Types and Programming Languages

Lecture 3. Untyped λ-calculus

Xiaojuan Cai

cxj@sjtu.edu.cn

BASICS Lab, Shanghai Jiao Tong University

Fall, 2016
The history of $\lambda$-calculus

- The $\lambda$-calculus was invented by Alonzo Church in 1920s.
- In 1960s, Peter Landin observed that a complex programming language can be understood by formulating it as a tiny core calculus — $\lambda$-calculus.
- Landin’s work and John McCarthy’s Lisp make $\lambda$-calculus the most widespread specifications of PL features, in both design and implementation.
Other important calculi

- **π-calculus** by Milner, Parrow, and Walker, for defining semantics of message-based concurrent languages.
- **Object calculus** by Abadi and Cardelli, for catching features of object-oriented languages.
Outline

Basics

Formalities

Programming in $\lambda$-calculus

Nameless representation of terms
Syntax

- In arithmetic expression, there is no function.
- In $\lambda$-calculus, everything is a function.

**Syntax.**

$$t ::= \text{terms}$$

- $x$ variable
- $\lambda x.t$ abstraction
- $t \ t$ application

$$v ::= \text{values}$$

- $\lambda x.t$ functionvalue
Abstract and concrete syntax

- The **concrete syntax** refers to the strings of characters. It is the input of a *lexical analyzer*.
- The **abstract syntax** is an internal representation of programs as labeled trees, also called abstract syntax trees. It is the output of a *parser*.

We focus on **abstract syntax**.

Two conventions of $\lambda$-terms:

1. $stu$ stands for $(st)u$. (left associative)
2. $\lambda x.\lambda y.s$ stands for $\lambda x.(\lambda y.s)$.

**Quiz.** Please draw the syntax tree of $(\lambda x.\lambda y.x y x)x$. 

Scope

- $x$ is **bound** if it occurs in the body $t$ of an abstraction $\lambda x.t$.
- $x$ is **free** if it is not bound.

$$(\lambda x. \lambda y. x \ y \ x) \ x$$

The third $x$ is free, and the first two occurrence of $x$ are bound.

- A term is **closed** if it has no free variables. A closed term is also called **combinators**.

$$id = \lambda x. x$$
(λx.s) t is called a redex (“reducible expression”).

\[ (\lambda x.s) t \rightarrow [x \mapsto t] s \]

- **Full \(\beta\)-reduction**: any redex may be reduced at any time.
- **Normal order strategy**: the leftmost, outermost redex is reduced first.
- **Call by name**: leftmost, outermost redex is reduced first, and no redex inside abstractions is allowed to reduce.
- **Call by value**: outermost redexes are reduced and only its argument part has already been reduced to a value.
Examples

Quiz. Find all the redex of this term:

\[ id (id (\lambda z. id z)) \]

1. \[ id (id (\lambda z. id z)) \]
2. \[ id (id (\lambda z. id z)) \]
3. \[ id (id (\lambda z. id z)) \]

\[ \text{▶ Full } \beta \text{-reduction allows all these redexes.} \]

\[ id (id (\lambda z. id z)) \rightarrow id (id (\lambda z.z)) \]

\[ \text{▶ Normal order strategy allows the first one.} \]

\[ id (id (\lambda z. id z)) \rightarrow (id (\lambda z. id z)) \]
\[ \rightarrow \lambda z.id z \rightarrow \lambda z.z \]

\[ \text{▶ Call-by-name is more restrictive than normal order} \]

\[ id (id (\lambda z. id z)) \rightarrow (id (\lambda z. id z)) \]
\[ \rightarrow \lambda z.id z \rightarrow \lambda z.z \]

\[ \text{▶ Call by value requires the right-hand side to be a value.} \]

\[ id (id (\lambda z. id z)) \rightarrow id (\lambda z.id z) \]
\[ \rightarrow \lambda z.id z \]
Which strategy?

- Full $\beta$-reduction is nondeterministic.
- Call-by-value strategy is used by most languages.
- Call-by-name is sometimes called lazy strategy.
- Haskell uses an optimized version of call-by-name, and called call-by-need.
Outline

Basics

Formalities

Programming in $\lambda$-calculus

Nameless representation of terms
Terms and variables

Terms. Let $V$ be a set of variables. The set of terms is the smallest set $T$ such that:

- $x \in T$ for every $x \in V$;
- if $t_1 \in T$ and $x \in V$, then $\lambda x . t_1 \in T$;
- if $t_1, t_2 \in T$, then $t_1 \ t_2 \in T$.

Free variables.

\[
FV(x) = \{x\} \\
FV(\lambda x . t_1) = FV(t_1) \setminus \{x\} \\
FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)
\]
Substitution

Can you find the mistake in this definition of substitution?

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s] \lambda y. t &= \lambda y.([x \mapsto s] t) \\
[x \mapsto s] (t_1 \ t_2) &= ([x \mapsto s] t_1 \ [x \mapsto s] t_2)
\end{align*}
\]

Revised one:

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s] \lambda x. t &= \lambda x. t \\
[x \mapsto s] \lambda y. t &= \lambda y.([x \mapsto s] t) \quad \text{if } y \neq x \land y \notin \text{FV}(s) \\
[x \mapsto s] (t_1 \ t_2) &= ([x \mapsto s] t_1 \ [x \mapsto s] t_2)
\end{align*}
\]
Terms that differ only in the names of bound variables are interchangeable in all contexts.

**Substitution, finally**

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y & \text{if } x \neq y \\
[x \mapsto s] \lambda y.t &= \lambda y.([x \mapsto s]t) & \text{if } y \neq x \land y \not\in \text{FV}(s) \\
[x \mapsto s](t_1 \; t_2) &= ([x \mapsto s]t_1 \; [x \mapsto s]t_2)
\end{align*}
\]
Operational semantics, formally

**Call by value.**

\[
\begin{align*}
\text{APP1} & \quad \frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2} \\
\text{APP2} & \quad \frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2} \\
\text{APPABS} & \quad \frac{(\lambda x . t) v \rightarrow [x \mapsto v] t}{(\lambda x . t) v \rightarrow [x \mapsto v] t}
\end{align*}
\]

**Quiz.** Please give the evaluation rules for call-by-name and full \(\lambda\)-calculus, respectively.
Outline

Basics

Formalities

Programming in λ-calculus

Nameless representation of terms
Multiple arguments

- We do not write $f = \lambda(x, y).s$, instead, we write $f = \lambda x. \lambda y. s$.
- These two are different things. Informally, the first function takes a pair and returns $s$. The second one takes an $x$ and returns a function which will take a $y$ then return $s$.
- The transformation of multi-argument functions into higher-order functions is called currying, in honor of Haskell Curry.
Church Booleans

\[
\text{true} = \lambda t.\lambda f. t \\
\text{false} = \lambda t.\lambda f. f
\]

**Operators**

\[
\text{test} = \lambda l.\lambda m.\lambda n. l m n \\
\text{and} = \lambda m.\lambda n. m n \text{false}
\]

**Quiz.**
1. Define boolean operators \text{or} and \text{and} \text{not}.
2. What is the difference between \text{if then else} and \text{test} we define here?
Pairs

\[
\text{pair} = \lambda f.\lambda s.\lambda b. b f s
\]

Operators

\[
\text{fst} = \lambda p. p \text{ tru}
\]
\[
\text{snd} = \lambda p. p \text{ fls}
\]

Example.

\[
\text{fst} (\text{pair } v w)
\]
\[
\rightarrow^* \text{fst} (\lambda b. b v w)
\]
\[
\rightarrow (\lambda b. b v w) \text{ tru}
\]
\[
\rightarrow \text{tru } v w
\]
\[
\rightarrow^* v
\]
Church numerals

\[\begin{align*}
0 & = \lambda s.\lambda z. z \\
1 & = \lambda s.\lambda z. s\ z \\
2 & = \lambda s.\lambda z. s\ (s\ z) \\
3 & = \lambda s.\lambda z. s\ (s\ (s\ z)) \\
& \vdots
\end{align*}\]

- A number \( n \) is a function that takes two arguments \( s \) and \( z \) (\( succ \) and \( zero \)) and applies \( s \), \( n \) times to \( z \).
- 0 is syntactically equivalent to \( fls \).
- Successor functions: \( succ = \lambda n.\lambda s.\lambda z. s(n\ s\ z) \)

**Quiz.** Give another way to define \( succ \).
Operators

\[ n = \lambda s. \lambda z. s \cdots (s \, z) \cdots \]

\[ \text{n times of } s \]

- \textit{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m \, s \, (n \, s \, z)
- \textit{times} = \lambda m. \lambda n. m \, (\text{plus} \, n) \, 0
- \textit{power} = \lambda m. \lambda n. n \, (\text{times} \, m) \, 1
- \textit{iszero} = \lambda m. m \, (\lambda x. \text{fls}) \, \text{tru}
- \textit{pred} is quite a bit more difficult than additions.

\[
\begin{align*}
zz & = \text{pair} \, 0 \, 0; \\
ss & = \lambda p. \text{pair} \, (\text{snd} \, p) \, (\text{plus} \, 1 \, (\text{snd} \, p)); \\
prd & = \lambda m. \text{fst} \, (m \, ss \, zz);
\end{align*}
\]
Recursion

Can all terms be evaluate to a normal form? In λ-calculus, no. Here is a diverge term:

\[ \Omega = (\lambda x.xx)(\lambda x.xx) \]

**Fix-point combinator, or Y-combinator.**

Call-by-name version  \( Y = \lambda f. (\lambda x.f (x x)) (\lambda x.f (x x)) \)

Call-by-value version  \( Y = \lambda f. (\lambda x.f (\lambda y.x x y)) (\lambda x.f (\lambda y.x x y)) \)

How to define a recursive function?

\[ g = \lambda fact. \lambda x. (if \ x = 0 \ then \ 1 \ else \ x * (fact (x - 1))) \]

\[ \text{factorial} = Y g \]

**Quiz.** Give the reduction of \( \text{factorial} \ 3 \).
Outline

Basics

Formalities

Programming in \( \lambda \)-calculus

Nameless representation of terms
Overview

- **Conventions** help us to discuss basic concepts.
- In implementations, we need to choose a single representation.

**Candidates:**

1. Renaming bound variables to “fresh” names;
2. Devising some “canonical” representation of variables and terms that does not require renaming
4. Combinatory logic [Curry and Feys, 1958; Barendregt, 1984]

We will use the formulation based on a well-known technique due to *Nicolas de Bruijn*. 
De Bruijn terms

We can represent terms more straightforwardly by making variable occurrences point directly to their binders, rather than referring to them by name.

- $\lambda x.x$ to $\lambda.0$
- $\lambda x.\lambda y.x (y x)$ to $\lambda.\lambda.1(0 1)$

**Definition 6.1.2. Terms**

Let $T$ be the smallest family of sets $\{T_0, T_1, T_2, \cdots\}$ such that

- $k \in T_n$ whenever $0 \leq k < n$;
- if $t_1 \in T_n$ and $n > 0$, then $\lambda.t_1 \in T_{n-1}$;
- if $t_1 \in T_n$ and $t_2 \in T_n$, then $(t_1 t_2) \in T_n$.

The elements of $T_n$ are terms with at most $n$ free variables.
Naming context

- Suppose we want to represent $\lambda x. y x$ as a nameless term. What’s the binder for $y$?
- To deal with terms containing free variables, we need the idea of naming context.

**Example.** Given a naming context

$$\Gamma = \{x \mapsto 4, y \mapsto 3, z \mapsto 2, a \mapsto 1, b \mapsto 0\}$$

- $x (y z)$ encoded into 4 (3 2);
- $\lambda w. y w$ encoded into $\lambda.4.0$;
- $\lambda w. \lambda a. x$ encoded into $\lambda.\lambda.6$.

**Definition.** A naming context $\Gamma = x_n, x_{n-1}, \ldots, x_1, x_0$ assigns to each $x_i$ the de Bruijn index $i$. 
Shifting and substitution

Example.

\[(\lambda b. b (\lambda a. b a)) (\lambda b. b a) \]
\[\rightarrow [b \mapsto (\lambda b. b a)](b (\lambda a. b a)) \]
\[= (\lambda b. b a) (\lambda c. (\lambda b. b a) c) \]

Nameless representation under \( \Gamma = a \):

\[(\lambda.0 (\lambda.1 0)) (\lambda.0 1) \rightarrow [0 \mapsto (\lambda.0 1)](0 (\lambda.1 0)) = (\lambda.0 1) (\lambda.(\lambda.0 2) 0) \]

In the substitution \([j \mapsto s]t\) where \(t\) is an abstraction \(\lambda.t'\),

- \(j\) needs to be increased in \(t'\)
- The free variables in \(s\) also need to be increased in the substitution applied to \(t'\).
Definition 6.2.1: The $d$-place shift of a term $t$ about cutoff $c$, written $\uparrow^d_c (t)$ is defined as

$$\uparrow^d_c (k) = \begin{cases} k & \text{if } k < c \\ k + d & \text{if } k \geq c \end{cases}$$

$$\uparrow^d_c (\lambda.t_1) = \lambda.\uparrow^d_{c+1} (t_1)$$

$$\uparrow^d_c (t_1 t_2) = \uparrow^d_c (t_1) \uparrow^d_c (t_2)$$

Definition 6.2.4: The substitution of a term $s$ for variable number $j$ in a term $t$, written $[j \mapsto s]t$, is defined as follows:

$$[j \mapsto s]k = \begin{cases} s & \text{if } k = j \\ k & \text{otherwise} \end{cases}$$

$$[j \mapsto s](\lambda.t_1) = \lambda.[j + 1 \mapsto \uparrow^1 s]t_1$$

$$[j \mapsto s](t_1 t_2) = [j \mapsto s]t_1 [j \mapsto s]t_2$$
Evaluation

\[(\lambda.t_1) \ t_2 \quad \rightarrow \quad \uparrow^{-1} \ [0 \mapsto \uparrow^1 \ t_2] t_1\]

Example.

\[(\lambda b. \ w (\lambda a. b \ a)) (\lambda b. b \ a)\]
\[\rightarrow \quad [b \mapsto (\lambda b. b \ a)](b \ (\lambda a. b \ a))\]
\[= \ w \ (\lambda c. (\lambda b. b \ a) \ c)\]

Nameless representation under \(\Gamma = wa\):

\[(\lambda. 2 \ (\lambda. 1 \ 0)) \ (\lambda. 0 \ 1)\]
\[\rightarrow \quad \uparrow^{-1} \ [0 \mapsto \uparrow^1 \ (\lambda. 0 \ 1)](2 \ (\lambda. 1 \ 0))\]
\[= \quad \uparrow^{-1} \ [0 \mapsto \ (\lambda. 0 \ 2)](2 \ (\lambda. 1 \ 0))\]
\[= \quad \uparrow^{-1} \ (2 \ [0 \mapsto \ (\lambda. 0 \ 2)](\lambda. 1 \ 0)))\]
\[= \quad \uparrow^{-1} \ (2 \ \lambda. [1 \mapsto \uparrow^1 \ (\lambda. 0 \ 2)](1 \ 0)))\]
\[= \quad \uparrow^{-1} \ (2 \ \lambda. ((\lambda. 0 \ 3)](1 \ 0)))\]
\[= \quad \uparrow^{-1} \ (2 \ \lambda. ((\lambda. 0 \ 3) \ 0))\]
\[= \quad (1 \ \lambda.((\lambda. 0 \ 2) \ 0))\]

Quiz. Given \(\Gamma = wa\), show the de Bruijn notation of \((\lambda x. \lambda a. axw)(\lambda x. x)\) and evaluate it.
Conclusion

- \(\lambda\)-calculus is one of the most important models for computation theory.
- Call-by-value strategy is used by most programming languages. Call-by-name strategy is also called lazy strategy. \(\lambda\)-calculus with either strategy is Turing complete.
- De Bruijn notation is a great way to tackle with bound names, and is especially useful in the implementation.
Homework

- 5.2.3, 5.2.7, 5.2.8, 6.2.2, 6.2.5, 6.2.8