COMS W4115
Programming Languages and Translators
Lecture 22: April 17, 2013
Introduction to Lambda Calculus

Outline

- The evolution of programming languages
- Programming paradigms
- History of lambda calculus and functional programming languages
- CFG for lambda calculus
- Function abstraction
- Function application
- Free and bound variables
- Beta reductions
- Evaluating a lambda expression
- Currying
- Renaming bound variables by alpha reduction
- Eta conversion
- Substitutions
- Disambiguating lambda expressions
- Normal form
- Evaluation strategies

1. The Evolution of Programming Languages

- There are thousands of programming languages in the world today. Why so many?
  - Software is used in virtually every domain of human endeavor. Each application domain has its distinct and often conflicting programming needs. No one language is ideal for all application domains.
    - Scientific programming demands support for matrices and efficient floating point computations. Fortran, first developed in the 1950s, is still widely used for scientific computation.
    - Systems programming demands low-level control of machine resources, often with real-time constraints. C and C++ are the dominant systems programming languages.
    - Business applications are characterized by data persistence and report generation. SQL is the dominant query language in business data processing.
- Why are there so many new programming languages?
  - Software reliability and programmer productivity are critical concerns.
  - Programmer training is a significant cost, so old languages tend to persist.
  - But new languages can arise to meet the demands of new application domains. For example, Java was designed in the 1990s to meet the demands of Internet programming. But to gain acceptance, Java was designed to look a lot like C++.
- What makes a good programming language?
  - This is a hotly debated question. There is no universally accepted metric for the goodness of a programming language. Technical excellence is rarely a criterion for adoption.
  - Functional programming languages have many ardent advocates and many features of functional languages have been incorporated into modern languages. The next sequence of lectures will discuss the theoretical underpinnings of functional languages and highlight a few important functional languages.

2. Programming Paradigms

- A programming paradigm is a style of computer programming.
- There are many paradigms, each with its distinctive characteristics and prototypical programming languages. The four most common are:
  - imperative also known as procedural programming
    - imperative programming tends to be the most popular paradigm
    - an imperative program tells the computer *how* to do a computation
    - the typical imperative languages are C and Fortran
  - object-oriented programming
    - an object-oriented program consists of a collection of interacting objects
    - C++, Java, and Smalltalk are prototypical object-oriented languages
  - functional programming
    - a functional program is typified by the recursive definition of functions
    - Haskell, Lisp, Scheme, ML, and OCaml are popular functional languages

    - declarative programming
        - a declarative program specifies *what* should be computed
        - SQL and Prolog are declarative languages
- Some programming languages are multi-paradigm languages in the sense that they can support more than one programming paradigm. For example, C#, OCaml, Ruby, and Scala support both imperative, object-oriented, and functional programming.

3. History of Lambda Calculus and Functional Programming Languages

- Lambda calculus was introduced in the 1930s by Alonzo Church at Princeton University as a mathematical system for defining computable functions.
- Lambda calculus is equivalent in definitional power to that of Turing machines.
- Lambda calculus serves as the theoretical model for functional programming languages and has applications to artificial intelligence, proof systems, and logic.
- Lisp was developed by John McCarthy at MIT in 1958 around lambda calculus.
- ML, a general-purpose functional language, was developed by Robin Milner at the University of Edinburgh in the early 1970s. Caml and OCaml are dialects of ML developed at INRIA in 1985 and 1996, respectively.
- Haskell, considered by many as one of the purest functional programming languages, was developed by Simon Peyton Jones, Paul Houdak, Phil Wadler and others in the late 1980s and early 90s.
- Because of its simplicity, lambda calculus is a very useful tool for the study and analysis of programming languages.

4. CFG for The Lambda Calculus

- The central concept in lambda calculus is an expression which we can think of as a program that when evaluated returns a result consisting of another lambda calculus expression.
- Here is the grammar for lambda expressions:

```
expr â†’ Î» variable . expr | expr expr | variable | ( expr ) | built_in
```

- A `variable` is an identifier.
- A `built_in` is a built-in function such as addition or multiplication, or a constant such as an integer or boolean. However, as we shall see, all programming language construct can be implemented as functions with the pure lambda calculus so these built-ins are unnecessary. However, we will use built-ins for notational simplicity.

5. Function Abstraction

- A function abstraction, often called a lambda abstraction, is a lambda expression that defines a function.
- A function abstraction consists of four parts: a lambda followed by a variable, a period, and then an expression as in `Î»x.expr`.
- In the function abstraction `Î»x.expr` the variable `x` is the formal parameter of the function and `expr` is the body of the function.
- For example, the function abstraction Î»*x*. + *x* 1 is a function of *x* that adds *x* to 1. Parentheses can be added to lambda expressions for clarity. Thus, we could have written this function abstraction as Î»*x*.(+ *x* 1) or even as (Î»*x*. (+ *x* 1)).
- In C this function definition might be written as

```
int addOne (int x)
{
 return (x + 1);
}
```

- Note that unlike C the lambda abstraction does not give a name to the function. The lambda expression itself is the function.
- We say that `Î»x.expr` *binds* the variable `x` in `expr` and that `expr` is the *scope* of the variable.

6. Function Application

- A function application, often called a lambda application, consists of an expression followed by an expression: `expr expr`. The first expression is a function abstraction and the second expression is an argument to which the function is applied.
- For example, the lambda expression Î»*x*. (+ *x* 1) 2 is an application of the function Î»*x*. (+ *x* 1) to the argument 2.
- This function application Î»*x*. (+ *x* 1) 2 can be evaluated by substituting the argument 2 for the formal parameter *x* in the body (+ *x* 1). Doing this we get (+ 2 1). This substitution is called a beta reduction.
- Beta reductions are like macro substitutions in C. To do beta reductions correctly we may need to rename bound variables in lambda expressions to avoid name clashes.
- Functions can be used as arguments to functions and functions can return functions as results.

7. Free and Bound Variables

- In the function definition Î»*x*.*x* the variable *x* in the body of the definition (the second *x*) is *bound* because its first occurrence in the definition is Î»*x*.

- A variable that is not bound in `expr` is said to be *free* in `expr`. In the function (Î»*x*.*xy*), the variable *x* in the body of the function is bound and the variable *y* is free.
- Every variable in a lambda expression is either bound or free. Bound and free variables have quite a different status in functions.
- In the expression (Î»*x*.*x*)(Î»*y*.*yx*):
  - The variable *x* in the body of the leftmost expression is bound to the first lambda.
  - The variable *y* in the body of the second expression is bound to the second lambda.
  - The variable *x* in the body of the second expression is free.
  - Note that *x* in second expression is independent of the *x* in the first expression.
- In the expression (Î»*x*.*xy*)(Î»*y*.*y*):
  - The variable *y* in the body of the leftmost expression is free.
  - The variable *y* in the body of the second expression is bound to the second lambda.
- Given an expression *e*, the following rules define FV(*e*), the set of free variables in *e*:
  - If *e* is a variable *x*, then FV(*e*) = {*x*}.
  - If *e* is of the form Î»*x*.*y*, then FV(*e*) = FV(*y*) - {*x*}.
  - If *e* is of the form *xy*, then FV(*e*) = FV(*x*) âˆª FV(*y*).
- An expression with no free variables is said to be *closed*.

8. Beta Reductions

- A function application `Î»x.e f` is evaluated by substituting the argument `f` for the formal parameter `x` in the body `e` of the function definition.
- We will use the notation `[f/x]e` to indicate that `f` is to be substituted for all free occurrences of `x` in the expression `e`.
- Examples:

1. (Î»*x*.*x*)*y* â†' [*y*/*x*]*x* = *y*.
2. (Î»*x*.*xzx*)*y* â†' [*y*/*x*]*xzx* = *yzy*.
3. (Î»*x*.*z*)*y* â†' [*y*/*x*]*z* = *z*.

- This substitution in a function application is called a *beta reduction* and we use a right arrow to indicate it.
- If expr1 â†' expr2, we say expr1 *reduces* to expr2 in one step.
- In general, `(Î»x.e)f â†' [f/x]e` means that applying the function `(Î»x.e)` to the argument expression `f` reduces to the expression `[f/x]e` where the argument expression `f` is substituted for the function's formal parameter `x` in the function body `e`.
- A lambda calculus expression (aka a "program") is "run" by computing a final result by repeatly applying beta reductions. We use â†'* to denote the reflexive and transitive closure of â†'; that is, zero or more applications of beta reductions.
- Examples:
- (Î»*x*.*x*)*y* â†' *y* (illustrating that Î»*x*.*x* is the identity function).
- (Î»*x*.*xx*)(Î»*y*.*y*) â†' (Î»*y*.*y*)(Î»*y*.*y*) â†' (Î»*y*.*y*); thus, we can write (Î»*x*.*xx*)(Î»*y*.*y*) â†'* (Î»*y*.*y*). Note that here we have applied a function to a function as an argument and the result is a function.

9. Evaluating a Lambda Expression

- A lambda calculus expression can be thought of as a program which can be executed by evaluating it. Evaluation is done by repeatedly finding a reducible expression (called a redex) and reducing it by a function evaluation until there are no more redexes.
- Example 1: The lambda expression (Î»*x*.*x*)*y* in its entirety is a redex that reduces to *y*.
- Example 2: The lambda expression (+ (* 1 2) (- 4 3)) has two redexes: (* 1 2) and (- 4 3). If we choose to reduce the first redex, we get (+ 2 (- 4 3)). We can then reduce (+ 2 (- 4 3)) to get (+ 2 1). Finally we can reduce (+ 2 1) to get 3.
- Note that if we had chosen the second redex to revaluate first, we would have ended up with the same result:

  (+ (* 1 2) (- 4 3)) â†' (+ (* 1 2) 1) â†' (+ 2 1) â†' 3.

10. Currying

- All functions in lambda calculus are prefix and take exactly one argument.
- If we want to apply a function to more than one argument, we can use a technique called *currying* that treats a function applied to more than one argument to a sequence of applications of one-argument functions. For example, to express the sum of 1 and 2 we can write (+ 1 2) as ((+ 1) 2) where the expression (+ 1) denotes the function that adds 1 to its argument. Thus ((+ 1) 2) means the function + is applied to the argument 1 and the result is a function (+ 1) that adds 1 to its argument: (+ 1 2) = ((+ 1) 2) â†' 3
- Note that function application associates to the left.

11. Renaming Bound Variables by Alpha Reduction

- The name of a formal parameter in a function definition is arbitrary. We can use any variable to name a parameter, so that the function Î»*x*.*x* is equivalent to Î»*y*.*y* and Î»*z*.*z*. This kind of renaming is called *alpha reduction*.
- Note that we cannot rename free variables in expressions.
- Also note that we cannot change the name of a bound variable in an expression to conflict with the name of a free variable in that expression.

12. Eta Conversion

- The two lambda expressions (Î»*x*.+ 1 *x*) and (+ 1) are equivalent in the sense that these expressions behave in exactly the same way when they are applied to an argument -- they add 1 to it. Î·-conversion is a rule that expresses this equivalence.
- In general, if *x* does not occur free in the function *F*, then (Î»*x*.*F x*) is Î·-convertible to *F*.

13. Substitutions

- For a beta reduction, we introduced the notation `[f/x]e` to indicate that the expression `f` is to be substituted for all free occurrences of the formal parameter `x` in the expression `e`:

  ```
  (Î»x.e) f â†’ [f/x]e
  ```

- To avoid name clashes in a substitution `[f/x]e`, first rename the bound variables in `e` and `f` so they become distinct. Then perform the textual substituion of `f` for `x` in `e`.
  - For example, consider the substitution `[y(Î»x.x)/x] Î»y.(Î»x.x)yx`.
  - After renaming all the bound variables to make them all distinct we get `[y(Î»u.u)/x] Î»v.(Î»w.w)vx`.
  - Then doing the substitution we get `Î»v.(Î»w.w)v(y(Î»u.u))`.
- The rules for substitution are as follows. We assume `x` and `y` are distinct variables, and `e`, `f` and `g` are expressions.
  - For variables

    ```
    [e/x]x = e
    [e/x]y = y
    ```

  - For function applications

    ```
    [e/x](f g) = ([e/x]f) ([e/x]g)
    ```

  - For function abstractions

    ```
    [e/x](Î»x.f) = Î»x.f
    ```

    `[e/x](Î»y.f) = Î»y.[e/x]f`, provided `y` is not free in `e` (the "freshness" condition).
- Examples:

1. `[y/y](Î»x.x) = Î»x.x`
2. `[y/x]((Î»x.y) x) = ([y/x](Î»x.y)) ([y/x]x) = (Î»x.y)y`
3. Note that the freshness condition does not allow us to make the substitution `[y/x](Î»y.x) = Î»y.([y/x]x) = Î»y.y` because `y` is free in the expression `y`.

14. Disambiguating Lambda Expressions

- The grammar we gave in section 4 for lambda expressions is ambgious. A few simple rules will remove the ambiguities.
- Function application is left associative: *f g h* = ((*f g*) *h*)
- Function application binds more tightly than lambda: Î»*x*.*f g x* = (Î»*x*.(*f g*) *x*)
- The body in a function abstraction extends as far to the right as possible: Î»*x*. + *x* 1 = Î»*x*. (+ *x* 1).

15. Normal Form

- An expression containing no possible beta reductions is said to be in normal form. A normal form expression is one containing no redexes.
- Examples of normal form expressions:
  - `x` where `x` is a variable.
  - `x e` where `x` is a variable and `e` is a normal form expression.
  - `Î»x.e` where `x` is a variable and `e` is a normal form expression.
- 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.

16. Evaluation Strategies

- An expression may contain more than one redex so there can be several reduction sequences. For example, the expression `(+ (* 1 2) (- 4 3))` can be reduced to normal form with two reduction sequences:

  ```
  (+ (* 1 2) (- 4 3))
  â†’ (+ 2 (- 4 3))
  â†’ (+ 2 1)
  â†’ 3
  ```

  or the sequence

  ```
  (+ (* 1 2) (- 4 3))
  â†’ (+ (* 1 2) 1)
  â†’ (+ 2 1)
  â†’ 3
  ```

- As we pointed out in (15), the expression `(Î»x.x x)(Î»x.x x)` does not have a terminating sequence of reductions.
- Some reduction sequences for a given expression may reach a normal form while others may not. For example, consider the expression `(Î»x.1)((Î»x.x x)(Î»x.x x))`. If we reduce the application of `(Î»x.1)` to `(Î»x.x x)(Î»x.x x)`, we get the result 1, but if we keep reducing the application of `(Î»x.x x)` to `(Î»x.x x)`, the evaluation will not terminate.
- We shall see that repeatedly reducing the leftmost outermost redex will always yield a normal form, if a normal form exists.

17. Practice Problems

1. Evaluate `(Î»x. Î»y. + x y)1 2`.
2. Evaluate `(Î»f. f 2)(Î»x. + x 1)`.
3. Evaluate `(Î»x. (Î»x. + (* 1)) x 2) 3`.
4. Evaluate `(Î»x. Î»y. + x((Î»x. * x 1) y))2 3`.
5. Is `(Î»x. + x x)` Î·-convertible to `(+ x)`?
6. Show how all bound variables in a lambda expression can be given distinct names.
7. Construct an unambiguous grammar for lambda calculus.

18. 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