

# Types and Programming Language

枫聆

2022 年 5 月 14 日

## 目录

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Untyped Systems</b>	<b>4</b>
2.1	Syntax . . . . .	4
2.2	Induction . . . . .	5
2.3	Semantic Styles . . . . .	6
2.4	Evaluation . . . . .	7
2.5	The Untyped Lambda-Calculus . . . . .	8
2.6	Programming in the Lambda-Calculus . . . . .	11
2.7	Normal Forms . . . . .	15
2.8	Recursion . . . . .	16
<b>3</b>	<b>Simple Types</b>	<b>18</b>
3.1	Typed Arithmetic Expressions . . . . .	18
3.2	Simply Typed Lambda-Calculus . . . . .	20
<b>4</b>	<b>Type Extensions</b>	<b>23</b>
4.1	Known Types . . . . .	23
4.2	Known Features . . . . .	25
4.3	Normalization . . . . .	27
4.4	References . . . . .	30
4.5	Recursion . . . . .	32

<b>5</b>	<b>Subtyping</b>	<b>34</b>
5.1	STLC . . . . .	34
5.2	Properties of Subtyping and Typing . . . . .	36
5.3	Bottom Type . . . . .	38
5.4	Coercion Semantics . . . . .	39
<b>6</b>	<b>Type Reconstruction</b>	<b>41</b>
6.1	Type Inference . . . . .	41
6.2	Let Polymorphism . . . . .	45

## Introduction

**Definition 1** A **type system** is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of value they compute.

type system 是一种用于证明某些确定的程序行为不会发生的方法，它怎么做呢？通过它们计算出值的类型来分类，有点抽象... 我想知道 the kinds of value they compute 是什么？如何分类？分类之后接下来该怎么做？

**Annotation 2** Being static, type systems are necessarily also **conservative**: they can categorically prove the absence of some bad program behaviors, but they can't prove their presence.

### Example 3

```
1 if <complex test> then 5 else <type error>
```

上面这个 annotation 在说 type system 只能证明它看到的一些 bad program behavior 不会出现，但是它们可能会 reject 掉一些 runtime time 阶段运行良好的程序，例如在 runtime 阶段上面的 else 可能永远都不会进。即 type system 无法证明它是否真的存在。

## Untyped Systems

### Syntax

**Definition 4** The set of terms is the smallest set  $\mathcal{T}$  such that

1.  $\{\text{true}, \text{false}, 0\} \subseteq \mathcal{T}$ ;
2. if  $t_1 \in \mathcal{T}$ , then  $\{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1\} \subseteq \mathcal{T}$ ;
3. if  $t_1 \in \mathcal{T}, t_2 \in \mathcal{T}, t_3 \in \mathcal{T}$ , then  $\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in \mathcal{T}$ .

**Definition 5** The set of terms is defined by the following rules:

$$\frac{\text{true} \in \mathcal{T}}{t_1 \mathcal{T}} \quad \frac{\text{false} \in \mathcal{T}}{t_1 \mathcal{T}} \quad \frac{0 \in \mathcal{T}}{t_1 \mathcal{T}} \quad \frac{\text{succ } t_1 \in \mathcal{T}}{t_1 \mathcal{T}} \quad \frac{\text{pred } t_1 \in \mathcal{T}}{t_1 \mathcal{T}} \quad \frac{\text{iszero } t_1 \in \mathcal{T}}{t_1 \mathcal{T}} \quad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T} \quad t_3 \in \mathcal{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3}$$

**Definition 6** For each natural number  $i$ , define a  $S(X)$  as follow:

$$\begin{aligned} S_0(X) &= X \\ S_1(X) &= \{\text{succ } t, \text{pred } t, \text{iszero } t \mid t \in X\} \cup \{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in X\} \\ &\vdots \\ S_{i+1}(X) &= S(S_i(X)). \end{aligned}$$

**Proposition 7**  $\mathcal{T} = \bigcup_{i=0}^{\omega} S_i(\{\text{true}, \text{false}, 0\})$ .

PROOF 我们设  $\bigcup_{i=0}^{\omega} S_i(\{\text{true}, \text{false}, 0\}) = S$  和  $\{\text{true}, \text{false}, 0\} = T$ , 证明过程分两步走 (1)  $S$  follow Definition2.1 (2)  $S$  is smallest.

proof (1).  $\{\text{true}, \text{false}, 0\} \in S$  这是显然的. 若  $t_1 \in S$ , 那么  $t_1 \in S_i(T)$ , 考虑  $\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1 \in S_{i+1}(T)$ . 同理 Definition2.1(3).

proof (2). 考虑任意 follow Definition2.1 的集合  $S'$ , 我们需要证明  $S \subseteq S'$ . 我们考虑任意的  $S_i \subseteq S$ , 若都有  $S_i \subseteq S'$ , 那么则有  $S \subseteq S'$ . 这里我们使用 induction 来证明, 首先有  $S_0(T) \subseteq S'$ , 假设  $S_n(T) \subseteq S'$ . 那么考虑  $S_{n+1}(T) = S(S_n(T))$ , 任意的  $t_1, t_2, t_3 \in S_n(T)$ , 那么 Definition2.1(1)(2)(3) 得到的结果都是属于  $S'$ , 因此  $S_{n+1}(T) \subseteq S'$ . Q. E. D.

**Definition 8** The **depth** of a term  $t$  is the smallest  $i$  such that  $t \in S_i(X)$ .

**Definition 9** If a term  $t \in S_i(X)$ , then all of its **immediate subterms** must be in  $S_{i-1}(X)$ .

**Theorem 10** **Structural induction** Suppose  $P$  is a predicate on terms. If for each term  $s$ , given  $P(r)$  for all immediate subterms  $r$  of  $s$ , we can show  $P(s)$ , then  $P(s)$  holds for all  $s$ .

## **Induction**

## Semantic Styles

**Annotation 11** 有三种方法来形式化语义:

1. Operational semantics(操作语义) 定义程序是如何运行的? 所以你需要一个 abstract machine 来帮助解释, 之所以 abstract 是因为它里面的 machine code 就是 the term of language. 其中又分为两种类型, big-step 和 small-step.
2. Denotational semantics(指称语义) 就是给定一个 semantic domain 和一个 interpretation function, 通过一个 function 把 term 映射到 semantic domain 里面, 这个 domain 里面可能是一堆数学对象. 它的优势是对求值进行抽象, 突出语言的本质. 我们可以在 semantic domain 里面做运算, 只要 interpretation function 建立的好, 运算结果可以表征程序本身的性质.
3. Axiomatic semantics(公理语义) 拿 axioms 堆起来的程序? 类似 Hoare logic.
4. Algebraic semantics(代数语义) 把程序本身映射到某个代数结构上, 转而研究这个代数?

## Evaluation

**Annotation 12** 这一章在讲 operational semantic of boolean expression, 这个过程会清晰的告诉你我们求值的结果是什么? 当我们对 term 求值时, term 之间的转换规则应该是什么? 既然有了转换, 那么一定有终止的时候, 这个终止的时刻就是我们求值的结果, 那我们要问什么时候停止呢? 开头的表格告诉了关于前面这些问题的答案. 当然有一些东西也没有出现在表格里面, 但是它们同样重要, 例如不能在对 false, true, 0 这些东西再求值; 求值的顺序等等.

**Definition 13** An instance of an inference rule is obtained by consistently replacing each metavariable by the same term in the rule' s conclusion and all its premises (if any).

一个推导规则的实例, 就是把里面的 metavariable 替换成具体的 terms, 但是一定需要注意对应关系.

**Definition 14** Evaluation relations: 一步求值 (基本 evaluation relation); 多步求值 (evaluation relation 的传递闭包产生的新的 relation, 这个 relation 包含原来的所有 evaluation relation);

**Definition 15** A term  $t$  is in normal form if no evaluation rule applies to it.

范式是一个 term 无法继续求值的状态.

**Definition 16** A closed term is stuck if it is in normal form but not a value, we often call it neutral form.

受阻项是一种特殊的范式, 这个范式不是一个合法的值.

## The Untyped Lambda-Calculus

**Annotation 17** 过程抽象 Procedural (or functional) abstraction is a key feature of essentially all programming languages

**Definition 18**  $\lambda$  演算的定义 The lambda-calculus (or  $\lambda$ -calculus) embodies this kind of function definition and application in the purest possible form. In the lambda-calculus everything is a function: the arguments accepted by functions are themselves functions and the result returned by a function is another function.

The syntax of the lambda-calculus comprises just three sorts of terms.

$$\begin{aligned} t ::= & \\ & x \\ & \lambda x. t \\ & t t. \end{aligned}$$

A variable  $x$  by itself is a term; the abstraction of a variable  $x$  from a term  $t_1$ , written  $\lambda x. t_1$ , is a term; and the application of a term  $t_1$  to another term  $t_2$ , written  $t_1 t_2$ , is a term.

在 pure lambda-calculus 里面所有的 terms 都是函数, 第一个 term 表示变量, 第二个 term 表示 abstraction, 第三个 term 表示 application. 言下之意一个 lambda 函数的参数和返回值也都是函数.

**Definition 19** 两个重要的约定 First, application associates to the left, means

$$s t u = (s t) u.$$

Second, the bodies of abstractions are taken to extend as far to the right as possible.

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

第一个是说函数的 apply 操作是左结合, 第二是说 lambda 函数的抽象体尽量向右扩展.

**Definition 20** 作用域 scope An occurrence of the variable  $x$  is said to be **bound** when it occurs in the body  $t$  of an abstraction  $\lambda x. t$ . (More precisely, it is bound by this abstraction. Equivalently, we can say that  $\lambda x$  is a binder whose scope is  $t$ .) An occurrence of  $x$  is **free** if it appears in a position where it is not bound by an enclosing abstraction on  $x$ . i.e.  $x$  in  $\lambda y. x y$  and  $x y$  are free.

A term with no free variables is said to be **closed**; closed terms are also called **combinators**. The simplest combinator, called the identity function,

$$\text{id} = \lambda x. x.$$



**Definition 21  $\alpha$  等价** A basic form of equivalence, definable on lambda terms, is alpha equivalence. It captures the intuition that the particular choice of a bound variable, in an abstraction, does not (usually) matter.

$$\lambda x. x \cong \lambda y. y$$

简而言之，同时对一个 lambda 函数替换所有 bound variable 得到的 term 是等价的,  $\alpha$  变换在进行  $\beta$  规约的时候，用于解决变量名冲突特别有用）。

**Definition 22 操作语义** Each step in the computation consists of rewriting an application whose left-hand component is an abstraction, by substituting the right-hand component for the bound variable in the abstraction's body. Graphically, we write

$$(\lambda x. t_{12}) t_2 \rightarrow [x \mapsto t_2] t_{12},$$

where  $[x \mapsto t_2]$  means "the term obtained by replacing all free occurrences of  $x$  in  $t_{12}$  by  $t_2$ ".

**Definition 23 可约表达式** A term of the form  $(\lambda x. t_{12}) t_2$  is called **redex** (reducible expression), and the operation of rewriting a redex according to the above rule is called  **$\beta$ -reduction**.

**Definition 24 几种规约策略** Each strategy defines which redex or redexes in a term can fire on the next step of evaluation.

1. Undering **full  $\beta$ -reduction**, any redex may be reduced at any time. i.e., consider the term

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

we can write more readably as  $\text{id} (\text{id} (\lambda z. \text{id } z))$ . This term contains three redexes:

$$\begin{array}{c} \text{id } (\text{id } (\lambda z. \text{id } z)) \\ \text{id } (\text{id } (\lambda z. \text{id } z)) \\ \text{id } (\text{id } (\lambda z. \text{id } z)) \end{array}$$

under full  $\beta$ -reduction, we might choose, for example, to begin with the innermost index, then do the one in the middle, then the outermost:

$$\begin{array}{l} \text{id } (\text{id } (\lambda z. \text{id } z)) \\ \rightarrow \text{id } (\text{id } (\lambda z. z)) \\ \rightarrow \text{id } (\lambda z. z) \\ \rightarrow \lambda z. z \\ \rightarrow \end{array}$$

2. Undering the **normal order** strategy, the leftmost, outermost redex is always reduced first. Under this strategy, the term above would be reduced as follows

$$\begin{aligned}
 & \underline{\text{id} (\text{id} (\lambda z. \text{id } z))} \\
 \rightarrow & \underline{\text{id} (\lambda z. \text{id } z)} \\
 \rightarrow & \lambda z. \underline{\text{id } z} \\
 \rightarrow & \lambda z. z \\
 \rightarrow & 
 \end{aligned}$$

3. The **call by name** strategy is yet more restrictive, allowing no reductions inside abstractions.

$$\begin{aligned}
 & \underline{\text{id} (\text{id} (\lambda z. \text{id } z))} \\
 \rightarrow & \underline{\text{id} (\lambda z. \text{id } z)} \\
 \rightarrow & \lambda z. \text{id } z \\
 \rightarrow & 
 \end{aligned}$$

4. Most languages use a **call by value** strategy, in which only outermost redexes are reduced and where a redex is reduced only when its right-hand side has already been reduced to a value—a term that is finished computation and cannot be reduced and further.

$$\begin{aligned}
 & \underline{\text{id} (\text{id} (\lambda z. \text{id } z))} \\
 \rightarrow & \underline{\text{id} (\lambda z. \text{id } z)} \\
 \rightarrow & \lambda z. \text{id } z \\
 \rightarrow & 
 \end{aligned}$$

注意 call by name 和 call by value 的区别, call by name 是在  $\lambda$  函数调用前不对参数进行规约而直接替换到函数 body 内, 换言之如果一个参数不会被用到, 那么它永远都不会被 evaluated, call by value 是其对立情况, 先对参数进行规约.

Evaluation strategies are used by programming languages to determine two things—when to evaluate the arguments of a function call and what kind of value to pass to the function.

**Definition 25** Given lambda abstraction  $\lambda x. e$ , then the  **$\eta$ -conversion** of it as follow

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

where  $y$  is not in  $e$ .

## Programming in the Lambda-Calculus

**Definition 26** 高阶函数 A higher order function is a function that takes a function as an argument, or returns a function.

$$f^{\circ n} = \underbrace{f \circ f \circ \dots \circ f}_{n \text{ times}}.$$

**Annotation 27** Define  $\circ$  itself as a function:

$$\circ = \lambda f. \lambda g. \lambda x. f(g(x)).$$

So function composition can be denoted by

$$\circ f g = \lambda x. f(g(x)).$$

非常漂亮.

**Annotation 28** 多参数柯里化 Motivation is that the lambda-calculus provides no built-in support for multi-argument functions. The solution here is higher-order functions.

Instead of writing  $f = \lambda(x, y). s$ , as we might in a richer programming language, we write  $f = \lambda x. \lambda y. s$ . we then apply  $f$  to its arguments one at a time, write  $f v w$ , which reduces to

$$f v w \rightarrow \lambda y. [x \mapsto v] s \rightarrow [x \mapsto v] [y \mapsto w] s.$$

This transformation of multi-arguments function into higher-order function is called **currying** in honor of Haskell Curry, a contemporary of Church.

**Annotation 29** Church 形式的布尔代数 Define the terms **tru** and **fls** as follows:

$$\text{tru} = \lambda t. \lambda f. t$$

$$\text{fls} = \lambda t. \lambda f. f$$

The terms **tru** and **fls** can be viewed as representing the boolean values “true” and “false,” then define a combinator **test** with the property that  $\text{test } b v w$  reduces to  $v$  when  $b$  is **tru** and reduces to  $w$  when  $b$  is **fls**.

$$\text{test} = \lambda l. \lambda m. \lambda n. l m n;$$

The **test** combinator does not actually do much:  $\text{test } b v w$  reduces to  $b v w$ . i.e., the term  $\text{test } \text{tru } v w$  reduces

as follows:

$$\begin{aligned}
& \text{test } \text{tru } v \ w \\
& = \text{tru } v \ w \\
& \rightarrow (\lambda t. \ \lambda f. \ t) \ v \ w \\
& \rightarrow (\lambda f. \ v) \ w \\
& \rightarrow v.
\end{aligned}$$

We can also define boolean operator like logical conjunction as functions:

$$\text{and} = \lambda b. \ \lambda c. \ b \ c \ \text{fls} = \lambda b. \ \lambda c. \ b \ c \ b$$

Define logical **or** and **not** as follows:

$$\begin{aligned}
\text{or} &= \lambda b. \ \lambda c. \ b \ \text{tru } c = \lambda b. \ \lambda c. \ b \ b \ c \\
\text{not} &= \lambda b. \ b \ \text{fls } \text{tru} \\
\text{xor} &= \lambda b. \ \lambda c. \ b \ (\text{not } c) \ c
\end{aligned}$$

$$\begin{aligned}
\text{tru} &= \lambda t. \ \lambda f. \ t \\
\text{xor} &= \lambda a. \ \lambda b. \ a \ (\text{not } b) \ b \\
\text{xor } \text{tru } b &= \text{tru } (\text{not } b) \ b \\
&= \text{not } b
\end{aligned}$$

**Annotation 30** 有序对 Using booleans, we can encode pairs of values as terms.

$$\begin{aligned}
\text{pair} &= \lambda f. \ \lambda s. \ \lambda b. \ b \ f \ s \\
\text{fst} &= \lambda p. \ p \ \text{tru} \\
\text{snd} &= \lambda p. \ p \ \text{fls}
\end{aligned}$$

$\text{pair}$  变成了一个函数，它可以接收一个  $\text{tru}$  或者  $\text{fls}$  来返回第一个值或者第二个值， $\text{fst}$  和  $\text{snd}$  就是  $\text{pair}$  的一个 applying 过程，比较有趣.

**Annotation 31** Church 形式的序数 Define the Church numerals as follows

$$\begin{aligned}
c_0 &= \lambda s. \ \lambda z. \ z \\
c_1 &= \lambda s. \ \lambda z. \ s \ z \\
c_2 &= \lambda s. \ \lambda z. \ s \ (s \ z) \\
c_3 &= \lambda s. \ \lambda z. \ s \ (s \ (s \ z)) \\
&\dots
\end{aligned}$$

这里我们使用高阶函数来描述这一性质

Number	Function definition	Lambda expression
0	$0\ f\ x = x$	$0 = \lambda f. \lambda x. x$
1	$1\ f\ x = f\ x$	$1 = \lambda f. \lambda x. f\ x$
2	$2\ f\ x = f\ (f\ x)$	$2 = \lambda f. \lambda x. f\ (f\ x)$
3	$3\ f\ x = f\ (f\ (f\ x))$	$3 = \lambda f. \lambda x. f\ (f\ (f\ x))$
$\vdots$	$\vdots$	$\vdots$
n	$n\ f\ x = f^n\ x$	$n = \lambda f. \lambda x. f^{\circ n}\ x$

参考皮亚诺公理，对应这里我们构建自然数需要有一个 0 和一个后继函数  $f$ 。你会注意到  $c_0$  和 **fls** 是同一个 term，常规编程语言里面很多情况下 0 和 false 确实也是一个东西。

**Annotation 32 Church 形式序数的运算符** We can define the successor function on Church numerals as follows

$$\text{succ} = \lambda n. \lambda s. \lambda z. s\ (n\ s\ z)$$

注意这里的后继函数接受对象是一个 Church numeral，从而返回新的 Church numeral，和我们构造 Church number 中的后继不是一个东西，它的作用就是让对应具体的数再复合一次  $f$ 。因此分解一下上面的 apply 过程，首先是  $(n\ s\ z)$  得到相对应的数，然后在对它复合一次  $f$ 。

另外一种形式

$$\text{succ} = \lambda n. \lambda s. \lambda z. n\ s\ (s\ z)$$

这个方式也很巧妙，相当于把  $0' = 0 + 1$  作为新的零元。

**Annotation 33** The addition of Church numerals can be preformed by a term **plus** that takes two Church numerals  $m$  and  $n$ , as arguments, and yields another Church numeral.

$$\text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m\ s\ (n\ s\ z)$$

这里遵循函数复合的结合律  $f^{\circ(m+n)}(z) = f^{\circ m}(f^{\circ n}(z))$ ，相对于把其中的一个 Church number 对应的具体数当做了另一个 Church numeral 的 zero。

**Annotation 34**

$$\text{times} = \lambda m. \lambda n. m\ (\text{plus}\ n)\ c_0$$

这个就非常有趣了，这里先固定  $m$ ，把它 succ 设为 plus  $n$  和 zero 设为  $c_0$ ，相当于  $(\text{plus}\ n)^m(c_0)$ 。

另一种更简洁的形式：

$$\text{times} = \lambda m. \lambda n. \lambda s. \lambda z. m\ (n\ s)\ z$$

这里的  $(n\ s)$  变成了一个特殊 abstraction  $s^{\circ n} = \lambda z. s(s(\dots(s\ z)\dots))$ ，它并不是一个标准的 succ 形式

### Annotation 35

$$\text{exp} = \lambda m. \lambda n. n m$$

推一个来看看，注意其中的几次  $\alpha$  变换，避免产生变量名的冲突.

$$\begin{aligned}
 \text{exp } c_3 \ c_2 &= c_2 \ c_3 \\
 &= (\lambda s. \lambda z. s \ (s \ z)) \ c_3 \\
 &= \lambda z. \ c_3 \ (c_3 \ z) \\
 \rightsquigarrow_{\alpha} &= \lambda z. \ (\lambda f. \lambda x. f \ (f \ (f \ x))) \ ((\lambda f. \lambda x. f \ (f \ (f \ x))) \ z) \\
 &= \lambda z. \ (\lambda f. \lambda x. f \ (f \ (f \ x))) \ (\lambda x. \ z \ (z \ (z \ x))) \\
 \rightsquigarrow_{\alpha} &= \lambda z. \ (\lambda f. \lambda x. f \ (f \ (f \ x))) \ (\lambda g. \ z \ (z \ (z \ g))) \\
 &= \lambda z. \ \lambda x. \ (\lambda g. \ z \ (z \ (z \ g))) \ ((\lambda g. \ z \ (z \ (z \ g))) \ ((\lambda g. \ z \ (z \ (z \ g))) \ x)) \\
 &= \lambda z. \ \lambda x. \ (\lambda g. \ z \ (z \ (z \ g))) \ ((\lambda g. \ z \ (z \ (z \ g))) \ (z \ z \ z \ x)) \\
 &= \lambda z. \ \lambda x. \ (\lambda g. \ z \ (z \ (z \ g))) \ (z \ z \ z \ z \ z \ z \ z \ x) \\
 &= \lambda z. \ \lambda x. \ z \ z \ z \ z \ z \ z \ z \ z \ z \ x \\
 &= \lambda s. \lambda z. \ s \ s \ s \ s \ s \ s \ s \ s \ s \ z \\
 &= c_9
 \end{aligned}$$

## Normal Forms

**Annotation 36** 前面提到的 neutral term-”neutral terms contain a free variable at a ‘head’ position”, 它是对 normal form 更细致的一种刻画, 形如  $x y$ , 其中  $x$  是一个 free variable, 而  $y$  是一个 lambda abstraction.

**Definition 37** In untyped lambda calculus, the neutral terms and the normal form are generated in the following rules.

$$\frac{t \text{ nf}}{\lambda x. t \text{ nf}} \quad \frac{t \text{ ne}}{t \text{ nf}} \quad \frac{t_1 \text{ ne} \quad t_2 \text{ nf}}{t_1 t_2 \text{ ne}} \quad \frac{}{x \text{ ne}}$$

**Annotation 38** 定义上述 normal form 的 generator 本想是根据它们来证明一些依赖 normal form 的命题, 例如 false 和 true 的刻画”if  $\vdash e : \alpha \rightarrow (\alpha \rightarrow \alpha)$  and  $e$  is normal form, then  $e = \text{true}$  or  $e = \text{false}$ ”, 对  $e$  使用 normal form structure induction, 仅仅使用上面第一个 inference rule, 实际上就可以了. 注意 normal form 的定义并不依赖 type system, 显然 neutral term 这种东西在 STLC 根本不可能出现...

## Recursion

**Annotation 39** 首先经历几个思考 recursion 历程. 首先给出一个标准 recursion 过程, 阶乘过程

$$\text{fact } \bar{n} = \text{if } (\text{iszero } \bar{n})(\bar{1}) \text{ then } (\text{mult } \bar{n} (\text{fact } (\text{pred } \bar{n}))).$$

其中  $\bar{n}$  代表 church number. 因此 fact 表示的 abstraction 为

$$\text{fact} = \lambda n. \text{if } (\text{iszero } n)(\bar{1}) \text{ then } (\text{mult } n (\text{fact } (\text{pred } n)))$$

这里我们只有一个等式, 那么我们应当如何准确地给出 fact 的定义呢? 如果我们尝试给左边的 fact 提出来, 即

$$\text{fact} = (\lambda f. \lambda n. \text{if } (\text{iszero } n)(\bar{1}) \text{ then } (\text{mult } n (f(\text{pred } n)))) \text{ fact}$$

将左边括号里面的 abstraction 看做一个函数, 那么 fact 实际上就是它的一个 fixed point. 我们的下一个目标就是想办法构造出这个函数的不动点, 如果这个不动点正好就是我们需要的 fact 那就太好了.

**Definition 40** A self-applicatin abstraction is

$$\omega = \lambda x. x x.$$

The divergent omega combinator is

$$\Omega = \omega\omega = (\lambda x. x x) (\lambda x. x x).$$

**Definition 41** A call-by-name Y combinator is

$$\mathbf{Y} = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)).$$

**Theorem 42** For all abstractions  $F \in \Lambda$ , we have

$$\mathbf{Y}F = F(\mathbf{Y}F)$$

PROOF 只需要对  $\mathbf{Y}F$  做两次 application 即可

$$\begin{aligned} \mathbf{Y}F &= (\lambda x. F (x x)) (\lambda x. F (x x)) \\ &= F((\lambda x. F (x x)) (\lambda x. F (x x))) \\ &= F(\mathbf{Y}F). \end{aligned}$$



**Annotation 43** 我们将使用 **Y** combinator 继续 fact 的 definition, 我们使

$$F = (\lambda f. \lambda n. \text{if } (\text{iszero } n)(\bar{1}) \text{ then } (\text{mult } n (f(\text{pred } n)))).$$

由前述的 theorem, 现在我们有一个 fixed point  $\mathbf{Y}F$ . 我们使  $\text{fact} = \mathbf{Y}F$ , 显然有

$$\begin{aligned} \text{fact } n &= \mathbf{Y}F \ n \\ &= F(\mathbf{Y}F) \ n \\ &= \text{if } (\text{iszero } n)(\bar{1}) \text{ then } (\text{mult } n (\mathbf{Y}F(\text{pred } n))) \\ &= \text{if } (\text{iszero } n)(\bar{1}) \text{ then } (\text{mult } n (\text{fact}(\text{pred } n))) \end{aligned}$$

当我们思考  $\mathbf{Y}F$  是不是我们想要的那个 fixed point 呢? 实际上它就是 fact 的准确定义. 这个问题我们这样思考: 对于每一个 church number  $\bar{n}$ , fact 都有对应的形式  $\text{fact}_{\bar{n}}$ , 展开  $\mathbf{Y}F\bar{n} = F^n(\mathbf{Y}F)\bar{n}$  也可以得到相同的结果. 那么整个定义就是 well-defined.

**Example 44** 如果  $F = \lambda x. x$ , 我们可以得到什么东西呢? 所有 term  $x$  都是其 fixed point. 即

$$x = \lambda x. x \Rightarrow x = x$$

Nothing we have done.

## Simple Types

### Typed Arithmetic Expressions

**Definition 45** The typing relation for arithmetic expressions, written

$$t : T$$

is defined by a set of inference rules assigning types to terms.

$$\begin{array}{c} \text{true} : \text{bool} \\ \text{false} : \text{bool} \\ \frac{t_1 : \text{bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \\ 0 : \text{nat} \\ \frac{t_1 : \text{nat}}{\text{succ } t_1 : \text{nat}} \\ \frac{t_1 : \text{nat}}{\text{pred } t_1 : \text{nat}} \\ \frac{t_1 : \text{nat}}{\text{iszero } t_1 : \text{bool}} \end{array}$$

**Annotation 46** 注意分支 terms 中的  $T$  表示任意的 types 即可能包括 `bool` 和 `nat`. 理论上两个分支的表达式的 type 可以不一样, 但是这一样以来似乎就不是 well-typed, 处理这样的情况需要等到我们学习更多的类型的 type 之后才能来重新构造.

**Annotation 47** 使用 inference rule 来描述 type 是为了更方便地证明 inductive theorem.

**Definition 48** A term  $t$  is **typable or well typed** if there is some  $T$  such that  $t : T$ . If  $t$  is typable, then its type is unique(**uniqueness of types**).

**Annotation 49** 这里很重要是理解如果给定一个 type relation  $t : T$ , 那么肯定是由上述 inference rule 推导出来的, 所以我们会经常看到从 conclude 推 premise 的过程, 也就是寻找合适的 inference rule 反向推导, 这个过程我们称其为 **derivation**, 其中反向寻找合适的 inference rule 的方法是利用了所谓 inversion lemma.

**Theorem 50** **progress** A well-typed term is not stuck.

PROOF 我们利用 structural induction 来证一下 progress. 首先基本的 terms `false`, `true`, `0`, `succ nv` 都是明显的 values, 其中 `nv` 表示一个 numeric value.

*Case 1*  $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$  subterm :  $t_1 = \text{bool}$   $t_2 = T$   $t_3 = T$ .

由归纳假设 subterm are not stuck term, 当  $t_1 = \text{true}$  或者  $t_1 = \text{false}$  时, 我们对  $t$  一步 evaluation 得到  $t_2$  或者  $t_3$ . 另外当  $t_1 \rightarrow t'_1$  时, 我们也可以得到  $t \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$ .

*Case 2*  $t = \text{succ } t_1$  subterm :  $t_1 = \text{nat}$ .

由归纳假设当  $t_1 = \text{nv}$  时, 那么  $\text{succ } t_1$  还是一个 numeric value. 另外当  $t_1 \rightarrow t'_1$ , 我们也可以得到  $t \rightarrow \text{succ } t'_1$

*Case 3*  $t = \text{pred } t_1$   $t_1 = \text{nat}$ .

同上.

*Case 4*  $t = \text{iszero } t_1$   $t_1 = \text{nat}$ .

同上.

**Annotation 51** 换言之 progress 保证是任意一个 well-typed term, 它可能是一个 value 或者可以进一步根据 evaluation rules 推导.

**Theorem 52** **preservation** If a well-typed term takes a step of evaluation, then the resulting term is also well typed.

PROOF 和证明 progress 类似, 对于每个 case 需要分析其所有的 one step, 然后根据 typing rules, 证明 type perservation. i.e. representative case.

*Case 1*  $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$  subterm :  $t_1 = \text{bool}$   $t_2 = T$   $t_3 = T$ .

当  $t_1 = \text{true}$  或者  $t_1 = \text{false}$  时, 我们对  $t$  一步 evaluation 得到  $t' = t_2$  或者  $t' = t_3$ , 显然  $t'$  with  $T$ . 另外当我们 take  $t_1 \rightarrow t'_1$  时, 此时对 subterm  $t_1$  可以 apply induction hypothesis 得到  $t'_1 : \text{bool}$ , 因此  $t' = \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$ , 此时依然有  $t'$  with  $T$ .

**Definition 53**

safty = progress + preservation.

## Simply Typed Lambda-Calculus

**Definition 54** Define the type of  $\lambda$ -abstraction(function) as follow

$$\lambda x. t : T_1 \rightarrow T_2$$

it classifies function that expect argument of type  $T_1$  and return result of type  $T_2$ . The type constructor  $\rightarrow$  is right-associative.

**Annotation 55** 试想我们应该怎样给一个 function 赋予一个 type 呢? 首先要解决是这个 function 需要的 argument 的 type 是怎样的? 这里自然地会想到两种方法, 一是直接给 argument 打上 annotation, 而是从 function body 推出 argument 的 type. 第一种 type annotation 通常称为 explicitly typed, 第二种则称其为 implicitly typed. 我们如果采用第一种方法, 假设给定  $x : T_1$ , 同时将  $t_2$  中的所有出现的  $x$  的 type 都表示为  $T_1$  得到  $x : T_2$ , 那么显然此时就可以构造出一个 abstraction 和它对应 type 为  $\lambda x. t_2 : T_1 \rightarrow T_2$ , 形式化的描述这个 type rule 即为

$$\frac{x : T_1 \vdash t_2 : T_2}{\lambda x. t_2 : T_1 \rightarrow T_2}$$

其中  $\vdash$  可以解释为 under, 即 obtain some type relations under some assumptions. 特别地  $\vdash x : T$  表示 assumptions 是空的.

**Definition 56** A typing context  $\Gamma$  is a sequence of distinct variables and thier types as follow

$$\Gamma = x_1 : T_1, x_2 : T_2, x_3 : T_3, \dots$$

**Annotation 57 rule of typing abstractions** 如果考虑上 nested abstraction 的情况, 我们扩展一下前面提到的 type inference

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x. t_2 : T_1 \rightarrow T_2.}$$

这里我们规定  $t_2$  中除  $x$  外的 free variables 均在  $\Gamma$  中.

**Annotation 58 rule of variables** A variable has whatever type we are currently assuming it to have,

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

**Annotation 59 rule of applications**

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}$$

**Annotation 60** rule of conditionals

$$\frac{\Gamma \vdash t_1 : \text{bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

**Annotation 61** We often use  $\lambda_{\rightarrow}$  to refer to the simply typed lambda-calculus.

**Theorem 62 uniqueness of types** In a given typing context  $\Gamma$ , a term  $t$  has at most one type. That is, if a term is typable, then its type is unique.

**Lemma 63 canonical forms**

1. If  $v$  is a value of type  $\text{bool}$ , then  $v$  is either `true` or `false`;
2. If  $v$  is a value of type  $T_1 \rightarrow T_2$ , then  $v = \lambda x : T_1. t_2$ .

**Lemma 64 weakening** If  $\Gamma \vdash t : T$  and  $x \notin \text{dom}(\Gamma)$ , then  $\Gamma, x : S \vdash t : T$ .

**Theorem 65 progress** Suppose  $t$  is a closed, well-typed term (that is  $\vdash t : T$ ). Then either  $t$  is a value or else there is some  $t'$  with  $t \rightarrow t'$ .

PROOF proved by structural induction.

Q. E. D.

**Theorem 66 preservation under substitution** If  $\Gamma, x : S \vdash t : T$  and  $\Gamma \rightarrow s : S$ , then  $\Gamma \vdash [x \rightarrow s]t : T$ .

PROOF 写几步 structural induction 找找感觉, 因为 substitution 是第一次出现. 这里我们依然对  $t$  来进行归纳.

*Case 1* 若  $t = v$ , 其中  $v$  为一个 variable.

分两种情况: (1 若  $v = x$ , 则  $[x \rightarrow s]t = [x \rightarrow s]v = s$ , 而根据命题条件  $\Gamma \rightarrow s : S$ , 显然成立. (2 其他情况下, 则有  $[x \rightarrow s]v = v$ , 即这个 substitution 没起作用, 显然还是成立.

**Annotation 67** 对于一个 language 有两种特别的刻画形式:

- **Curry-style** 首先我们定义 terms, 再定义关于它们的求值规则 (evaluation rules), 来确定 terms 的语义. 然后在定义一个类型系统来拒绝一些不符合我们预期的 terms. 因此语义刻画是在类型之前, 即它是一种 implicit typing, 也可以看做一种 polymorphism. 例如对于 identity abstraction, 在 curry-style 下可以使用一种 type variables, 它可以代指所有可能的 type,

$$I = \lambda x. x : \sigma \rightarrow \sigma$$

- **Church-style** 首先我们定义 terms, 再确定一些 well-typed 的 terms. 然后只给 well-typed terms 制定求值规则, 来确定其语义. 因此类型先于语义, 即它是一种 explicit typing. 例如对于 identity abstraction, 对每一种可能 type, 在 church-style 下我们都可以得到 family of identity,

$$I_A = \lambda x : A. x : A \rightarrow A.$$

它们两个最大的不同就是我们在谈论一个 term 的语义的时候到底是否关系它此时是 well-typed. Curry-style 通常适用于刻画 implicitly typed system, 而 Church-style 通常用于刻画 explicitly typed system.

## Type Extensions

### Known Types

**Definition 68** **base type** Something like bool, nat, float and string, these type are for describing simple and unstructured values and appropriate primitive operation for manipulating these values.

**Definition 69** **unit type** a constant unit with unique type Unit, the type can be only from this constant. we often use \* for the unique unit term and 1 for the unit type

$$\overline{\Gamma \vdash * : 1}$$

**Definition 70** The **sequencing** notation  $t_1; t_2$  has the effect of evaluating  $t_1$ , throwing away its trivial result(unit), and going on to evaluate  $t_2$ .

**Annotation 71** **first way to formalize sequencing** Add  $t_1; t_2$  as a new alternative in the syntax of terms, and then add two evaluation rules

$$\frac{t_1 \rightarrow t'_1}{t_1; t_2 \rightarrow t'_1; t_2}$$
$$\text{unit}; t_2 \rightarrow t_2$$

and a typing rule

$$\frac{\Gamma \vdash t_1 : \text{Unit} \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1; t_2 : T_2}$$

**Annotation 72** **second way to formalize sequencing** Regard  $t_1; t_2$  as an abbreviation for the term  $(\lambda x : \text{Unit}. t_2) t_1$ , where  $x \in \text{FV}(t_2)$ .

**Theorem 73** Suppose  $\lambda^E$  for the simply typed lambda-calculus with the first way of sequencing formalization and  $\lambda^I$  for the simply typed lambda-calculus with Unit. Let  $e : \lambda^E \rightarrow \lambda^I$  be the elaboration function that translates from the  $\lambda^E$  to  $\lambda^I$  by replacing every occurrence of  $t_1; t_2$  with  $(\lambda x : \text{Unit}. t_2) t_1$ , where  $x \in \text{FV}(t_2)$ . Then for each  $t$  of  $\lambda^E$ , we have

1.  $t \rightarrow_E t'$  iff  $e(t) \rightarrow_I e(t')$ ;
2.  $\Gamma \vdash_E t : T$  iff  $\Gamma \vdash_I e(t) : T$ .

**Annotation 74** 这个 sequencing 目前来说和我们现代下的语言里面对应的概念还是有差别的. 根据第一个 formalization, 也就是我们定义里面提到的它是依赖  $t_1$  的 evaluation result, 我们对一个 sequencing 能做的就是首先对  $t_1$  进行 evaluating, 只有它的 result 是一个 Unit 的时候, 我们可以尝试丢掉它转而去处理  $t_2$ . 显然当  $t_1$  不是 trivial 的时候,  $t_2$  永远得不到的 evaluating, 就停在了某个  $t'_1; t_2$ . 这是就目前而言的我们可以做的事情.

再关于第二个 formalization 而言，它是一个很特别的带注解的 application，会有一个自然地疑问，如果此时  $t_1$  的 evaluation result 不是 Unit，怎么让这个 application make sense? 是卡在这里，还是怎样? 显然在前述的 corresponding theorem 下我更倾向于是卡在这里.



## Known Features

**Definition 75** **Ascription** is simple feature for ascribe a particular type to a given term. We write "t as T" for the "the term t, to which we ascribe the type T".

**Definition 76** **Let Bindings** let  $x = t_1$  in  $t_2$ , 它们的 evaluation rule 和 type rule 跟 lambda abstraction 是差不多的, 即

$$\text{let } x = t_1 \text{ in } t_2 = (\lambda x : T_1. t_2) t_1.$$

**Definition 77** **Pair** Pairing, written  $t = \langle t_1, t_2 \rangle$  and projection, written  $t.1$  for the  $t_1$  and  $t.2$  for the  $t_2$ . One new type constructor,  $T_1 \times T_2$ , called the product of  $T_1$  and  $T_2$ .

**Definition 78** **Tuple** is general formalization of Pair.

**Definition 79** **Record** Recording, written  $\{l_1 = t_1, \dots, l_n = t_n\}$  and thier type  $\{l_1 : T_1, \dots, l_n : T_n\}$ .

**Definition 80** **pattern matching** Given two kinds of patterns, variable pattern  $x$  and record pattern  $\{l_1 = p_1, \dots, l_n = p_n\}$  (so it can be nested). Plus a match function  $match : P \times V \rightarrow \text{Subs} \cup \text{Fail}$ , where  $P$  are patterns,  $V$  is values, Subs are substitutions and Fail means matching fails. The matching rules as follow

$$\frac{\begin{array}{c} match(x, v) = [x \rightarrow v] \\ \text{for each } i \text{ } match(p_i, v_i) = \sigma_i \end{array}}{match(\{l_1 = p_1, \dots, l_n = p_n\}, \{l_1 = v_1, \dots, l_n = v_n\}) = \sigma_1 \circ \dots \circ \sigma_n} \quad \begin{array}{c} M - Var \\ \\ M - Rcd \end{array}$$

The computation rule for pattern matching generalizes the let-binding as follow

$$\text{let } p = v \text{ in } t = match(p, v) t_1.$$

**Definition 81** A **sum type** is written as  $T_1 + T_2$ , there are two terms can be desribed this type:

1. Assume  $t_1 : T_1$ , then **inl**  $t_1 : T_1 + T_2$ ;
2. Assume  $t_2 : T_2$ , then **inr**  $t_2 : T_1 + T_2$ .

There is a **case** construct that allows us to distinguish whether a given value comes from the left or right branch of a sum,

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \quad \Gamma, x_1 : T_1 \vdash t_1 : T \quad \Gamma, x_2 : T_2 \vdash t_2 : T}{\text{case } t_0 \text{ of inl } x_1 \mapsto t_1 : T \mid \text{inr } x_2 \mapsto t_2 : T}$$

where  $x_1 \mapsto t_1 : T$  means  $[x_1 \rightarrow T_1]t_1$ , another one is similar.

**Annotation 82** 上述 case 分几步理解

1. 首先得有一个  $t_0 : T_1 + T_2$ ,
2. 然后我们有两个 type 相同的 terms  $t_1 : T$  和  $t_2 : T$ , 它们分别各自都有一个 bounded variables  $x_1 :: T_1$  和  $x_2 :: T_2$ , 因此这里实际上一个隐含的 abstraction, 最后
3. 我们确定  $t_0$  是  $\text{inl}x$  和  $\text{inr}y$ , 根据其选择  $t_1$  和  $t_2$  apply 上对应的  $x$  和  $y$ , 这里我们使用  $x$  和  $y$  目的是将其和两个 bounded variables 区分开来.

**Annotation 83** 这里存在一个类型唯一性的问题, 如果  $t_1 : T_1$ , 那么对于任意  $T_2$ , 都有  $\text{inl } t_1 : T_1 + T_2$ , 显然  $\text{inl } t_1$  的类型就不唯一了.

这里有三种解决办法

1. 留着  $T_2$  符号化, typechecker 继续往后推, 如果遇到某个地方  $T_2$  可能在当前 context 需要成为某个特定的值;
2. 给所有可能的  $T_2$  一个 unified representation(ocaml);
3. 在语法上要求显式地给  $T_2$  一个 type annotation.

**Definition 84** **variant** is generalization of sum  $\langle l_1 : T_1, l_2 : T_2 \rangle$ .

**Definition 85** **option**  $\langle \text{none} : \text{Unit}, \text{some} : \text{Nat} \rangle$ .

**Definition 86** **enumeration** An enumerated type (or enumeration) is a variant type in which the field type associated with each label is Unit.

**Definition 87** **single-field variant**  $\langle l : T \rangle$ .

**Annotation 88** single-field variant 的主要作用由一个 type 构造出多个不一样的 types 但是仅仅是用附加的 labels 来刻画的, 这就可以描述具有相同 type 但是不同对象.

**Definition 89** **void type** An emptyset of term with unique type  $\text{void}^1$ , we often use 0 for the unique void type. When we reach a void type, we have reached an error, and thus we can throw any typed exception.

$$\frac{t_1 : 0}{\vdash \text{abort}_A t_1 : A}$$

**Annotation 90** The abort rule extracts a term of any type from the void type.

---

<sup>1</sup>There is no term for void type

## Normalization

**Theorem 91** If  $\vdash t : T$ , then  $t \rightarrow^* v$ , where  $v$  is a value, or abbreviate  $t \Downarrow$ .

**Remark 92** 上述 normalization theorem 使用的是 simply typed lambda calculus.

**Annotation 93** Normalization 又名 termination, 它在描述一个 well-typed 的 term 通过 evaluation 最终可以变成一个 value. 这里 values 包括 false, true 和 lambda abstraction. 自然地, 这里考虑使用 induction hypothesis 来证明, 但是处理不了 application. 对于 application 我们需要使用 reduction rule

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 t_2 : T_2}$$

根据假设  $t_1, t_2$  都是 normalizable, 那么设  $t_1 \rightarrow^* t'_1 = \lambda x : T_1. t_3$  (这里用了一下 value of function type 的 canonical form) 和  $t_2 \rightarrow^* t'_2$ , 其中  $t'_2$  是 normalized. 再来一个  $\beta$  reduction, 则有

$$t'_1 t'_2 = [x \rightarrow t'_2] t'_3$$

这里有两个问题: (1)  $t_3$  是一个怎样的形式? (2) substitution 干了什么?

**Definition 94** Suppose the logical predicate for strong normalization as follow

$$\begin{aligned} \text{SN}_A(t) &\iff \vdash t : A \wedge t \Downarrow, \\ \text{SN}_{T_1 \rightarrow T_2}(t) &\iff \vdash t : T_1 \rightarrow T_2 \wedge t \Downarrow \wedge \forall t_1. \text{SN}_{T_1}(t_1) \Rightarrow \text{SN}_{T_2}(t t_1) \end{aligned}$$

where  $A$  is base type.

**Annotation 95** 观察上述 definition 是加强了 application 的 conclude(?), 可以通过这两个 logical predicate 来继续我们的证明, 接下来的证明分两步走:

1. 首先证明  $\vdash t : T \Rightarrow \text{SN}_T(t)$ , 即所有 closed well-typed 的 term 都复合上述定义的 logical predicate,
2. 然后  $\text{SN}_T(t) \Rightarrow t \Downarrow$ .

这种手法就是所谓 **logical relation** 证明方法.

**Lemma 96**  $\text{SN}_T(t) \Rightarrow t \Downarrow$

PROOF 根据定义这是显然的.

Q. E. D.

**Annotation 97** 证明过程的第一步又会拆成两步:

1.  $\text{SN}_T(t)$  将会在  $t$  的 evaluation 过程中保持,

2. 再做根据 type derivations 的 induction, 但是于证明 abstraction  $t = \lambda x : T_1. t_2$  满足  $SN_{T_1 \rightarrow T_2}(t)$  的时候, 注意这里我们 SN 对 closed term 而言的, 因此我们这里根据 derivation 是

$$\frac{x : T_1 \vdash t_2 : T_2}{\vdash \lambda x : T_1. t_2}$$

问题来了这个 inference rule 的 premise 不是 empty, 因此我们没法继续用 induction hypothesis 来继续我们的证明, 这里需要做一个推广 (generalization), 即  $\Gamma \vdash t : T \Rightarrow SN_T(t)$ . 这里又会出现一个问题是的  $t$  可能不是 closed 了, 因此我们考虑将这个 open term  $t$  实例化, 即从  $\Gamma$  出发构造 substitution 给  $t$ , 让它重新变成 closed. 最终我们所需要的结论只是 generalization 的一个推论.

**Lemma 98** If  $t : T$  and  $t \rightarrow t'$ , then  $SN_T(t) \iff SN_T(t')$

PROOF 首先由  $t \rightarrow t'$ , 那么有  $t \Downarrow \iff t' \Downarrow$ . 再分情况, 若  $T = A$ , 证明就结束了; 若  $T = T_1 \rightarrow T_2$ , 由  $t t_1 \rightarrow t' t_1$ , 则  $t t_1 \Downarrow \iff t' t_1 \Downarrow$ , 又回到第一种情况, 证明了 function type 额外需要的条件. Q. E. D.

**Lemma 99** If  $x_1 : T_1, x_2 : T_2, \dots, x_n : T_n \vdash t : T$  and  $v_1, v_1, \dots, v_n$  are closed values of  $T_1, T_2, \dots, T_n$  with  $SN_{T_i}(v_i)$ , then  $SN_T([x_1 \rightarrow v_1, x_2 \rightarrow v_2, \dots, x_n \rightarrow v_n]t)$

PROOF structural induction as follow

Case 1

$$\begin{aligned} t &= x_i \\ T &= T_i \end{aligned}$$

显然成立.

Case 2

$$\begin{aligned} t &= \lambda x : S_1. s_2 \\ T &= S_1 \rightarrow S_2 \\ x_1 : T_1, x_2 : T_2, \dots, x_n : T_n, x : S_1 &\vdash s_2 : S_2 \end{aligned}$$

显然此时  $[x_1 \rightarrow v_1, x_2 \rightarrow v_2, \dots, x_n \rightarrow v_n]t$  已经一个 value 了, 因为  $t$  本来就是一个 abstraction. 此时需要额外证明 applying 过程, 即给定任意的  $SN_{S_1}(s)$ , 有  $SN_{S_2}([x_1 \rightarrow v_1, x_2 \rightarrow v_2, \dots, x_n \rightarrow v_n]t) s$ . 根据 Lemma96, 我们有  $s \rightarrow^* v$ , 根据归纳假设即有

$$SN_{S_2}([x_1 \rightarrow v_1, x_2 \rightarrow v_2, \dots, x_n \rightarrow v_n, x \rightarrow v]t)$$

而

$$([x_1 \rightarrow v_1, x_2 \rightarrow v_2, \dots, x_n \rightarrow v_n]t) s \rightarrow^* [x_1 \rightarrow v_1, x_2 \rightarrow v_2, \dots, x_n \rightarrow v_n, x \rightarrow v]t,$$

再用一下 Lemma98, 即可得到我们想要的.

Case 3

$$\begin{aligned}
t &= t_1 t_2 \\
x_1 : T_1, x_2 : T_2, \dots, x_n : T_n &\vdash t_1 : T_{11} \rightarrow T_{12} \\
x_1 : T_1, x_2 : T_2, \dots, x_n : T_n &\vdash t_2 : T_{11} \\
T &= T_{12}
\end{aligned}$$

根据归纳假设有  $\text{SN}_{T_{11} \rightarrow T_{12}}([x_1 \rightarrow v_1, x_2 \rightarrow v_2, \dots, x_n \rightarrow v_n]t_1)$  和  $\text{SN}_{T_{11}}([x_1 \rightarrow v_1, x_2 \rightarrow v_2, \dots, x_n \rightarrow v_n]t_2)$ . 再根据  $\text{SN}_{T_{11} \rightarrow T_{12}}$  的 definition, 有

$$\begin{aligned}
&\text{SN}_{T_{12}}([x_1 \rightarrow v_1, x_2 \rightarrow v_2, \dots, x_n \rightarrow v_n]t_1[x_1 \rightarrow v_1, x_2 \rightarrow v_2, \dots, x_n \rightarrow v_n]t_2) \\
&= \text{SN}_{T_{12}}([x_1 \rightarrow v_1, x_2 \rightarrow v_2, \dots, x_n \rightarrow v_n]t_1 t_2)
\end{aligned}$$

得证.

**Annotation 100** Lemma99中 substitution 可以记为  $\gamma = [x_1 \rightarrow v_1, x_2 \rightarrow v_2, \dots, x_n \rightarrow v_n]$ , 也可以直接记为  $\gamma \models \Gamma$ , 理解为”the substitution  $\gamma$  statisfies the type environment,  $\Gamma$ ”.

**Corollary 101**  $\vdash t : T \Rightarrow \text{SN}_T(t)$ .

PROOF 直接从 Lemma99可得.

Q. E. D.

## References

**Definition 102** A **reference value** represents mutable cell. The basic operations on reference are allocation, dereferencing and assignment.

To allocate a reference, we use the **ref** operator, providing an initial value for the new cell

$$r = \text{ref } 5 \Rightarrow r : \text{Ref Nat.}$$

To read a current value of this cell, we use the dereferencing operator **!**

$$!r \Rightarrow 5 : \text{Nat.}$$

To change the value stored in the cell, we use the assignment operator

$$r := 7 \Rightarrow \text{unit} : \text{Unit.}$$

The result of the assignment is the trivial **unit** value.

**Definition 103** The references **r** and **s** are said to be aliases for the same cell.

**Annotation 104** 在这里就正式的引入了 sequencing 带来的 side effort, 关于 references 的 evaluation rule 非常冗余, 这里简单记关键几点

1. references 会被抽象成 location indexes  $l \in \mathcal{L}$ . states 会被抽象成 store function  $\mathcal{L} \rightarrow \text{values}$ ;
2. 之前的所有 evaluation 都会附近上额外 store function;
3. dereference 一个不存在的 location, 会给出一个错误. dereference operator 要等到它右边的 term 被 evaluated 成一个 value 才能起作用, 同理 allocation 也一样;
4. 对于 assignment, 需要先 evaluate 左边 term.

**Definition 105** A **store typing** is a finite function mapping locations to types, we use the metavariable  $\Sigma$  to range over such functions. the typing rule for locations can be formalized as follow

$$\frac{\Sigma(l) = T_1}{\Gamma | \Sigma \vdash l : \text{Ref } T}$$

**Annotation 106** 这里为什么要构造一个这样的 function 呢? 因为自然地考虑  $l$  应该依赖于 store function  $\mu$ , 这里对应的 typing rule 为

$$\frac{\Gamma | \mu \vdash \mu(l) : T_1}{\Gamma | \mu \vdash l : \text{Ref } T}$$

如果  $\mu$  的结构是这样

$$(l_1 \rightarrow \lambda x : \text{Nat}. !l_2 \times, l_2 \rightarrow \lambda x : \text{Nat}. !l_1 \times)$$

这里 cyclic reduction 的过程,  $l_1$  的 type 依赖  $l_2$  的 type 依赖, 反过来  $l_2$  的 type 依赖  $l_1$  的 type. 那么如何构造这样一个  $\Sigma: \mathcal{L} \rightarrow T$  的 map 呢? 它是可以在 evaluation 过程动态构造的, 因为只要一个 location 第一次被 allocated, 那么在它对应的位置上一定有一个具体的 type, 同样无论后面经历 assignment 多少次都只有唯一的 type 对应, 这样我们可以一开始就将  $\Sigma$  置为一个 empty map, 再根据对应的操作是维护它就可以了.

**Definition 107 (Connection between  $\mu$  and  $\Sigma$ )** A store  $\mu$  is said to be well typed with respect to a typing context  $\Gamma$  and a store typing  $\Sigma$ , written  $\Gamma \mid \Sigma \vdash \mu$ , if  $\text{dom}(\mu) = \text{dom}(\Sigma)$  and  $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$  for every  $l \in \text{dom}(\mu)$ .

**Theorem 108 Preservation** If

$$\begin{aligned} \Gamma \mid \Sigma &\vdash t : T \\ \Gamma \mid \Sigma &\vdash \mu \\ t \mid \mu &\rightarrow t' \mid \mu' \end{aligned}$$

then, for some  $\Sigma' \supseteq \Sigma$ ,

$$\begin{aligned} \Gamma \mid \Sigma' &\vdash t' : T \\ \Gamma \mid \Sigma' &\vdash \mu' \end{aligned}$$

**Annotation 109** 其中  $\Sigma' \supseteq \Sigma$  产生的原因是 allocation operator 会带来新的 location, 同时不用考虑 assignment operator, 因为 sequencing 没有在当前的语法中, 它的 side effort 也无法起作用, 所以一个包含关系就够了.

**Theorem 110 Progress** Suppose  $t$  is closed, well-typed term, that is  $\cdot \mid \Sigma \vdash t : T$  for some  $T$  and  $\Sigma$ . Then either  $t$  is a value or else, for any store  $\mu$  such that  $\cdot \mid \Sigma \vdash \mu$ , there is some term  $t'$  and store  $\mu'$  with  $t \mid \mu \rightarrow t' \mid \mu'$ .

## Recursion

**Definition 111** There are two basic approaches to recursive types.

1. **equi-resursive** approach: given two types expression as definition-ally equal-interchangeable in all contexts.
2. **iso-resursive** approach: takes a recursive type and its unfolding as different, but isomorphic.

**Definition 112** An explicit **recursion operator**  $\mu$  for types:

$$\mu\alpha.\tau$$

it has two interpretation from above two approaches.

1. equi-resursive: A recursive type  $\mu\alpha.\tau$  is the infinite type satisfying the equation  $\alpha = \tau$ .
2. iso-resursive: A recursive type  $\mu\alpha.\tau$  was regarded as an infinite type and consider equal to its unfolding  $[\alpha \rightarrow \mu\alpha.\tau]\tau$ .

**Annotation 113** equi-recursive 性质可以告诉 typechecker 对应的 recursion type 它可以适当的转换, 例如我们可以 recursion type  $T$  满足等式  $X = X \rightarrow X$ , 给定两个 type 均为  $T$  的 terms  $M, N$ , 当考虑 application 时  $MN$ , typechecker 就知道  $M : T \rightarrow T$  和  $N : T$ .

而 iso-resursive 性质就是将 equi-recursive 里面隐式的转换用 unfolding 和 unfold 变成显式的了. 这里有一个小小问题探讨, 以上的所有 recursion 的描述都是针对 type 而言的, 不是之前在 untyped  $\lambda$ -calculus 里面针对 term 而言的. 但是 unfold 和 fold 会作为 primitives 出现在 terms 里面, 那么这里需要对它们有一个准确的描述.

首先每个 recursive type 都有 unfold 和 fold, 因此它们形式化的定义如下

$$\text{unfold}[\mu\alpha.\tau] : \mu\alpha.\tau \rightarrow [\alpha \rightarrow \mu\alpha.\tau]\tau$$

$$\text{fold}[\mu\alpha.\tau] : [\alpha \rightarrow \mu\alpha.\tau]\tau \rightarrow \mu\alpha.\tau$$

例如某个 term 具有 unfolded formation, 对其进行 fold 可以写作  $\text{fold}[\mu\alpha.\tau] t$ . 那么它们的 isomorphism 体现在

$$\text{fold}[\mu\alpha.\tau](\text{unfold}[\mu\alpha.\tau] t) = t$$

**Example 114** The type natural list are defined as follow

$$\text{NatList} = \mu\alpha. \langle \text{nil} : \text{Unit}, \text{cons} : \{\text{Nat}, \alpha\} \rangle.$$

**Definition 115** A fixed constructor for function type  $\tau$  is defined as follow

$$\text{fix}_\tau = \lambda f : \tau \rightarrow \tau. (\lambda x : (\mu\alpha.\alpha \rightarrow \tau). f(x x)) (\lambda x : (\mu\alpha.\alpha \rightarrow \tau). f(x x)).$$



**Annotation 116** 去掉  $\text{fix}_\tau$  中所有的 type annotation, 就可以得到在 untyped lambda calculus 里面的 fix constructor. 这里  $x$  必须是一个 arrow type, 同时它的 domain 也是  $x$  它自己, 显然我们找不到这样 finite type, recursion operator 在这里就神奇的起作用了.

**Definition 117** A well-typed term whose evaluation will diverge.

$$\text{diverge}_\tau = \lambda : \text{Unit}. \text{fix}_\tau \text{ id}$$

**Annotation 118** 这意味 recursive type 的引入将会破坏 strong normalization.

**Example 119** 利用 recursive type 可以完美地将 untyped lambda calculus  $\Lambda$  (only include variables, abstraction and application) 变成 typed. 给每个在  $\Lambda$  里面的 term 都 assign 上一个 recursive type  $D$

$$D = \mu\alpha. \alpha \rightarrow \alpha.$$

因为 arrow type 的天然存在, 会导致出现  $D \rightarrow D$ , 因此我们需要一个 unifier. 其中 abstraction 对应了 fold 操作, 我们需要将  $D \rightarrow D$  变成  $D$  保持一致, 即

$$\text{lam} = \lambda f : D \rightarrow D. f \text{ as } D,$$

而 application 对应 unfold 操作, 我们需要将第一个 term 从  $D$  变成  $D \rightarrow D$ , 这样才能 apply, 即

$$\text{ap} = \lambda f : D. \lambda x : D. f \ a.$$

给定 closed term  $M \in \Lambda$ , 用  $M^*$  表示  $M$  对应的 typed term, 两者对应如下

$$x^* = x$$

$$(\lambda x. M)^* = \text{lam } (\lambda x : D. M^*)$$

$$(MN)^* = \text{ap } M^* N^*$$

## Subtyping

### STLC

**Annotation 120** **Motivation** 考虑下面 application

$$(\lambda r : \{x : \text{Nat}\}. r.x) \{x = 0, y = 1\},$$

它在前面的 STLC 里面不是 typable 的, 显然 argument 不满足 abstraction 里面的 explicit annotation, 我们希望兼容这种问题.

**Definition 121** Let  $S, T$  be any terms,  $S$  is a **subtype** of  $T$  if any term of type  $S$  can safely be used in a context where a term of type  $T$  is expected, simply written  $S <: T$ .

**Annotation 122** 首先举个例子  $\{x : \tau_1, y : \tau_2\} <: \{x : \tau_1\}$ , 你如果直接思考两个 record 的势来比较显然是不符合直接的, 但是这个你得用 cast 的想法来理解, 这就相当于  $S$  的类型蕴含着  $T$  的类型, 这个理解在逻辑上其实比较好理解,  $S$  里面的 limits 实际上要比  $T$  多, 因此 "the element of  $S$  are a subset of the elements of  $T$ ".

**Definition 123** Subsumption rule

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}.$$

**Lemma 124** Subtype relation satisfies reflexivity and transitivity.

**Definition 125** Let type  $Top$  satisfies  $S <: Top$  for any type  $S$ .

**Definition 126** Subtyping rule for records.

$$\begin{array}{l} \{l_1 : T_1, l_2 : T_2, \dots, l_{n+k} : T_{n+k}\} <: \{l_1 : T_1, l_2 : T_2, \dots, l_n : T_n\} \quad \text{width subtyping} \\ \frac{\forall i. S_i <: T_i}{\{l_1 : S_1, l_2 : S_2, \dots, l_n : S_n\} <: \{l_1 : T_1, l_2 : T_2, \dots, l_n : T_n\}} \quad \text{depth subtyping} \end{array}$$

**Definition 127** Subtyping rule for functions.

$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}.$$

**Annotation 128** 关于 function 的 subtyping, 通常描述为 "the function arguments are contravariant and the function results are covariant", 其中 covariant 的意思就是说如果原本我们有  $T_2 <: S_2$  做为 premise, 那么由上述 inference rule 的 conclude 里面同样保持这个 relation, 即  $T_2$  在  $<:$  左边, 而  $S_2$  在右边; 类似地 contravariant 得到的结果是相反的.

上述的 inference rule 可以解释为:

1. 一个函数我可以 accept 相比于 original argument type 所含 elements 更多的 new argument type, 这是 safe 的. 因为假设 function 的 input 不变, 我现在可以接受更多的元素, 其中原来可以元素也可以 cover 到, 显然是安全的.
2. 一个函数我们可以 return 相比于 original result type 所含 elements 更少一点的 new argument, 这是 safe 的. 因为假设 function 的 requirements of result 不变, 我现在  $\text{dom}(Res_{new})$  是原来  $\text{dom}(Res_{old})$  的一个子集, 原来可以满足前述的 requirement, 那么现在当然可以满足, 显然是安全的.

## Properties of Subtyping and Typing

**Annotation 129** 当引入 subtyping 之后我们需要重新研究 current type system 已经证明过的 properties, 例如 preservation and progress.

**Lemma 130** (inversion)

- If  $S <: T_1 \rightarrow T_2$ , then  $S$  has the form  $S_1 \rightarrow S_2$ , with  $T_1 <: S_1$  and  $S_2 <: T_2$ .
- If  $S <: \{(l_i : T_i)_{i \in 1 \dots n}\}$ , then  $S$  has the form  $\{(k_j : S_j)_{j \in 1 \dots m}\}$ , with  $\{(l_i)_{i \in 1 \dots n}\} \subseteq \{(k_j)_{j \in 1 \dots m}\}$  and for  $\forall l_i = k_j. S_i <: T_j$ .

**Lemma 131** If  $\Gamma \vdash \lambda x : S_1. s_2 : T_1 \rightarrow T_2$ , then  $T_1 <: S_1$  and  $\Gamma, x : S_1 \vdash s_2 : T_2$ .

PROOF 直接根据 typing derivation

$$\frac{\frac{\Gamma, x : S_1 \vdash s_2 : T_2}{\Gamma \vdash \lambda x : S_1. s_2 : S_1 \rightarrow T_2} \quad \frac{T_1 <: S_1 \quad T_2 <: T_2}{S_1 \rightarrow T_2 <: T_1 \rightarrow T_2}}{\Gamma \vdash \lambda x : S_1. s_2 : T_1 \rightarrow T_2}$$

关键是将  $\Gamma \vdash \lambda x : S_1. s_2 : T_1 \rightarrow T_2$  看做一个 abstraction, 它的 argument annotation 显然和它的 type 里面的不一样, 不能直接拆, 因此首先整体要用一下 function subtyping. Q. E. D.

**Lemma 132** If  $\Gamma \vdash \{(k_j = s_j)_{j \in 1 \dots m}\} : \{(l_i = T_i)_{i \in 1 \dots n}\}$ , then  $\{(l_i)_{i \in 1 \dots n}\} \subseteq \{(k_j)_{j \in 1 \dots m}\}$  and  $\forall l_i = k_j. \Gamma \vdash s_j : T_i$ .

**Lemma 133** (substitution) If  $\Gamma, x : S \vdash t : T$  and  $\Gamma \vdash s : S$ , then  $\Gamma \vdash [x \mapsto s]t : T$ .

**Lemma 134** (canonical forms)

- If  $v$  is closed value of type  $T_1 \rightarrow T_2$ , then  $v$  has the form  $\lambda x : S_1. t_2$ .
- If  $v$  is closed value of type  $\{(l_i : T_i)_{i \in 1 \dots n}\}$ , then  $v$  has the form  $\{(k_i : v_i)_{i \in 1 \dots n}\}$ , with  $\{(l_i)_{i \in 1 \dots n}\} \subseteq \{(k_i)_{i \in 1 \dots n}\}$ .

**Theorem 135** (preservation) If  $\Gamma \vdash t : T$  and  $t \rightarrow t'$ , then  $\Gamma \vdash t' : T$

PROOF 证明过程是根据 typing rules 归纳 well-typed  $t : T$ , 这里只需要重点关心 new typing rules, 和稍微有些区别的 canonical forms 和 substitution. i.e. representative cases.

Case 1

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$

由归纳假设  $\Gamma \vdash t : S$  and  $t \rightarrow t'$ , then  $\Gamma \vdash t' : S$ , 此时在对  $t' : S$  用一下 subsumption rule, 即有  $\Gamma \vdash t' : T$ .

Case 2

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}$$

evalutaion 现在只考虑 app-abs, 设  $t_1 = \lambda x : S_{11}. t_{12}$ , 注意这里用的 argument annotation 是  $S_{11}$ , 因为这里没有东西可以说明它是  $T_1$ , 然后我们根据 Lemma 131, 才能得到关于  $S_{11}$  的信息  $T_1 <: S_{11}$  和  $\Gamma, x : S_{11} \vdash t_{12} : T_2$ . 于是再对  $t_2 : T_1$  用一下 subsumption rule 得到  $t_2 : S_{11}$ , 最后用 Lemma 133即可.

**Theorem 136** (**progress**) If  $t$  is closed, well-typed term, then either  $t$  is value or else there is some  $t'$  with  $t \rightarrow t'$ .

PROOF 与在 STLC 证明类似, 此时注意下 canonical forms 即可.

Q. E. D.

## Bottom Type

**Definition 137** Let type  $Bot$  satisfies  $Bot <: T$  for any type  $T$ .

**Lemma 138** There are no closed values of type  $Bot$ .

PROOF 可以用 canonical forms 两个情况来说明: 若  $v$  是一个  $\text{abs } \lambda x : T_1. t_2$ , 那么对于任意地 type  $S_1 \rightarrow S_2$  我们应当有如下的 derivation:

$$\frac{\frac{x : T_1 \vdash t_2 : T_2}{\lambda x : T_1. t_2 : T_1 \rightarrow T_2} \quad S_1 <: T_1 \quad T_2 <: S_2}{\lambda x : T_1. t_2 : S_1 \rightarrow S_2} T_1 \rightarrow T_2 <: S_1 \rightarrow S_2$$

换个思路我们希望找到是没有 subtyp 的 type

Q. E. D.

## Coercion Semantics

**Annotation 139** 前面提到的 subtyping 只不一种针对类型的 extension, 并没有影响到我们的 evaluation 过程, 但是在实际操作可能会出问题. 假设我们给定  $T <: S$ , 在实际中  $T$  和  $S$  可能有不一样的 internal structure, 例如常见的 bool 和 int 类型, 那么在实际 evaluation 的过程我们应当实现类型转换过程的结构转换. 其中有一种想法就是将带有 subtyping 语法的 language 翻译成不带 subtyping language, 还是使用原来的 evaluation 过程. 通常来说编译器里面的这种翻译是从 high-level language 到 low-level language 的过程.

这里的翻译过程需要配合 subtyping derivation 和 normal typing derivation. 例如给定一个 source language(带 subtyping) 里面的 term  $e$  和对应的 type  $T$ , 我们要将他翻译成 target language 里面的 term  $e'$ . 那么我们需要将  $e$  按照 derivation 把  $e$  里面的所有 subterm 也都要翻译到 target language 里面. 其中涉及到 subtyping derivation 的翻译, 每一个 subtyping rule 对应一个一个 abstraction, 例如  $\lambda x : T. s : T \rightarrow S$ , 我们再将需要做 type 转换的 term 作为 argument 传入就行. 还有一个比较特殊点就是两个 language 之间的 type 可能不一致, 例如前面提到了  $Top$  这一类型, 对应目标语言里面它就是  $Unit$ , 因此我们还需要构造一下两个语言里面类型的映射.

下述 translation 的 source language 为 pure STLC 带上 record 和 subtyping, target language 为 pure STLC 只有 record 和 type  $Unit$ .

**Definition 140** Function  $\llbracket - \rrbracket$  of type translation

$$\begin{aligned}\llbracket Top \rrbracket &= Unit \\ \llbracket T_1 \rightarrow T_2 \rrbracket &= \llbracket T_1 \rrbracket \rightarrow \llbracket T_2 \rrbracket \\ \llbracket \{(l_i : T_i)_{i \in 1 \dots n}\} \rrbracket &= \{(l_i : \llbracket T_i \rrbracket)_{i \in 1 \dots n}\}\end{aligned}$$

**Definition 141** If  $C$  is a subtyping derivation tree whose conclusion is  $S <: T$ , then we have  $\mathcal{C} :: S <: T$ . Similarly,  $\mathcal{D} :: S <: T$  for typing derivation.

**Definition 142** Coercion for subtyping.

$$\begin{aligned}\llbracket \overline{T <: T} \rrbracket &= \lambda x : \llbracket T \rrbracket. x \\ \llbracket \overline{S <: Top} \rrbracket &= \lambda x : \llbracket S \rrbracket. unit \\ \llbracket \frac{C_1 :: S <: U \quad C_2 :: U <: T}{S <: T} \rrbracket &= \lambda x : \llbracket S \rrbracket. \llbracket C_2 \rrbracket (\llbracket C_1 \rrbracket x) \\ \llbracket \frac{C_1 :: T_1 <: S_2 \quad C_2 :: S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \rrbracket &= \lambda f : \llbracket S_1 \rightarrow S_2 \rrbracket. \lambda x : \llbracket T_1 \rrbracket. \llbracket C \rrbracket (f (\llbracket C \rrbracket_1)) \\ \llbracket \overline{\{(l_i : T_i)_{i \in 1 \dots n+k}\} <: \{(l_i : T_i)_{i \in 1 \dots n}\}} \rrbracket &= \lambda r : \{(l_i : \llbracket T_i \rrbracket)_{i \in 1 \dots n+k}\}. \{(l_i = r.i)_{i \in 1 \dots n}\} \\ \dots &\end{aligned}$$

**Definition 143** Function  $\llbracket - \rrbracket$  of typing derivation

$$\begin{aligned} \llbracket \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \rrbracket &= x \\ \llbracket \frac{\mathcal{D} :: \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \rrbracket &= \lambda x : \llbracket T_1 \rrbracket. \llbracket \mathcal{D} \rrbracket \\ \llbracket \frac{\mathcal{D} :: \Gamma \vdash t : S \quad \mathcal{C} :: S <: T}{\Gamma \vdash t : T} \rrbracket &= \llbracket \mathcal{C} \rrbracket \llbracket \mathcal{D} \rrbracket \\ &\dots \end{aligned}$$

**Theorem 144** If  $\mathcal{D} :: \Gamma \vdash t : T$ , then  $\llbracket \Gamma \rrbracket \vdash \llbracket t \rrbracket : \llbracket T \rrbracket$ , where  $\llbracket \Gamma \rrbracket$  is the pointwise extension of the type translation to contexts  $\llbracket \emptyset \rrbracket = \emptyset$  and  $\llbracket \Gamma, x : T \rrbracket = \llbracket \Gamma \rrbracket, x : \llbracket T \rrbracket$ .

**Annotation 145** 如果我们有关于  $\tau_1 <: \tau_2$ ,  $\tau_1 <: \tau_3$ ,  $\tau_2 <: \tau_4$  and  $\tau_3 <: \tau_4$  的 primitive coercions, 那么显然关于  $\tau_1 <: \tau_4$  的 subtyping derivation 就有两种, 有一个疑问是它们翻译到 target language 经过 evaluation 之后是不是有一样的结果呢?

**Definition 146** A translation  $\llbracket - \rrbracket$  from typing derivations in one language to term in another is **coherent** if, for every pair of derivations  $\mathcal{D}_1$  and  $\mathcal{D}_2$  with same conclusion  $\Gamma \vdash t : T$ , the translations  $\llbracket \mathcal{D}_1 \rrbracket$  and  $\llbracket \mathcal{D}_2 \rrbracket$  are behaviorally equivalent terms of the target language.



## Type Reconstruction

### Type Inference

**Annotation 147** 注意到前面讨论 typechecker 的时候, 出现了很多 annotations, 这些 annotations 帮助我们解决了一些问题, 特别是对 abstraction 的 argument annotation, 那么这里要探讨一类 typechecker 会建立在没有这些 annotation 的基础上, 作为 polymorphism 的前奏.

**Definition 148** [1] **Type templates** are ductively generated by

$$\text{TypeTemp} ::= \tau \mid \text{Tvar} \mid \text{TypeTemp} \rightarrow \text{TypeTemp}.$$

where  $\tau$  is base type and Tvar is type variable.

**Definition 149** A **type substitution**  $f$  is any function from type variables to type templates. Any type substitution  $f$  can be extended to a function between type templates and defined inductively by

$$f(T) = \begin{cases} T & \text{if } T \text{ is any basic type } \tau \\ f(T) & \text{if } T \text{ is any type variable } x \\ f(A) \rightarrow f(B) & \text{if } T \text{ is } A \rightarrow B \text{ for any two type templates } A \text{ and } B \end{cases}$$

**Annotation 150** 一个 type substitution 相当于是对缺少 annotations 的补全, 从此我们又可以尝试使用之前的 typechecker 欢快的玩耍了. 同时一个含 type variables 的 term 如果它是 well-typed, 那么它的所有 instance 也是 well-typed.

**Theorem 151** If  $\Gamma \vdash t : T$ , then for any type substitution  $\sigma$  we have  $\sigma\Gamma \vdash \sigma t : \sigma T$ .

**Annotation 152** 上述 theorem 中的  $\sigma t$  表示给  $t$  加上 annotation.

**Annotation 153** Suppose that  $t$  is a term containing type variables and  $\Gamma$  is an associated context (possibly also containing type variables). There are two quite different questions that can ask about  $t$ :

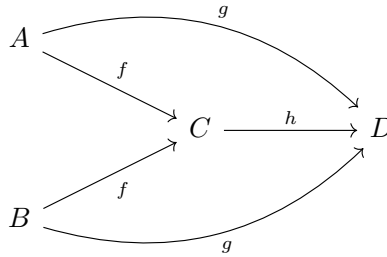
1. **for every  $\sigma$ , do we have  $\sigma\Gamma \vdash \sigma t : T$ ?** i.e.  $\lambda f : X \rightarrow X. \lambda a : X. f (f a)$  for any concrete type  $T$ , the instance is well typed, that is actually parametric polymorphism.
2. **for some  $\sigma$ , do we have  $\sigma\Gamma \vdash \sigma t : T$ ?** i.e.  $\lambda f : Y. \lambda a : X. f (f a)$  is not typable as it stands, but if replace  $Y$  by  $X \rightarrow X$ , we obtain well typed term.

**Annotation 154** 上面提到的第二类问题就是这里需要探究的问题，尽管我们没有了 term 里面的存在的 annotation，但是 term 里面依然暗含一些 constraints. 例如给定一个 application  $e_1 e_2$  和对应的  $\Gamma \vdash e_1 : \tau_1$  and  $\Gamma \vdash e_2 : \tau_2$ , 那么这里就有一个 constraint 用等式来描述就是  $\tau_1 = \tau_2 \rightarrow X$ , 其中  $X$  是一个 fresh variable. 因此我们需要注意到这些 constraints 来限定可用的 substitutions, 于是就提出了 unifier 的概念.

**Definition 155** A type template to be more general than other if the latter can be obtained by applying a substitution to the former. That is, if type templates  $A$  is more general than type template  $B$ , then there exists a substitution  $\sigma$  such that  $B = \sigma A$ .

**Definition 156** A substitution  $f$  is called an unifier of two sequences of type templates  $A_1, \dots, A_n$  and  $B_1, \dots, B_n$  if  $f(A_i) = f(B_i)$  for all  $i = 1, \dots, n$ . We say that it is the most general unifier if given any other unifier  $g$  exists a substitution  $h$  such that  $g = h \circ f$ .

**Annotation 157** 这里的 the most general unifier 刻画应该是这样一个性质,



如果存在两种 unifier, 那么显然它们是上述这种关系, 当考虑所有 unifier 的时候, 就存在一个 most general unifier(?).

**Lemma 158** (how to construct the most general unifier) If an unifier of  $A_1, \dots, A_n$  and  $B_1, \dots, B_n$  exists, the most general unifier is  $\text{unify}(A_1, \dots, A_n, B_1, \dots, B_n)$ , which is partially defined by induction as follow, where  $x$  is any type variable.

1.  $\text{unify}(x; x) = \text{id}$  and  $\text{unify}(\tau; \tau) = \text{id}$ ;
2.  $\text{unify}(x; B) = (x \rightarrow B)$ , the substitution that only changes  $x$  by  $B$ ; if  $x$  does not occur in  $B$ . The algorithm fails if  $x$  occurs in  $B$ ;
3.  $\text{unify}(A; x)$  is defined symmetrically;
4.  $\text{unify}(A \rightarrow A'; B \rightarrow B') = \text{unify}(A, A'; B, B')$ ;
5.  $\text{unify}(A, A_1, \dots, A_n; B, B_1, \dots, B_n) = f \circ g$  where  $g = \text{unify}(A_1, \dots, A_n; B_1, \dots, B_n)$  and  $f = \text{unify}(g(A); g(B))$ .

6. unify fails in any other case.

PROOF 这里需要证明 3 个部分:(1 上述描述的确实是一个 unifier (2 the most general unifier (3 termination.

对于 (3 termination, 我们需要 model 一个 degree 关于当且需要解决的 constraints, 这个 degree 用两个维度来表示: the number of distinct type variables 和 the number of whole types. 对于上述算法的每一步只会出现两种情况: (1 要么直接结束了 i.e. step1 或者 step6 (2 要么就是 degree 在减少 i.e. step2,3,5 比较明显, step4 中减少了两个 arrow type. step 5 相当于从一堆 constraints 里面取一个出来接近. 因此我们的 degree 是 finite, 最终算法会停止.

对于 (2 还是通常手法 hypothesis induction, 对 constraints 的规模做归纳. Base: 对于 empty sequence, unifier 实际上得到的是一个 identity id, 那么对于任意的 substitution  $\delta$ , 都有  $\delta = \delta \circ \text{id}$ . Induction: 当  $n+1$  规模时, 即  $\mathcal{C} = (A, B) \cup \mathcal{C}'$ , 其中  $\mathcal{C}' = (A_1, \dots, A_n; A_1, \dots, A_n)$ . 根据假设我们有  $\sigma' = \text{unify}(\mathcal{C}')$ , 这里要对  $(\sigma' A, \sigma' B)$  的结构继续按照 unifier 算法讨论, 现在仅考虑  $\sigma' A = x$  时, 即 step2. 对于任意  $\delta$  unifies  $\mathcal{C}$ , 也有  $\delta$  unifies  $\mathcal{C}'$ , 那么这里存在  $\gamma$  使得  $\delta = \gamma \circ \sigma'$ , 这里提前假设给我们带来的. 而这里的  $\sigma = \text{unify}(\mathcal{C}) = (\sigma' A \rightarrow \sigma' B) \circ \sigma'$ , 要是能证明

$$\delta = \gamma \circ (\sigma' A \rightarrow \sigma' B) \circ \sigma'$$

就好了. 对于任意的 type variable  $x$ , 当  $\sigma' x \neq \sigma' A$  时, 有  $(\gamma \circ (\sigma' A \rightarrow \sigma' B) \circ \sigma')x = (\gamma \circ \sigma')x = \delta x$ ; 当  $\sigma' x = \sigma' A$  时, 有

$$(\gamma \circ (\sigma' A \rightarrow \sigma' B) \circ \sigma')x = \gamma(\sigma' B) = \delta(B),$$

而  $\delta(A) = \delta(B)$ , 因此  $\delta(B) = (\gamma \circ \sigma')A = \delta(x)$ .

Q. E. D.

**Annotation 159** 你可能会在评估 unifier 算法的时候, 想如果不同的 type templates 含有相同的 type variables 是否出现问题? 只要对应 type templates, 不出现上面提到的那种问题, 就不会有问题. 因为我们处理的过程是 one by one, 那么处理下一个的时候, 前面的所有已经被某个  $\sigma$  给弄成一样了, 这样其中一个变, 全部都得变, 最后还是一样的.

**Theorem 160 (type inference)** The function  $\text{typeinfer}(M, B)$ , partially defined as follows, finds the most general substitution  $\sigma$  such that  $x_1 : \sigma A_1, \dots, x_n : \sigma A_n \vdash M : \sigma B$  is a valid typing judgment if it exists; and fails otherwise.

1. (var)  $\text{typeinfer}(\Gamma, x_i : A_i \vdash x_i : B) = \text{unify}(A_i, B)$ ;
2. (app)  $\text{typeinfer}(\Gamma \vdash MN : B) = f \circ g$ , where  $g = \text{typeinfer}(\Gamma \vdash M : X \rightarrow B)$  and  $f = \text{typeinfer}(g\Gamma \vdash N : gX)$  for a fresh type variable  $X$ .
3. (abs)  $\text{typeinfer}(\Gamma \vdash \lambda x. M : B) = f \circ g$  where  $g = \text{unify}(B, z \rightarrow z')$  and  $f = \text{typeinfer}(g\Gamma, x : gz \vdash M : gz')$  for fresh type variables  $z, z'$ .

**Annotation 161** 关于 type inference 有两种方法, 但实质都是一样的.

1. 我们上述提到的方法实际上在一边构造 constraints of types, 同时也在解决这些 constraints.
2. 另外一种方法是首先把某个 term 对应的 constraints 都构造出来, 最后我们可以得到一个关于该 term 的一个 shape of type  $S$ , 然后我们再去找到满足该 constraints 的  $\sigma$ , 最后对应 term 的 type 为  $\sigma S$ . 构造 constraints 的过程采用 inference rules.

无论哪种方法, 我们都需要证明该算法的 soundness 和 completeness. 其中

1. soundness 指该算法得到的  $\sigma$  和  $\sigma B$  是 well-defined.
2. completeness 指任意一个满足上述  $\sigma$ , 有一个利用上述算法得到的  $\sigma'$  使得  $\sigma B = \sigma' B$ .

**Definition 162** Suppose  $\Gamma \vdash M : \pi$ , where  $\pi$  is type template. If every possible type  $T$  for  $M$  is a instance of  $\pi$ , that is,  $T = \sigma\pi$ , we say  $\pi$  is a **principle type** of  $M$ .

**Theorem 163** If  $\Gamma \vdash M : B$  has possible solution, then  $M$  has principle type.

## Let Polymorphism

**Annotation 164** 如果想要实现如下语法

```
let double = λf : τ → τ. λx : τ. f(f(x)) in
let a = double (λx : nat. succ (succ x)) 1 in
let b = double (λx : bool. x) false in ...
```

在现有 let 语义下有些问题，这里会给出两个矛盾的 constraints 即

$$\tau \rightarrow \tau = \text{nat} \rightarrow \text{nat}, \tau \rightarrow \tau = \text{bool} \rightarrow \text{bool}.$$

出现这种问题的原因在于 let 本身的 typing inference rule

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \text{T-LET}$$

这里将  $e_1$  和  $x$  的 type 绑定起来了，因此首先我们首先去掉这种绑定状态。

**Definition 165** Let-polymorphism typing rule as follow

$$\frac{\Gamma \vdash [x \rightarrow e_1]e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \text{T-LETPOLY}$$

**Annotation 166** 这样我们直接将 let-body 中的  $x$  替换成  $e_1$ ，去掉了单独关于  $e_1$  的单独 typechecking，等到它在 let-body 中出现的时候在考虑其 type。

这样还是有一点问题就是 double 里面还是存在 argument annotation，还是解决不了问题，我们得想办法把不同 case 里面的 argument annotation 用不同的 variables 表示。这里我们在语法上引入不带 annotation 的 lambda abstraction，在 parser typing 阶段再给 assign 一个 annotation。这样做的好处就是对同一个 abstraction，我们能拷贝不同 annotation 的 abstraction。

**Definition 167** Typing rule for unannotated abstraction.

$$\frac{X \notin \mathcal{X} \quad \Gamma, x : A \vdash e_1 : B|_{\mathcal{X}}}{\Gamma \vdash \lambda x. e_1 : A \rightarrow B}$$

where  $\mathcal{X}$  represents type variable used in  $\Gamma, e_1$  and  $B$ .

**Annotation 168** 现在就可以正常工作了，但是还有一些不那么明显的问题：

1. 如果  $e_1$  没有在 let-body 中出现，那么它永远不会被 typecheck。我们可以改写一下 let-poly 来解决这个问题：

$$\frac{\Gamma \vdash [x \rightarrow e_1]e_2 : \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \text{T-LETPOLY}$$

2. 因为  $e_1$  将会被替换到 let-body 每一处它出现的地方, 那么每个地方都会经历 typecheck, 无论其是否含有 type variables. 我们最好首先对  $e_1$  做一个 simplify, 可以先把  $e_1$  提出来, 构造关于  $e_1$  的 principle type  $\pi$ ; 然后 generalize  $\pi$  里面出现的 type variables  $X_1, \dots, X_2$ , 注意不包含  $\Gamma$  里面出现的 type variables, 仅仅是对  $e_1$  的刻画, 利用这些 type variables 构造一个 type scheme  $\forall X_1, \dots, X_2 \pi$ ; 最后开始 typecheck let-body, 对于每一个  $x$  出现的地方, 我们都实例化一下这个 type scheme, 即给里面每个 type variables 都 assign 一个 fresh variables 得到一个 type, 再把这个 type 给  $x$ .

## 参考文献

- [1] Mario Román García. Category Theory and Lambda Calculus. <https://mroman42.github.io/ctlc/ctlc.pdf>