




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/tp/>


2



Theoretical Foundations

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


3



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λ演算(typed lambda calculus)的框架来研究程序设计语言的各种概念


4



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: $\lambda x:\text{nat}.x$ ($\text{Id}(x:\text{nat}) = x$) 无须给函数命名
 - 后继函数: $\lambda x:\text{nat}.x+1$
 - 常函数: $\lambda x:\text{nat}.10$
 - Untyped λ calculus $\lambda x.x$

5



Free and Bound Variables

- λ项 $\lambda x:\sigma.M$
 - λ是一个约束算子, 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 $\lambda x:\sigma.x+y$ 中, x是约束的, y是自由的;
 - α-conversion: $\lambda x:\sigma.x+y$ 等同于 $\lambda z:\sigma.z+y$
- Application: 左结合: MNP应看成 (MN) P
 - $(\lambda x.(x+y)) 3 = 3 + y$
 - $(\lambda z.(x + 2*y + z)) 5 = x + 2*y + 5$

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

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

6

Higher-Order Functions

高阶函数 (Higher-Order Functions)

- Given function f , return function $f \circ f$
 $\lambda f. \lambda x. f (f x)$
- 举例

$$\begin{aligned}
 & (\lambda f. \lambda x. f (f x)) (\lambda y. y + 1) \\
 &= \lambda x. (\lambda y. y + 1) ((\lambda y. y + 1) x) \\
 &= \lambda x. (\lambda y. y + 1) (x + 1) \\
 &= \lambda x. (x + 1) + 1
 \end{aligned}$$

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
 - Apply basic computation rule to any subexpression
 - Repeat

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

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)
- 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

证明系统

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 formula)
 - 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
- 演绎

定理

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

证明

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

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



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。

- 推理规则的表达



13



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 代替 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

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

- D2: Rule of Composition

$$\frac{P\{Q_1\}R_1, R_1\{Q_2\}R}{P\{Q_1; Q_2\}R}$$

14



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

```

1 PROCEDURE FACT ( N:INTEGER; VAR Y:INTEGER);
2 VAR X: INTEGER;
3 BEGIN
4 X := 0;
5 Y := 1;
6 ASSERT ( Y = X! & X ≤ N )
7 WHILE X < N DO BEGIN
8 X := X + 1;
9 Y := Y * X
10 END
11 END;
ENTRY: N ≥ 0
EXIT: Y = N!

```

15



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

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

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

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

函数应用-在函数体中用实在变元代替形式变元

16



公理语义

- 一个等式公理系统

- 约束变元改名公理

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

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

- 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

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

17



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) $\xrightarrow{1} (st, s, ((x_1 * x_2) + 1) / c)$
 - s: 环境区(数据) $\xrightarrow{1} (st, s, x_1 / x_2 * 1 / c)$
 - c: 控制区(程序) $\xrightarrow{3} (2 / st, s, x_2 * 1 / c)$
- 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

18



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

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

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词上有区别。

- 对实现编译器或解释器有用

19



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

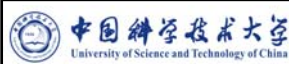
- 每个语言构造的每个实例都有对应的指称
- 复合语言构造的实例的指称只依赖于它的子构造的指称

- 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 $\sigma \rightarrow \tau$ 的值集是函数集合, 项 $\lambda x:\sigma. M$ 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

20



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

More on Lambda Calculus



Non-terminating reduction

$$\begin{aligned}
 & (\lambda x. x x) (\lambda x. x x) && (\lambda x. f (x x)) (\lambda x. f (x x)) \\
 \rightarrow & (\lambda x. x x) (\lambda x. x x) && \rightarrow f ((\lambda x. f (x x)) (\lambda x. f (x x))) \\
 \rightarrow & \dots && \rightarrow \dots \\
 & (\lambda x. x x y) (\lambda x. x x y) && \\
 \rightarrow & (\lambda x. x x y) (\lambda x. x x y) y && \\
 \rightarrow & \dots &&
 \end{aligned}$$

22



Terminating & non-terminating

Term may have both terminating and non-terminating reduction sequences

$$\begin{aligned}
 & (\lambda u. \lambda v. v) ((\lambda x. x x)(\lambda x. x x)) \\
 \rightarrow & \lambda v. v
 \end{aligned}$$

$$\begin{aligned}
 & (\lambda u. \lambda v. v) ((\lambda x. x x)(\lambda x. x x)) \\
 \rightarrow & (\lambda u. \lambda v. v) ((\lambda x. x x)(\lambda x. x x)) \\
 \rightarrow & \dots
 \end{aligned}$$

23



Reduction strategies

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

$$\begin{aligned}
 & (\lambda u. \lambda v. v) ((\lambda x. x x)(\lambda x. x x)) \quad \text{Normal-order reduction will} \\
 \rightarrow & \lambda v. v \quad \text{find normal form if exists}
 \end{aligned}$$

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

$$\begin{aligned}
 & (\lambda u. \lambda v. v) ((\lambda x. x x)(\lambda x. x x)) \\
 \rightarrow & (\lambda u. \lambda v. v) ((\lambda x. x x)(\lambda x. x x)) \\
 \rightarrow & \dots
 \end{aligned}$$

24

Reduction strategies

Examples

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

25

Reduction strategies

Examples

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

Normal-order	Applicative-order
$(\lambda x. p) ((\lambda y. y) (\lambda z. z))$	$(\lambda x. p) ((\lambda y. y) (\lambda z. z))$
$\rightarrow p$	$\rightarrow (\lambda x. p) (\lambda z. z)$
	$\rightarrow p$

26

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"
- ...

27

Main points till now

- Syntax: notation for defining functions
 - "Pure": without adding any additional syntax
 - (Terms) $M, N ::= x \mid \lambda x. M \mid M N$
- Semantics (reduction rules)
 - $(\lambda x. M) N \rightarrow [N/x]M \quad (\beta)$
- Next: programming in "pure" λ -calculus
 - Encoding data and operators

28

Programming in λ -calculus

Encoding Boolean values and operators

- True $\equiv \lambda x. \lambda y. x$
- False $\equiv \lambda x. \lambda y. y$

29

Programming in λ -calculus

Encoding Boolean values and operators

- True $\equiv \lambda x. \lambda y. x$
- False $\equiv \lambda x. \lambda y. y$
- not $\equiv \lambda b. b \text{ False True}$
 - not True $\rightarrow \text{True False True} \rightarrow \text{False}$
 - not False $\rightarrow \text{False False True} \rightarrow \text{True}$

30

Programming in λ -calculus

- Encoding Boolean values and operators
 - True $\equiv \lambda x. \lambda y. x$
 - False $\equiv \lambda x. \lambda y. y$
 - not $\equiv \lambda b. b \text{ False True}$
 - and $\equiv \lambda b. \lambda b'. b \text{ b' False}$

and True b
 $\rightarrow^* \text{True } b \text{ False}$
 $\rightarrow b$

and False b
 $\rightarrow^* \text{False } b \text{ False}$
 $\rightarrow \text{False}$

31

Programming in λ -calculus

- Encoding Boolean values and operators
 - True $\equiv \lambda x. \lambda y. x$
 - False $\equiv \lambda x. \lambda y. y$
 - not $\equiv \lambda b. b \text{ False True}$
 - and $\equiv \lambda b. \lambda b'. b \text{ b' False}$
 - or $\equiv \lambda b. \lambda b'. b \text{ True b'}$

or True b
 $\rightarrow^* \text{True True } b$
 $\rightarrow \text{True}$

or False b
 $\rightarrow^* \text{False True } b$
 $\rightarrow b$

32

Programming in λ -calculus

- Encoding Boolean values and operators
 - True $\equiv \lambda x. \lambda y. x$
 - False $\equiv \lambda x. \lambda y. y$
 - not $\equiv \lambda b. b \text{ False True}$
 - and $\equiv \lambda b. \lambda b'. b \text{ b' False}$
 - or $\equiv \lambda b. \lambda b'. b \text{ True b'}$
 - if b then M else N $\equiv b \text{ M N}$

Not unique encoding

33

Programming in λ -calculus

- Encoding Boolean values and operators
 - True $\equiv \lambda x. \lambda y. x$
 - False $\equiv \lambda x. \lambda y. y$
 - not $\equiv \lambda b. b \text{ False True}$
 - and $\equiv \lambda b. \lambda b'. b \text{ b' False}$
 - or $\equiv \lambda b. \lambda b'. b \text{ True b'}$
 - if b then M else N $\equiv b \text{ M N}$
 - not' $\equiv \lambda b. \lambda x. \lambda y. b \text{ y } x$

not' True
 $\rightarrow \lambda x. \lambda y. \text{True } y \text{ } x$
 $\rightarrow \lambda x. \lambda y. y = \text{False}$

not' False
 $\rightarrow \lambda x. \lambda y. \text{False } y \text{ } x$
 $\rightarrow \lambda x. \lambda y. x = \text{True}$

34

Programming in λ -calculus

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

35

Programming in λ -calculus

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

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

36



Programming in λ -calculus

□ Church numerals

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

37



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

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

iszero (succ \underline{n}) \rightarrow * False

38



Programming in λ -calculus

□ Church numerals

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

39



Programming in λ -calculus

- **Booleans**
- **Natural numbers**
- **Pairs**
- **Lists**
- **Trees**
- **Recursive functions**
- ...

Read supplementary materials: [A](#)

40