# Types and Programming Language

枫聆

2022 年 3 月 26 日

## 目录

# Introduction

**Definition 1** A type system is a tractable syntactic method for proving the absence of certain program behaviors by classlying 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 cant prove their presence.

**Example 3**

```
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 $\texttt{if} t_1 \texttt{then} t_2 \texttt{else} t_3 \in \mathcal{T}$.

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

$$\begin{array}{ccc} \text{true} \in \mathcal{T} & \text{false} \in \mathcal{T} & 0 \in \mathcal{T} \\ \dfrac{t_1 \mathcal{T}}{\text{succ} t_1 \in \mathcal{T}} & \dfrac{t_1 \mathcal{T}}{\text{succ} t_1 \in \mathcal{T}} & \dfrac{t_1 \mathcal{T}}{\text{succ} t_1 \in \mathcal{T}} \\ & \dfrac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T} \quad t_3 \in \mathcal{T}}{\texttt{if} t_1 \texttt{then} t_2 \texttt{else} t_3} & \end{array}$$

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

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

**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 Definiton2.1(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 $\mathsf{t} \in S_i(X)$, 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 $\mathsf{s}$, given $P(\mathsf{r})$ for all immediate subterms $\mathsf{r}$ of $\mathsf{s}$, we can show $P(\mathsf{s})$, then $P(\mathsf{s})$ holds for all $\mathsf{s}$.

# 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 mechine 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 interpretaion function, 通过这个 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. Alegbraic 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 pro-gramming languages

**Definition 18** $\lambda$ 演算的定义 The lambda-calculus (or -calculus) embodies this kind of function defi-nition 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.

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

A variable $\mathsf{x}$ by itself is a term; the abstraction of a variable $\mathsf{x}$ from a term $\mathsf{t}_1$, written $\lambda x.\,t_1$, is a term; and the application of a term $\mathsf{t}_1$ to another term $\mathsf{t}_2$, written $\mathsf{t}_1\ \mathsf{t}_2$, 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

$$\mathsf{s}\ \mathsf{t}\ \mathsf{u} = (\mathsf{s}\ \mathsf{t})\ \mathsf{u}.$$

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 $\mathsf{x}$ is said to be bound when it occurs in the body $\mathsf{t}$ of an abstraction $\lambda x.\,\mathsf{t}$.(More precisely, it is bound by this abstraction. Equivalently, we can say that $\lambda x$ is a binder whose scope is $\mathsf{t}$.) 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,

$$\mathsf{id} = \lambda x.\,x.$$

**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

$$(\lambda x.\, \mathsf{t}_{12})\ \mathsf{t}_2 \to [x \mapsto \mathsf{t}_2]\, \mathsf{t}_{12},$$

where $[x \mapsto \mathsf{t}_2]$ means "the term obtainted by replacing all free occurences of $x$ in $\mathsf{t}_{12}$ by $t_2$".

**Definition 23** 可约表达式 A term of the form $(\lambda x.\, \mathsf{t}_{12})\ \mathsf{t}_2$ 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 $\mathsf{id}\ (\mathsf{id}(\lambda z.\, \mathsf{id}\ z))$. This term contains three redexes:

   $$\underline{\mathsf{id}\ (\mathsf{id}\ (\lambda z.\, \mathsf{id}\ z))}$$
   $$\mathsf{id}\ (\underline{\mathsf{id}\ (\lambda z.\, \mathsf{id}\ z)})$$
   $$\mathsf{id}\ (\mathsf{id}\ (\lambda z.\, \underline{\mathsf{id}\ z}))$$

   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:

   $$\begin{aligned} &\mathsf{id}\ (\mathsf{id}\ (\lambda z.\, \underline{\mathsf{id}\ z})) \\ \to &\mathsf{id}\ (\underline{\mathsf{id}\ (\lambda z.\, z)}) \\ \to &\underline{\mathsf{id}\ (\lambda z.\, z)} \\ \to &\lambda z.\, z \\ \not\to & \end{aligned}$$

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

$$
\begin{aligned}
&\underline{\mathsf{id}\ (\mathsf{id}\ (\lambda z.\, \mathsf{id}\ z))} \\
\to&\underline{\mathsf{id}\ (\lambda z.\, \mathsf{id}\ z)} \\
\to&\lambda z.\, \underline{\mathsf{id}\ z} \\
\to&\lambda z.\, z \\
\not\to&
\end{aligned}
$$

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

$$
\begin{aligned}
&\underline{\mathsf{id}\ (\mathsf{id}\ (\lambda z.\, \mathsf{id}\ z))} \\
\to&\underline{\mathsf{id}\ (\lambda z.\, \mathsf{id}\ z)} \\
\to&\lambda z.\, \mathsf{id}\ z \\
\not\to&
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathsf{id}\ \underline{(\mathsf{id}\ (\lambda z.\, \mathsf{id}\ z))} \\
\to&\underline{\mathsf{id}\ (\lambda z.\, \mathsf{id}\ z)} \\
\to&\lambda z.\, \mathsf{id}\ z \\
\not\to&
\end{aligned}
$$

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.

# Programming in the Lambda-Calculus

**Definition 25** 高阶函数 A higher order function is a function that takes a function as an argument, or returns a function.

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

**Annotation 26** 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 27** 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 $f = \lambda(x, y).\, \mathsf{s}$, as we might in a richer programming language, we write $f = \lambda x.\, \lambda y.\, \mathsf{s}$. we then apply $f$ to it arguments one at times, write $f\ v\ w$, which reduces to

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

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

**Annotation 28** 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 test $b\ v\ w$ reduces to $v$ when $b$ is **tru** and reduces $w$ when $b$ is **fls**.

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

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

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

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:

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

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

**Annotation 29** 有序对 Using booleans, we can encode pairs of values as terms.

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

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

**Annotation 30** 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 31** 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 32** 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 $f^{\circ(m+n)}(z) = f^{\circ m}(f^{\circ n}(x))$，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 33**

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

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 $s^{\circ n} = \lambda z.\, s(s(\cdots(s\ z)\cdots))$，它并不是一个标准的 succ 形式

**Annotation 34**

$$\exp = \lambda m.\, \lambda n.\, n\ m$$

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.

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

## Normal Forms

**Annotation 35** 前面提到的 neural 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 $\mathsf{x}\ \mathsf{y}$，其中 $\mathsf{x}$ 是一个 free variable，而 $\mathsf{y}$ 是一个 lambda abstraction.

**Definition 36** In untyped lambda calculus, the neutral terms and the normal form are generated in the following rules.

$$\frac{\mathsf{t}\ nf}{\lambda\mathsf{x}.\,\mathsf{t}\ nf} \qquad \frac{\mathsf{t}\ ne}{\mathsf{t}\ nf} \qquad \frac{\mathsf{t}_1\ ne \quad \mathsf{t}_2\ nf}{\mathsf{t}_1\ \mathsf{t}_2\ \mathsf{ne}} \qquad \frac{}{\mathsf{x}\ \mathsf{ne}}$$

**Annotation 37** 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 $\vdash e : \alpha \to (\alpha \to \alpha)$ and $e$ is normal form, then $e = true$ or $e = false$”，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...

# Simple Types

## Typed Arithmetic Expressions

**Definition 38** The typing relation for arithmetic expressions, written

$$\mathsf{t} : \mathrm{T}$$

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

$$\mathsf{true} : \mathrm{bool}$$

$$\mathsf{false} : \mathrm{bool}$$

$$\frac{\mathsf{t}_1 : \mathrm{bool} \quad \mathsf{t}_2 : \mathrm{T} \quad \mathsf{t}_3 : \mathrm{T}}{\textbf{if } \mathsf{t}_1 \textbf{ then } \mathsf{t}_2 \textbf{ else } \mathsf{t}_3 : \mathrm{T}}$$

$$\mathsf{0} : \mathrm{nat}$$

$$\frac{\mathsf{t}_1 : \mathrm{nat}}{\mathsf{succ}\ \mathsf{t}_1 : \mathrm{nat}}$$

$$\frac{\mathsf{t}_1 : \mathrm{nat}}{\mathsf{pred}\ \mathsf{t}_1 : \mathrm{nat}}$$

$$\frac{\mathsf{t}_1 : \mathrm{nat}}{\mathsf{iszero}\ \mathsf{t}_1 : \mathrm{bool}}$$

**Annotation 39** 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 40** 使用 inference rule 来描述 type 是为了更方便地证明 inductive theorem.

**Definition 41** A term $\mathsf{t}$ is typbale or well typed if there is some T such that $\mathsf{t} : \mathrm{T}$. If $\mathsf{t}$ is typable, then its type is unique(uniquness of types).

**Annotation 42** 这里很重要是理解如果给定一个 type relation $\mathsf{t} : \mathrm{T}$，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 43** 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 $\mathsf{false}, \mathsf{true}, 0, \mathsf{succ}\ \mathsf{nv}$ 都是明显的 values, 其中 $\mathsf{nv}$ 表示一个 numeric value.

*Case 1* $\mathsf{t} = \textbf{if}\ \mathsf{t}_1\ \textbf{then}\ \mathsf{t}_2\ \textbf{else}\ \mathsf{t}_3 \quad \mathsf{t}_1 = \text{bool} \;\; \mathsf{t}_2 = \text{T} \;\; \mathsf{t}_3 = \text{T}.$

由归纳假设当 $\mathsf{t}_1 = \mathsf{true}$ 或者 $\mathsf{t}_1 = \mathsf{false}$ 时，我们对 $\mathsf{t}$ 一步 evaluation 得到 $\mathsf{t}_2$ 或者 $\mathsf{t}_3$. 另外当 $\mathsf{t}_1 \to \mathsf{t}_1'$ 时，我们也可以得到 $\mathsf{t} \to \textbf{if}\ \mathsf{t}_1'\ \textbf{then}\ \mathsf{t}_2\ \textbf{else}\ \mathsf{t}_3$.

*Case 2* $\mathsf{t} = \mathsf{succ}\ \mathsf{t}_1 \quad \mathsf{t}_1 = \text{nat}.$

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

*Case 3* $\mathsf{t} = \mathsf{pred}\ \mathsf{t}_1 \quad \mathsf{t}_1 = \text{nat}.$

同上.

*Case 4* $\mathsf{t} = \mathsf{iszero}\ \mathsf{t}_1 \quad \mathsf{t}_1 = \text{nat}.$

同上.

**Annotation 44** 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 45** preservation If a well-typed term takes a step of evaluation, then the resulting term is also well typed.

**Definition 46**

$$\text{safty} = \text{progress} + \text{preservation}.$$

# Simply Typed Lambda-Calculus

**Definition 47** Define the type of $\lambda$-abstraction(function) as follow

$$\lambda x.\, t : T_1 \to T_2$$

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

**Annotation 48** 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 $\lambda x.\, t_2 : T_1 \to T_2$，形式化的描述这个 type rule 即为

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

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

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

**Annotation 50** 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

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

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

**Annotation 51** 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 52** rule of applications

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

**Annotation 53** rule of conditionals

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathrm{bool} \quad \Gamma \vdash \mathsf{t}_2 : \mathrm{T} \quad \Gamma \vdash \mathsf{t}_3 : \mathrm{T}}{\Gamma \vdash \textbf{if } \mathsf{t}_1 \textbf{ then } \mathsf{t}_2 \textbf{ else } \mathsf{t}_3 : \mathrm{T}}$$

**Annotation 54** We often use $\lambda_{\rightarrow}$ to refer to the simply typed lambda-calculus.

**Theorem 55** uniquness of types In a given typing context $\Gamma$, a term $\mathsf{t}$ has at most one type. That is, if a term is typable, then it's type is unique.

**Lemma 56** canonical forms

1. If $\mathsf{v}$ is a value of type bool, then $\mathsf{v}$ is either $\mathsf{true}$ or $\mathsf{false}$;
2. If $\mathsf{v}$ is a value of type $\mathrm{T}_1 \rightarrow \mathrm{T}_2$, then $\mathsf{v} = \lambda \mathsf{x} : \mathrm{T}_1 . \mathsf{t}_2$.

**Lemma 57** weakening If $\Gamma \vdash \mathsf{t} : \mathrm{T}$ and $x \notin \mathrm{dom}(\Gamma)$, then $\Gamma, \mathsf{x} : \mathrm{S} \vdash \mathsf{t} : \mathrm{T}$.

**Theorem 58** progress Suppose $\mathsf{t}$ is a closed, well-typed term(that is $\vdash \mathsf{t} : \mathrm{T}$). Then either $\mathsf{t}$ is a value or else there is some $\mathsf{t}'$ with $\mathsf{t} \rightarrow \mathsf{t}'$.

Proof proved by structural induction. Q. E. D.

**Theorem 59** preservation under substitution If $\Gamma, \mathsf{x} : \mathrm{S} \vdash \mathsf{t} : \mathrm{T}$ and $\Gamma \rightarrow \mathsf{s} : \mathrm{S}$, then $\Gamma \vdash [\mathsf{x} \rightarrow \mathsf{s}]\mathsf{t} : \mathrm{T}$.

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 $\mathsf{t}$ 来进行归纳.

*Case 1* 若 $\mathsf{t} = \mathsf{v}$，其中 $\mathsf{v}$ 为一个 variable.

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

**Annotation 60** 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. 然后我们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.
- 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.

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 Curry-style 通常用于刻画 explicitly typed system.

# Type Extensions

## Known Types

**Definition 61** base type Something like bool, nat, float and string, these type are for describing simple and unstructured values and approriate primitive operation for manipulating these values.

**Definition 62** unit type a constant $\mathsf{unit}$ with unique type Unit, the type can be only from this constant.

**Definition 63** The sequencing notation $\mathsf{t}_1;\mathsf{t}_2$ has the effect of evaluating $\mathsf{t}_1$ , throwing away its trivial result($\mathsf{unit}$), and going on to evaluate $\mathsf{t}_2$.

**Annotation 64** first way to formalize sequencing Add $\mathsf{t}_1;\mathsf{t}_2$ as a new alternative in the syntax of terms, and then add two evaluation rules

$$\frac{\mathsf{t}_1 \to \mathsf{t}_1'}{\mathsf{t}_1;\mathsf{t}_2 \to \mathsf{t}_1';\mathsf{t}_2}$$

$$\mathsf{unit};\mathsf{t}_2 \to \mathsf{t}_2$$

and a typing rule

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathrm{Unit} \quad \Gamma \vdash \mathsf{t}_2 : \mathrm{T}_2}{\Gamma \vdash \mathsf{t}_1;\mathsf{t}_2 : \mathrm{T}_2}$$

**Annotation 65** second way to formalize sequencing Regard $\mathsf{t}_1;\mathsf{t}_2$ as an abbreviation for the term $(\lambda\mathsf{x} : \mathrm{Unit}.\, \mathsf{t}_2)\ \mathsf{t}_1$, where $\mathsf{x} \in \mathrm{FV}(\mathsf{t}_2)$.

**Theorem 66** Suppose $\lambda^E$ for the simply typed lambda-calcuclus with the first way of sequencing formalization and $\lambda^I$ for the simply typed lambda-calculus with Unit. Let $e\colon \lambda^E \to \lambda^I$ be the elaboration function that translates from the $\lambda^E$ to $\lambda^I$ by replacing every occurrence of $\mathsf{t}_1;\mathsf{t}_2$ with $(\lambda\mathsf{x} : \mathrm{Unit}.\, \mathsf{t}_2)\ \mathsf{t}_1$, where $\mathsf{x} \in \mathrm{FV}(\mathsf{t}_2)$. Then for each $\mathsf{t}$ of $\lambda^E$, we have

1. $\mathsf{t} \to_E \mathsf{t}'$ iff $e(\mathsf{t}) \to_I e(\mathsf{t}')$;
2. $\Gamma \vdash_E \mathsf{t} : \mathrm{T}$ iff $\Gamma \vdash_I e(\mathsf{t}) : \mathrm{T}$.

**Annotation 67** 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 $\mathsf{t}_1$ 的 evaluation result, 我们对一个 sequencing 能做的就是首先对 $\mathsf{t}_1$ 进行 evaluating，只有它的 result 是一个 Uint 的时候，我们可以尝试丢掉它转而去处理 $\mathsf{t}_2$. 显然当 $\mathsf{t}_1$ 不是 trivial 的时候，$\mathsf{t}_2$ 永远得不到的 evaluating，就停在了某个 $\mathsf{t}_1';\mathsf{t}_2$. 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 $\mathsf{t}_1$ 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 68** Ascription is simple feature for ascribe a particular type to a given term. We write "$\mathsf{t}$ as T" for the "the term $\mathsf{t}$, to which we ascribe the type T".

**Definition 69** Let Bindings $\mathsf{let}\ \mathsf{x} = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2$, 它们的 evaluation rule 和 type rule 根 lambda abstraction 是差不多的，即

$$\mathsf{let}\ \mathsf{x} = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2 = (\lambda x : \mathrm{T}_1.\, \mathsf{t}_2)\ \mathsf{t}_1.$$

**Definition 70** Pair Pairing, written $\mathsf{t} = \langle \mathsf{t}_1, \mathsf{t}_2 \rangle$ and projection, written $\mathsf{t}.1$ for the $\mathsf{t}_1$ and $\mathsf{t}.2$ for the $\mathsf{t}_2$. One new type constructor, $\mathrm{T}_1 \times \mathrm{T}_2$, called the product of $\mathrm{T}_1$ and $\mathrm{T}_2$.

**Definition 71** Tuple is general formalization of Pair.

**Definition 72** Record Recording, written $\{\mathsf{l}_1 = \mathsf{t}_1, \cdots, \mathsf{l}_n = \mathsf{t}_n\}$ and thier type $\{\mathsf{l}_1 : \mathrm{T}_1, \cdots, \mathsf{l}_1 : \mathrm{T}_1\}$.

**Definition 73** pattern matching Given two kinds of patterns, varible pattern $\mathsf{x}$ and record pattern $\{\mathsf{l}_1 = \mathsf{p}_1, \cdots, \mathsf{l}_n = \mathsf{p}_n\}$(so it can be nested). Plus a match function $match$: $P \times V \to \mathrm{Subs} \cup \mathrm{Fail}$, where $P$ are patterns, $V$ is values, Subs are substitutions and Fail means matching fails. The matching rules as follow

$$match(\mathsf{x}, \mathsf{v}) = [\mathsf{x} \to \mathsf{v}] \qquad M - Var$$

$$\frac{\text{for each } i\ match(\mathsf{p}_i, \mathsf{v}_i) = \sigma_i}{match(\{\mathsf{l}_1 = \mathsf{p}_1, \cdots, \mathsf{p}_n = \mathsf{p}_n\}, \{\mathsf{l}_1 = \mathsf{v}_1, \cdots, \mathsf{l}_n = \mathsf{v}_n\}) = \sigma_1 \circ \cdots \circ \sigma_n} \qquad M - Rcd$$

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

$$\mathsf{let}\ \mathsf{p} = \mathsf{v}\ \mathsf{in}\ \mathsf{t} = match(\mathsf{p}, \mathsf{v})\ \mathsf{t}_1.$$

**Definition 74** A sum type is written as $\mathrm{T}_1 + \mathrm{T}_2$, there are two terms can be desribed this type:

1. Assume $\mathsf{t}_1 : \mathrm{T}_1$, then $\mathrm{inl}\ \mathsf{t}_1 : \mathrm{T}_1 + \mathrm{T}_2$;
2. Assume $\mathsf{t}_2 : \mathrm{T}_2$, then $\mathrm{inr}\ \mathsf{t}_2 : \mathrm{T}_1 + \mathrm{T}_2$.

There is a $\mathsf{case}$ construct that allows us to distinguish whether a given value comes from the left or right branch of a sum,

$$\textbf{case}\ a\ \textbf{of}\ \mathrm{inl}\ x_1 \mapsto x_1.1 \mid \mathrm{inr}\ x_2 \mapsto x_2.1$$

**Annotation 75** 这里存在一个类型唯一性的问题，如果 $\mathsf{t}_1 : \mathrm{T}_1$，那么对于任意 $\mathrm{T}_2$，都有 $\mathrm{inl}\ \mathsf{t}_1 : \mathrm{T}_1 + \mathrm{T}_2$, 显然 $\mathrm{inl}\ \mathsf{t}_1$ 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 76** variant is generalization of sum $\langle \mathsf{l}_1 : \mathsf{T}_1, \mathsf{l}_2 : \mathsf{T}_2 \rangle$.

**Definition 77** option $\langle \mathsf{none} : \mathsf{Unit}, \mathsf{some} : \mathsf{Nat} \rangle$.

**Definition 78** enumeration An enumerated type (or enumeration) is a variant type in which the field type associated with each label is Unit.

**Definition 79** single-field variant $\langle \mathsf{l} : \mathsf{T} \rangle$.

**Annotation 80** 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.

# Normalization

**Theorem 81** If $\vdash \mathsf{t} : \mathrm{T}$, then $\mathsf{t} \to^* \mathsf{v}$, where $\mathsf{v}$ is a value, or abbreviate $\mathsf{t} \Downarrow$.

**Remark 82** 上述 normalization theorem 使用是 simply typed lambda calculus.

**Annotation 83** 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

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathrm{T}_1 \to \mathrm{T}_2 \quad \Gamma \vdash \mathsf{t}_2 : \mathrm{T}_2}{\Gamma \vdash \mathsf{t}_1 \mathsf{t}_2 : \mathrm{T}_2}$$

根据假设 $\mathsf{t}_1, \mathsf{t}_2$ 都是 normalizable，那么设 $\mathsf{t}_1 \to^* \mathsf{t}'_1 = \lambda x : \mathrm{T}_1 . \mathsf{t}_3$(这里用了一下 value of function type 的 canonical form) 和 $\mathsf{t}_2 \to^* \mathsf{t}'_2$，其中 $\mathsf{t}'_2$ 是 normalized. 再来一个 $\beta$ reduction，则有

$$\mathsf{t}'_1 \mathsf{t}'_2 = [x \to \mathsf{t}'_2]\mathsf{t}'_3$$

这里有两个问题: (1)$\mathsf{t}_3$ 是一个怎样的形式? (2) substitution 干了什么?

**Definition 84** Suppose the logical predicate for strong normalization as follow

$$\begin{aligned} &\mathrm{SN}_{\mathrm{A}}(\mathsf{t}) \iff \vdash \mathsf{t} : \mathrm{A} \ \wedge \ \mathsf{t} \Downarrow, \\ &\mathrm{SN}_{\mathrm{T}_1 \to \mathrm{T}_2}(\mathsf{t}) \iff \vdash \mathsf{t} : \mathrm{T}_1 \to \mathrm{T}_2 \ \wedge \ \mathsf{t} \Downarrow \wedge \forall \mathsf{t}_1 . \, \mathrm{SN}_{\mathrm{T}_1}(\mathsf{t}_1) \Rightarrow \mathrm{SN}_{\mathrm{T}_2}(\mathsf{t} \ \mathsf{t}_1) \end{aligned}$$

where A is base type.

**Annotation 85** 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 $\vdash \mathsf{t} : \mathrm{T} \Rightarrow \mathrm{SN}_{\mathrm{T}}(\mathsf{t})$，即所有 closed well-typed 的 term 都复合上述定义的 logical predicate,
2. 然后 $\mathrm{SN}_{\mathrm{T}}(\mathsf{t}) \Rightarrow \mathsf{t} \Downarrow$.

这种手法就是所谓logical relation证明方法.

**Lemma 86** $\mathrm{SN}_{\mathrm{T}}(\mathsf{t}) \Rightarrow \mathsf{t} \Downarrow$

Proof 根据定义这是显然的. Q. E. D.

**Annotation 87** 证明过程的第一步又会拆成两步:

1. $\mathrm{SN}_{\mathrm{T}}(\mathsf{t})$ 将会在 t 的 evalution 过程中保持,

2. 再做根据 type derivations 的 induction，但是于证明 abstraction $\mathsf{t} = \lambda x : \mathrm{T}_1 . \mathsf{t}_2$ 满足 $\mathrm{SN}_{\mathrm{T}_1 \to \mathrm{T}_2}(\mathsf{t})$ 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

$$\frac{\mathsf{x} : \mathrm{T}_1 \vdash \mathsf{t}_2 : \mathrm{T}_2}{\vdash \lambda x : \mathrm{T}_1 . \mathsf{t}_2}$$

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 $\Gamma \vdash \mathsf{t} : \mathrm{T} \Rightarrow \mathrm{SN}_{\mathrm{T}}(\mathsf{t})$. 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 88** If $\mathsf{t} : \mathrm{T}$ and $\mathsf{t} \to \mathsf{t}'$, then $\mathrm{SN}_{\mathrm{T}}(\mathsf{t}) \iff \mathrm{SN}_{\mathrm{T}}(\mathsf{t}')$

Proof 首先由 $\mathsf{t} \to \mathsf{t}'$，那么有 $\mathsf{t} \Downarrow \iff \mathsf{t}' \Downarrow$. 再分情况，若 $\mathrm{T} = \mathrm{A}$，证明就结束了; 若 $\mathrm{T} = \mathrm{T}_1 \to \mathrm{T}_2$，由 $\mathsf{t}\ \mathsf{t}_1 \to \mathsf{t}'\ \mathsf{t}_1$，则 $\mathsf{t}\ \mathsf{t}_1 \Downarrow \iff \mathsf{t}'\ \mathsf{t}_1 \Downarrow$，又回到第一种情况，证明了 function type 额外需要的条件. Q. E. D.

**Lemma 89** If $\mathsf{x}_1 : \mathrm{T}_1, \mathsf{x}_2 : \mathrm{T}_2, \cdots, \mathsf{x}_n : \mathrm{T}_n \vdash \mathsf{t} : \mathrm{T}$ and $\mathsf{v}_1, \mathsf{v}_1, \cdots, \mathsf{v}_1$ are closed values of $\mathrm{T}_1, \mathrm{T}_2, \cdots, \mathrm{T}_n$ with $\mathrm{SN}_{\mathrm{T}_i}(\mathsf{v}_i)$, then $\mathrm{SN}_{\mathrm{T}}([\mathsf{x}_1 \to \mathsf{v}_1, \mathsf{x}_2 \to \mathsf{v}_2, \cdots, \mathsf{x}_n \to \mathsf{v}_n]\mathsf{t})$

Proof structural induction as follow

*Case 1*

$$\mathsf{t} = \mathsf{x}_i$$
$$\mathrm{T} = \mathrm{T}_i$$

显然成立.

*Case 2*

$$\mathsf{t} = \lambda \mathsf{x} : \mathrm{S}_1 . \mathsf{s}_2$$
$$\mathrm{T} = \mathrm{S}_1 \to \mathrm{S}_2$$
$$\mathsf{x}_1 : \mathrm{T}_1, \mathsf{x}_2 : \mathrm{T}_2, \cdots, \mathsf{x}_n : \mathrm{T}_n, \mathsf{x} : \mathrm{S}_1 \vdash \mathsf{s}_2 : \mathrm{S}_2$$

显然此时 $[\mathsf{x}_1 \to \mathsf{v}_1, \mathsf{x}_2 \to \mathsf{v}_2, \cdots, \mathsf{x}_n \to \mathsf{v}_n]\mathsf{t}$ 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 $\mathrm{SN}_{\mathrm{S}_1}(\mathsf{s})$，有 $\mathrm{SN}_{\mathrm{S}_2}(([\mathsf{x}_1 \to \mathsf{v}_1, \mathsf{x}_2 \to \mathsf{v}_2, \cdots, \mathsf{x}_n \to \mathsf{v}_n]\mathsf{t})\ \mathsf{s})$. 根据 Lemma86，我们有 $\mathsf{s} \to^* \mathsf{v}$，根据归纳假设即有

$$\mathrm{SN}_{\mathrm{S}_2}([\mathsf{x}_1 \to \mathsf{v}_1, \mathsf{x}_2 \to \mathsf{v}_2, \cdots, \mathsf{x}_n \to \mathsf{v}_n, \mathsf{x} \to \mathsf{v}]\mathsf{t})$$

而

$$([\mathsf{x}_1 \to \mathsf{v}_1, \mathsf{x}_2 \to \mathsf{v}_2, \cdots, \mathsf{x}_n \to \mathsf{v}_n]\mathsf{t})\ \mathsf{s} \to^* [\mathsf{x}_1 \to \mathsf{v}_1, \mathsf{x}_2 \to \mathsf{v}_2, \cdots, \mathsf{x}_n \to \mathsf{v}_n, \mathsf{x} \to \mathsf{v}]\mathsf{t},$$

再用一下 Lemma88, 即可得到我们想要的.

*Case 3*

$$\begin{aligned}&\mathsf{t} = \mathsf{t}_1\mathsf{t}_2\\&\mathsf{x}_1 : \mathrm{T}_1, \mathsf{x}_2 : \mathrm{T}_2, \cdots, \mathsf{x}_n : \mathrm{T}_n \vdash \mathsf{t}_1 : \mathrm{T}_{11} \to \mathrm{T}_{12}\\&\mathsf{x}_1 : \mathrm{T}_1, \mathsf{x}_2 : \mathrm{T}_2, \cdots, \mathsf{x}_n : \mathrm{T}_n \vdash \mathsf{t}_2 : \mathrm{T}_{11}\\&\mathrm{T} = \mathrm{T}_{12}\end{aligned}$$

根据归纳假设有 $\mathrm{SN}_{\mathrm{T}_{11}\to\mathrm{T}_{12}}([\mathsf{x}_1 \to \mathsf{v}_1, \mathsf{x}_2 \to \mathsf{v}_2, \cdots, \mathsf{x}_n \to \mathsf{v}_n]\mathsf{t}_1)$ 和 $\mathrm{SN}_{\mathrm{T}_{11}}([\mathsf{x}_1 \to \mathsf{v}_1, \mathsf{x}_2 \to \mathsf{v}_2, \cdots, \mathsf{x}_n \to \mathsf{v}_n]\mathsf{t}_2)$. 再根据 $\mathrm{SN}_{\mathrm{T}_{11}\to\mathrm{T}_{12}}$ 的 definition，有

$$\begin{aligned}&\mathrm{SN}_{\mathrm{T}_{12}}([\mathsf{x}_1 \to \mathsf{v}_1, \mathsf{x}_2 \to \mathsf{v}_2, \cdots, \mathsf{x}_n \to \mathsf{v}_n]\mathsf{t}_1[\mathsf{x}_1 \to \mathsf{v}_1, \mathsf{x}_2 \to \mathsf{v}_2, \cdots, \mathsf{x}_n \to \mathsf{v}_n]\mathsf{t}_2)\\&= \mathrm{SN}_{\mathrm{T}_{12}}([\mathsf{x}_1 \to \mathsf{v}_1, \mathsf{x}_2 \to \mathsf{v}_2, \cdots, \mathsf{x}_n \to \mathsf{v}_n]\mathsf{t}_1\ \mathsf{t}_2)\end{aligned}$$

得证.

**Annotation 90** Lemma89中 substitution 可以记为 $\gamma = [\mathsf{x}_1 \to \mathsf{v}_1, \mathsf{x}_2 \to \mathsf{v}_2, \cdots, \mathsf{x}_n \to \mathsf{v}_n]$，也可以直接记为 $\gamma \models \Gamma$，理解为"the substitution $\gamma$ statisfies the type environment, $\Gamma$".

**Corollary 91** $\vdash \mathsf{t} : \mathrm{T} \Rightarrow \mathrm{SN}_{\mathrm{T}}(\mathsf{t})$.

Proof 直接从 Lemma89可得. Q. E. D.

## References

**Definition 92** 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

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

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

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

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

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

The result of the assignment is the trivial unit value.

**Definition 93** The references r and s are said to be aliases for the same cell.

**Annotation 94** 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 indexs $l \in \mathcal{L}$. states 会被抽象成 store function $\mathcal{L} \to$ 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 95** 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

$$\frac{\Sigma(l) = \mathrm{T}_1}{\Gamma|\Sigma \vdash l : \text{Ref T}}$$

**Annotation 96** 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

$$\frac{\Gamma|\mu \vdash \mu(l) : \mathrm{T}_1}{\Gamma|\mu \vdash l : \text{Ref T}}$$

如果 $\mu$ 的结构是这样

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

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 $\Sigma$: $\mathcal{L} \to T$ 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 97** (Connection between $\mu$ and $\Sigma$) A store $\mu$ is said to be well typed with respest to a typing context $\Gamma$ and a store typing $\Sigma$, written $\Gamma \mid \Sigma \vdash \mu$, if $\text{dom}(\mu) = \text{dom}(\mu)$ and $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ for every $l \in \text{dom}(\mu)$.

**Theorem 98** Preservation If

$$\Gamma \mid \Sigma \vdash \mathsf{t} : \mathrm{T}$$
$$\Gamma \mid \Sigma \vdash \mu$$
$$\mathsf{t} \mid \mu \to \mathsf{t}' \mid \mu'$$

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

$$\Gamma \mid \Sigma' \vdash \mathsf{t}' : \mathrm{T}$$
$$\Gamma \mid \Sigma' \vdash \mu'$$

**Annotation 99** 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 100** Progress Suppose $\mathsf{t}$ is closed, well-typed term, that is $\cdot \mid \Sigma \vdash \mathsf{t} : \mathrm{T}$ for some $\mathrm{T}$ and $\Sigma$. Then either $\mathsf{t}$ is a value or else, for any store $\mu$ such that $\cdot \mid \Sigma \vdash \mu$, there is some term $\mathsf{t}'$ and store $\mu'$ with $\mathsf{t} \mid \mu \to \mathsf{t}' \mid \mu'$.

# Subtyping

**Definition 101** 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 102** 首先举个例子 $\{x : \tau_1, y : \tau_2\} <: \{x : \tau_1\}$，这就相当于 $S$ 的类型蕴含着 $T$ 的类型，直觉上"the element of $S$ are a subset of the elements of $T$"，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 $T$.

**Definition 103** Subsumpting rule

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

**Lemma 104** Subtype relation satifies reflexivity and transitivity.

**Definition 105** Subtyping rule for records.

$$\{l_1 : T_1, l_2 : T_2, \cdots, l_n : T_n\} <: \{l_1 : T_1, l_2 : T_2, \cdots, l_{n+k} : T_{n+k}\} \quad \text{width subtyping}$$

$$\frac{\forall i.\ S_i <: T_i}{\{l_1 : T_1, l_2 : T_2, \cdots, l_n : T_n\} <: \{l_1 : S_1, l_2 : S_2, \cdots, l_n : S_n\}} \quad \text{depth subtyping}$$