中国科学技术大学
University of Science and Technology of China

Fundamentals

《程序设计语言理论》

张昱

0551-63603804，yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

参考资料

- PFPL
  - Chapter 4 Statics, 5 Dynamics, and 6 Type Safety
- TAPL
  - Chapter 8 Typed Arithmetic Expressions
  - Chapter 9 Simply Typed Lambda-Calculus
- http://staff.ustc.edu.cn/~yuzhang/tpl

Theoretical Foundations

- Computability Theory (halting problem)
- Program Logics (Axiomatic Semantics)
- Lambda Calculus (syntax, operational semantics)
- Denotational Semantics
- Operational Semantics
- Type Theory

Lambda Calculus

- **对程序语言进行数学分析: 从语言建模开始**
  - 突出感兴趣的程序构造，忽略一些无关的细节
  - **Lambda Calculus:** 抓住语言本质的很小的核心演算
    1930s, Alonzo Church & Stephen Cole Kleene
- **Lambda Calculus：源于可计算理论**
  - 奠定语言中函数定义和命名约定的基本机制
  - 既可看成一种简单的语言(用于描述计算)，又可看成一种数学对象（可证明）
  - 用类型化l演算(typed lambda calculus) 的框架来研究程序设计语言的各种概念

Lambda Calculus

- λ表示法的主要特征
  - λ抽象(abstraction): 用于定义函数
  - λ应用(application): 使用所定义的函数
  - 用λ表示法写出的表达式叫做λ表达式或λ项
- 举例
  - Typed λ calculus（自然数类型上的几个例子）
    - 恒等函数： λx:nat.x (Id(x:nat) = x) 无须给函数命名
    - 后继函数： λx:nat.x+1
    - 常函数： λx:nat.10
  - Untyped λ calculus λx.x

Free and Bound Variables

- λ项λx:σ.M
  - λ是一个约束算子
  - Bound Variable（约束变元） x 是占位符
    可以重新命名λ约束变元而不改变表达式的含义
  - 在λx:σ.x+y 中，x是约束的，y是自由的；
  - α-conversion：λx:σ.x+y 等同于 λz:σ.z+y

Free and Bound Variables

- Application：左结合：MNP应看成 (MN) P
  - (λx. (x+y)) 3 = 3 + y
  - (λz. (x + 2*y + z)) 5 = x + 2*y + 5

λ的约束范围应尽可能地大，直到表达式结束或碰到不能配对的右括号为止

  - λx. f (f x) = λx.(f (f (x)))

Higher-Order Functions

高阶函数（Higher-Order Functions）

- **Given function f, return function f ∘ f**

  $\lambda f.\ \lambda x.\ f\ (f\ x)$

- **举例**

  $(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y + 1)$

  $= \lambda x.\ (\lambda y.\ y + 1)\ ((\lambda y.\ y + 1)\ x)$

  $= \lambda x.\ (\lambda y.\ y + 1)\ (x+1)$

  $= \lambda x.\ (x+1) + 1$

Reduction 归约

- Basic computation rule is β-reduction

  $(\lambda x.\ e1)\ e2 \rightarrow [e2/x]\ e1$

  Rename bound variables to avoid name collisions

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

  - Substitute “blindly”

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

  - Rename bound variables

    $(\lambda f.\ \lambda z.\ f\ (f\ z))\ (\lambda y.\ y + x)$

    $= \lambda z.\ [(\lambda y.\ y + x)\ ((\lambda y.\ y + x)\ z))] = \lambda z.\ z + x + x$

Reduction 归约

- Basic computation rule is β-reduction

  $(\lambda x.\ e1)\ e2 \rightarrow [e2/x]\ e1$

- Reduction
  - Apply basic computation rule to any subexpression
  - Repeat

Formal Semantics（形式语义）

- 公理语义
- 操作语义
- 指称语义

Formal Semantics

以数学为工具，利用符号和公式，精确定义和解释计算机程序语言的语义，使语义形式化的学科.

- 公理语义(Axiomatic Semantics)
  - 推导表达式之间等式的形式系统
- 操作语义(Operational Semantics)
  - 将等式确定为有向规则的推理，称为归约 Reduction（符号求值）

证明系统

- 指称语义(Denotational Semantics)
  - 称为模型。一个模型是一组集合，每种类型一个集合，每个良类型的表达式可解释为相应集合中的一个元素

公理语义

- 1969,Hoare, An Axiomatic Basis for Computer Programming
  - 语言的数学理论，提供证明程序性质的逻辑基础
    - 语法规则：确定什么是合式公式(well-formed fomula)
    - 公理：是不加证明地被接受的基本定理
    - 推理规则：从已确定的定理演绎新定理的机理
    - 基本的逻辑系统，如，带有等式的一阶谓词演算
  - 对证明程序正确性有用
  - 示例：整数运算、程序执行

公理语义-整数运算

整数运算公理

$$
\begin{array}{ll}
A1 & x + y = y + x \\
A2 & x \times y = y \times x \\
A3 & (x + y) + z = x + (y + z) \\
A4 & (x \times y) \times z = x \times (y \times z) \\
A5 & x \times (y + z) = x \times y + x \times z \\
A6 & y \leq x \supset (x - y) + y = x \\
A7 & x + 0 = x \\
A8 & x \times 0 = 0 \\
A9 & x \times 1 = x \\
 & \ldots\ldots
\end{array}
$$

演绎

定理

$$y \leq r \supset r + y \times q = (r - y) + y \times (1 + q)$$

证明

$$
\begin{array}{lll}
 & (r - y) + y \times (1 + q) & \\
= & (r - y) + (y \times 1 + y \times q) & (A5) \\
= & (r - y) + (y + y \times q) & (A9) \\
= & ((r - y) + y) + y \times q & (A3) \\
= & r + y \times q \quad \text{provided } y \leq r & (A6)
\end{array}
$$

中国科学技术大学
University of Science and Technology of China

公理语义-程序执行

- 公式：P {Q} R （{P} Q {R}） P和R都是一阶公式

  如果前条件(断言)P在程序Q执行前的状态成立，则执行Q后将得到满足后条件(断言)R的状态。

  部分正确性断言：如果P在Q执行前为真，那么，如果Q的执行终止，则终止在使R为真的某个状态。

  终止性断言：如果P在Q执行前为真，那么Q将终止在使R为真的某个状态。

- 推理规则的表示

$$\frac{f_0, f_1, \dots f_n}{f_0}$$

前提(premise)

推论(conclusion)

公理语义-程序执行

- 赋值公理 $\vdash P_0\{x := f\}P$

  其中x是变量，f是表达式，P_0可以通过用f代换P中的每一个x而得到

  $$\vdash y > 8\{x := y+4\}x > 12$$

- 推理规则

  - **D1: Rules of Consequence**

  $$\frac{P\{Q\}R,\ R \rightarrow S}{P\{Q\}S} \qquad \frac{P\{Q\}R,\ S \rightarrow P}{S\{Q\}R}$$

  - **D2: Rule of Composition**

  $$\frac{P\{Q1\}R1,\ R1\{Q2\}R}{P\{Q1;Q2\}R}$$

公理语义-程序执行

□ 推理规则

- **D3: Rules of Iteration**

$$\frac{P \ \& \ B\{S\}\ P}{P\ \{\text{while } B \text{ do } S\}\ \neg B \ \& \ P}$$

□ 例子

```
PROCEDURE FACT ( N:INTEGER; VAR Y:INTEGER);
VAR X: INTEGER;
BEGIN
X := 0;
Y := 1;
ASSERT ( Y = X! & X ≤ N )
WHILE X < N DO BEGIN
X := X + 1;
Y := Y * X
END
END;
ENTRY: N ≥ 0
EXIT: Y = N!
```

公理语义

一个等式公理系统

- 代换：[N/x]M 表示M中的自由变元x用N代换的结果

注意:N中的自由变元不能代换到M中后成为约束变元

- 约束变元改名公理

$\lambda x{:}\sigma.\, M = \lambda y{:}\sigma.\, [\, y/x\,]M$, M中无自由出现的y (α)

例如， $\lambda x{:}\sigma.x+y = \lambda z{:}\sigma.z+y$

- 等价公理：$(\lambda x{:}\sigma.\, M)\, N = [N/x]\, M$ $(\beta)_{eq}$

函数应用–在函数体中用实在变元代替形式变元

公理语义

- 一个等式公理系统
  - 约束变元改名公理

    $\lambda x{:}\sigma.\, M = \lambda y{:}\sigma.\, [\, y/x\,]M$, M中无自由出现的y (α)
  - 等价公理：$(\lambda x{:}\sigma.\, M)\, N = [N/x]\, M$ $(\beta)_{eq}$
  - 同余性规则：相等的函数应用于相等的变元产生相等的结果

$$\frac{M_1 = M_2,\, N_1 = N_2}{M_1 N_1 = M_2 N_2}$$

操作语义

1964, P.J.Landin, The mechanical evaluation of expressions

SECD (Stack, Environment, Code, Dump) machine

- 定义一个抽象机，给出在抽象机上的执行规则
  - 抽象机的大状态 (st, s, c)
    - st: 栈区(工作区)
    - s: 环境区(数据)
    - c: 控制区(程序)
  - 状态转移规则
    - 例： $(x_1*x_2)+1$的求值
    - 在x_1,x_2值为2和3时
    - 符号" /"用于分割存放的信息

$$
\begin{aligned}
& (st, s, ((x_1 * x_2)+1)/c) \\
\overset{1}{\Longrightarrow}\ & (st, s, (x_1 * x_2)/1/+/c) \\
\overset{1}{\Longrightarrow}\ & (st, s, x_1/x_2/*/1/+/c) \\
\overset{3}{\Longrightarrow}\ & (\mathbf{2}/st, s, x_2/*/1/+/c) \\
\overset{3}{\Longrightarrow}\ & (\mathbf{3}/\mathbf{2}/st, s, */1/+/c) \\
\overset{4}{\Longrightarrow}\ & (\mathbf{6}/st, s, 1/+/c) \\
\overset{2}{\Longrightarrow}\ & (\mathbf{1}/\mathbf{6}: st, s, +/c) \\
\overset{4}{\Longrightarrow}\ & (\mathbf{7}/st, s, c)
\end{aligned}
$$

操作语义

操作语义(Operational Semantics)

- 是演绎出最终结果的证明系统，或者说是通过一系列步骤变换一个表达式的证明系统
- 由等式公理的单向形式给出了归约规则

β归约

$$(\lambda x{:}\sigma.\ M)\ N \rightarrow_{\beta} [N/x]\ M \qquad (\beta)_{red}$$

例如， $(\lambda x{:}nat.\ x+4)\ 4 \rightarrow 4+4$

β归约是定义在α等价上的，即β归约的结果不是唯一确定的，但是归约产生的任何两个项仅在约束变元的名字上有区别。

- 对实现编译器或解释器有用

指称语义

源于Christopher Strachey 和 Dana Scott 在1960年代的工作

又称为不动点语义(fixed-point semantics),Scott-Strachey语义

- 先确定**指称物**(多为数学对象，如整数、集合、函数)，然后给出语言构造到指称物的语义映射，该映射满足
  - 每个语言构造的每个实例都有对应的指称
  - 复合语言构造的实例的指称只依赖于它的子构造的指称
- 类型化λ演算的指称语义
  - 每个类型表达式对应到一个集合，称为该类型的值集
  - 类型σ 的项解释为其值集上的一个元素
  - 类型$\sigma \to \tau$ 的值集是函数集合，项λx:σ. M解释为一个数学函数
- 无类型化λ演算可以从类型化λ演算中派生

More on Lambda Calculus

Non-terminating reduction

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

$\rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

$\rightarrow \ldots$

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

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

$\rightarrow \ldots$

$(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$\rightarrow f\ \big((\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))\big)$

$\rightarrow \ldots$

Terminating & non-terminating

Term may have both terminating and non-terminating reduction sequences

$(\lambda u.\ \lambda v.\ v)\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))$

$\to \lambda v.\ v$

$(\lambda u.\ \lambda v.\ v)\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))$

$\to (\lambda u.\ \lambda v.\ v)\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))$

$\to \ldots$

Reduction strategies

- **Normal-order reduction: choose the left-most, outer-most redex first**

$(\lambda u.\ \lambda v.\ v)\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))$

$\to \lambda v.\ v$

Normal-order reduction will find normal form if exists

- **Applicative-order reduction: choose the left-most, inner-most redex first**

$(\lambda u.\ \lambda v.\ v)\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))$

$\to (\lambda u.\ \lambda v.\ v)\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))$

$\to \ldots$

Reduction strategies

- Examples

Normal-order	*Applicative-order*
$(\lambda x.\ x\ x)\ ((\lambda y.\ y)\ (\lambda z.\ z))$	$(\lambda x.\ x\ x)\ ((\lambda y.\ y)\ (\lambda z.\ z))$
$\to ((\lambda y.\ y)\ (\lambda z.\ z))\ ((\lambda y.\ y)\ (\lambda z.\ z))$	$\to (\lambda x.\ x\ x)\ (\lambda z.\ z)$
$\to (\lambda z.\ z)\ ((\lambda y.\ y)\ (\lambda z.\ z))$	$\to (\lambda z.\ z)\ (\lambda z.\ z)$
$\to (\lambda y.\ y)\ (\lambda z.\ z)$	$\to \lambda z.\ z$
$\to \lambda z.\ z$	

Reduction strategies

Examples

Applicative-order may ***not*** be as efficient as normal-order when the argument is not used.

Normal-order

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

$\rightarrow p$

Applicative-order

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

$\rightarrow (\lambda x.\ p)\ (\lambda z.\ z)$

$\rightarrow p$

Reduction strategies

- Similar to (but subtly different from) *evaluation strategies* in language theories
  - Call-by-name (like normal-order)
    - ALGOL 60
  - arguments are not evaluated, but directly substituted into function body
  - Call-by-need ("memorized version" of call-by-name)
    - Haskell, R, …
  - *called "lazy evaluation"*
  - Call-by-value (like applicative-order)
    - C, …
  - *called "eager evaluation"*
  - …

Main points till now

- Syntax: notation for defining functions
  - "Pure": without adding any additional syntax

$$\text{(Terms)} \quad M, N \ ::= \ x \ | \ \lambda x.\, M \ | \ M\, N$$

- Semantics (reduction rules)

$$(\lambda x.\, M)\, N \ \rightarrow \ [N/x]M \qquad (\beta)$$

- Next: programming in "pure" λ-calculus
  - Encoding data and operators

Programming in λ-calculus

- **Encoding Boolean values and operators**
  - **True ≡ λx. λy. x**
  - **False ≡ λx. λy. y**

Programming in λ-calculus

- Encoding Boolean values and operators
  - **True ≡ λx. λy. x**
  - **False ≡ λx. λy. y**
  - **not ≡ λb. b False True**

not True
→ True False True
→ False

not False
→ False False True
→ True

Programming in λ-calculus

- Encoding Boolean values and operators
  - **True ≡ λx. λy. x**
  - **False ≡ λx. λy. y**
  - **not ≡ λb. b False True**
  - **and ≡ λb. λb'. b b' False**

and True b
→* True b False
→ b

and False b
→* False b False
→ False

Programming in λ-calculus

Encoding Boolean values and operators

- **True ≡ λx. λy. x**
- **False ≡ λx. λy. y**
- **not ≡ λb. b False True**
- **and ≡ λb. λb'. b b' False**
- **or ≡ λb. λb'. b True b'**

or True b
→* True True b
→ True

or False b
→* False True b
→ b

Programming in λ-calculus

- Encoding Boolean values and operators
  - True ≡ λx. λy. x
  - False ≡ λx. λy. y
  - not ≡ λb. b False True
  - and ≡ λb. λb’. b b’ False
  - or ≡ λb. λb’. b True b’
  - if b then M else N ≡ b M N

Not unique encoding

Programming in λ-calculus

- **Encoding Boolean values and operators**
  - **True ≡ λx. λy. x**
  - **False ≡ λx. λy. y**
  - **not ≡ λb. b False True**
  - **and ≡ λb. λb'. b b' False**
  - **or ≡ λb. λb'. b True b'**
  - **if b then M else N ≡ b M N**
  - **not' ≡ λb. λx. λy. b y x**

not' True
→ λx. λy. True y x
→ λx. λy. y = False

not' False
→ λx. λy. False y x
→ λx. λy. x = True

中国科学技术大学
University of Science and Technology of China

Programming in λ-calculus

- Church numerals
  - $\underline{0} \equiv \lambda f.\ \lambda x.\ x$ *(the same as False!)*
  - $\underline{1} \equiv \lambda f.\ \lambda x.\ f\ x$
  - $\underline{2} \equiv \lambda f.\ \lambda x.\ f\ (f\ x)$
  - $\underline{n} \equiv \lambda f.\ \lambda x.\ f^n\ x$

Programming in λ-calculus

- **Church numerals**
  - **$\underline{0} \equiv \lambda f.\ \lambda x.\ x$** ***(the same as False!)***
  - **$\underline{1} \equiv \lambda f.\ \lambda x.\ f\ x$**
  - **$\underline{2} \equiv \lambda f.\ \lambda x.\ f\ (f\ x)$**
  - **$\underline{n} \equiv \lambda f.\ \lambda x.\ f^n\ x$**
  - **$\text{succ} \equiv \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$**

$\text{succ}\ \underline{n}$
$\rightarrow \lambda f.\ \lambda x.\ f\ (\underline{n}\ f\ x)$
$= \lambda f.\ \lambda x.\ f\ ((\lambda f.\ \lambda x.\ f^n\ x)\ f\ x)$
$\rightarrow \lambda f.\ \lambda x.\ f\ (f^n\ x)$
$= \lambda f.\ \lambda x.\ f^{n+1}\ x$
$= \underline{n+1}$

Programming in λ-calculus

- Church numerals
  - $\underline{0} \equiv \lambda f.\ \lambda x.\ x$ *(the same as False!)*
  - $\underline{1} \equiv \lambda f.\ \lambda x.\ f\ x$
  - $\underline{2} \equiv \lambda f.\ \lambda x.\ f\ (f\ x)$
  - $\underline{n} \equiv \lambda f.\ \lambda x.\ f^n\ x$
  - $\text{succ} \equiv \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$
  - $\text{succ}' \equiv \lambda n.\ \lambda f.\ \lambda x.\ n\ f\ (f\ x)$

Programming in λ-calculus

Church numerals

- $\underline{0} \equiv \lambda f.\ \lambda x.\ x$
- $\underline{1} \equiv \lambda f.\ \lambda x.\ f\ x$
- $\underline{2} \equiv \lambda f.\ \lambda x.\ f\ (f\ x)$
- $\underline{n} \equiv \lambda f.\ \lambda x.\ f^{n}\ x$
- $\text{succ} \equiv \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$
- $\text{iszero} \equiv \lambda n.\ \lambda x.\ \lambda y.\ n\ (\lambda z.\ y)\ x$

iszero $\underline{0}$
$\to \lambda x.\ \lambda y.\ \underline{0}\ (\lambda z.\ y)\ x$
$= \lambda x.\ \lambda y.\ (\lambda f.\ \lambda x.\ x)\ (\lambda z.\ y)\ x$
$\to \lambda x.\ \lambda y.\ (\lambda x.\ x)\ x$
$\to \lambda x.\ \lambda y.\ x$ = True

iszero $\underline{1}$
$\to \lambda x.\ \lambda y.\ \underline{1}\ (\lambda z.\ y)\ x$
$= \lambda x.\ \lambda y.\ (\lambda f.\ \lambda x.\ f\ x)\ (\lambda z.\ y)\ x$
$\to \lambda x.\ \lambda y.\ (\lambda x.\ (\lambda z.\ y)\ x)\ x$
$\to \lambda x.\ \lambda y.\ ((\lambda z.\ y)\ x)$
$\to \lambda x.\ \lambda y.\ y$ = False

iszero (succ $\underline{n}$) $\to^{*}$ False

Programming in λ-calculus

- Church numerals
  - $\underline{0} \equiv \lambda f.\ \lambda x.\ x$
  - $\underline{1} \equiv \lambda f.\ \lambda x.\ f\ x$
  - $\underline{2} \equiv \lambda f.\ \lambda x.\ f\ (f\ x)$
  - $\underline{n} \equiv \lambda f.\ \lambda x.\ f^n\ x$
  - $\text{succ} \equiv \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$
  - $\text{iszero} \equiv \lambda n.\ \lambda x.\ \lambda y.\ n\ (\lambda z.\ y)\ x$
  - $\text{add} \equiv \lambda n.\ \lambda m.\ \lambda f.\ \lambda x.\ n\ f\ (m\ f\ x)$
  - $\text{mult} \equiv \lambda n.\ \lambda m.\ \lambda f.\ n\ m\ f$

Programming in λ-calculus

- Booleans
- Natural numbers
- Pairs
- Lists
- Trees
- Recursive functions
- …

Read supplementary materials: A