Type System + Security = ?

枫聆

2022 年 7 月 14 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 follows:

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

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 follows:

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

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

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-caculus is defined as follows:

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

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

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

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

Annotation 1.21. 相比于 untyped lambda calculus 每个 term 现在是有一个 type τ 标签了, 特别地, abstraction 的 type 形如 $\tau \to \tau$. 我们这里可能与你经常看见的 typed lambda calculus 不太一样，我们给 abstraction 的 arguments 加上了 annotation，这样可以省去了 type inference，实际上这里也不需要. 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 $\Gamma = \Gamma_1 \cup \Gamma_2$. 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，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: *progess 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 $t \to t'$.

Theorem 1.25. If $\Gamma \vdash t : \tau$ and $t \to t'$, 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 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. 但是我还是简要地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. 最后我们简单的来提及一下关于 *subtype polymorphism* 概念.

Definition 1.27. Let τ_1, τ_2 be any types in a system, τ_1 is a *subtype* of τ_2 if any term of type τ_1 can safely be used in context where a term of type τ_2 is expected, simply written $\tau_1 <: \tau_2$.

Example 1.28. 例如我们有这样一个 term $t = \{x_1 : int, x_2 : int\}$ (tuple) 和一个 abstraction $f = \lambda z : \{x_1 : int\}. z.1$, 其中 $z.1$ 表示 fetch tuple 中的第一个元素，即这就是 x_1 元素. 那么 application $f\ t$ 是 well-typed 的吗? 显然是的，因为 t 完全可以被当做 type $\{x_1 : int\}$ 来用，即 $\{x_1 : int, x_2 : int\} <: \{x_1 : int\}$. 前面的思路就是 subtyping 中最重要的一个想法 *subsumption*.

Definition 1.29. The Subsumpting rule defined as:

$$\frac{\Gamma \vdash t : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash t : \tau_2}\ \text{T-Sub}$$

Annotation 1.30. 其中 $\tau_1 <: \tau_2$ 表示一个 subtype relation，这是需要我们额外地来定义，前面例子中提到的 tuple 就是根据它的 width 来定义的，同理你也可以来定义其他 data structures 之间的相关内容. subtype relation 有几个比较明显的性质例如: reflexivity 和 transitivity，这都是很容易证明的. 还有一个比较有趣的是关于 abstractions 之间的 subtype relation 定义:

$$\frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

这就是所谓的"the function arguments are contravariant and the function results are covariant".

Annotation 1.31. 好了以上就是你在继续往下读所需要的关于 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 1.32. 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 $\overline{x}$ 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 $\overline{x} = \{H, U\}$, 而 $\overline{y} = \{L, T\}$, 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 proporties. 这样的思路已经非常明显地对应了用 lattice 来分析问题 [2]. 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 1.33. Information flow policies are defined by a lattice $(SC, \vee, \wedge)$, where SC is a finite set of security classes partially ordered by $\leq$. For any $a, b \in SC$, we define $\vee$ and $\wedge$ as follows

$$b = a \vee b \text{ if } a \leq b$$
$$a = a \wedge b \text{ if } a \leq b$$

Definition 1.34. We define the set of security classes $\overline{e}$ for expression e such that

$$\overline{e} = \bigvee_{x \text{ is free varible in } e} \overline{x}$$

Definition 1.35. We define the set of security classes $\overline{c}$ for command c as follows:

$$\overline{c} = \bigwedge_{x \text{ is assigned to in } c} \overline{x}$$

Definition 1.36. We can give a programming certification condition for representing the information flow policy as follows:

$$\overline{e} \ \leq \ \overline{c}.$$

Noninterference

Definition 1.37. We say that a command c satisfies noninterference if equivalent initial memories produce equivalent final memories [3].

Annotation 1.38. 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, 而我们的验证思路是在一些特定的 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:

$$
\begin{aligned}
&(\textit{phrases}) && p ::= e \mid c \\
&(\textit{expressions}) && e ::= x \mid l \mid n \mid e_1 \oplus e_2 \\
&(\textit{command}) && c ::= e_1 := e_2 \mid c_1; c_2 \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } e \textbf{ do } c \mid \\
& && \qquad \textbf{letvar } x := e \textbf{ in } c
\end{aligned}
$$

where x ranges over variables, l over locations, and n over integer literals, and $\oplus$ includes binary operators $+, -, =, <$. Integers are the obly values. We use 0 for false and 1 for true, and assume that locations are well ordered [4].

Definition 2.2. We define the evalution rules for above language as follows:

$$\frac{}{\mu \vdash n \Rightarrow n}\ \text{E-Base} \qquad \frac{l \in dom(\mu)}{\mu \vdash l \Rightarrow \mu(l)}\ \text{E-Contents}$$

$$\frac{\mu \vdash e_1 \Rightarrow n_1 \quad \mu \vdash e_1 \Rightarrow n_2}{\mu \vdash e_1 \oplus e_2 \Rightarrow n_1 \oplus n_2}\ \text{E-Add}$$

$$\frac{\mu \vdash e \Rightarrow n \quad l \in dom(\mu)}{\mu \vdash l := e \Rightarrow \mu[l \mapsto n]}\ \text{E-Update} \qquad \frac{\mu \vdash c_1 \Rightarrow \mu_1 \quad \mu_1 \vdash \mu_2}{\mu \vdash c_1; c_2 \Rightarrow \mu_2}\ \text{E-Seq}$$

$$\frac{\mu \vdash e \Rightarrow 1 \quad \mu \vdash c_1 \Rightarrow u_1}{\mu \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \Rightarrow \mu_1}\ \text{E-Branch}_1 \qquad \frac{\mu \vdash e \Rightarrow 0 \quad \mu \vdash c_2 \Rightarrow u_2}{\mu \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \Rightarrow \mu_2}\ \text{E-Branch}_2$$

$$\frac{\mu \vdash e \Rightarrow 0}{\mu \vdash \textbf{while } e \textbf{ do } c \Rightarrow u}\ \text{E-Loop}_1 \qquad \frac{\mu \vdash e \Rightarrow 1 \quad \mu \vdash c \Rightarrow \mu_1 \quad \mu_1 \vdash \textbf{while } e \textbf{ do } c \Rightarrow \mu_2}{\mu \vdash \textbf{while } e \textbf{ do } c \Rightarrow u_2}\ \text{E-Loop}_2$$

$$\frac{\mu \vdash e \Rightarrow n \quad l \text{ is the fresh location not in } dom(\mu) \quad \mu[l := n] \vdash [l \mapsto x]c \Rightarrow \mu_1}{\mu \vdash \textbf{letvar } x := e \textbf{ in } c \Rightarrow \mu_1 - l}\ \text{E-Letvar}$$

where μ: *locations*→*values* is a memeory map, using $\mu \vdash e \Rightarrow n$ for producing value by expression (that is expression has no side-effect) and $\mu \vdash c \Rightarrow \mu_1$ for producing new memeory map by command, $\mu[l := n]$ means $\mu(l) = n$ and otherwise $\mu[l \Rightarrow n](l') = \mu(l')$, $\mu - l$ is new meomery map without $l \in dom(\mu)$ and $[l \mapsto x]c$ means that replacing all occurences of x in c with l.

Annotation 2.3. 上述关于 operational semantics 是比较清楚的，值得一提是 letvar 中重新构造了一个 fresh location 来替换掉了原来的 local variable，这样做的好处是省去了构造 local variable 的环节.

Typing Rules

Definition 2.4. The types of above lanaguge are defined as follows:

$$(datatypes) \qquad \tau ::= s$$
$$(phrasetypes) \quad \rho ::= \tau \mid \tau\ var \mid \tau\ cmd$$

where s range over the sef of SC of security classes, which is assumed to be partially ordered by $\leq$. Type $\tau\ var$ is the type of a variable and $\tau\ cmd$ is the type of a command.

Definition 2.5. The typing judgements have the form

$$\Gamma, \Delta \vdash p : \rho$$

where Γ is the context of location typing and Δ is the context of variable typing. The judgement means that phrase p has type ρ under Γ for typing free location in p and Δ for typing free variable in p.

Definition 2.6. The tpying rules of above language are defined as follows:

$$\frac{}{\Gamma, \Delta \vdash n : \tau}\ \text{T-Int} \qquad \frac{x : \tau \in \Delta}{\Gamma, \Delta \vdash x : \tau\ var}\ \text{T-Var} \qquad \frac{l : \tau \in \Gamma}{\Gamma, \Delta \vdash l : \tau\ var}\ \text{T-Loc}$$

$$\frac{\Gamma, \Delta \vdash e_1 : \tau \quad \Gamma, \Delta \vdash e_2 : \tau}{\Gamma, \Delta \vdash e_1 \oplus e_2 : \tau}\ \text{T-Arith} \qquad \frac{\Gamma, \Delta \vdash e : \tau\ var}{\Gamma, \Delta \vdash e : \tau}\ \text{T-RVal}$$

$$\frac{\Gamma, \Delta \vdash e_1 : \tau\ var \quad \Gamma, \Delta \vdash e_2 : \tau}{\Gamma, \Delta \vdash e_1 := e_2 : \tau\ cmd}\ \text{T-Assign} \qquad \frac{\Gamma, \Delta \vdash c_1 : \tau\ cmd \quad \Gamma, \Delta \vdash c_2 : \tau}{\Gamma, \Delta \vdash c_1; c_2 : \tau\ cmd}\ \text{T-Seq}$$

$$\frac{\Gamma, \Delta \vdash e : \tau \quad \Gamma, \Delta \vdash c_1 : \tau\ cmd \quad \Gamma, \Delta \vdash c_2 : \tau}{\Gamma, \Delta \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau\ cmd}\ \text{T-If}$$

$$\frac{\Gamma, \Delta \vdash e : \tau \quad \Gamma, \Delta \vdash c : \tau\ cmd}{\Gamma, \Delta \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau\ cmd}\ \text{T-While}$$

$$\frac{\Gamma, \Delta \vdash e : \tau_1 \quad \Gamma, \Delta, x : \tau_1 \vdash c : \tau\ cmd}{\Gamma, \Delta \vdash \textbf{letvar}\ x := e\ \textbf{in}\ c : \tau\ cmd}\ \text{T-Letvar}$$

$$\frac{\tau_1 \leq \tau_2}{\vdash \tau_1 <: \tau_2}\ \text{T-Base}$$

$$\frac{\tau_2 <: \tau_1}{\vdash \tau_1\ cmd <: \tau_2\ cmd}\ \text{T-Cmd}^{-}$$

$$\frac{\Gamma, \Delta \vdash p : \rho_1 \quad \rho_1 <: \rho_2}{\Gamma, \Delta \vdash p : \rho_2}\ \text{T-Sub}$$

Annotation 2.7. 这里我们实际上仅仅是将 secure information flow analysis 的算法转换称了 typing rules，最终构成了一个 type system，后面我们将在这个 type system 下研究一些有趣的性质.

Soundness

Lemma 2.8. (Structural Subtyping) If $\vdash \rho_1 <: \rho_2$, then either

- if ρ_1 is of the form τ_1 and ρ_2 is of the form τ_2, then $\tau_1 \leq \tau_2$;
- if ρ_1 is of the form τ_1 *var* and ρ_2 is of the form τ_2 *var*, then $\tau_1 = \tau_2$;
- if ρ_1 is of the form τ_1 *cmd* and ρ_2 *cmd* is of the form τ_2 *cmd*, then $\tau_2 \leq \tau_1$;

Annotation 2.9. 我并不打算证明上面的 *structural subtyping lemma*, 因为比较简单但是比较🅴嗦，只需要使用 subtyping rules 即可. 但是它是比较有用的，可以用来帮我们 close 掉 $\rho_1 <: \rho_2$ 所在 branch 的 proof，后续是可以看见的.

Annotation 2.10. 第一个 *simple security lemma* 严格证明了 well-typed expression $e : \tau$ 和前面提到的 $\bar{e}$ 是等价的.

Lemma 2.11. (Simple Security) If $\Gamma, \Delta \vdash e : \tau$, then for every location l in e and $l : \tau_1 \in \Gamma$, we have $\tau_1 \leq \tau$.

证明. 我们可以根据 e 的 structure 来讨论，核心还是在 height of derivation 上做 induction.

Base case: 分别对 last rules 为 T-Int, T-Var 和 T-Loc，这些结果都是显然的.

Induction case: 若 $e = e_1 \oplus e_2$, 并且其 last rule 为 T-Assign:

$$\frac{\Gamma, \Delta \vdash e_1 : \tau \quad \Gamma, \Delta \vdash e_2 : \tau}{\Gamma, \Delta \vdash e_1 \oplus e_2 : \tau}\ \text{T-Arith}$$

对其两个 primises 使用 induction hypothesis 得到命题中结论. 同理 last rule 为T-RVal也一样证明.

特别地，我们最后考虑一下 last rule 为 T-Sub 的 case:

$$\frac{\Gamma, \Delta \vdash e : \tau_1 \quad \tau_1 <: \tau}{\Gamma, \Delta \vdash e : \tau}\ \text{T-Sub}$$

其 right side primise 我们可以用 *structural subtyping lemma* 来 close 掉，因此满足 $\tau_1 \leq \tau$，然后继续对 left side 做 derivation 回到了前面的 cases. □

Annotation 2.12. 第二个 *confinement lemma* 严格证明了 well-typed command $c : \tau$ 和前面提到的 $\bar{c}$ 是等价的.

Lemma 2.13. (Confinement) If $\Gamma, \Delta \vdash c : \tau$, then for every location l assigned to in c and $l : \tau_1 \in \Gamma$, we have $\tau_1 \geq \tau$.

证明. 和证明 *simple security lemma* 做 structural induction, 这里只记录一下几个代表性的 cases:

- 若 $c = e_1 := e_2$, 若 last rule 为 apply T-Assign:

$$\frac{\Gamma, \Delta \vdash e_1 : \tau\ var \quad \Gamma, \Delta \vdash e_2 : \tau}{\Gamma, \Delta \vdash e_1 := e_2 : \tau\ cmd}\ \text{T-Assign}$$

  注意 left side primise，只有 T-Var 可以对其 apply, 这意味着当 assignment 左边被确定之后，它的 type 是不能被更改，这是一个小细节 (从 structural typing lemma 的第二个 case 也能看出来)，再对其用一下 induction hypothesis 即有命题中的结论.

- 若 $c = \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_1$, 若 last rule 为 apply T-If:

$$\frac{\Gamma, \Delta \vdash e : \tau \quad \Gamma, \Delta \vdash c_1 : \tau\ cmd \quad \Gamma, \Delta \vdash c_2 : \tau}{\Gamma, \Delta \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau\ cmd}\ \text{T-If}$$

  显然 assignment 只可能存在于 c_1 或者 c_2, 再对后面两个 primise 使用一下 induction hypothesis 即可得到命题结论.

- 若直接 apply T-Sub:

$$\frac{\Gamma, \Delta \vdash c : \tau_1\ cmd \quad \tau_1 <: \tau\ cmd}{\Gamma, \Delta \vdash c : \tau\ cmd}\ \text{T-Sub}$$

  对其中 right side primise 用一下 *structural typing lemma* 可以得到 $\tau \le \tau_1$, left side primise 又回到了前面的 cases.

□

Definition 2.14. We write $\mu_1 \approx_\tau \mu_2$ to mean $\mu_1(l) = \mu_2(l)$ that for all $l : \tau_1 \in \Gamma$ such that $\tau_1 \le \tau$, where μ_1 and μ_2 are memory maps with same domain.

Theorem 2.15. (Type Soundness) Suppose $\Gamma \vdash c : \rho$, for any evaluation $\mu_1 \vdash c \Rightarrow \mu_1', \mu_2 \vdash c \Rightarrow \mu_2'$ and $\mu_1 \approx_\tau \mu_2$, then $\mu_1' \approx_\tau \mu_2'$.

$$\begin{array}{ccc} \mu_1 & \xrightarrow{\ c\ } & \mu_1' \\ \approx_\tau \| & & \| \approx_\tau \\ \mu_2 & \xrightarrow{\ c\ } & \mu_2' \end{array}$$

证明. 这个证明过程和我们最前面在 pure STLC 中 progress 和 preservation 是非常相似的. 也得先证明一下 progress，你才能对 c 做 structural induction. progress 是显然的，假设 progress 已经被证明了，我们来证一下这里 preservation 中的一个 case: 若 $c = l := e$, 并且 last evaluation rule 为 E-Update. 那么

$$\frac{\mu_1 \vdash e \Rightarrow n_1 \quad l \in dom(\mu_1)}{\mu_1 \vdash l := e \Rightarrow \mu_1[l \mapsto n_1]}\ \text{E-Update} \qquad \frac{\mu_2 \vdash e \Rightarrow n_2 \quad l \in dom(\mu_2)}{\mu_2 \vdash l := e \Rightarrow \mu_2[l \mapsto n_2]}\ \text{E-Update}$$

这里有一个不太方便的地方，就是在给 c 选 typing rules 的时候，总有两种选择:(1 选 T-Assign (2 选 T-Sub. 在原文章 [4] 里面，作者把我们前面介绍的 typing rules 精炼了一下，构造出了一个等价的 machine-orient typing system. 就是 subtyping rule 不再单独存在，而是直接放到了其余的 typing rules 里面，例如 T-Assign 就变成了

$$\frac{\Gamma, \Delta \vdash e_1 : \tau_2 \ var \quad \Gamma, \Delta \vdash e_2 : \tau_2 \quad \tau_1 \leq \tau_2}{\Gamma, \Delta \vdash e_1 := e_2 : \tau_1 \ cmd} \ \text{T-Assign}^*$$

这样我们的选择就变成确定了. 这里不这样做也没有关系，无法就是把 T-Sub 提出来再证一次就行. 设 $\rho = \tau_1 \ cmd$. 若 last typing rule 均为 T-Assign

$$\frac{\Gamma, \Delta \vdash l : \tau_1 \ var \quad \Gamma, \Delta \vdash e : \tau_1}{\Gamma, \Delta \vdash l := e : \tau_1 \ cmd} \ \text{T-Assign}$$

因为命题中我们不考虑 Δ, 因此这里 assignment 的左边只有 locations，这里我们需要分两种情况:

- $\tau_1 \leq \tau$, 由 $\mu_1 \approx_\tau \mu_2$ 显然有 $\mu_1[l \mapsto n_1] \approx_\tau \mu_2[l \mapsto n_2]$. 因为两个 memory maps 都分别新增了一个 type 为 τ_1 的 l.
- $\tau_1 \not\leq \tau$, 和上面情况一样，也有 $\mu_1[l \mapsto n_1] \approx_\tau \mu_2[l \mapsto n_2]$, 只不过现在两个 memory maps 都新增了一个不影响命题判断的 location l，因为我们只 care memory map 上 type" 小于" τ 的 locations.

  若 last typing rule 均为 T-Sub, 即

$$\frac{\Gamma, \Delta \vdash l := e : \tau_2 \ cmd \quad \dfrac{\tau_1 \leq \tau_2}{\tau_2 \ cmd <: \tau_1 \ cmd} \ \text{T-Cmd}^-}{\Gamma, \Delta \vdash l := e : \tau_1 \ cmd} \ \text{T-Sub}$$

依然是分两种情况讨论 τ_2 和 τ 的关系. 这里有比较有趣的情况是一个 last rule 为 T-Assign, 另一个为 T-Sub. 或者两个都 last rule 都是 T-Sub, 但是上面的 τ_2 不一样. 这两个情况我们都可以通过 τ_1 配合 subtyping relation transitivity 来做. □

参考文献

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

[2] D. Bell, L. LaPadula, Secure Computer System: Mathematical Foundations and Model.

[3] Andrew Myers. Proving noninterference for a while-language using small-step. operational semantics.

[4] Dennis Volpano, Geoffrey Smith, Cynthia Irvine. A sound system for secure flow analysis.