CS 314 Principles of Programming Languages

Lecture 19: Lambda Calculus

Zheng (Eddy) Zhang

Rutgers University

April 11, 2018
• The **imperative** and **functional** models grew out of work undertaken by Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, and etc in 1930s.

- Different formalizations of the notion of an algorithm, or “effective procedure”, based on **automata**, **symbolic manipulation**, **recursive function definitions**, and **combinatorics**.

• These results led Church to conjecture that:

“Any intuitively appealing model of computing would be equally powerful as well.”

— Church’s thesis
• Turing’s model of computing was the *Turing machine* a sort of pushdown automaton using an unbounded storage “tape”

The Turing machine computes in an imperative way, by changing the values in cells of its tape – like variables just as a high level imperative program computes by changing the values of variables.
• Church’s model of computing is called the lambda calculus

It is based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter $\lambda$ — hence the notation’s name). Lambda calculus was the inspiration for functional programming: one uses it to compute by *substituting parameters into expressions*, just as one computes in a high level functional program by *passing arguments to functions*. 
Functional Programming

- Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are an attempt to realize Church's lambda calculus in practical form as a programming language.

- **The key idea: do everything by composing functions**
  - No mutable state
  - No side effects
  - Function as first-class values
\textbf{Lambda Calculus}

\textbf{\(\lambda\)-terms} are inductively defined.

A \textbf{\(\lambda\)-term} is:

\begin{itemize}
  \item a variable \(x\)
  \item \((\lambda x. M) \Rightarrow\) where \(x\) is a variable and \(\lambda\) is a \(\lambda\)-term (abstraction)
  \item \((M N) \Rightarrow\) where \(M\) and \(N\) are both \(\lambda\)-terms (application)
\end{itemize}
The context-free grammar for \(\lambda\)-terms:

\[
\begin{align*}
\lambda\text{-term} & \rightarrow \text{expr} \\
\text{expo} & \rightarrow \text{name} \mid \text{number} \mid \lambda \text{name} . \text{expr} \mid \text{func} \ \text{arg} \\
\text{func} & \rightarrow \text{name} \mid ( \lambda \text{name} . \text{expr} ) \mid \text{func} \ \text{arg} \\
\text{arg} & \rightarrow \text{name} \mid \text{number} \mid ( \lambda \text{name} . \text{expr} ) \mid ( \text{func} \ \text{arg} )
\end{align*}
\]

\[\lambda y . \ y + y\]

name (as parameter) \hspace{1em} \text{expr (another } \lambda\text{-term)}
The context-free grammar for λ-terms:

\[
\begin{align*}
\lambda\text{-term} & \rightarrow \text{expr} \\
\text{expo} & \rightarrow \text{name} \mid \text{number} \mid \lambda \text{name} . \text{expr} \mid \text{func arg} \\
\text{func} & \rightarrow \text{name} \mid ( \lambda \text{name} . \text{expr} ) \mid \text{func arg} \\
\text{arg} & \rightarrow \text{name} \mid \text{number} \mid ( \lambda \text{name} . \text{expr} ) \mid ( \text{func arg} )
\end{align*}
\]

\[
( \text{y} \mid \text{z} )
\]

\[
\text{func} \quad \text{arg}
\]
Associativity and Precedence

• Function application is left associative: \((f \ g \ z)\) is \(((f \ g) \ z)\)

• Function application has precedence over function abstraction. “function body” extends as far to the right as possible:
  \((\lambda x. yz)\) is \((\lambda x.(yz))\)

• Multiple arguments: \((\lambda xy. z)\) is \((\lambda x(\lambda y.z))\)
Abstraction ($\lambda x. M$) “binds” variable $x$ in “body” $M$. You can think of this as a declaration of variable $x$ with scope $M$. 

$$(\lambda y . y z) y$$
Free and Bound Variables

Let M, N be λ-terms and x is a variable. The set of free variable of M, free(M), is defined inductively as follows:

- free(x) = \{x\}
- free(M N) = free(M) ∪ free(N)
- free (\text{λ}x.M) = free(M) - free(x)
Free and Bound Variables

Note:
A variable can occur **free** and **bound** in a λ-term.

Example:

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

- **y** is free
- **y** is bound

“free” is relative to a λ-sub-term.
Function Application as Substitution

The result of applying an abstraction ($\lambda x. M$) to an argument $M$ is formalized by a special form of textual substitution.

$$(\lambda x. M) \ N \quad \equiv \quad [N/x]M$$

Informally, $N$ replaces all free occurrences of $x$ in $M$. 
Function Application as Substitution

The result of applying an abstraction \((\lambda x. M)\) to an argument \(M\) is formalized by a special form of textual substitution.

\[
(\lambda x. M) N \equiv [N/x]M
\]

Informally, \(N\) replaces all free occurrences of \(x\) in \(M\).

Example:

\[
(\lambda a. \lambda b. a+b)2 \ x \equiv (\lambda b.2+b)x \\
\equiv 2 + x
\]
Computation in the lambda calculus is based on the concept or reduction (rewriting rules). The goal is to “simplify” an expression until it can no longer be further simplified.

\[
(\lambda x. M) N \quad \Rightarrow_\beta \quad [N/x] M \quad (\beta\text{–reduction})
\]

\[
(\lambda x. M) \quad \Rightarrow_\alpha \quad \lambda y. [y/x] M \quad (\alpha\text{–reduction}), \text{ if } y \notin \text{free}(M)
\]

Note:

• An equivalence relation can be defined based on \(\equiv\)-convertible \(\lambda\)-terms. “Reduction” rules really work both ways, but we are interested in reducing the complexity of \(\lambda\)-term (forward direction)

• \(\alpha\)-reduction does not reduce the complexity of \(\lambda\)-term

• \(\beta\)-reduction: corresponds to application, models computation
Reduction

• A subterm of the form \((\lambda x.M)N\) is called a \textit{redex} (reduction expression)
• A reduction is any sequence of \textit{\(\alpha\)-reductions} and \textit{\(\beta\)-reductions}
• A term that cannot be \textit{\(\beta\)-reduced} are said to be in \textit{\(\beta\) normal form}
• A subterm that is an abstraction or a variable is said to be in \textit{head normal form}.

Question: Does a normal form always exist?

Example:

\[((\lambda x.xx)) (\lambda x.xx))\]
Remember: Computation in the lambda calculus is a sequence of applications of reduction rules (mostly $\beta$–reductions).

Logical constants and operations (incomplete list):

\[
\begin{align*}
\text{true} &\equiv \lambda a. \lambda b. a \\
\text{false} &\equiv \lambda a. \lambda b. b
\end{align*}
\]

\[
\text{cond} \equiv \lambda m. \lambda n. \lambda p. (p \ m \ n)
\]

\[
\begin{array}{c}
\text{if p is true} \\
\text{return m}
\end{array} \quad 
\begin{array}{c}
\text{cond} \ p \ m \ n \\
\Rightarrow p \ m \ n \\
\Rightarrow \lambda a. \lambda b. a \ m \ n \\
\Rightarrow m
\end{array} \quad 
\begin{array}{c}
\text{if p is false} \\
\text{return n}
\end{array} \quad 
\begin{array}{c}
\text{cond} \ p \ m \ n \\
\Rightarrow p \ m \ n \\
\Rightarrow \lambda a. \lambda b. b \ m \ n \\
\Rightarrow n
\end{array}
\]
Programming in Lambda Calculus

Remember: Computation in the lambda calculus is a sequence of applications of reduction rules (mostly $\beta$–reductions).

Logical constants and operations (incomplete list):

- **true** $\equiv \lambda a. \lambda b. a$
- **false** $\equiv \lambda a. \lambda b. b$

- **not** $\equiv \lambda x. (x \text{ false true})$

  - if x is **true**
    - return **false**
    - $\equiv \lambda x. (x \text{ false true}) x$
    - $\equiv x \text{ false true}$
    - $\equiv \lambda a.\lambda b.a \text{ false true}$
    - $\equiv \text{ false}$

  - if x is **false**
    - return **true**
    - $\equiv \lambda x. (x \text{ false true}) x$
    - $\equiv x \text{ false true}$
    - $\equiv \lambda a.\lambda b.b \text{ false true}$
    - $\equiv \text{ true}$
Programming in Lambda Calculus

Remember: Computation in the lambda calculus is a sequence of applications of reduction rules (mostly \(\beta\)-reductions).

Logical constants and operations (incomplete list):

\[
\text{true} \equiv \lambda a. \lambda b. a
\]
\[
\text{false} \equiv \lambda a. \lambda b. b
\]

\[
\text{and} \equiv \lambda x. \lambda y. (x \ y \ \text{false})
\]

\[
\begin{align*}
\text{if } x \text{ is } \text{true} & \quad \text{and } x \ y \\
& \equiv x \ y \ \text{false} \\
& \equiv \lambda a. \lambda b. a \ y \ \text{false} \\
& \equiv y
\end{align*}
\]

\[
\begin{align*}
\text{if } x \text{ is } \text{false} & \quad \text{and } x \ y \\
& \equiv x \ y \ \text{false} \\
& \equiv \lambda a. \lambda b. b \ y \ \text{false} \\
& \equiv \text{false}
\end{align*}
\]
Programming in Lambda Calculus

Remember: Computation in the lambda calculus is a sequence of applications of reduction rules (mostly $\beta$–reductions).

Logical constants and operations (incomplete list):

\[
\text{true} \equiv \lambda a. \lambda b. a
\]
\[
\text{false} \equiv \lambda a. \lambda b. b
\]
\[
\text{cond} \equiv \lambda m. \lambda n. \lambda p. (p \ m \ n)
\]
\[
\text{not} \quad \equiv \lambda x. (x \ \text{false} \ \text{true})
\]
\[
\text{and} \quad \equiv \lambda x. \lambda y. (x \ y \ \text{false})
\]
\[
\text{or} \quad \equiv \text{homework}
\]
What about data structures?

**Data structures:**

**pairs** can be represented as:

\[
[M.N] \equiv \lambda z. (z M N)
\]

**first** \(\equiv\) \(\lambda x. (x \text{ true})\)  
**second** \(\equiv\) \(\lambda x. (x \text{ false})\)  
**build** \(\equiv\) \(\lambda x.\lambda y.\lambda z. (z x y)\)
What about the encoding of arithmetic constants?

**Church Numerals:**

0 ≡ \( \lambda f x. \ x \)

1 ≡ \( \lambda f x. \ (f \ x) \)

2 ≡ \( \lambda f x. \ (f \ (f \ x)) \)

...  

\( n \equiv \lambda f x. (f (f (\ldots (f x) \ldots))) \equiv \lambda f x. (f^n x) \)

The natural number \( n \) is represented as a function that applies a function \( f \) \( n \)-times to \( x \).

\begin{align*}
\text{succ} & \equiv \lambda m. \ (\lambda f x. (f (m f x))) \\
\text{add} & \equiv \lambda mn. \ (\lambda f x. ((m f) (n f x))) \\
\text{mult} & \equiv \lambda mn. \ (\lambda f x. ((m (n f)) x)) \\
\text{isZero?} & \equiv \lambda m. \ (m \ \text{false} \ \text{not} \ \text{false})
\end{align*}
Reading:

- Scott, Chapter 11.7 (4th Edition supplementary chapters)