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

Fundamentals

《程序语言设计和程序分析》

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/pldpa

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λ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，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
- 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

  λf. λx. f (f x)

- 举例

  (λf. λx. f (f x)) (λy. y +1)

  = λx. (λy. y +1) ((λy. y +1) x)

  = λx. (λy. y +1) (x+1)

  = λx. (x+1) +1

Reduction 归约

- **Basic computation rule is β-reduction**

  (λx. e1) e2 → [e2/x] e1

  Rename bound variables to avoid name collisions

  (λf. λx. f (f x)) (λy. y +x)

  - **Substitute "blindly"**

    λx. (λy. y + x) ((λy. y + x) x) = λx. x + x + x

  - **Rename bound variables**

    (λf. λz. f (f z)) (λy. y +x)

    = λz. [(λy. y + x) ((λy. y + x) z))] = λz. 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）

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

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

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

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

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

```
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!
```

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 $\quad(\alpha)$

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

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

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

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**

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

More on Lambda Calculus

Non-terminating reduction

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

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

$\rightarrow \ldots$

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

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

$\rightarrow \ldots$

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

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

$\rightarrow \ldots$

Terminating & non-terminating

Term may have both terminating and non-terminating reduction sequences

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

$\rightarrow \lambda v.\ v$

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

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

$\rightarrow \ldots$

Reduction strategies

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

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

$\rightarrow \lambda v.\ v$

Normal-order reduction will find normal form if exists

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

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

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

$\rightarrow \ldots$

Reduction strategies

Examples

Normal-order

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

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

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

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

$\rightarrow \lambda z.\ z$

Applicative-order

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

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

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

$\rightarrow \lambda z.\ z$

Reduction strategies

Examples

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

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

Reduction strategies

- Similar to (but subtly different from) *evaluation strategies* in language theories
  - **Call-by-name (like normal-order)**
    - ALGOL 60
  - **Call-by-need ("memorized version" of call-by-name)**
    - Haskell, R, …
  - **Call-by-value (like applicative-order)**
    - C, …
  - **…**

arguments are not evaluated, but directly substituted into function body

called "lazy evaluation"

called "eager evaluation"

Main points till now

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

    (Terms) M, N ::= x | λx. M | 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

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** $\equiv \lambda x.\lambda y.\ x$
- **False** $\equiv \lambda x.\lambda y.\ y$
- **not** $\equiv \lambda b.\ b\ \mathbf{False}\ \mathbf{True}$

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

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!)*
- $\underline{1} \equiv \lambda f.\, \lambda x.\, f\ x$
- $\underline{2} \equiv \lambda f.\, \lambda x.\, f\ (f\ x)$
- $\underline{n} \equiv \lambda f.\, \lambda x.\, f^n\ x$
- $\mathbf{succ} \equiv \lambda n.\, \lambda f.\, \lambda x.\, f\ (n\ f\ x)$

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

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

Programming in λ-calculus

Church numerals

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

Programming in λ-calculus

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

Read supplementary materials: A

中国科学技术大学

University of Science and Technology of China

Lambda Expression in C++

C++ Lambda

Since C++ 2011-N3242

5.1 Primary expressions, p89

5.1.2 Lambda expressions, p91

多个可选项

```
[capture list] (params list) mutable exception-> return type
{ function body }
```

捕获外部变量列表

```
#include <algorithm>
#include <cmath>
#include <array>
#include <iostream>
#include <string_view>
void abssort(int *x, unsigned N) {
  std::sort(x, x + N,
    [](int a, int b) {
      return std::abs(a) < std::abs(b);
    });
}
```

```
int main()
{
    std::array<int, 10> s =
      {5, 7, -4, 2, 8, -6, 1, -9, 0, 3};
    auto print = [&s]
     (std::string_view const rem) {
       for (auto a : s) {
          std::cout << a << ' ';
       }
       std::cout << ": " << rem << '\n';
    };
    stdabssort(s.begin(), 10);
}
```

Lambda Capture List

C++2011: Lambda-capture: & | = | id | & id | this

- Captured by value
- Captured by reference
- Captured implicitly
  - [=] captured by value
  - [&] captured by reference
- Mixed
  - [&, x]captured by reference implicitly, but x by value
  - [=, &y] captured by value implicitly, but y by reference

```
int x = 0; int y = 42;
auto qqq = [x, &y] {
  std::cout << "x: " << x << std::endl;
  std::cout << "y: " << y << std::endl;
  ++y; ++x;
};
qqq(); // x==0, y==43
```

```
int x = 0; int y = 42;
auto qqq = [x, &y]() mutable {
++y; ++x;
};
qqq(); // x==0, y==43
```

C++ Lambda Evolution

- C++ 2014 – N3797
  - Capture: & | = | id [*init*] | & id [*init*] | this
  - Introduce capture initializer

```
int x = 4;
auto y = [&r = x, x = x+1]()->int {
    r += 2;
    return x+2;
  }(); // Updates ::x to 6, and initializes y to 7.
```

C++ Lambda Evolution

- C++ 2017 – N4659
  - Capture: add *this P0018r3 (2016.3.4)

Issue: Lambda expressions cannot capture `*this` by value

Lambda expressions declared within a non-static member function explicilty or implicitly captures the `this` pointer to access to member variables of `this`. Both capture-by-reference `[&]` and capture-by-value `[=]` *capture-defaults* implicitly capture the `this` pointer, therefore member variables are always accessed by reference via `this`. Thus the capture-default has no effect on the capture of `this`.

```
struct S {
        int x ;
        void f() {
                // The following lambda captures are currently identical
                auto a = [&]() { x = 42 ; } // OK: transformed to (*this).x
                auto b = [=]() { x = 43 ; } // OK: transformed to (*this).x
                a();
                assert( x == 42 );
                b();
                assert( x == 43 );
        }
};
```

C++ Lambda Evolution

C++ 2017 – N4659

Capture: add *this P0018r3 (2016.3.4)

Asynchronous dispatch of closures is a cornerstone of parallelism and concurrency. When a lambda is asynchronously dispatched from within a non-static member function, via `std::async` or other concurrency / parallelism dispatch mechanism the `*this` object *cannot* be captured by value. Thus when the `std::future` (or other handle) to the dispatched lambda outlives the originating class the lambda's captured `this` pointer is invalid.

```
class Work {
private:
        int value ;
public:
        Work() : value(42) {}
        std::future spawn()
        { return std::async( [=]()->int{ return value ; }); }
};

std::future foo()
{
        Work tmp ;
        return tmp.spawn();
        // The closure associated with the returned future
        // has an implicit this pointer that is invalid.
}

int main()
{
        std::future f = foo();
        f.wait();
        // The following fails due to the
        // originating class having been destroyed
        assert( 42 == f.get() );
        return 0 ;
}
```

C++ Lambda Evolution

C++ 2017 – N4659

Capture: add *this P0018r3 (2016.3.4)

Asynchronous dispatch of closures is a cornerstone of parallelism and concurrency. When a lambda is asynchronously dispatched from within a non-static member function, via `std::async` or other concurrency / parallelism dispatch mechanism the `*this` object *cannot* be captured by value. Thus when the `std::future` (or other handle) to the dispatched lambda outlives the originating class the lambda's captured `this` pointer is invalid.

```
class Work {
private:
        int value ;
public:
        Work() : value(42) {}
        std::future spawn()
        { return std::async( [=]()->int{ return value ; }); }
};

std::future foo()
{
        Work tmp ;
        return tmp.spawn();
        // The closure associated with the returned future
        // has an implicit this pointer that is invalid.
}

int main()
{
        std::future f = foo();
        f.wait();
        // The following fails due to the
        // originating class having been destroyed
        assert( 42 == f.get() );
        return 0 ;
}
```

C++ Lambda Evolution

C++ 2017 – N4659

Capture: add *this P0018r3 (2016.3.4)

Current and future hardware architectures specifically targeting parallelism and concurrency have heterogeneous memory systems. For example, NUMA regions, attached accelerator memory, and processing-in-memory (PIM) stacks. In these architectures it will often result in significantly improved performance if the closure is copied to the data upon which it operates, as opposed to moving the data to and from the closure.

通过复制闭包，而不是从闭包移入或移出，来改善性能

For example, parallel execution of a closure on large data spanning NUMA regions will be more performant if a copy of that closure residing in the same NUMA region acts upon that data. If a full (self-contained) capture-by-value lambda closure were given to a parallel dispatch, such as in the parallelism technical specification, then the library could create copies of that closure within each NUMA region to improve data locality for the parallel computation. For another example, a closure dispatched to an attached accelerator with separate memory must be copied to the accelerator's memory before execution can occur. Thus current and future architectures *require* the capability to copy closures to data.

C++ Lambda Evolution

- C++ 2017 – N4659
  - Capture: add *this P0018r3 (2016.3.4)

Error-prone and onerous work-around: `[=, tmp=*this]`

A potential work-around for this deficiency is to explicitly capture a copy the originating class.

```
class Work {
private:
        int value ;
public:
        Work() : value(42) {}
        std::future spawn()
        {
                return std::async( [=,tmp=*this]()->int{ return tmp.value ; });
        }
};
```

```
class Work {
public:
        void do_something() const {
                // for ( int i = 0 ; i < N ; ++i )
                foreach( Parallel , 0 , N , [=,tmp=*this]( int i )
                {
                        // A modestly long loop body where
                        // every reference to a member must be modified
                        // for qualification with 'tmp.'
                        // Any mistaken omissions will silently fail
                        // as references via 'this->'.
                }
                );
        }
};
```

繁重的
必须修改代码中成员的每个引用以添加 tmp.资格

Capture *this


```
class Task {
private:
  double a; double b;
public:
  Task() : a(0.0), b(0.0) {}
  ~Task() { a = std::nan(""); b = std::nan(""); }
  void set_a(double a) {this.a = a; }
  void set_b(double b) {this.b = b; }
  std::future<double> run() {
    return std::async(std::launch::async, [=, *this]() {
      return std::sqrt(a* a + b * b);
    });
  }
};
```

```
int main() {
  Task t;
  t.set_a(5.0);
  t.set_b(10.0);
  std::cout <<t.run().get();
}
```

May run the lambda expression after the task object is destructed

Captured the object pointed by this by value

C++ Core Guidelines

- lambda expression: when to use
  - F.50: Use a lambda when a function won't do (to capture local variables, or to write a local function)
  - F.52: Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms
  - F.53: Avoid capturing by reference in lambdas that will be used non-locally, including returned, stored on the heap, or passed to another thread
  - ES.28: Use lambdas for complex initialization, especially of const variables

- **C.170: If you feel like overloading a lambda, use a generic lambda**

```
void f(int);
void f(double);
auto f = [](char); // error: cannot overload variable and function
auto g = [](int) { /* ... */ };
auto g = [](double) { /* ... */ }; // error: cannot overload variables
auto h = [](auto) { /* ... */ }; // OK
```

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

Lambda Expression in Python

4.7.6. Lambda Expressions

- Small anonymous functions can be created with the lambda keyword.

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)  42
>>> f(1)  43
```

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

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

Others

LAMBDA: The ultimate Excel worksheet function

- =LAMBDA(X, Y, SQRT(X*X+Y*Y))

- =LAMBDA(X, Y, LET(XS, X*X, YS, Y*Y, SQRT(XS+YS)))

	A	B	C	D	E
1	HEAD	=LAMBDA(str, IF(str="","error: HEAD of empty string", LEFT(str, 1)))			
2	TAIL	=LAMBDA(str, IF(str="","error: TAIL of empty string", RIGHT(str, LEN(str)-1)))			
3	REVERSE	=LAMBDA(str, IF(str="", "", REVERSE(TAIL(str)) & HEAD(str))))			
4					
5	12345	1	2345	54321	12345
6		=HEAD(A5)	=TAIL(A5)	=REVERSE(A5)	=REVERSE(D5)