Types and Programming Language

枫聆

2022 年 3 月 18 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 $\mathtt{if} t_1 \mathtt{then} t_2 \mathtt{else} t_3 \in \mathcal{T}$.

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

$$\text{true} \in \mathcal{T} \qquad \text{false} \in \mathcal{T} \qquad 0 \in \mathcal{T}$$

$$\frac{t_1 \mathcal{T}}{\text{succ} t_1 \in \mathcal{T}} \qquad \frac{t_1 \mathcal{T}}{\text{succ} t_1 \in \mathcal{T}} \qquad \frac{t_1 \mathcal{T}}{\text{succ} t_1 \in \mathcal{T}}$$

$$\frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T} \quad t_3 \in \mathcal{T}}{\mathtt{if} t_1 \mathtt{then} t_2 \mathtt{else} t_3}$$

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

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

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 s, given $P(\mathsf{r})$ for all immediate subterms r of s, we can show $P(\mathsf{s})$, then $P(\mathsf{s})$ 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 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.

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 λ 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.

$$\mathsf{t} ::=$$

$$\mathsf{x}$$

$$\lambda x.\,\mathsf{t}$$

$$\mathsf{t}\ \mathsf{t}.$$

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 $\mathsf{t}_1\ \mathsf{t}_2$, 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 x is said to be bound when it occurs in the body t of an abstraction $\lambda x.\,\mathsf{t}$.(More precisely, it is bound by this abstraction. Equivalently, we can say that λx 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,

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

Definition 21 α 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, α 变换在进行 β 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

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

where $[x \mapsto \mathsf{t}_2]$ means "the term obtainted by replacing all free occurences of x in 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 β-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 β-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 β-reduction, we might choose, for example, to begin with the innermost index, then do the one in the middle, then the outermost:

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

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

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

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

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

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.

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

注意 call by name 和 call by value 的区别，call by name 是在 λ 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:

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

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

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

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

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 α 变换，避免产生变量名的冲突.

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

Simple Types

Typed Arithmetic Expressions

Definition 35 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 36 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.

Definition 37 A term t is typbale or well typed if there is some T such that $\mathsf{t} : \mathrm{T}$. If t is typable, then its type is unique(uniquness of types).

Annotation 38 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 39 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 nv 表示一个 numeric value.

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

由归纳假设当 $\mathsf{t}_1 = \mathsf{true}$ 或者 $\mathsf{t}_1 = \mathsf{false}$ 时，我们对 t 一步 evaluation 得到 t_2 或者 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 $\mathsf{t} = \mathsf{succ}\ \mathsf{t}_1 \quad \mathsf{t}_1 = \mathrm{nat}.$

由归纳假设当 $\mathsf{t}_1 = \mathrm{nv}$ 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 $\mathsf{t} = \mathsf{pred}\ \mathsf{t}_1 \quad \mathsf{t}_1 = \mathrm{nat}.$

同上.

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

同上.

Annotation 40 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 41 preservation If a well-typed term takes a step of evaluation, then the resulting term is also well typed.

Definition 42

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

Simply Typed Lambda-Calculus

Definition 43 Define the type of λ-abstraction(function) as follow

$$\lambda \mathsf{x}.\, \mathsf{t} : \mathrm{T}_1 \to \mathrm{T}_2$$

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 44 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 $\mathsf{x} : \mathrm{T}_1$，同时将 t_2 中的所有出现的 x 的 type 都表示为 T_1 得到 $\mathsf{x} : \mathrm{T}_2$，那么显然此时就可以构造出一个 abstraction 和它对应 type 为 $\lambda \mathsf{x}.\, \mathsf{t}_2 : \mathrm{T}_1 \to \mathrm{T}_2$，形式化的描述这个 type rule 即为

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

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

Definition 45 A typing context Γ is a sequence of distinct variables and thier types as follow

$$\Gamma = \mathsf{x}_1 : \mathrm{T}_1, \mathsf{x}_2 : \mathrm{T}_2, \mathsf{x}_3 : \mathrm{T}_3, \cdots$$

Annotation 46 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

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

这里我们规定 t_2 中除 x 外的 free variables 均在 Γ 中.

Annotation 47 rule of variables A variable has whatever type we are currently assuming it to have,

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

Annotation 48 rule of applications

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

Annotation 49 rule of conditionals

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

Annotation 50 We often use $\lambda_{\rightarrow}$ to refer to the simply typed lambda-calculus.

Theorem 51 uniquness of types In a given typing context Γ, a term t has at most one type. That is, if a term is typable, then it's type is unique.

Lemma 52 weakening If $\Gamma \vdash \mathsf{t} : \mathrm{T}$ and $x \notin \text{dom}(\Gamma)$, then $\Gamma, \mathsf{x} : \mathrm{S} \vdash \mathsf{t} : \mathrm{T}$.

Theorem 53 progress Suppose t is a closed, well-typed term(that is $\vdash \mathsf{t} : \mathrm{T}$). Then either t is a value or else there is some t' with $\mathsf{t} \rightarrow \mathsf{t}'$.

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

Theorem 54 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 t 来进行归纳.

Case 1 若 $\mathsf{t} = \mathsf{v}$，其中 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 55 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 56 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 57 unit type a constant with unique type, the type can be only from this constant.