Types and Programming Languages

Lecture 3. Untyped λ-calculus

Xiaojuan Cai

cxj@sjtu.edu.cn

BASICS Lab, Shanghai Jiao Tong University

Spring, 2016

The history of λ-calculus

- The λ-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 — λ-calculus.
- *Landin*'s work and *John McCarthy*'s Lisp make λ-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 λ-calculus

Nameless representation of terms

Syntax

- In arithmetic expression, there is no function.
- In λ-calculus, everything is a function.

Syntax.

$$\begin{array}{lcll} t & ::= & & \textit{terms} \\ & & x & \textit{variable} \\ & & \lambda x.t & \textit{abstraction} \\ & & t\ t & \textit{application} \\ \\ v & ::= & & \textit{values} \\ & & \lambda x.t & \textit{functionvalue} \end{array}$$

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 λ-terms:

1. $s\ t\ u$ stands for $(s\ t)\ u$. (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*.

  $$id = \lambda x.x$$

Operational semantics, informally

$(\lambda x.s)\,t$ is called a *redex* ("reducible expression").

$$\beta\text{-reduction}\ \frac{}{(\lambda x.s)\,t \longrightarrow [x \mapsto t]s}$$

- Full β-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:

$$id\ (id\ (\lambda z.id\ z))$$

$$1.\underline{id\ (id\ (\lambda z.id\ z))}\quad 2.id\ (\underline{id\ (\lambda z.id\ z)})\quad 3.id\ (id\ (\lambda z.\underline{id\ z}))$$

- Full β-reduction allows all these redexes.

$$id\ (id\ (\lambda z.id\ z)) \longrightarrow id\ (id\ (\lambda z.z))$$

- Normal order strategy allows the first one.

$$id\ (id\ (\lambda z.id\ z)) \longrightarrow (id\ (\lambda z.id\ z))$$
$$\longrightarrow \lambda z.id\ z \longrightarrow \lambda z.z$$

- Call-by-name is more restrictive than normal order

$$id\ (id\ (\lambda z.id\ z)) \longrightarrow (id\ (\lambda z.id\ z))$$
$$\longrightarrow \lambda z.id\ z \not\longrightarrow$$

- Call by value requires the right-hand side to be a value.

$$id\ (id\ (\lambda z.id\ z)) \longrightarrow id\ (\lambda z.id\ z)$$
$$\longrightarrow \lambda z.id\ z$$

Which strategy?

- Full β-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

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.

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(\lambda x.t_1) & = & FV(t_1) \setminus \{x\} \\ FV(t_1\, t_2) & = & FV(t_1) \cup FV(t_2) \end{array}$$

Substitution

Can you find the mistake in this definition of substitution?

$$
\begin{array}{rcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \textit{if } x \neq y \\
[x \mapsto s]\lambda y.t & = & \lambda y.([x \mapsto s]t) & \\
[x \mapsto s](t_1\, t_2) & = & ([x \mapsto s]t_1\, [x \mapsto s]t_2) &
\end{array}
$$

Revised one:

$$
\begin{array}{rcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \textit{if } x \neq y \\
[x \mapsto s]\lambda x.t & = & \lambda x.t & \\
[x \mapsto s]\lambda y.t & = & \lambda y.([x \mapsto s]t) & \textit{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s](t_1\, t_2) & = & ([x \mapsto s]t_1\, [x \mapsto s]t_2) &
\end{array}
$$

Convention

Terms that differ only in the names of bound variables are interchangeable in all contexts.

Substitution, finally

$$
\begin{array}{rcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \textit{if } x \neq y \\
[x \mapsto s]\lambda y.t & = & \lambda y.([x \mapsto s]t) & \textit{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1\ [x \mapsto s]t_2) &
\end{array}
$$

Operational semantics, formally

Call by value.

$$\textsc{App1}\ \frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \ \longrightarrow\ t_1'\ t_2}$$

$$\textsc{App2}\ \frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \ \longrightarrow\ v_1\ t_2'}$$

$$\textsc{AppAbs}\ \frac{}{(\lambda x.t)\, v \ \longrightarrow\ [x \mapsto v]\, t}$$

Quiz. Please give the evaluation rules for *call-by-name* and *full* λ-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 return s. The second one takes an x return a function which will take a y then return s.
- The transformation of multi-argument functions into higher-order fucntions is called currying, in honor of *Haskell Curry*.

Church Booleans

$$
\begin{array}{rcl}
\mathit{tru} & = & \lambda t.\lambda f.t \\
\mathit{fls} & = & \lambda t.\lambda f.f
\end{array}
$$

Operators

$$
\begin{array}{rcl}
\mathit{test} & = & \lambda l.\lambda m.\lambda n.l\; m\; n \\
\texttt{and} & = & \lambda m.\lambda n.m\; n\; \mathit{fls}
\end{array}
$$

Quiz.

1. Define boolean opertors `or` and `not`.
2. What is the difference between *if then else* and *test* we define here?

Pairs

$$pair \quad = \quad \lambda f.\lambda s.\lambda b.b\, f\ s$$

Operators

$$\begin{array}{rcl} fst & = & \lambda p.p\; tru \\ snd & = & \lambda p.p\; fls \end{array}$$

Example.

$$\begin{array}{l} fst\,(pair\; v\; w) \\ \longrightarrow^{*} \quad fst\,(\lambda b.b\; v\; w) \\ \longrightarrow \quad (\lambda b.b\; v\; w)\; tru \\ \longrightarrow \quad tru\; v\; w \\ \longrightarrow^{*} \; v \end{array}$$

Church numerals

$$\begin{array}{rcl} 0 & = & \lambda s.\lambda z.z \\ 1 & = & \lambda s.\lambda z.s\,z \\ 2 & = & \lambda s.\lambda z.s\,(s\,z) \\ 3 & = & \lambda s.\lambda z.s\,(s\,(s\,z)) \\ \ldots & & \end{array}$$

- 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

$$n = \lambda s.\lambda z.\underbrace{s\ \cdots (s\,z)\cdots)}_{n \text{ times of} s}$$

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

$$\begin{aligned} zz &= pair\ 0\ 0; \\ ss &= \lambda p.pair\,(snd\ p)\,(plus\,1\,(snd\ p)); \\ prd &= \lambda m.fst(m\ ss\ zz); \end{aligned}$$

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?

$$\begin{array}{rcl} g & = & \lambda fact.\lambda x.(if\;\; x = 0\;\; then\; 1\;\; else\;\; x * (fact\,(x-1))) \\ factorial & = & Y\,g \end{array}$$

Quiz. Give the reduction of *factorial* 3.

Outline

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
3. Explicit substitution [Abadi, Cardelli, Curien, and Lvy, 1991]
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* $\Gamma = x_n, x_{n-1}, \cdots, x_1, x_0$ assigns to each x_i the *de Bruijn* index i.

Shifting and substitution

Example.

$$
\begin{aligned}
&(\lambda b.b\,(\lambda a.b\,a))\,(\lambda b.b\,a)\\
&\longrightarrow\ [b \mapsto (\lambda b.b\,a)](b\,(\lambda a.b\,a))\\
&= (\lambda b.b\,a)\,(\lambda c.(\lambda b.b\,a)\,c)
\end{aligned}
$$

Nameless representation under $\Gamma = a$:

$$(\lambda.0\,(\lambda.1\,0))\,(\lambda.0\,1)\ \longrightarrow\ [0 \mapsto\ (\lambda.0\,1)](0\,(\lambda.1\,0)) = (\lambda.0\,1)\,(\lambda.(\lambda.0\,2)\,0)$$

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'.

Shift and substitution

Definition 6.2.1: The d-place shift of a term t about cutoff c, written $\uparrow_c^d(t)$ is defined as

$$
\begin{array}{rcl}
\uparrow_c^d(k) & = & \begin{cases} k & \text{if } k < c \\ k+d & \text{if } k \geq c \end{cases} \\
\uparrow_c^d(\lambda.t_1) & = & \lambda.\uparrow_{c+1}^d(t_1) \\
\uparrow_c^d(t_1\ t_2) & = & \uparrow_c^d(t_1)\ \uparrow_c^d(t_2)
\end{array}
$$

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:

$$
\begin{array}{rcl}
[j \mapsto s]k & = & \begin{cases} s & \text{if } k = j \\ k & \text{otherwise} \end{cases} \\
[j \mapsto s](\lambda.t_1) & = & \lambda.[j+1 \mapsto \uparrow^1 s]t_1 \\
[j \mapsto s](t_1\ t_2) & = & [j \mapsto s]t_1\ [j \mapsto s]t_2
\end{array}
$$

Evaluation

$$(\lambda.t_1)\,t_2 \;\longrightarrow \uparrow^{-1}\,[0 \mapsto \uparrow^{1}\; t_2]t_1$$

Example.

$$\begin{aligned}
&(\lambda b.w\,(\lambda a.b\,a))\,(\lambda b.b\,a)\\
&\longrightarrow\;\; [b \mapsto (\lambda b.b\,a)](b\,(\lambda a.b\,a))\\
&= w\,(\lambda c.(\lambda b.b\,a)\,c)
\end{aligned}$$

Nameless representation under $\Gamma = wa$:

$$\begin{aligned}
&(\lambda.2\,(\lambda.1\,0))\,(\lambda.0\,1)\\
&\longrightarrow \uparrow^{-1}\,[0 \mapsto \uparrow^{1}\,(\lambda.0\,1)](2\,(\lambda.1\,0))\\
&= \uparrow^{-1}\,[0 \mapsto\;\; (\lambda.0\,2)](2\,(\lambda.1\,0))\\
&= \uparrow^{-1}\,(2\,[0 \mapsto\;\; (\lambda.0\,2)](\lambda.1\,0)))\\
&= \uparrow^{-1}\,(2\,\lambda.[1 \mapsto \uparrow^{1}\,(\lambda.0\,2)](1\,0)))\\
&= \uparrow^{-1}\,(2\,\lambda.[1 \mapsto\;\; (\lambda.0\,3)](1\,0)))\\
&= \uparrow^{-1}\,(2\,\lambda.((\lambda.0\,3)\,0))\\
&=\;\; (1\,\lambda.((\lambda.0\,2)\,0))
\end{aligned}$$

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

Conclusion

- λ-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. λ-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, 5.3.6, 6.1.4, 6.1.5, 6.2.2, 6.2.5, 6.2.8, 6.3.1
- Read Chapter 7: *An ML Implementation of the Lambda-Calculus*, and extend `arith` with untyped λ-calculus.