\ e_2)[x \to e] = (e_1[x \to e])(e_2[x \to e])$

Capture-avoiding substitution

α-conversion (bound variables can be renamed)

$\lambda x.e = \lambda y.(e[x \to y]$ if $y \notin FV(e)$

Then we can substitute $\lambda y.e_1[x \to e_2]$ also when $y \in FV(e_2)$:

first rename y to some fresh variable z: $\lambda y.e_1 = \lambda z.e_1[y \to z]$

then substitute x with e_1: $\lambda z.e_1[y \to z][x \to e_1]$

Reductions: Computing with lambda expressions

β-conversion (or β-eduction)

$$(\lambda x.e_1)\ e_2 = e_1[x \rightarrow e_2]$$

is the *evaluation* step for lambda expressions. We write:

$$(\lambda x.e_1)\ e_2 \longrightarrow_{\beta} e_1[x \rightarrow e_2]$$

η-conversion: simplifies application + abstraction

$\lambda x.e\ x = e$ if $x \notin FV(e)$

Equivalence and Confluence

Two terms are *equivalent* if one can be converted to each other by the three conversion rules.

A λ-expressions may have several β-reducible subexpressions (*redexes*)
$\Rightarrow$ which one to apply first ?

Church-Rosser theorem: if a term reduces to two different terms, these in turn reduce to a common term (diamond property).

$$e \longrightarrow_\beta^* e_1 \wedge e \longrightarrow_\beta^* e_2 \Rightarrow \ \exists e' \ . \ e_1 \longrightarrow_\beta^* e' \wedge e_2 \longrightarrow_\beta^* e'$$

Reduction strategies

normal-order reduction

leftmost outermost redex first

also reduces under λ

if any reduction terminates, then normal order terminates

call-by-name

leftmost outermost redex first

does not reduce under λ

applicative order reduction (call by value)

only reduce $(\lambda x.e_1)\ e_2$ when argument e_2 is value

In programming language practice: *lazy* evaluation: only reduce argument if needed, but do not duplicate expressions (evaluate at most once)