Lambda Calculus

Theoretical Foundations of Functional Programming

Raj Sunderraman
Computer Science Department
Georgia State University
Functions

- A computer program can be considered as a **function** from input values to output values. What does it mean for a function to be **computable**? The following 3 models are equivalent!

  - **Alonzo Church** defined **Lambda Calculus** in the 1930s to answer this question. He claimed that a function is computable if and only if it can be written as a λ-term.

  - **Alan Turing** devised **Turing machines** as a mechanism to define computability. He claimed that a function is computable if and only if it can be computed using a Turing machine.

  - **Kurt Gödel** introduced **Recursive Function Theory** to define computability. He claimed that a function is computable if and only if it is general recursive.
Lambda Calculus

• With its simple syntax and semantics, Lambda Calculus is an excellent vehicle to study the meaning of programming languages.

• All functional programming languages (Haskel, LISP, Scheme, etc) are syntactic variations of the Lambda Calculus; so their semantics can be discussed in the context of Lambda Calculus.

• Denotational Semantics, an important method for the formal specification of programming languages, grew out of Lambda Calculus.
Three Observations About Functions

1. Functions need not be named
   \[ x \mapsto x \times x \]

2. The choice of name for the function parameter is irrelevant
   \[ x \mapsto x \times x \]
   \[ y \mapsto y \times y \]
   both are the same function (both return the square of their inputs)

3. Functions may be rewritten to have exactly one parameter
   \[(x,y) \mapsto x+y\]
   may be written as
   \[ x \mapsto (y \mapsto x+y) \]
Concepts and Examples

Consider the function:

\[
\text{cube}: \text{Integer} \rightarrow \text{Integer}
\]

where \(\text{cube}(n) = n^3\)

What is the value of the identifier “cube”? How can we represent the object bound to “cube”? Can we define this function without giving it a name? like a literal?

In Lambda Calculus, such a function would be represented by the expression:

\[\lambda n. n^3\]

This is an anonymous function (function literal) mapping its input \(n\) to \(n^3\)
Concepts and Examples

Consider another function:

\[ f : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \]

where \( f(m,n) = n^2 + m \)

Lambda Calculus allows functions to have exactly one parameter

\( f \) would be represented by the expression:

\[ \lambda m. \lambda n. (n^2 + m) \]

This is an anonymous function (function literal) mapping its input \((m,n)\) to \((n^2 + m)\)
by “currying”: \( m \Rightarrow (n \Rightarrow n^2 + m) \)
Lambda Calculus Syntax

A \lambda-term is defined inductively as follows:
1. A variable is a \lambda-term (e.g. x, y, m, n, etc)
2. If M is a \lambda-term and x is a variable, then (\lambda x. M) is a \lambda-term
3. If M and N are \lambda-terms then (M N) is a \lambda-term

In the above definition,
(\lambda x. M) is called a lambda abstraction; or in programming terminology the definition of a function. Here x is the input parameter (bound variable) and M is the body of the function.
(M N) is called a function application; or in programming terminology a function call. M is called the rator and N is called the rand (operator, operand)
Lambda Calculus Syntax continued

We introduce two other types of \( \lambda \)-terms:

4. A number is a \( \lambda \)-term (e.g. 10, 2, -5, 6.5, etc)

5. If \( M \) and \( N \) are \( \lambda \)-terms then \((\text{op} \ M \ N)\) is a \( \lambda \)-term, where \( \text{op} \) is +, -, *, or /

These two are not part of the original “pure” Lambda Calculus.

Well-formed \( \lambda \)-terms:

\( x \)

\( 5 \)

\( (\lambda x. x) \)

\( (\lambda x. (* x x)) \)

\( ((\lambda x. (* x x)) \ 5) \)
Parentheses; Lots of them!

\[(\lambda x.(\lambda y.(\lambda z.((x z)(y z))))))\]

Let us see how this is constructed from the definition:

- \(x, y, z\) are \(\lambda\)-terms using rule 1
- \((x z)\) is a \(\lambda\)-term using rule 3
- \((y z)\) is a \(\lambda\)-term using rule 3
- \(((x z)(y z))\) is a \(\lambda\)-term using rule 3
- \((\lambda z.((xz)(y z)))\) is a \(\lambda\)-term using rule 2
- \((\lambda y.(\lambda z.((x z)(y z))))\) is a \(\lambda\)-term using rule 2
- \((\lambda x.(\lambda y.(\lambda z.((x z)(y z))))))\) is a \(\lambda\)-term using rule 2
Expression Trees

Variable

Number

(op M N)

(M N)

(λx.M)
Expression tree for

\((\lambda x. (\lambda y. (\lambda z. ((x z) (y z))) )))\)
Conventions for omitting parentheses

1. Omit outermost parentheses. For example \((\lambda x.x)\) can be written as \(\lambda x.x\)

2. Function applications are left-associative; So, omit parentheses when not necessary. For example \((M N) P\) can be written as \(M N P\)

3. Body of function abstractions extend as far right as possible. So, we can write \(\lambda x.(MN)\) as \(\lambda x.MN\)

Using the above conventions, \((\lambda x.(\lambda y.(\lambda z.((x z)(y z))))))\) can be written as \(\lambda x.\lambda y.\lambda z.x z (y z)\)
Lambda Calculus Interpreter (PLY Specification)

expr :
   NUMBER
| NAME
| LPAREN expr expr RPAREN
| LPAREN LAMBDA NAME expr RPAREN
| LPAREN OP expr expr RPAREN

NUMBER = r’[0-9]+ | [0-9]+"."[0-9]* | "."[0-9]*
LPAREN = r’(‘
RPAREN = r’)’
OP = r’+|−|*|/’
LAMBDA = r’[Ll][Aa][Mm][Bb][Dd][Aa]’
NAME = r’[a-zA-Z][a-zA-Z0-9]’
Lambda Calculus Interpreter continued

(\lambda x. x) is written as (lambda x x)
(\lambda x. (* x x)) is written as (lambda x (* x x))
((x y)(x z)) is written as ((x y)(x z))

The two syntactic differences are that
- the “.” after \lambda x is left out
- \lambda is spelt out as lambda
Lambda Calculus Semantics

What is the meaning (semantics, or value) of $\lambda$-terms?

e.g. what is the meaning of $((\lambda x.(\ast x x)) \, 5)$?

Informally, it looks like we are calling the function $(\lambda x.(\ast x x))$ with the argument $5$. The function should return $(\ast \, 5 \, 5) = 25$

Before we formally define the semantics of $\lambda$-terms, we need a few definitions.
- Free and Bound Variables
- $\alpha$-equivalence
- Substitutions
- $\beta$-reductions
Free and Bound Variables

In the $\lambda$-term $(\lambda x. M)$
- $x$ is a bound variable
- $\lambda$ is said to bind $x$ in $M$
- Any occurrence of $x$ in $M$ is said to be bound in $(\lambda x. M)$
- This concept is not novel! We have seen this in CSC 2510/Math 2420 in Predicate Calculus; e.g. in $\exists x \ P(x)$, $x$ in $P(x)$ is bound to the $x$ next to $\exists$.
- Also seen in programming languages such as Python in a formal parameter of a function (the occurrence of $x$ in the function body is bound to the parameter $x$)
  ```python
def f(x):
    return x*x
```
Free and Bound Variables - Examples

(1) In the $\lambda$-term, $\lambda x. \; x \; y$

- $x$ next to $\lambda$ is bound
- $x$ in the body of the $\lambda$-term is bound to the $x$ next to $\lambda$
- $y$ in the body of the $\lambda$-term is free

(2) In the $\lambda$-term, $(\lambda x. \; x \; y)(\lambda y. \; z \; y)$

\[ \begin{array}{cccc}
    & b & b & f \\
\end{array} \]

\[ \begin{array}{cccc}
    b & f & b \end{array} \]

The variable next to $\lambda$ is always bound!

(3) In the $\lambda$-term, $(\lambda x. (\lambda x.x) \; x)$, the $x$ in the body of the inner $\lambda$-term is bound to the $x$ of that $\lambda$-term and the last $x$ is bound to the $x$ of the outer $\lambda$-term.
Free Variable Definition

FV(M), the set of free variables in M is inductively defined as follows:

1. FV[x] = \{ x \}
2. FV[\lambda x.M] = FV[M] - \{ x \}
3. FV[MN] = FV[M] \cup FV[N]
4. FV[number] = \{ \}
5. FV[(op M N)] = FV[M] \cup FV[N]
Free Variables Example

\[ \text{FV[}\lambda x.\lambda y.((\lambda z.\lambda v.z(v))(xy)(zu)))] \]

\[ = \text{FV[}((\lambda z.\lambda v.z(v))(xy)(zu))\text{]} - \{ x, y \} \]

\[ = (\text{FV[}(\lambda z.\lambda v.z(v))\text{]} \cup \text{FV[}(xy)\text{]} \cup \text{FV[}(zu)\text{]}\} - \{ x, y \} \]

\[ = (\text{FV[}(\lambda z.\lambda v.z(v))\text{]} \cup \{ x, y \} \cup \{ z, u \}) - \{ x, y \} \]

\[ = ((\text{FV[}z(v)\text{]} - \{ z, v \}) \cup \{ x, y, z, u \}) - \{ x, y \} \]

\[ = \{ x, y, z, u \} - \{ x, y \} \]

\[ = \{ z, u \} \]
\(\alpha\)-equivalence

(\(\lambda x.x\)) is the same as (\(\lambda y.y\))

(\(\lambda x.(\ast x x)\)) is the same as (\(\lambda u.(\ast u u)\))

All we have done is change the parameter name (\textit{bound variable}) next to the \(\lambda\) as well as in the body of the function.

Renaming the bound variable does not change the abstraction.

Formally,

\[
(\lambda x. M) =_\alpha (\lambda y. M\{x \leftarrow y\})
\]

where \(y\) is a “brand new” variable not appearing in \(M\), and \(M\{x \leftarrow y\}\) is \(M\) with all occurrences of \(x\) replaced by \(y\).
The same idea is present in programming languages as well. We do this often, i.e. we name a parameter of a function one way and after some time decide to give it a better name. To do this we consistently change all references to the old name with the new name!

e.g.

```python
def isPrime(n):
    for i in range(1, n):
        if n % i == 0:
            return False
    return True
def isPrime(num):
    for i in range(1, num):
        if num % i == 0:
            return False
    return True
```

\(\alpha\)-equivalence continued
Substitution

• Substitution is defined for **free** variables
• We will substitute a **free variable** with a \( \lambda \)-term.
• Substitution will be used during a “function call” when we provide an actual parameter value for the formal parameter
• For example, when we call the isPrime function with the actual argument 17, i.e. isPrime(17), the formal parameter \( n \) would have to be substituted by 17 in the body of the function:

```python
def isPrime(n):
    for i in range(1, n):
        if n % i == 0:
            return False
    return True
```

```python
for i in range(1, 17):
    if 17 % i == 0:
        return False
return True
```
Substitution

\((\lambda x. (x \ y)) [y = 5] \ = \ (\lambda x. (x \ 5))\)
\((\lambda x. (x \ y)) [y = (u \ v)] \ = \ (\lambda x. (x \ (u \ v)))\)

Substitution must be done carefully so as not to alter the meaning of the \(\lambda\)-term!

\((\lambda x. (x \ y)) [y = x] \neq (\lambda x. (x \ x))\)

As can be seen, \(y\) was a free-variable before, but after the substitution \(y\)’s value has become bound! Such a case is called a “capture” case.

\((\lambda x. (x \ y)) [y = x] =_{\alpha} (\lambda x'. (x' \ y)) [y = x] = (\lambda x'. (x' \ x))\)

Another “capture” example:

\((\lambda x. (y \ x)) [y = (\lambda z. (x \ z))] \neq (\lambda x. ((\lambda z. (x \ z)) \ x))\)
\((\lambda x. (y \ x)) [y = (\lambda z. (x \ z))] =_{\alpha} (\lambda x'. (y \ x')) [y = (\lambda z. (x \ z))] = (\lambda x'. ((\lambda z. (x \ z)) \ x'))\)
Substitution Definition

1. \( x[x = P] = P \)
2. \( y[x = P] = y \) if \( x \neq y \)
3. \( (M N)[x = P] = (M[x = P] N[x = P]) \)
4. \( (\lambda x. M)[x = P] = (\lambda x. M) \)
5. \( (\lambda y. M)[x = P] = (\lambda y. M[x = P]) \) if \( x \neq y \) and \( y \notin FV[P] \)
6. \( (\lambda y. M)[x = P] = (\lambda y'. (M\{y\leftarrow y'\}[x = P])) \) if \( x \neq y \) and \( y \in FV[P] \) and \( y' \) is brand new

Case 6 is the “capture” case! Bound variable \( y \) is “renamed” to \( y' \) using \( \alpha \)-equivalence and then the substitution is applied.
Substitution Example

\[(\lambda y. (((\lambda x. x) y) x)) [x = (y (\lambda x. x))]\]
\[= \]
\[(\lambda y'. (((\lambda x. x) y') x)) [x = (y (\lambda x. x))]\]
\[= \]
\[(\lambda y'. (((\lambda x. x)y'[x = (y (\lambda x. x))]) x[x = (y (\lambda x. x))]))\]
\[= \]
\[(\lambda y'. (((\lambda x. x)y') (y (\lambda x. x))))\]
Consider the λ-term, (λx. (* x x)), that denotes the “square” function.
To call this function with argument 5, we will construct the “apply” λ-term:
( (λx. (* x x)) 5)

β-reduction allows us to “execute” this function call. We “substitute” the bound variable (parameter), x, of the function abstraction with 5 in the body of the function abstraction.

( (λx. (* x x)) 5) = β (* x x) [x = 5] = (* 5 5) = 25

β-reduction can be applied **only** to a λ-term of the form ((λx.M) N)

**Note:** The formal definition of substitution does not have rules for the impure λ-terms which involve arithmetic operators; but the definition can be easily extended.
\[ (\lambda x. M) N = \beta M[x = N] \]

A \textbf{\(\beta\)-redex} is of the form \((\lambda x. M) N\)

The result of \(\beta\)-reduction is called a \textbf{reduct}.

To “execute” a \(\lambda\)-term, \(\beta\)-reduction is applied repeatedly until there are no more \(\beta\)-redexes to be found in the \(\lambda\)-term.

A \(\lambda\)-term without any \(\beta\)-redexes is said to be in \textbf{\(\beta\)-normal-form}.
β-reduction Examples

\[(\lambda x. y) (\lambda z. (z \ z)) \] = \[\beta\] \[y[x = (\lambda z. (z \ z))] = y\]

\[(\lambda w. w) \ (\lambda w. w) \] = \[\beta\] \[w[w = (\lambda w. w)] = (\lambda w. w)\]

\[(\lambda x. y) ((\lambda z. (z \ z)) \ (\lambda w. w))\] = \[\beta\] \[(\lambda x. y) ((z \ z)[z = (\lambda w. w)])\]

\[(\lambda x. y) ((\lambda w. w) \ (\lambda w. w))\] = \[\beta\] \[(\lambda x. y) (w[w = (\lambda w. w)])\]

\[(\lambda x. y) (\lambda w. w)\] = \[\beta\] \[(y[x = (\lambda w. w)])\]

\[= y\]

The order of applying β-reductions is not significant. The end result is the same, especially if it terminates.
\( ((\lambda x.y) (\lambda z.(z z))) = \beta y[x = (\lambda z.(z z))] = y \)
\[
((\lambda w.w) \ (\lambda w.w)) = \beta \ w[w = (\lambda w.w)] = (\lambda w.w)
\]
\( \beta \)-reduction Examples using Expression Trees

\[
((\lambda x. y) ((\lambda z. (z z)) (\lambda w. w))) \quad \xrightarrow{\beta} \quad (\lambda x. y) (\lambda w. w) \quad \Rightarrow \quad y
\]
Using the Lambda Calculus Interpreter Notation:

\[ ((\text{lambda } x \ (\ast x x)) \ 2) \]

\[ \Rightarrow_{\beta} \]

\[ (\ast 2 2) \]

\[ = \]

\[ 4 \]
\[ \lambda f \lambda x (f (f x)) \lambda y (* y (* y y)) 2 \]
\[ = \beta \]
\[ ((\lambda x ((\lambda y (* y (* y y))) ((\lambda y (* y (* y y))) x))) 2) \]
\[ = \beta \]
\[ ((\lambda y (* y (* y y))) ((\lambda y (* y (* y y))) 2)) \]
\[ = \beta \]
\[ ((\lambda y (* y (* y y))) (* 2 (* 2 2)) = ((\lambda y (* y (* y y))) 8) \]
\[ = \beta \]
\[ (* 8 (* 8 8)) = 512 \]
\[
((\lambda \, ((\lambda f \, (\lambda x \, (f \, (f \, x)))) \, (\lambda y \, (* \, (* \, y \, y)))) \, 2)
\]

\[
\beta
\]

\[
\rightarrow
\]

\[
((\lambda y \, (* \, (* \, y \, y))) \, ((\lambda y \, (* \, (* \, y \, y))) \, 2))
\]

\[
\rightarrow
\]

\[
((\lambda x \, ((\lambda y \, (* \, (* \, y \, y))) \, ((\lambda y \, (* \, (* \, y \, y))) \, x))) \, 2)
\]

\[
\beta
\]
\[ ((\text{lambda } y \ (\ast \ y \ (\ast \ y \ y))) \ (\text{lambda } y \ (\ast \ y \ (\ast \ y \ y))) \ 2) \]

\[ \beta \]

\[ ((\text{lambda } y \ (\ast \ y \ (\ast \ y \ y))) \ (\text{lambda } y \ (\ast \ y \ (\ast \ y \ y))) \ 2) \]

\[ \beta \]

\[ ((\text{lambda } y \ (\ast \ y \ (\ast \ y \ y))) \ (\text{lambda } y \ (\ast \ y \ (\ast \ y \ y))) \ 2) \]

\[ (\ast \ 8 \ (\ast \ 8 \ 8)) \]

\[ 512 \]
Try this out!

$$(((\lambda x (\lambda y (\lambda z (* (x z)(y z)))) (\lambda x (* x)))) (\lambda x (+ x x))) 5)$$

see if you can evaluate this to 250?