

Type System + Security = ?

枫聆

2022 年 7 月 13 日

目录

1	Introduction	2
1.1	Type System	2
1.2	Secure Information Flow	6
1.3	Noninterference	7
2	Type System for Secure Information flow analysis	8
2.1	Operational Semantics	8
2.2	Typing Rules	8

Introduction

Type System

Definition 1.1. 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 [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

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

where $[x \mapsto N]$ is substitution means "replacing all free occurrences 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:

$$\lambda x. M \rightarrow M$$

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 reductions 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* (variable and abstraction), and allows no reduction inside abstraction.

Annotation 1.12. 注意这两个 reduction 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 arguments 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 reduction 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 calculus 的引入只是为了进一步说明 evaluation 过程中所需要的 *operational semantics*, 即 small step $t_1 \rightarrow t'_1$. 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:

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

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2} \text{ E-APP1}$$

$$\frac{t_2 \rightarrow t'_2}{v_1 t'_2 \rightarrow v_1 t_2} \text{ E-APP2}$$

$$\frac{}{(\lambda x. t_1) v_1 \rightarrow [x \mapsto v_1] t_1} \text{ E-APPABS}$$

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: $t_1 t_2 \rightarrow t'_1 t_2$ if $t_1 \rightarrow t'_1$, 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 semantics: 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，我们可以继续使用 E-APP1 rule 往右做 reduction. 后面我们就将用 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 $(\lambda x. x \lambda y. y) z \rightarrow (\lambda y. y) z$ 例子:

$$\frac{\frac{}{\lambda x. x \lambda y. y \rightarrow \lambda y. y} \text{E-APPABS}}{(\lambda x. x \lambda y. y) z \rightarrow (\lambda y. y) z} \text{E-APP1}$$

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

$$\frac{?}{(\lambda x. x \lambda y. y) z \rightarrow z} ?$$

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:

$$\lambda x. x \lambda y. y z \rightarrow z \lambda y. y$$

最后的结果依然是一个 normal form，准确地说是一个 neutral form，即它最左边并不是一个 abstraction. 但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-calculus is defined as follow:

$$\begin{aligned}
t &::= x \mid \lambda x : \tau. t \mid t \ t \\
v &::= \lambda x : \tau. t \\
\tau &::= \tau \rightarrow \tau \\
\Gamma &::= \emptyset \mid \Gamma, x : \tau \\
\\
\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2} &\text{E-APP1} \\
\frac{t_2 \rightarrow t'_2}{v_1 \ t'_2 \rightarrow v_1 \ t'_2} &\text{E-APP2} \\
\frac{}{(\lambda x : t. t_1) \ v_1 \rightarrow [x \mapsto v_1] t_1} &\text{E-APPABS} \\
\\
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} &\text{T-VAR} \\
\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1. t : \tau_1 \rightarrow \tau_2} &\text{T-ABS} \\
\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 \ t_2 : \tau_2} &\text{T-APP}
\end{aligned}$$

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 $\tau \rightarrow \tau$. 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:

$$\frac{}{c : \tau_2, x : \tau_1 \vdash c : \tau_2} \text{T-VAR} \\
\frac{}{c : \tau_2 \vdash \lambda x : \tau_1. c : \tau_2} \text{T-ABS}$$

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, 因为有可能很显然的 structural rule, 你直接引入或者消去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 联系起来呢? 那就不得不提到两个非常非常重要的 theorems: *prograss theorem* and *perservation theorem*. 前者是说如果某个 term 是 well-typed, 那么它是可以继续 evaluation 或者它本身是一个 value. 后者是就更进一步了, 同时这个 term 在 evaluation 的过程中, 它将继续保持对应 type. 有了这两个 theorems, 我们就可以在一开始就做 refuse (i.e., some terms are not well-typed), 这也是一开始就说到的 type system 可以用来抓住我们想要的某些 properties. 它们形式化地表示如下.

Theorem 1.24. Suppose t is well-typed term (this is $\vdash t : \tau$), then either t is value or else there is some t' with $t \rightarrow t'$.

Theorem 1.25. If $\Gamma \vdash t : \tau$ and $t \rightarrow t'$, then $\Gamma \vdash t' : \tau$.

Annotation 1.26. 我并不打算来严格证明的它们, 因为会花费很多篇幅, 同时你可能还需要证明其他的一些 lemma, 例如一些 some structural rules are admissible 和 substitution lemma 等等. 有兴趣的同学可以直接去研究一下, 总体上过程还是非常 straightforward 的. 但是我还是提一下关于证明这些类似 structural proof 结论的时候, 经常会用到一种 structural induction 的方法: 首先给出 height of derivation (the greatest number of successive application of rules in it, where T-VAR have height 0), 然后我们在 height 上做 induction. 再者如果涉及到更精细地证明方式, 我们还会定义 weight of term, 然后在它上面做 induction, 这一类的 induction 可以在证明 cut-elimination 的地方看见.

好了以上就是你在继续往下读所要的关于 type system 的预备知识, 或许很简单, 又或许不太简单, 但是我相信一定会给那些第一次接触到它的人打开另一扇窗.

Secure Information Flow

Annotation 1.27. Information flow 是指程序执行的过程中信息流动, 那么 secure information flow 就是指信息在流动的过程中遵守一些 policies, i.e., 信息我们可以根据 sensitivity 分为两个部分 low 和 high, 简称 L 和 H , 我们允许 L 往 H 流动, 而不允许 H 往 L 流动, 因为 L 往往是假设是可以被观察的, 因此 H 往 L 流动就意味着敏感信息泄露. 同样的思路我们也可以用来描述信息的 integrity 分为 trusted 和 untrusted 吗, 简称为 T 和 U , 我们允许 T 往 U 流动, 而不允许 U 往 T 流动. 那么一个很自然地问题就是如何 model information flow policy? 这也是我们最为感兴趣的地方.

我们将 owners of imformation 表示为程序中的 variables. 对于任意一个 variable x , 我们用 \bar{x} 表示其所具有的 the set of security classes, 其中包含我们前面提到的 sensitivity 和 integrity. 假设 $\bar{x} = \{H, U\}$, 而 $\bar{y} = \{L, T\}$, 在现有 policies 下我们是运行 y 到 x 的流动, 因为我们分别比较了对应 security propoities. 如果某个 expression z 里面包含 x 与 y , 那么我们也可以计算出 $\bar{z} = \{H, U\}$. 这样的思路已经非常明显地对应了用 lattice 来分析问题. 首先我们可以定义一个两个 partial order 分别包含 $L \leq H$ 和 $T \leq U$, 然后再利用前面这两个 poset 定义一个 product partial order:

$$(a_1, b_1) \leq (a_2, b_2) \iff a_1 \leq a_2 \text{ and } b_1 \leq b_2$$

Definition 1.28. Information flow policies are defined by a lattice (SC, \leq) , where SC is a finite set of security classes partially ordered by \leq .

Noninterference

Definition 1.29. We say that a command c satisfies noninterference if equivalent initial memories produce equivalent final memories [2].

Annotation 1.30. 简而言之就是说一段程序在等价的初始状态下，总可以得到等价的结束状态. 其中等价状态和在程序执行过程状态的变化，使我们需要刻画的东西. 在某种程度上它可以作为验证 analysis 的 correctness, 因为我们在做 analysis 时候总是伴随这一些 assumptions, 我们需要验证时在在一定的 assumptions 下我们总能得到正确结果，而不受其他的因素影响.

Type System for Secure Information flow analysis

Operational Semantics

Definition 2.1. We consider a simply common language described below:

$$\begin{array}{ll} (\textit{phrases}) & p ::= e \mid c \\ (\textit{expressions}) & e ::= x \mid l \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 = e_2 \mid e_1 < e_2 \\ (\textit{command}) & c ::= e_1 := e_2 \mid c_1; c_2 \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \end{array}$$

where x ranges over variables, l over locations, and n over integer literals. Integers are the obly values. We use 0 for false and 1 for true, and assume that locations are well ordered [3].

Typing Rules

参考文献

- [1] Benjamin C. Pierce. Types and Programming Languages.
- [2] Andrew Myers. Proving noninterference for a while-language using small-step. operational semantics.
- [3] Dennis Volpano, Geoffrey Smith, Cynthia Irvine. A sound system for secure flow analysis.