计算机研究与发展
Journal of Computer Research and Development
ISSN 1000-1239/CN 11-1777/TP
43(1):15~22, 2006

一种用于 Java 虚拟机的类型化低级语言

陈 晖 陈意云 吴 萍 项 森
(中国科学技术大学计算机科学与技术系 合肥 230027)
(hchen@mail.ustc.edu.cn)

A Typed Low-Level Language Used in Java Virtual Machine

Chen Hui, Chen Yiyun, Wu Ping, and Xiang Sen
(*Department of Computer Science and Technology, University of Science and Technology of China, Hefei* 230027)

Abstract In the past ten years, there has been a trend in the field of trustworthy computing: building high-assurance software system based on programming languages and compilers. The most obvious advantage of these techniques is reducing trusted computing base of software system. Moreover the language-based techniques are suitable to describe and verify fine-grained safety policies. Inspired by these researches TLL is designed. It is expected to be a type-safe intermediate language used in the just-in-time compiler of Java virtual machine. The work described in this paper is based on Intel ORP, and aims at building a smaller trusted computing base. Compared with JVML, TLL is closer to the assemble language, and hence is convenient to encode high-level primitive efficiently. TLL type system is derived on polymorphic typed lambda calculus, which is expressive and general to encode various high-level language features. For case study, the self-application semantic, one of the most important safety properties of object-oriented language, is expressed and assured. A prototype using TLL as intermediate language in the just-in-time compiler can be granted as a starting point for building Java virtual machine with tiny trusted computing base.

Key words typed language; code safety; certifying compilation

摘 要 为了能够减小运算系统的需信任计算基础、描述较小粒度的安全策略,目前的研究倾向于从程序设计语言和编译器入手来提高软件的安全性. 基于以上研究背景设计了一种类型化的低级语言 TLL. TLL 是一种为 Java 虚拟机即时编译器设计的类型安全中间语言,以构造一个具有更小需信任计算基础的 Java 虚拟机系统为目的. TLL 的类型系统基于多态的类型化 λ 演算,它具有丰富的表现力且能够编码各种高级语言的抽象. 基于 TLL 的一个虚拟机原型系统已经实现,它可以作为实现一个高安全且面向多种源语言的运行时系统的起点.

关键词 类型化语言;代码安全;验证编译

中图法分类号 TP312

1 引 言

由于 Internet 的爆发式增长和广泛应用,移动计算吸引了越来越多的关注,在这个领域中程序的安全问题显得至关重要. 当用户从网上下载一段代码(浏览器插件、Java 小程序)到主机上,总是希望它的运行不会危害主机. 一个安全主机(safe host)应

收稿日期:2004-07-19;修回日期:2005-04-15
基金项目:国家自然科学基金项目(60173049,60473068)

该能够拒绝可能会破坏主机的代码,而允许那些符合一定的安全策略的不信任方代码在其上运行.检验并执行从网络获得的移动代码的 Java 虚拟机正是一个典型的例子.这样的安全主机一方面要对外来的代码进行检验,以保护自己;另一方面检验的正确与否又依赖于对某些代码(代码检验程序)的信任.需信任计算基础(trusted computing base,TCB)这个概念通常被用来评价一个系统的易被攻击性.TCB 指的是实现系统的代码中不能证明为安全而不得不给予信任的那些部分.这就意味着 TCB 中可能含有安全漏洞,系统的安全依赖于 TCB 中代码的正确性.一个系统的 TCB 包含的代码越多,可能存在的安全问题就越多,因而被击破的可能性就越大.一个 Java 虚拟机的实现通常包含了大量的代码,它的各个部分例如即时编译器(just-in-time compiler,JIT)、垃圾收集器通常都含有近万行代码.在大多数 Java 虚拟机的实现中,这些代码都包含在 TCB 之内.

在过去的 10 年中,出现了许多利用基于语言的技术来确保低级语言代码安全性的研究.其中较具影响的研究包括携带证明代码(proof-carrying code,PCC)[1]、类型化汇编语言(typed assembly language,TAL)[2]和高阶类型化语言 FLINT[3].这些研究工作表明了利用类型和逻辑能够有效地验证代码的安全性质.PCC 使用一种通用逻辑来表示机器代码的安全属性及其证明,通过对证明的验证来确保对应的机器代码是满足代码消费者定义的安全策略的.TAL 的类型系统保证那些通过类型检查的良类型的汇编程序不会出现意想不到的错误行为.基于 PCC 思想构造的一个 Java 程序运行系统的 TCB 只包含了不到 2700 行代码[4],而 TAL 也将编译器排除出 TCB.这些工作的共同之处是通过验证编译器将从源程序获得的有用信息以不同的表示形式(断言注释、类型注释)保持到目标代码中,这些信息将被用于目标代码安全属性的验证.

出于同样目的,我们设计了一种用于 Java 虚拟机的类型化低级语言(typed low-level language,TLL),作为 JIT 编译器的中间语言.在现有的 Java 虚拟机上,将 Java 虚拟机语言(简称 JVML,又称 JVM 字节码)程序翻译成本的机器代码在字节码验证之后,它并不能保证经过验证的属性在翻译过程中能够得到保持,因此 JIT 在 Java 虚拟机系统的 TCB 中.使用 TLL 可以降低代码验证的层次,同时也减小了虚拟机的 TCB,因为 JIT 的前端可以排除出 TCB.我们基于 Intel 的 Java 虚拟机实现开放运行平台[5](open runtime platform,ORP),构建了一个用 TLL 作为验证语言的 JIT 编译器.由于 TLL 的低级和通用性,使 ORP 有可能成为支持多种语言程序验证和运行的平台.

我们基本实现了对 JVML 语言的支持.为了更清晰和简练地描述所用的方法,本文的讨论基于 JVML 的一个精简子集 $JVML_0$.本文的第 2 节将简要介绍 $JVML_0$;第 3 节将形式描述目标语言 TLL 的语法、操作语义和定型规则(typing rule);第 4 节与相关的工作做比较;第 5 节对本文工作做一个总结.

2 源语言 $JVML_0$

源语言 $JVML_0$ 是 JVML 语言的一个精简子集,图 1 显示了 $JVML_0$ 的语法.它包含了 JVML 的面向对象语言的基本特征:$JVML_0$ 程序的基本元素为类文件;每个类文件描述了一个 Java 类及类之间的继承关系;描述了数据、方法成员的类型以及方法的字节码序列.这个子集对 JVML 做了一些简化,例如仅保留了类的单继承关系、原子类型中的整数类型,以及类成员的访问属性都为公共"public".这个简化并不妨碍在 Java 虚拟机上讨论从面向对象语言到一个有高阶类型的低级语言的类型保持(type preserving)翻译.

Class File::= C {*super_class* C
fields(f_i : $Type_i$)*
Methods(m_i : $Type_i$; $Body_i$)*}
Type::= C | *int* | *void* | ($Type$)* → $Type$
Body::= ($Inst$)*
Inst::= pop | pop2 | dup | maths_op | iload | istore | if(op) | goto | return | new | invokespecial | getfield | putfield | invokevirtual

Fig. 1 Syntax of the source language JVML0.

图 1 源语言 JVML0 的语法

JVML 语言是一个与平台无关的语言,它用于描述在具有程序计数器、能够动态分配空间的堆、操作数栈、局部变量集合的抽象机器上进行的计算.在 Freud 等人的文章[6]中证明了类似于 $JVML_0$ 的一个 JVML 子集的类型安全性质.

3 类型化低级语言 TLL

TLL 语言比 JVML 语言更接近机器代码,但为了兼顾表示的简洁性及 JIT 编译模式的需要,它又

保持了某些高级语言的特征. 一个 TLL 程序大致可以描绘为若干个全局变量声明了函数和静态分配的数据;函数由指令序列构成;全局变量、函数以及函数内跳转目标标签有显式的类型标注. 一些 JIT 编译器中间语言也保留了部分 Java 源程序类型信息. 与它们不同,TLL 的类型系统采用了从系统 F (system F)[7] 演化而来的高阶类型系统. 这样的高阶类型系统更具有一般形式,能够表示多种高级数据抽象和安全属性,为使 TLL 能够面向多种源语言提供了基础. 本节余下内容给出 TLL 的形式描述.

3.1 语　法

图 2 展示了 TLL 的语法. 一个 TLL 程序是由以下 4 个元素描述:堆(H)、变量表(E)、寄存器表(R)和语句(包括全局变量声明 g 和指令 ι)序列(I). H,E,R 分别为堆值标签(h)、全局变量(x)和虚拟寄存器到它们所对应的值的映射集合. 堆值标签对应数据元组和代码块(通称堆值,在图 2 中标识为 b);全局变量和虚拟寄存器则对应单值(w 可以是整数或是指向某个堆值的指针). 简单来说,一个 TLL 程序的执行就是语句序列作用在堆、变量表、寄存器表上,使它们发生状态的转变.

Kinds	$\kappa ::= T \mid Row \mid Code \mid \kappa_1 \rightarrow \kappa_2$
Types	$t ::= \alpha \mid c \mid \Gamma \mid \tau \mid \forall \alpha \leqslant t_1 :\kappa . t_2 \mid \mathrm{rec}\ \alpha :\kappa . t \mid \exists \alpha \leqslant t_1 :\kappa . t_2$
Small Velue Types	$c ::= void \mid int \mid \forall \alpha \leqslant \kappa . c_1 \times \cdots \times c_n \rightarrow c' \mid \mathrm{ref}\ \tau \mid (\ominus)^+ \mid \mathrm{null}$
Register File Types	$\Gamma ::= \forall \overrightarrow{\alpha :\kappa \leqslant t} . \{\overrightarrow{r :c}\}^*$
Heap Value Types	$\tau = \langle (l :c^{\varphi})^* \rangle \varphi = + \mid -$
Type Coercions	$\delta ::= [t] \mid \mathrm{roll}^c \mid \mathrm{unroll}^c \mid \mathrm{epack}^c [t, v]$
Statements	$I ::= (L :\Gamma)^+ \iota \mid g\ ;I$
Global Variants	$g ::= x :c = w \mid x = Fn[\alpha](c_1 \times \cdots \times c_n) :c \{I\}$
Instructions	$\iota ::= \mathrm{bop}\ r_1, v_1, v_2 \mid \mathrm{assign}\ v_1, v_2 \mid \mathrm{jmp}[t]\ L \mid b\_con\ r, L[t] \mid \mathrm{call}\ r, v[t], v_1, \cdots, v_n \mid \mathrm{eopen}\ \alpha, r, v \mid \mathrm{halloc}\ r, v, \tau \mid \mathrm{hread}\ r_1, r, v \mid \mathrm{hwrite}\ r, v_1, v_2 \mid \mathrm{halt}$
Operands	$v ::= w \mid r \mid x \mid \delta[v] \quad w = \mathrm{const} \mid l \mid \delta[w] \mid ns \quad l ::= \mathrm{off} \quad r = r_1 \mid r_2 \mid \cdots$
RegisterFile	$R ::= \{r_1 \rightarrow w_1, \cdots, r_n \rightarrow w_n\}$
Variables Tables	$E ::= \{x_1 \rightarrow w_1, \cdots, x_n \rightarrow w_n\}$
Heap	$H ::= \{h_1 :\rightarrow b_1, \cdots, h_n \rightarrow b_n\}$
Heap Values	$b ::= \langle l_1 :w_1, \cdots l_n :w_n \rangle \mid \mathrm{code}\langle I \rangle$
Program	$P ::= \langle H, E, R, I \rangle$

Fig. 2　Syntax of target language TLL.

图 2　目标语言 TLL 语法

TLL 抽象机器中的寄存器数目被认为是无限多个,用来存放运算的中间结果. 在代码发射过程中它们将作为物理寄存器分配的候选,因此被称为“虚拟”寄存器.

TLL 的指令要比 JVML 指令更加低级:堆值操作指令 halloc,hread,hwrite 取代了针对面向对象的对象产生指令和对象成员访问指令;更一般的函数调用指令 call 取代了对象方法调用指令 invokevirtual, invokespecial. 除了基本的赋值指令、二元运算指令以及分支转移指令,TLL 指令集还包括一些涉及类型操作的指令,例如指令 eopen 用来打开一个存在类型的包,它在操作语义上与赋值语句并无差别,只是目的操作数的类型发生了变化. 指令的操作数可以是单值、虚拟寄存器、全局变量或者是经过某种类型强制(type coercion)的单值.

TLL 类型系统采用了从系统 F 演变而来的高阶类型系统. 种类(kind)定义了高阶类型中类型变量 α 的论域(domain). 种类 T 包含了单值类型:原子类型、指针类型、函数类型;种类 Row 包含了标注元组中各个域的值类型的堆值类型 τ;而种类 $\kappa_1 \rightarrow \kappa_2$ 则包含了各种类型构造符,例如指针类型构造符 ref 属于种类 $Row \rightarrow T$. F-bounded 中的高阶类型,例如多态类型($\forall \alpha \leqslant t_1 :\kappa . t_2$)、存在类型($\exists \alpha \leqslant t_1 :\kappa . t_2$)以及递归类型($\mathrm{rec}\ \alpha :\kappa . t$)具有丰富的表现力. 多态类型概括了具有相似结构的一类类型的集合,用类型变量来抽象其中可变的部分;当具体到一个类型实例时则用具体的类型来取代抽象类型变量. 存在类型则为数据抽象和信息隐藏提供了基础. 一个存在类型 $\exists \alpha :\kappa . c$ 的值可以由一个原来类型为 $c[t/\alpha]$ 的值 w 通过类型强制 $epack^c[t, w]$ 隐藏了类型 t 得到. 在使用这个存在包(existential package)时隐藏类型 t 将不为用户代码所知. 在 TLL 的类型系统中采用了同构递归(iso-recursive)(另一种递归为等价递归 equi-recursive),它的特征是递归类型 $rec\ \alpha . t$ 与它的一步展开 $t[rec\ \alpha . t/\alpha]$ 需要显式的 roll 和 unroll 类型强制来转换,采用这种显式类型强制的好处是避免了 F-bounded 多态类型子定型不可判定的缺陷[8].

而在函数内部,TLL 通过检查标签类型来确保控制流的安全转移. 标签类型表现为寄存器表类型(Γ),它是对一个寄存器集合的类型指派,也就是指令序列中用到的虚拟寄存器的类型前条件.

3.2 操作语义

语言的类型安全性质由语言操作语义和静态语义之间的一种关系来描述,具备这种性质的语言将保证良形(well-formed)的程序不会出错. 因此在讨论 TLL 类型安全属性之前必须先给出 TLL 的操作语义和静态语义. TLL 的操作语义由一组求值和归

约规则来描述,在表 1 中给出. 表中的标记 $E(x)$ 表示在变量表 E 中绑定在变量 x 上的值;标记 $E\{x\rightarrow w\}$ 表示了变量表 E',它与 E 的不同仅在于 x 的值绑定为 w. 类似的还有关于堆和寄存器表的标记 $H(h)$,$R(r)$,$H\{h\rightarrow w\}$,$R\{r\rightarrow w\}$. 标记 $|v|$ 表示 $E(x)$,$R(r)$ 或是一个 w.

Table 1 Operational Semantics of TLL

表 1 TLL 的操作语义

$P=\langle H,E,R,g'\mid \iota;I\rangle\rightarrow P'$	
$g'=x:c=w$	$P'=\langle H,E\cup\{x\rightarrow w\},R,I\rangle$
$g'=x=Fn[\alpha](c_1\times\cdots\times c_n):c\{I\}$	$P'=\langle H\{h\rightarrow code\langle I\rangle\},E\cup\{x\rightarrow h\},R,I\rangle$
$\iota=$ assign v_1,v_2	If $v_1=r$ then $P'=(H,E,R\{r\rightarrow w\},I)$
	If $v_1=x$ then $P'=(H,E\{x\rightarrow w\},R,I)$ 其中,$w=\mid v_2\mid$
$\iota=$ bop r_1,v_1,v_2	$P'=(H,E,R\{r\rightarrow w\},I)$ where $w=\text{bop}(\mid v_1\mid,\mid v_2\mid)$
$\iota=b\_con\ r,\ L[t]$	If $R(r)$ satisfies condition then $P'=(H,E,R,I'[t/\alpha])$
	else $P'=(H,E,R,I)$ where$E(L)=h\ H(h)=code\langle I'\rangle$
$\iota=$ call $r,x[t],v_1,\cdots,v_n$	If$(H,E\cup\{arg_1\rightarrow w_1,\cdots,arg_n\rightarrow w_n\},\varnothing,I'[t/\alpha])$
	$\rightarrow(H',E'\cup\{arg_1\rightarrow w'_1,\cdots,arg_n\rightarrow w'_n\},R',w')$
	then $P'=(H',E',R\{r\rightarrow w'\},I)$ where $w_i=\mid v_i\mid$
$\iota=$ return r	$P'=(H,E,R,w)$ where $w=R(r)$
$\iota=$ halloc $r,v,\langle l_1:c_1,\cdots l_n:c_n\rangle$	$P'=(H\cup\{h\rightarrow\langle l_1:ns,\cdots l_n:ns\rangle\},E,R\{r\rightarrow h\},I)$
	where $\mid v\mid=\text{sizeof}(\langle l_1:ns,\cdots l_n:ns\rangle)$
$\iota=$ hread r,v,l_i	$P'=(H,E,R\{r\rightarrow w_i\},I)$
	where $\mid v\mid=h\ H(h)=\langle l_1:w_1,\cdots,l_i:w_i,\cdots,l_n:w_n\rangle$
$\iota=$ hwrite v_1,r,l_i	$P'=(H\{h\rightarrow\langle l_1:w_1\cdots,l_i:w_i,\cdots,l_n:w_n\rangle\},E,R,I)$
	where $\mid v_1\mid=h\ R(r)=w_i$
$\iota=$ eopen α,r,v	$P'=(H,E,R\{r\rightarrow w\},I[t/\alpha])$ where $\mid v\mid=epack^c[t,w]$

关系 $P\rightarrow P'$ 描述一个 TLL 程序到另一个 TLL 程序的转换,反映在变量声明和指令作用下程序状态四元组的变化. 一个 TLL 程序的执行总是从一个初始状态$\langle\varnothing,\varnothing,\varnothing,I\rangle$开始,其中良形的程序总是以 halt 指令结束或是求值成功,终结在某个状态$\langle H,E,R,\text{halt}\rangle$或$\langle H,E,R,v\rangle$. 表 1 中的每条规则描述了 TLL 语句对程序四元组的改变. 例如,堆运算指令则改变了堆的状态:指令 halloc $r,v,\langle l_1:c_1,\cdots l_n:c_n\rangle$ 在堆中分配一个大小为 $|v|$ 未初始化数据元组,而类型$\langle l_1:c_1,\cdots l_n:c_n\rangle$则决定了这个数据元组中每个域的类型;堆值读写指令 hread r_1,r,l_i 和 hwrite r,v,l_i,将访问由值 r 所指向的数据元组中特定的域,域在数据元组的偏移由 l_i 给出. 赋值指令、算术运算指令以及函数调用与返回指令具有通常语言中的意义. 而对于 TLL 中的类型操作指令 eopen α,r,v,在擦除类型后它的操作语义和赋值指令并无区别,将引入一个自由类型变量来代替存在包中的隐藏类型.

3.3 静态语义

TLL 的静态语义定义怎样的 TLL 程序是良形的. 对于类型化语言而言,良形即意味着良类型(well-typed),静态语义规则也就是定型规则. 图 3 中列出了部分 TLL 的定型规则. 类型检查器将根据定型规则来检查程序是否是良类型的. 这些判定规则的判定上下文中包括了以下 4 种符号:类型上下文 Δ 描述上下文中自由类型变量;具有形式 $\{h_1:t_1,\cdots,h_n:t_n\}$ 的堆类型 Ψ 描述堆中各个堆标签所对应堆值的类型;具有形式 $\{x_1:c_1,\cdots,x_n:c_n\}$ 的全局变量表类型 B 描述变量表中各个变量的类型以及寄存器表类型 Γ.

TLL 的定型规则可以分为 4 类判定. 第 1 类是关于类型的判定规则,包括判定类型是否为良形及其所属种类的规则以及判定类型之间子定型(subtyping)关系的规则. 子定型规则保证函数的实参是形参的子类型时程序仍然能够正常执行. 例如规则 T7 说明了数据元组的子定型规则:如果元组

τ_1 包含 τ_2 的所有偏移,且这些域的类型是 τ_2 中相应域的类型或是子类型,那么 τ_1 为 τ_2 的子类型. 这条规则允许一个运算能够对那些在相同偏移处具有相同类型的元组进行相同的操作. 在 TLL 中元组类型的域索引为整数偏移量 l_i. 为了叙述简便和易于理解,本文把 l_i 写成字符串标识的形式,它对应着某个固定的整数的偏移量. 元组类型是否被初始化由标志 φ 标识:+ 为已初始化;- 为未初始化.

TLL 程序的良形性由它的 4 个组成部分(H, E, R, I)的良形性来定义,规则 P1 正说明这点. 第 2 类判定给出这 4 个组成部分的良形性规则;第 3 类判定为值类型判定,图 3 列出的值类型判定主要说明在类型强制作用下值的类型变化;第 4 类判定递归地定义指令序列的良形性:如果在上下文 Δ', Ψ', B', Γ'及 c 中判定指令序列 I 为良形的,那么在上下文 Δ, Ψ, B, Γ, c 中指令序列 ι, I 是良形的,其中$\langle\Delta',\Psi',B',\Gamma'\rangle$为$\langle\Delta,\Psi,B,\Gamma\rangle$在指令 ι 作用下产生的上下文,而类型 c 则指出期望指令序列返回的值类型. 因此对函数中代码序列的检查遵循这样的过程:从初始的判定上下文开始依次检查每条指令;检查完一条指令后根据指令的定型规则为下一条指令的检查产生一个新的上下文. 每个指令序列总是以如以下 3 条指令结束:控制流转移指令;返回指令 return;程序终结指令 halt. 因此序列的检查总是以相应的 3 种检查结束:当前的上下文是否满足跳转目标的标签类型(规则 I2 及 I4);返回值类型是否符合函数声明(规则 I3);程序终结指令 halt 无条件为良类型(规则 I9)结束.

Type and Subtyping Judgments:

$$\frac{}{\triangleright int :T}(\mathrm{T1})\quad \frac{\Delta\triangleright t_i :T}{\Delta\triangleright\langle l_1 :t_1,\cdots,l_n :t_n\rangle :Row}(\mathrm{T2})\quad \frac{\Delta\triangleright\tau :Row}{\Delta\triangleright\mathrm{ref}\,\tau :T}(\mathrm{T3})\quad \frac{\triangleright\tau :H}{\triangleright\mathrm{null}\leqslant\mathrm{ref}\,\tau ?}(\mathrm{T4})\quad \frac{\triangleright\tau :H}{\triangleright\mathrm{ref}\,\tau\leqslant\mathrm{ref}\,\tau ?}(\mathrm{T5})\quad \frac{\Delta\triangleright\tau_1\leqslant\tau_2}{\Delta\triangleright\mathrm{ref}\,\tau_1\leqslant\mathrm{ref}\,\tau_2}(\mathrm{T6})$$

$$\frac{\Delta\triangleright c_i^{\varphi_i}\leqslant c_i'^{\varphi_i} :T(\text{for } 1\leqslant i\leqslant n)\qquad m\geqslant n\ l_i'\in\{l_1,\cdots,l_m\}}{\Delta\triangleright\langle l_1 :c_1^{\varphi_1},\cdots,l_m :c_m^{\varphi_m}\rangle\leqslant\langle l'_1 :c_1^{\phi_1},\cdots,l'_n :c_n'^{\varphi_n}\rangle}(\mathrm{T7})$$

$$\frac{\Delta\triangleright c_1\leqslant c_2 :T}{\Delta\triangleright c_1^{+}\leqslant c_2^{-}}(\mathrm{T8})\qquad \Gamma_1=\forall\overrightarrow{\alpha\leqslant t :\kappa}.\ \{\overrightarrow{r_i :t_i},\overrightarrow{r_j :t_j}\}\ \Gamma_2=\forall\overrightarrow{\beta\leqslant t :\kappa}.\ \{\overrightarrow{r_i :t_i}\}$$

$$\frac{\Delta\ \{\overrightarrow{\alpha\leqslant t :\kappa},\overrightarrow{\beta\leqslant t :\kappa}\},B,\Psi\triangleright t_i\alpha\neq\beta}{\Delta,B,\Psi\triangleright\Gamma_1\leqslant\Gamma_2}(\mathrm{T9})$$

$$\frac{\Delta,\alpha :\kappa\triangleright s_1\leqslant s_2\Delta,\alpha\leqslant s_1\triangleright t_1\leqslant t_2}{\Delta\triangleright\exists\alpha\leqslant s_1 :\kappa.\,t_1\leqslant\exists\alpha\leqslant s_2 :\kappa.\,t_2}(\mathrm{T10})\quad \frac{\alpha\leqslant\beta :\kappa,\Delta\triangleright t_1\leqslant t_2}{\Delta\triangleright\mathrm{rec}\ \alpha.\,t_1\leqslant\mathrm{rec}\ \beta.\,t_2 :\kappa}(\mathrm{T11})$$

$$\alpha :\kappa_1,\Delta\triangleright t_{21}\leqslant t_{11} :\kappa_1\qquad \frac{\alpha\leqslant t_{21} :\kappa_1,\Delta\triangleright t_{12}\leqslant t_{22} :\kappa_2}{\Delta\triangleright\forall\alpha\leqslant t_{11}.\,t_{12}\leqslant\forall\alpha\leqslant t_{21}.\,t_{22} :\kappa_1\to\kappa_2}(\mathrm{T12})$$

Program Judgments:

$$\triangleright H :\Psi\ \varnothing,\Psi\triangleright E :B\ \varnothing,\Psi,B\triangleright g_i :c_i\ g_i ::= x_i :c_i=w_i$$

$$\frac{\varnothing,\Psi\triangleright R :\Gamma\ \varnothing,\Psi,B\cup\{\overrightarrow{c_i :c_i}\},\Gamma,void\triangleright I}{\triangleright\langle H,E,R,g_i ;\cdots ;g_n\ I\rangle}(\mathrm{P1})\quad \frac{\triangleright b_i :t_i\triangleright t_i :Row\ \text{or}\triangleright t_i :Code}{\triangleright H=\{\!\{h_i\to b_i\}\!\}*\}\ :\Psi=\{\!\{h_i\to t_i\}\!\}*\}}(\mathrm{P2})\quad \frac{\Delta\triangleright c :\kappa\ \Delta,\Psi\triangleright w :c}{\Delta,\Psi\triangleright x :c=w :c}(\mathrm{P3})$$

$$\frac{\Delta,\Psi\triangleright E :B\ \Delta,\Psi,B\triangleright g :c\ g ::= x :c=w}{\Delta,\Psi,\triangleright E\cup\{x\to w\} :B\cup\{x :c\}}(\mathrm{P4})\quad \frac{\Delta\triangleright c_i :\kappa\ \Delta\cup\{\alpha\},\Psi,B\cup\{f :c_1\times\cdots\times c_n\to c,arg_1 :c_1,\cdots,arg_n :c_n,L\},\varnothing,C\triangleright I}{\Delta,\Psi\triangleright x=Fn[\alpha,\kappa](c_1\times\cdots\times c_n) :c\ \{I\}\ :\forall\alpha :\kappa.\,c_1\times\cdots\times c_n\to c}(\mathrm{P5})$$

Value Judgements:

$$\frac{\Delta\triangleright t_1\leqslant t_2 :\kappa\ \Delta,\Psi,B,\Gamma\triangleright v :t_1}{\Delta,\Psi,B,\Gamma\triangleright v :t_2}(\mathrm{V1})\quad \frac{\Delta,\Psi,B,\Gamma\triangleright v :t[\mathrm{rec}\ \alpha.\,t/\alpha]}{\Delta,\Psi,B,\Gamma\triangleright roll(v) :rec\ \alpha.\,t}(\mathrm{V2})\quad \frac{\Delta,\Psi,B,\Gamma\triangleright v :rec\ \alpha.\,t}{\Delta,\Psi,B,\Gamma\triangleright unroll(v) :t[rec\ \alpha.\,t/\alpha]}(\mathrm{V3})$$

$$\frac{\Delta,\Psi,B,\Gamma\triangleright v :\forall\alpha\leqslant t_1 :\kappa_1.\,t_2 :\kappa_2\ \alpha\in FV(t_1)\qquad \Delta\triangleright t'\leqslant[t'/\alpha]\,t :\kappa}{\Delta\triangleright unroll(t')=t_3\,\Delta\triangleright t_3\leqslant t_1[t'/\alpha] :\kappa_1}(\mathrm{V4})\quad \frac{\Delta,\Psi,B,\Gamma\triangleright v :[t'/\alpha]t_1}{\Delta,\Psi,B,\Gamma\triangleright epack^{\exists\alpha\leqslant t.t_1}[t',v] :\exists\alpha\leqslant t :\kappa.\,t_1}(\mathrm{V5})$$

Instruction Judgments:

$$\frac{\Delta,\Psi,B,\Gamma\triangleright v_1 :c\ \Delta,\Psi,B,\Gamma\triangleright v_2 :c}{\Delta,\Psi,B,\Gamma\triangleright\mathrm{bop}\ r,v_1,v_2 ;I}(\mathrm{I1})\qquad \Delta,\Psi,B,\Gamma\triangleright r :int\qquad \Delta,B\triangleright f :c_1\times\cdots\times c_n\to c$$

$$\frac{\Delta,B\triangleright L :\Gamma'\ \Delta\triangleright\Gamma\leqslant\Gamma'\ \Delta,\Psi,B,\Gamma,c\triangleright I}{\Delta,\Psi,B,\Gamma,c\triangleright b\_con\ r,L ;I}(\mathrm{I2})\quad \frac{\Delta,\Psi,B,\Gamma,c\triangleright v :c}{\Delta,\Psi,B,\Gamma,c\triangleright\mathrm{return}\ v}(\mathrm{I3})$$

$$\Delta,B\triangleright L :\Gamma'\ \Delta\triangleright\Gamma\leqslant\Gamma'\qquad \Delta,\Psi,B,\Gamma\triangleright v :int\qquad \Delta\triangleright\tau\leqslant\langle l_1 :c_1^{\varphi},\cdots,l_i :c_i^{+},\cdots,l_n :c_n^{\varphi}\rangle$$

$$\frac{\Delta,\Psi,B,\Gamma,c\triangleright I}{\Delta,\Psi,B,\Gamma,c\triangleright\mathrm{jmp}\ L ;I}(\mathrm{I4})\quad \frac{\Delta,\Psi,B,\Gamma\{r=\mathrm{ref}\langle l_1 :c_1,\cdots,l_n :c_n\rangle^{-}\},c\triangleright I}{\Delta,\Psi,B,\Gamma,c\triangleright\mathrm{halloc}\ r,v,\langle l_1 :c_1,\cdots,l_n :c_n\rangle ;I}(\mathrm{I5})\quad \frac{\Delta,\Psi,B,\Gamma\triangleright r :\mathrm{ref}\ \tau\ \Delta,\Psi,B,\Gamma,\{r_1 :c_i\},c\triangleright I}{\Delta,\Psi,B,\Gamma,c\triangleright\mathrm{hread}\ r_1,r,l_i ;I}(\mathrm{I6})$$

$$\frac{\Delta,B\triangleright x :\forall\alpha\leqslant t' :\kappa.\,c_1\times\cdots\times c_n\to c\ \Delta,\Psi,B,\Gamma\triangleright v_i :c_i'\qquad \Delta,\Psi,B,\Gamma\triangleright v :\exists\alpha\leqslant t :\kappa.\,t'\ \alpha\notin\Delta}{\Delta\triangleright t\leqslant t' :\kappa\ \Delta\triangleright c_i'\leqslant c_i[t/\alpha]\ \Delta,\Psi,B,\Gamma\{r :c\},c\triangleright I}(\mathrm{I7})\quad \frac{\Delta\{\alpha\leqslant t :\kappa\},\Psi,B,\Gamma\{r :t'\}\triangleright I}{\Delta,\Psi,B,\Gamma\triangleright\mathrm{eopen}\ \alpha,r,v ;I}(\mathrm{I8})\quad \frac{}{\Delta,\Psi,B,\Gamma,c\triangleright halt ;I}(\mathrm{I9})$$

Fig. 3　TLL static semantics.

图 3　目标语言 TLL 语法

在定义了 TLL 的操作语义和静态语义之后,按照类型安全性质的一般证明方式,通过对以下两个定理的证明将进一步得到 TLL 是类型安全的这一推论.

定理 1. 保持性(preservation). 如果 $\triangleright P$ 并且 $P\to P'$,那么 $\triangleright P'$.

定理 2. 前进性(progress). 如果 $\triangleright P$,那么必然存在着 p' 使得 $P\to P'$,或是 P 具有程序终结状态:

①$\langle H,E,R,\text{halt}\rangle$;②$\langle H,E,R,v\rangle$.

保持性定理描述了这样一个性质:如果以一个良类型的程序状态开始,那么每一步的执行也会到达一个良类型的状态.前进性定理则说明这种执行不会突然终止,除非计算已经全部完成.这两个定理的证明通过对语句归纳定义证得.类型安全性质是以上这两个定理的推论:

推论 1.类型安全性或类型可靠性.如果 P 是良类型的,即 $\triangleright P$,那么 P 的执行不会终结于非良类型的程序状态 P'.

4 对象编码(object encoding)

将 JVML_0 翻译成 TLL 后,TLL 的类型检查需要确保高级属性在 TLL 代码中仍然得到满足.对于文章的源语言 JVML_0 来说,对象的自应用语义(self-application semantic)是 TLL 需要确保的重要属性.对象的自应用语义要求:调用对象的方法时,传给方法的第 1 个参数必须是对象本身[9].本节介绍 TLL 如何用一般的类型系统进行对象编码保持高级语言中的这一性质.

4.1 编码思想

对象编码最早出现在对面向对象语言特性的研究中,它以类型论的观点来考察对象系统,用基于类型论的语言结构刻画对象抽象.对象编码不仅是研究面向对象语言特性的重要方法,而且在面向对象语言的类型保持翻译中扮演重要角色,它为低级语言用更加基本的类型和操作原语来实现面向对象特征提供了方法.

基于一些面向对象语言特性分析的研究工作[10],League 等人[11]和 Glew[8]分别提出了针对 Flint 和 TAL 的实用且可扩展的对象编码方案.这些编码语言的类型系统同样也是从 F 系统衍变而来,因此这两种编码方法都能够在 TLL 上实现.我们沿用了 Glew 的 self 类型编码方法.这种方法引入了类型限定符 self,它能够用图 2 给出的语法表示:$\text{self}\ \alpha:\kappa.F(\alpha)\equiv\exists\alpha\leqslant F(\alpha):\kappa..\alpha$.从 F-bounded 存在类型的消除和引入规则能够推导出类似于文献[8]所描述的 self 限定符的相应规则(pack 和 unpack 相对于存在类型中的 epack 和 eopen):

$$\frac{\Delta\triangleright\text{unroll}(t_1)=t_2\quad\Delta\triangleright t_2\leqslant t[t_1/\alpha]\quad\Delta,\Psi,B,\Gamma\triangleright v:t_1}{\Delta,\Psi,B,\Gamma\triangleright pack^{\text{self}\ \alpha:\kappa.t}(v):\text{self}\ \alpha:\kappa.t}\ (\text{self-pack})$$

$$\frac{\Delta\triangleright v:\text{self}\ \alpha:\kappa.c\ \alpha\neq\Delta\quad\Delta\{\alpha\leqslant c:\kappa\},\Psi,B,\Gamma\{r:c\},c\triangleright I}{\Delta,\Psi,B,\Gamma,c\triangleright\text{unpack}\ \alpha,r,v;I}\ (\text{self-unpack})$$

下面的叙述仍然使用 self 限定符,可以把它当做 TLL 语言的一个类型宏.同时使用标记[.]TLL 来表示"…的 TLL 翻译".类 C 的对象引用的类型翻译为

$$ObjTP_C\equiv\text{self}\ \alpha:\text{T}.ref\ [C]_{rf}(\alpha),$$

其中,$[C]_{rf}(\alpha)\equiv\langle vtb:\text{ref}\ VtbTp_C[\alpha],(l_{fi}:[Type_i]_{TLL^+})*\rangle$ $[C]_{rf}(\alpha)$ 体现了一个对象元组中各个域的类型,其中 $Type_i$ 为类 C 对象各个数据成员的 Java 类型.一个元组除了包含实例数据之外还包含了该类对象的方法表指针.类的方法表类型为

$$VtbTp_C\equiv\forall\alpha\leqslant\text{ref}\ [C]_{rf}(\alpha):T.\ \langle(l_{mi}:\alpha\times t_1\times\cdots\times t_n\to t)*\rangle.$$

方法表中的每个方法的第 1 个参数都为对象本身,也就是"this".而方法表的 F-bounded 多态则体现了子类可以调用父类定义的方法,子类对象可以作为第 1 个参数传递给函数.$ObjTP_C$ 中的方法表类型由 self 限定符绑定的 α 实例化,类型变量 α 表示了对象本身的类型.

4.2 实例分析

下面的例子将说明 self 对象编码方法是如何实现对象的自应用语义的.

```
class Wind {
  superclass Object;
  fields weight:int;
  methods play:void→void {…};
class Brass {
  superclass Wind;
  fields color:int;
  methods play:void→void {…},
          adjust:void→void {…};
```

类 Brass 为类 Wind 的子类,与类 Wind 相比,它增加了新的成员域 color 和方法 adjust,并且重写了方法 play.根据如上所述的对象编码规则,类 Wind 和类 Brass 的对象引用被翻译成的元组指针类型分别为

$ObjTp_{\text{Wind}}\equiv\text{self}\ \alpha:Row.\langle vtb:\text{ref}\langle play:\alpha\to void\rangle,weight:\text{int}\rangle$

$ObjTp_{Brass}\equiv\text{self}\ \alpha:Row.\langle vtb:\text{ref}\langle play:\alpha\to void,adjust:\alpha\to void\rangle,weight:\text{int},color:\text{int}\rangle$

根据元组和存在类型的子定型判定规则 T7 和 T10 可得 $ObjTp_{\text{Brass}}\leqslant ObjTp_{\text{Wind}}$,与类的继承关系

保持一致. 一个 Brass 对象的 paly 方法调用翻译为图 4 所显示的 TLL 指令序列. 第 1 条指令 unpack 具有和赋值语句一样的操作语义,目标操作数 r_0 和 r_1 具有相同的值:同一个 Brass 对象的指针不同之处在于目标操作数 r_1 的类型是一个新引入的类型变量 α(规则 self-unpack). 我们可以从图 4 右栏看到类型检查时类型上下文的这一变化:指令 unpack 在原有的上下文 Δ_0 增加 α 得到新的上下文 Δ_1. 我们用 $\Delta\{\cdots\}$ 表示用花括号中的类型变量约束来更新类型上下文 Δ 得到的新的上下文,图中 $\Gamma\{\cdots\}$ 也具有类似的含义. α 是一个单元素类型,它仅包含一个元素,即刚刚执行完 unpack 操作的对象. α 出现在方法表的类型中,方法表的每一个函数的类型限定了第 1 个参数类型必须是 α,而不能是 Brass 类的其他对象. 而在一些面向对象语言的翻译并不能准确地描述这种限制.

TLL Instructions	Type Context (Δ,Γ)During Type-Check
	Δ_0,Γ_0 $\Gamma_0(r_0)=\text{self}\ \alpha.\text{ref}\langle vtb:\text{ref}\langle play:\alpha\rightarrow void, adjust:\alpha\rightarrow void\rangle, weight:\text{int}, color:\text{int}\rangle$
unpack α,r_1,r_0	$\Delta_1=\Delta_0\{\alpha\leqslant\text{ref}\langle mtb:\text{ref}\langle play:\alpha\rightarrow void, adjust:\alpha\rightarrow void\rangle, weight:\text{int}, color:\text{int}\rangle: T\}, \Gamma_1=\Gamma_0\{r_1:\alpha\}$,
hread r_2,r_1,vtb	$\Delta_2=\Delta_1,\Gamma_2=\Gamma_1\{r_1:\text{ref}\langle play:\alpha\rightarrow void, adjust:\alpha\rightarrow void\rangle\}$
hread $r_3,r_2,play$	$\Delta_3=\Delta_2,\Gamma_3=\Gamma_2\{r_3:\alpha\rightarrow void\}$
call $r_3,\langle r_1\rangle$	

Fig. 4　TLL instructions to invoke Brass.play.

图 4　调用 Brass.play 的 TLL 指令序列及类型检查过程中上下文的变化

5　相关工作

本文的工作借鉴了 PCC,TAL 和 FLINT 这些关于验证编译和携带证明代码的研究. 整个验证编译领域的研究基于这样的思想:验证编译结果具备某个属性要比验证整个编译器实现的正确性要简单得多,这也是这篇文章中描述的 TLL 工作的出发点.

最有名的验证编译器即 Sun 的从 Java 到 JVML 的 javac 编译器,本文已提到 JVML 并不适合用来达到低级代码安全性的目的. 它的原语过于高层,不能进行高效的编码. 而一些用于现有 Java 虚拟机的中间表示虽然具有低级的原语,但是往往沿用面向对象的类型信息. TLL 的原语比 JVML 低级,并且采用一般形式的类型系统.

FLINT 作为一种类型化的通用中间语言被提出来,它能够支持包括 SML,Java 在内的高阶语言. 但是由于它是一种函数式语言,在从栈式的 JVML 到 FLINT 编译时,需要引入另一种中间语言 λJVML[12]. 相比较而言,接近三地址指令的 TLL 指令较为低级,从 JVML 到 TLL 的翻译更加直接.

PCC 提出了一种移动代码传送和执行的模式:代码产生方用验证编译器产生具有注释的代码和证明;代码使用方利用注释来检查携带的证明是否是代码安全属性的一个合法证明;在网络上传递的是携带证明的代码. 与 PCC 不同本文的工作旨在利用基于语言的技术,在不改变 Java 程序运行模式的前提下,提高现有虚拟机的安全性. 虽然 JVML 仍然是用于网络传递的语言,但是我们将 JVML 的类型信息保持到低级语言,在这个更低级语言的程序上做安全性验证,将达到减少系统 TCB 和提高系统安全性的目的.

与更加低级的 TAL 相比,TLL 具有一些高级语言的特征. 目前实现的一些基于 TAL 的验证编译器都是静态编译器,而 TLL 是用于 Java 虚拟机中的 JIT 编译器. TLL 和 TAL 最大的区别在对函数类型的处理上. 在 TAL 程序中函数调用点活动栈和寄存器的变化被反应在类型信息中,它的函数入口标签类型表现为函数调用点的栈及寄存器的类型前条件和后条件. 这些函数入口标签类型是相当一致的,而且很容易转化为由调用参数类型和返回值类型构成的高级函数类型. 因此 TLL 中保持了高级函数类型,隐藏了函数调用时的具体操作,能够具有更加紧凑的类型表达方式.

我们在原型系统上进行了测试(由于源语言子集选取的原因,测试例的规模都较小). 实验结果显示,由于增加了类型信息维护和类型检查,原型系统运行这些例子的时间要比 ORP 原有的 O3JIT 编译器平均多 1/3. 性能的降低体现了用性能换取安全的策略. 在未来的工作中,TLL 代码的优化可以使性能提高. 而且当运行规模更大的程序时,编译阶段花费的时间在整个运行时间比例将会大大下降,性能降低这一缺憾可以被弥补.

6　结　　论

用基于语言的技术来考虑软件安全是目前产生高可信软件的研究热点. 从编程语言和编译器出发考虑软件安全,有利于减小运算系统的 TCB、描述较小粒度的安全策略. 本文工作设计了一种用于

Java 虚拟机的类型化中间语言 TLL. 它具有较低级的原语和由系统 F 演化而来的一般化的类型系统. 在此基础上构造的虚拟机原型系统具有比原有虚拟机更小的 TCB,可以作为构建高安全、面向多种源语言的运行时系统的基础.

参 考 文 献

1 G. Necula. Proof-carrying code. ACM Symp. Principles of Programming Language, New York, 1997

2 G. Morrisett, D. Walker, K. Crary, *et al*. From system F to typed assembly language. ACM Trans. Programming Languages and Systems, 1999, 21(3):528~569

3 Z. Shao. Typed common intermediate format. USENIX Conf. Domain-Specific Languages, Santa Barbara, California, 1997

4 A. W. Appel, N. Michael, A. Stump, *et al*. A trustworthy proof checker. Princeton CS, Tech. Rep.:TR-648-02, 2002

5 Intel Microprocessor System Lab. Open runtime platform, open source dynamic computing research platform. http://orp.sourceforge.net/, 2001

6 S. Freund, John Mitchell. A type system for object initialization in the Java bytecode language. ACM Trans. Programming Languages and Systems, 1999, 21(6):1196~1250

7 J. Y. Girard, P. Taylor, Y. Lafont. Proofs and Types. London:Cambridge University Press, 1989

8 Neal Glew. An efficient class and object encoding. STAR Lab, Tech. Rep.:STAR-TR-00.07-02, 2000

9 Samuel Kamin. Inheritance in Smalltalk-80: A denotational definition. The 15th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, 1988

10 K. B. Bruce, L. Cardelli, B. C. Pierce. Comparing object encodings. Information and Computation, 1999, 155(1/2):108~133

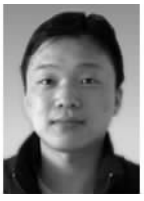
11 C. League, Z. Shao, V. Trifonov. Type-preserving compiler of feather-weight Java. ACM Trans. Programming Language and Systems, 2002, 24(2):112~152

12 C. League, Z. Shao, V. Trifonov. Precision in practice:A type-preserving Java compiler. The 12th Int'l Conf. Compiler Construction (CC'03), Warsaw, Poland, 2003

Chen Hui, born in 1979. Ph. D. candidate. Her main research interests include type theory, and language-based code safety.
陈晖,1979 年生,博士研究生,主要研究方向为类型论、基于语言的代码安全.

Chen Yiyun, born in 1946. Professor and Ph. D. supervisor. His major research interests includes programming languages theory and implementation, formal description and verification, and software safety.
陈意云,1946 年生,教授,博士生导师,主要研究方向为程序设计语言理论和实现技术、形式化描述技术、软件安全等.

Wu Ping, born in 1978. Ph. D. candidate. Her main research interests include model checking, program analysis technique.
吴萍,1978 年生,博士研究生,主要研究方向为模型检查、程序分析技术.



Xiang Sen, born in 1979. Ph. D. candidate. His main research interests include formal verification of low-level code.
项森,1979 年生 . 博士研究生,主要研究方向为低层代码的形式化验证.

Research Background

The explosive growth of World Wide Web makes software penetrate every corner of the society. The safety and security properties of software are more crucial than ever before. To increase software soundness, it is an important approach for developers to strictly specify, reason, and verify software based on programming language design. A popular view of trustworthy software is that software, and even operating system itself, should be implemented on the basis of a safe and secure language and its runtime environment. We have proposed a research on generating trustworthy software by combining the state-of-the-art logic-based and type-based techniques. This research work is supported by the Natural Science Foundation of China under grant No. 60173049 and No.60473068. As one part of this research we propose a scheme to build safer virtual machine based on a typed low-level language, TLL. This paper introduces TLL and gives its formal description. Refined type system helps preserve safety information from source down to low-level code to support safety verification. A prototype is built by reconstructing Intel's JVM implementation, ORP, in which the main part of JIT compiler is removed from TCB. Applying the refined type system and Hoare-style reasoning in the verification of other parts of the runtime system (e.g. garbage collector, thread control module) is our future work. When actually accomplished, a virtual machine with tiny TCB will be the basis of high-assurance runtime system.