Type System + Security = ?

枫聆

2022 年 7 月 12 日

目录

Introduction

Type System

Definition 1.1. 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 [1].

Annotation 1.2. 个人的题外话: type system 就是一种非常巧妙工具能帮你抓住那些你可以尽可能抓住的东西，这些东西就是指那些用 types 所刻画的我们感兴趣的 properties. 我觉得这是一种艺术，一种难以言于笔下的艺术，凝聚了一代又一代计算理论先驱们的一种智慧 “我们还可以做到更好，我们还可以往前再走一点...”，这些东西需要你慢慢地在他们的字里行间去感受.

Annotation 1.3. 在 computation 里面有两个东西很重要: (1) 怎么做 encoding? i.e., 给定一些输入我们如何将其转换成我们当且 system 中可以接受的东西 (2) 怎么做 computing? i.e., 我们如何将输入转换成其对应的结果. 我们先用简单的 untyped lambda calculus 来慢慢说明.

Definition 1.4. Let Λ be the set of terms in lambda calculus, it is defined by the follow inductive process:

- If x is a variable, then $x \in \Lambda$.
- If x is a variable and $M \in \Lambda$, then $\lambda x.\, M \in \Lambda$.
- If $M, N \in \Lambda$, then $(MN) \in \Lambda$.

Annotation 1.5. 上述关于 lambda calculus 的定义实际上就是一个 encoding，其中 $\lambda x.\, M$ 称其为 abstraction. 如果我们从 Λ 任意地取一个 term t 出来如何对其对 computing 呢? 我们将赋予其两个重要的 reductions β 和 η，其实分别对应我们常见的 application (函数调用) 和 extensionality (函数等价).

Definition 1.6. if a variable x is in such term $\lambda x.\, M$, then we call x is bound, otherwise x is free.

Definition 1.7. A term of the form $(\lambda x.\, M)\ N$ is called redex (reducible expression), and the operation of rewriting a redex according to the above rule is called β-reduction, written as

$$(\lambda x.\, M)\ N \to [x \mapsto N]M$$

where $[x \mapsto N]$ is substitution means ”replacing all free occurences of x in M by N”.

Definition 1.8. Given a term $\lambda x.\, M$ in Λ, if x is not free in M, then we have a η-reduction as follow:

$$\lambda x.\, M \to M$$

Annotation 1.9. 那么现在拿到一个 term 之后就可以尝试按照上面的规则来做 reduction，实际上这个里面还缺一个 α-conversion, 它是用来处理 variables 重名冲突的. 最后我们可以也许另外一个新的 term, 但是这个里面忽略了一个重要细节，就是当一个 term 里面存在多个地方可以应用上述规则的时候，我们应当如何选择应用顺序呢? 这里就可以引出两个经典的 reduction strategies: *call by name* 和 *call by value*.

Definition 1.10. In *call by name* reduction strategy, the leftmost redex is always reduced first, and allows no reducations inside abstraction.

Definition 1.11. In *call by value* reduction strategy, a redex is reduced only when its right hand side has reduced to a *value* (varible and abstraction), and allows no reducation inside abstraction.

Annotation 1.12. 注意这两个 reducation strategies 都是不允许在 abstraction 里面做 reduction 的，其实差异就是在做 application 的时候，call by name 是 agruments 接把值替换到 abstraction 里面，而 call by value 是先对 arguments 做 reduction. 当我们选择了一个 reducation strategy，然后对一个 finite term 不断地做 reduction，最终我们会得到一个已经无法再继续做 reduction 的 term，这个 term 就称其为 *normal form*.

Definition 1.13. A term N is in *normal form* is no reduction rule applies to it.

Annotation 1.14. 我们并不打算在先前的 lambda calculus 上建立一个完整的 language，例如引入 bool, natural number 和 test 的定义等. lambda caculus 的引入只是为了进一步说明 evaluation 过程中所需要的 *operational sementics*, 即 small step $t_1 \to t_1'$. 下面我们将 *call by value* 以 inference rules 巧妙地融入 lambda calculus.

Definition 1.15. The untyped lambda calculus is defined as follow:

$$t ::= x \mid \lambda x.\, t \mid t\ t$$
$$v ::= \lambda x.\, t$$

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}\ \text{E-App1}$$
$$\frac{t_2 \to t_2'}{v_1\ t_2' \to v_1\ t_2'}\ \text{E-App2}$$
$$\frac{}{(\lambda x.\, t_1)\ v_1 \to [x \mapsto v_1]t_1}\ \text{E-AppAbs}$$

Annotation 1.16. 关于对 inference rule 的理解，对初次接触它的人并不太好理解. 对于一个 inference rule 中间横线之上我们称为 premises，横线之下我们称其为 conclusion，通常 premises 可以有多个，而 conclusion 只有一个. 例如对于 E-App1 我们可以读作: $t_1\ t_2 \to t_1'\ t_2$ if $t_1 \to t_1$, 因此我们通常是从 conclusion 来考虑 premises, 简单地说就是从下往上读, 这一点尤为重要，它是我们用 inference rules 做 derivation 的基础.

我们来简单地解释一下上面定义的 untyped lambda calculus 的 operational sementics: 首先给出了 terms 和 values(只有 abstractions) 的准确定义，这里只引入了 β-reduction 即 E-AppAbs rule，并且将 *call by value* 也融

入了进去，这体现在 E-App1 rule 规定要先对 leftmost 做 reduction，直到 leftmost 变成了才能 value，我们才可以继续使用 E-App1 rule 往右做 reduction. 后面我们就就将用 evaluation 来代替 reduction 在如今 untyped lambda calculus 中的使用.

Definition 1.17. A *derivation* in up is either an the instance of E-AppAbs or an application of a evalution rule to derivations concluding its premises.

Example 1.18. 我们来举一个关于 $(\lambda x.\, x\ \lambda y.\, y)\ z \to (\lambda y.\, y)\ z$ 例子:

$$\dfrac{\dfrac{}{\lambda x.\, x\ \lambda y.\, y \to \lambda y.\, y}\ \text{E-AppAbs}}{(\lambda x.\, x\ \lambda y.\, y)\ z \to (\lambda y.\, y)\ z}\ \text{E-App1}$$

再想想我们是否能得到关于 $(\lambda x.\, x\ \lambda y.\, y)z \to z$ 的 derivation 呢?

$$\dfrac{?}{(\lambda x.\, x\ \lambda y.\, y)\ z \to z}\ ?$$

显然这里没有合适的 evaluation rule 可以 apply，这就是前面定义 untyped calculus lambda 的精髓，我们只能做 small step.

Annotation 1.19. 我们现在来思考另外一个问题: 如果给定一个 $\lambda x.\, x\lambda y.\, y\ z$，其中 x, y, z 均为 variables. 我们对其做 evaluation 会得到:

$$\lambda x.\, x\ \lambda y.\, yz \to z\ \lambda y.\, y$$

最后的结果奇怪，这并不是我们想要的东西，因为我们通常希望 evaluation 的结果是一个 value. 这就是涉及到我们是否可以在一开始就 refuse 掉可能会产生一个我们不期望的看到的结果呢? 而不是在 evaluation 进行到一半的时候，才恍然大悟. 这时候 type system 将会作为一个最有利的工具来帮助我们完成这个工作. 为了更清晰说明问题，我们还是先给出一个常见的 *pure simply typed lambda calculus*，这就是我们常说的 STLC 的简化版. 我们会直接给出定义，不会在像前面推出 untyped lambda calculus 那样，因为整个过程是相似的，清晰的.

Definition 1.20. The pure simply typed lambda-caculus is defined as follow:

$$t ::= x \mid \lambda x : \tau. t \mid t\ t$$
$$v ::= \lambda x : \tau. t$$

$$\tau ::= \tau \to \tau$$
$$\Gamma ::= \emptyset \mid \Gamma, x : \tau$$

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}\ \text{E-App1}$$
$$\frac{t_2 \to t_2'}{v_1\ t_2' \to v_1\ t_2'}\ \text{E-App2}$$
$$\frac{}{(\lambda x : t. t_1)\ v_1 \to [x \mapsto v_1]t_1}\ \text{E-AppAbs}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{T-Var}$$
$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1. t : \tau_1 \to \tau_2}\ \text{T-Abs}$$
$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}\ \text{T-App}$$

Annotation 1.21. 相比于 untyped lambda calculus 多了关于 type τ, 所有 terms 在 Λ 都可以看做一个 abstraction，它对应的 type 就是 $\tau \to \tau$. 其中 Γ 表示 contexts, i.e., 我们需要某个 term 里面所有 free variables 的 types 才能进一步推导 term 它具有什么 type, 它是可以为 empty set 的, 题外话 Γ 实际上也是可以看做 multi-set 的，即 Γ_1, Γ_2. 这里 judgement(也可以叫 sequent) $\Gamma \vdash t : \tau$ 表示 term t 在 contexts Γ 下具有 type τ，中间这个 $\vdash$ 叫 turnstile. 如果我们要验证这个 judgement 是 valid，就需要一个关于它的 derivation(这里 derivation 的定义和前面类似)，这里 derivation 就需要按照我们这里最下面的三个 typing rules. 这里我给的解释是比较简单，但是对于初次接触的人来说并没有那么容易，我将配合几个例子帮助你理解.

Example 1.22. 我们可以尝试推一下关于 $c : \tau_2 \vdash \lambda x : \tau_1. c : \tau_2$, 其中 c 表示一个 type 为 τ_2 的 constant:

$$\frac{\dfrac{}{c : \tau_2, x : \tau_1 \vdash c : \tau_2}\ \text{T-Var}}{c : \tau_2 \vdash \lambda x : \tau_1. c : \tau_2}\ \text{T-Abs}$$

这里其实隐藏关于 context permutation 的 structural rule 在里面，虽然它看起来还是很自然的，但是我们必须提醒一下未来需要小心的注意这些 structural rule，有可能很显然的 structural rule，你直接引入或者消去你将会得到可能与原来并不等价的 system.

Annotation 1.23. 那么我们将如何把 typing rules 和 evaluation rules 联系起来呢? 我将通过两个非常非常重要的 theorems: *perservation theorem* and *progess theorem* 来说明它们的联系

Information Flow

Noninterference

参考文献

[1] Benjamin C. Pierce. Types and Programming Languages.