COMS W4115
Programming Languages and Translators
Lecture 23: April 22, 2013
The Lambda Calculus

Outline

- Reduction orders
- The Church-Rosser theorems
- The Y combinator
- Implementing factorial using the Y combinator
- Church numerals
- Arithmetic
- Logic
- Other programming language constructs
- The influence of the lambda calculus on functional languages

1. Reduction Orders

- Programming languages use many different techniques to pass parameters to procedures such as call by value, call by reference, call by value-return, call by name, and so on. We can model many of these parameter-passing mechanisms using the lambda calculus.
- The order in which reductions are applied within a lambda expression can affect the final result. We will use the terms reduction order and evaluation order synonymously.
- A reducible expression (redex) in a lambda expression is a subexpression that can be reduced using beta reduction.
- An expression that contains no redexes is said to be in normal form.
- Some reduction orders for an expression may yield a normal form expression while other orders may not. For example, consider the expression
  `(Î»x.1)((Î»x.x x)(Î»x.x x))`
- This expression has two redexes:

1. The entire expression is a redex in which we can apply the function `(Î»x.1)` to the argument `((Î»x.x x)(Î»x.x x))` to yield the value 1.
2. The rightmost subexpression `((Î»x.x x)(Î»x.x x))` is also a redex in which we can apply the function `(Î»x.x x)` to the argument `(Î»x.x x)`. But if we do this reduction we get same subexpression: `(Î»x.x x)(Î»x.x x)` â†' `(Î»x.x x)(Î»x.x x)`. Thus, continuing this order of evaluation will not terminate in a normal form.

- A remarkable property of lambda calculus is that every lambda expression has a unique normal form if one exists.
- The expression `(Î»x.1)((Î»x.x x)(Î»x.x x))` has the normal form 1.
- The expression `(Î»x.x x)(Î»x.x x)` does not have a normal form because it always evaluates to itself. We can think of this expression as a representation for an infinite loop.
- There are two common reduction orders for lambda expressions: normal order evaluation and applicative order evaluation.
- Normal order evaluation
- In normal order evaluation we always reduce the leftmost outermost redex at each step.
- The first reduction order above is a normal order evaluation.
- If an expression has a normal form, then normal order evaluation will always find it.
- Applicative order evaluation
- In applicative order evaluation we always reduce the leftmost innermost redex at each step.
- The second reduction order above is an applicative order evaluation.
- Thus, even though an expression may have a normal form, applicative order evaluation may fail to find it.

2. The Church-Rosser Theorems

- A remarkable property of lambda calculus is that every expression has a unique normal form if one exists.
- **Church-Rosser Theorem I:** If `e â†'* f` and `e â†'* g` by any two reduction orders, then there always exists a lambda expression h such that `f â†'*` h and `g â†'*` h.
  - A corollary of this theorem is that no lambda expression can be reduced to two distinct normal forms. To see this, suppose `f` and `g` are in normal form. The Church-Rosser theorem says there must be an expression h such that `f` and `g` are each reducible to h. Since `f` and `g` are in normal form, they cannot have any redexes so `f = g =` h.
  - This corollary says that all reduction sequences that terminate will always yield the same result and that result must be a normal form.
  - The term *confluent* is often applied to a rewriting system that has the Church-Rosser property.
- **Church-Rosser Theorem II:** If `e â†'* f` and `f` is in normal form, then there exists a normal order reduction sequence from `e` to `f`.

3. The Y Combinator

- The Y combinator (sometimes called the paradoxical combinator) is a function that takes a function G as an argument and returns G(YG). With repeated applications we can get G(G(YG)), G(G(G(YG))),... .
- We can implement recursive functions using the Y combinator.
- Y is defined as follows:

  ```
  (Î»f.(Î»x.f(x x))(Î»x.f(x x)))
  ```

- Let us evaluate YG where G is any expression:

  ```
  (Î»f.(Î»x.f(x x))(Î»x'.f(x' x'))) G
  â†’ (Î»x.G(x x))(Î»x'.G(x' x'))
  â†’ G((Î»x'.G(x' x'))(Î»x'.G(x' x')))
  â†” G((Î»f.(Î»x.f(x x))(Î»x.f(x x)))G)
  = G(YG)
  ```

- Thus, YG â†’* G(YG); that is, YG reduces to a call of G on (YG).
- We will use Y to implement recursive functions.
- Y is an example of a fixed-point combinator.

4. Implementing Factorial using the Y Combinator

- If we could name lambda abstractions, we could define the factorial function with the following recursive definition:

  ```
  FAC = (Î»n.IF (= n 0) 1 (* n (FAC (- n 1 ))))
  ```

  where IF is a conditional function.
- However, functions in lambda calculus cannot be named; they are anonymous.
- But we can express recursion as the fixed-point of a function G. To do this, let us simplify the essence of the problem. We begin with a skeletal recursive definition:

  ```
  FAC = Î»n.(... FAC ...)
  ```

- By performing beta abstraction on FAC, we can transform its definition to:

  ```
  FAC = (Î»f.(Î»n.(... f ...))) FAC
     = G FAC
  ```

  where

  ```
  G = Î»f.Î»n.IF (= n 0) 1 (* n (f (- n 1 )))
  ```

  Beta abstraction is just the reverse of beta reduction.
- The equation

  ```
  FAC = G FAC
  ```

  says that when the function G is applied to FAC, the result is FAC. That is, FAC is a fixed-point of G.
- We can use the Y combinator to implement FAC:

  ```
  FAC = Y G
  ```

- As an example, let compute FAC 1:

  ```
  FAC 1 = Y G 1
       = G (Y G) 1
       = Î»f.Î»n.IF (= n 0) 1 (* n (f (- n 1 ))))(Y G) 1
       â†’ Î»n.IF (= n 0) 1 (* n ((Y G) (- n 1 ))))1
       â†’ IF (= n 0) 1 (* n ((Y G) (- 1 1 )))
       â†’ * 1 (Y G 0)
       = * 1 (G(Y G) 0)
       = * 1((Î»f.Î»n.IF (= n 0) 1 (* n (f (- n 1 ))))(Y G 0)
       â†’ * 1((Î»n.IF (= n 0) 1 (* n ((Y G) (- n 1 ))))0
       â†’ * 1(IF (= 0 0) 1 (* 0 ((Y G) (- 0 1 )))
       â†’ * 1 1
       â†’ 1
  ```

5. Church Numerals

- Church numberals are a way of representing the integers in lambda calculus.
- Church numerals are defined as functions taking two parameters:

  0 is defined as Î»f.Î»x. x

  1 is defined as Î»f.Î»x. f x

  2 is defined as Î»f.Î»x. f (f x).

  3 is defined as Î»f.Î»x. f (f (f x)).

  n is defined as Î»f.Î»x. f^n x
- n has the property that for any lambda expressions g and y, ngy â†’* $g^n y$. That is to say, ngy causes g to be applied to y *n* times.

6. Arithmetic

- In lambda calculus, arithmetic functions can be represented by corresponding operations on Church numerals.
- We can define a successor function `succ` of three arguments that adds one to its first argument:

  ```
  Î»n.Î»f.Î»x. f (n f x)
  ```

  - Example: Let us evaluate `succ 0`:

  ```
  (Î»n.Î»f.Î»x. f (n f x)) 0
    â†’ Î»f.Î»x. f (0 f x)
    = Î»f.Î»x. f ((Î»f.Î»x. x) f x)
    â†’ Î»f.Î»x. f (Î»x. x) x)
    â†’ Î»f.Î»x. f x
    = 1
  ```

- We can define a function `add` using the identity $f^{(m+n)} = f^m \circ f^n$ as follows:

  ```
  Î»m.Î»n.Î»f.Î»x. m f (n f x)
  ```

  - Example: Let us evaluate `add 1 2`:

  ```
  Î»m.Î»n.Î»f.Î»x. m f (n f x) 1 2
    â†’  Î»n.Î»f.Î»x. 1 f (n f x) 2
    â†’* Î»f.Î»x. f (f (f x))
    =  3
  ```

7. Logic

- The boolean value true can be represented by a function of two arguments that always selects its first argument: `Î»x.Î»y.x`
- The boolean value false can be represented by a function of two arguments that always selects its second argument: `Î»x.Î»y.y`
- An if-then-else statement can be represented by a function of three arguments `Î»c.Î»i.Î»e. c i e` that uses its condition `c` to select either the if-part `i` or the else-part `e`.
  - Example: Let us evaluate if true then 1 else 2:

  ```
  (Î»c.Î»i.Î»e. c i e) true 1 2
    â†’ (Î»i.Î»e. true i e) 1 2
    â†’ (Î»e. true 1 e) 2
    â†’ true 1 2
    = (Î»x.Î»y.x) 1 2
    â†’ (Î»y.1) 2
    â†’ 1
  ```

- The boolean operators and, or, and not can be implemented as follows:

  ```
  and = Î»p.Î»q. p q p
  or  = Î»p.Î»q. p p q
  not = Î»p.Î»a.Î»b. p b a
  ```

  - Example: Let us evaluate `not true`:

  ```
  (Î»p.Î»a.Î»b. p b a) true
    â†’ Î»a.Î»b. true b a
    = Î»a.Î»b. (Î»x.Î»y.x) b a
    â†’ Î»a.Î»b. (Î»y.b) a
    â†’ Î»a.Î»b. b
    = false (under renaming)
  ```

8. Other Programming Language Constructs

- We can readily implement other programming language constructs in lambda calculus. As an example, here are lambda calculus expressions for various list operations such as cons (constructing a list), head (selecting the first item from a list), and tail (selecting the remainder of a list after the first item):

  ```
  cons = Î»h.Î»t.Î»f. f h t
  head = Î»l.l (Î»h.Î»t. h)
  tail = Î»l.l (Î»h.Î»t. t)
  ```

9. The Influence of The Lambda Calculus on Functional Languages

- Our next lecture will be by Maria Taku who will talk about the influence of the lambda calculus on functional languages and her experiences implementing the PLT compiler project using OCaml.

10. Practice Problems

1. Evaluate `(Î»x.((Î»w.Î»z. + w z)1)) ((Î»x. xx)(Î»x. xx))) ((Î»y. * y 1) (- 3 2))` using normal order evaluation and applicative order evaluation.
2. Give an example of a code optimization transformation that has the Church-Rosser property.
3. Evaluate `FAC 2`.
4. Evaluate `succ two`.

5. Evaluate `add two three`.
6. Let `mul` be the function

   ```
   Î»m.Î»n.Î»f.Î»x. m (n f x)
   ```

Evaluate `mul two three`.

- Write a lambda expression for the boolean predicate `isZero` and evaluate `isZero one`.

11. References

- Simon Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.
- Stephen A. Edwards: The Lambda Calculus
- http://www.inf.fu-berlin.de/lehre/WS01/ALPI/lambda.pdf
- http://www.soe.ucsc.edu/classes/cmps112/Spring03/readings/lambdacalculus/project3.html

---

aho@cs.columbia.edu